TRUN

An Arduino Programming Language for Primary School Students

Project Report

SW414F19

Aalborg University
Electronics and IT

AALBORG UNIVERSITY
STUDENT REPORT

Electronics and IT
Aalborg University
http://www.aau.dk

Title:
TRUN

Theme:
Design, definition and implementation of programming languages

Project Period:
Spring Semester 2019

Project Group:
sw414f19

Participant(s):
Abiram Mohanaraj
Christoffer Hansen
Elisabeth Niemeyer Laursen
Fredrik De Frène
Simon Manojlovic
Simon Park Kærgaard

Supervisor(s):
Razvan-Gabriel Cirstea

Copies: 1

Page Numbers: 151

Date of Completion:
May 28, 2019

Abstract:

The purpose of this project is to design a new programming language suitable for teaching 7-9th grade students to learn programming through physical visualisation. The visualisation used for the programming language is an Arduino car constructed of LEGO. The designed language has statements for driving and turning the car.
A syntactical specification is given using a CFG and a semantic specification is given using structural operational semantic. For the programming language, a working compiler is implemented in Java using ANTLR for constructing the lexer and parser part of the compiler. The compiler translates to C code.
A usability test was conducted to evaluate the ease of the new programming language compared to the existing language for Arduino. The language is found to be more user friendly compared to the existing Arduino language.

Preface

This report is written by group sw414f19, 4th semester on Aalborg University, and turned in 28th May 2019. The project theme for the semester is "*Design, Definition and Implementation of Programming Languages*". The report is split into four parts: Introduction & Analysis, Design & Implementation, Evaluation, and Appendices.

Credits
We would like to thank our supervisor Razvan-Gabriel Cirstea for his guidance throughout the project. Also a thank to the lecturers Hans Hüttel and Bent Thomsen for giving advice.

We would also like to give credits to the tools utilised for this project. Firstly ANTLR, which has helped us generating the lexer and parser for the compiler. Secondly www.mshang.ca/syntree as a tool that generated a visual abstract syntax tree for the report.

Contents

Part I

Introduction & Analysis

1 Introduction

Since 2014, Dansk Erhverv has tried to incorporate creative Information Technology (IT) courses in the Danish public schools. Programming is intended to be introduced to the students in a creative way to encourage their interest in IT, which might help to further inform the students' choice of higher education, should they wish to pursue a more technical education. With an introduction to programming and different IT-systems, the students should also learn about security, which can contribute to the safety in their daily use of technology [1].

Recently, Merete Riisager, the Education Minister of Denmark, has pushed to include programming in the current syllabus, so the students can achieve a greater understanding of computers and other technologies. With the current rapid technological development, a high influx of people with a prominent knowledge of software will be needed in the labour market, which also makes it relevant for school children to learn programming. Already next school year (fall 2019), the students in 40-50 different public schools will be introduced to a course called "Teknologiforståelse" (Understanding of Technology), where they will be face topics such as programming and IT-security. Should the course after a three year trial be acknowledged it may be included in the syllabus for all public schools [2]. Globally, the programming course has already been added to the syllabus of public schools in Bulgaria, Estonia, Greece, Poland, Portugal and England [1].

Teknologiforståelse will become an independent course in some schools. Other schools will try the course as an extension to other already existing courses, such as Danish, Math, Social-studies, Arts, Physics, Chemistry, Crafts and Design, and Biology/Technology [3].

1.1 Initial Problem

Programming languages helps a programmer write software in an efficient manner, granted the programmer is familiar with the language. A programming language can be defined as a set of grammatical rules for instructing a computer to perform specific tasks [4]. However, the process of learning a programming language is long, and the learning curve is often steep. Beginners in particular struggle to learn coding when they need to implement ideas and con-

cepts of programming [5]. A programming language suitable for beginners should therefore be easy to understand.

Learning how to program in general proves to hold a lot of difficulties due to the required skills such as abstraction, generalisation etc. the complex syntax of programming languages cause problems for beginners, mainly because most programming languages are developed for professional use and not to support learning [6]. Programming with Arduino is often thought of as a good introduction to programming as it gives visual feedback. When programming with the Arduino, a programming language with a similar syntax to the high-level language C++ is currently used. C++ was designed for high-level enterprise programming which is not suited for beginners. Therefore, a new simplified programming language would be a suitable tool for beginners to learn programming.

This leads to the following initial problem:

The Arduino programming language is difficult to understand for beginners in terms of the functionality. Ideas and concepts of programming is hard to learn efficiently as it is time consuming and complicated.

2 | Problem Analysis

To achieve a greater understanding of the initial problem, further knowledge regarding certain aspects of the problem should be analysed. This chapter will therefore focus on explaining the Arduino and belonging programming language. Thereafter, the target group for a solution will also be specified. Lastly, four existing programming languages will be analysed followed by an analysis of programming paradigms. A conclusion to the problem analysis will be provided in Chapter 3.

2.1 Arduino

Arduino is an open-source electronics platform created with accessible hardware and software alike. The Arduino boards are programmable and are single-board microcontrollers, making them ideal for creating both simple and complex projects [7]. There is a wide variety of available boards from Arduino that provide great amounts of flexibility in regards to projects for the user. However the specific board used for the project and presented in this section is the *Arduino UNO*.

Arduino microcontrollers are primarily programmed using the Arduino programming language, which is a C-like subset of C++. Any constructs supported by the compiler (avr-g++ [8]) can be used when programming the Arduino [9]. However, as the Arduino microcontrollers run compiled code and not interpreted code, any high level language can be used if there exists a compiler for it.

2.1.1 Arduino UNO

The Arduino is a full microcontroller with a microprocessor, memory, and I/O pins, both digital and analog. The Arduino UNO microcontroller is the ATmega328p [10].

The Arduino UNO board has physical pins that restricts the capacity of the boards functions. **Figure 2.1** gives an overview of the Arduino Uno's interfaces.

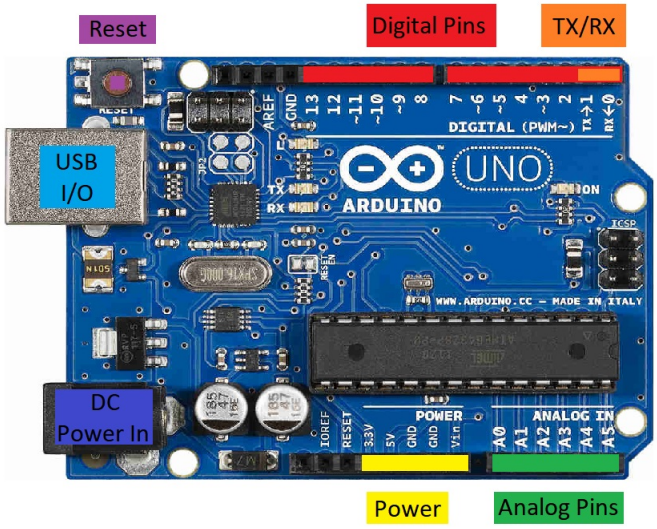

Figure 2.1: Overview of Arduino board.

The Arduino UNO has 14 digital pins and 6 analog pins. The pins can be seen on **Figure 2.1**. Of the 14 digital pins there are 6 pins that allows for Pulse Width Modulation (PWM). Digital pins 0 and 1 are used to receive and transmit serial data. Furthermore some of the digital pins can work as interrupt channels.

When creating an Arduino program it is the digital and analog pins that are used to light LEDs, spin motors or read data from sensors. An example of a program can be to set digital pin 12 to output 5V which in turn can power an LED or a motor. Moreover the analog pins can be used to read data from a sensor that measures distance to objects.

2.2 Target Group

For this project the chosen target group is children attending public schools to fit the issue elaborated in **Chapter 1**. When creating a programming language suitable for this particular target group, it is important to make it easy to learn in an environment that supports the children's interest in programming and IT systems. The project will primarily take into consideration students in the grades 7-9, therefore the mathematical complexity should be fitting for a student at that level. It should be simple to create small programs, which are easy for the user to both write and understand.

2.3 Language Evaluation Criteria

When designing or evaluating a programming language there are three general criteria to evaluate the language. In this section these criteria will be presented and discussed in regards to the targeted language. Lastly, a conclusion will be given deciding the most relevant aspects of the criteria for the chosen target group.

Before presenting the criteria it is important to note that the definition of criteria is vague and partly subjective. The criteria definition used in this report is as described in Sebesta[11, Section 1.3].

2.3.1 Readability

Readability is the ease with which a programmer can read and understand a program written in a specific language [11, p. 31]. For a language to be readable it must reflect a degree of simplicity. For example having relatively few basic constructs will make the language both easier to read and also easier to learn. Readability also increases the opportunity for programs being maintainable, which for many problem domains is essential [11, p. 32].

Readability is an important criteria considering the target group. For beginners it is vital to understand the elementary programming principles before being able to be efficient at writing programs.

2.3.2 Writability

"Writability is a measure of how easily a language can be used to create programs for a chosen problem domain" [11, p. 36]. Writing programs easily is most affected by the expressivity of a language. The expressivity of a language is the way the language is typed. A language has a high

expressivity if it has a convenient, typically meaning short, way of writing an expression or a construct.

When considering the target group of the language, writability is of less importance compared to readability. When learning a language it is more important to first understand the language and general programming principles before being efficient at writing code.

2.3.3 Reliability

Reliability is affected by the two other categories, readability & writability, by influencing the users' natural ways of expressing themselves. Being able to write code that is both easy to read and understand decreases programming errors [11, p. 39]. Besides, having exception handling allows for ensuring programs work under the expected circumstances [11, p. 38]. Having a solid framework of type checking ensures the programmer gets the errors during compile time and can fix their mistakes in the early process of writing programs. In general a compile time error is preferred over a run-time error.

Aliasing is a common source of programming errors. Aliasing is defined as having the same name for multiple variables that has a reference to the same piece of memory. Because of the many cases where a programming error can arise aliasing is preferred not to be allowed in a language targeted at beginners. In any case restricting aliasing increases reliability [11, p. 38].

2.3.4 Conclusion

Some important points have been made regarding how to design the syntax of the language. All three criteria are important to consider when designing the language. Readability is considered very important as the goal is to teach programming to beginners. If the language has a high level of readability, it is easier for beginners to acquire the understanding of programming concepts and constructs.

Writability in turn is also important mainly in regard to expressivity. When programming as a beginner it is important to be able to get thoughts and ideas expressed in a programming language with ease. Distractions should therefore be minimal. However, it is not preferred to compromise readability for writability.

Also reliability is important to reduce programming errors. It is desired to catch errors during compile time rather than at run-time since run-time errors are more expensive. Aliasing is a common source of errors and is preferred to be disallowed in a language targeted at beginners.

2.4 Arduino Programming Language

The difficulties faced when programming with the Arduino programming language will be discussed to get familiar with what to improve.

As mentioned briefly in **Section 2.1**, the Arduino programming language is a C-like subset of C++. It is a stripped down version of C++, meaning some of the standard libraries are unsupported [9]. Many of the restrictions made are to help the programmer avoid using too many resources, because of the lack of said resources on the Arduino hardware. In practice, this means the syntax is identical to that of C++. Hence, when analysing the difficulties with the syntax of the Arduino programming language, one can instead analyse C++ and in turn C.

C++ is a high-level programming language which adds object-oriented features to its predecessor, C. Although both are high-level languages, they are closer to assembly language than most other high-level languages [12][13]. This allows programmers to write very efficient code in terms of resources required, which is why it is a popular language to use when programming micro controllers and embedded systems. This can, however, make the language difficult to learn for entry-level or inexperienced programmers, when comparing its low-level nature to other more English-like programming languages.

```
/* Small Arduino program that
*  can be used to drive a car
*  forward for 10 seconds */

const int leftwheel_pin8 = 8;
const int rightwheel_pin7 = 7;

void setup() {
  pinMode(leftwheel_pin8, OUTPUT);
  pinMode(rightwheel_pin7, OUTPUT);
}

void loop() {
  digitalWrite(leftwheel_pin8, HIGH); // Make both
  digitalWrite(rightwheel_pin7, HIGH); // wheels spin

  delay(10000); // Delay for 10 seconds

  digitalWrite(leftwheel_pin8, LOW); // Stop
  digitalWrite(rightwheel_pin7, LOW); // both wheels
}
```

Figure 2.2: Code for small Arduino program.

The code snippet in **Figure 2.2** depicts a small program in the Arduino programming language that emits a high signal to two pins on an Arduino board. This could for example power two motors which would be connected to a wheel each in order to drive a robotic vehicle forward.

The code in **Figure 2.2** is rather long compared to the functionality that the car executes. The code is tainted by repetition when specifying which pins to set high and low. The code in the snippet does not relay to the reader that it is moving a car (except for the comments).

When programming with the Arduino, the code must consist of the two code blocks setup() and loop(). The setup() block only runs once in the beginning, which can be used as preparation to the main block. Setup() is used to do initial task like initialising variables, pin modes or receiving initial input from a user or sensor. Thereafter, the loop() block runs, which includes the active control of the Arduino board. The lines will be run sequentially in the block and at the end return to the top of the block to run again. [14]

To summarise, C/C++ is generally more difficult to understand for beginners with little or no programming experience. The Arduino programming language can add to this difficulty by including libraries that introduces the user to new, unexplained functions. Furthermore the programmer needs a basic understanding of the Arduino hardware, which adds an additional layer of complexity between the mapping from code to real-world execution.

2.5 Analysis of Languages

Three existing programming languages will be analysed. The analysis will consist of an examination of syntax and features as well as advantages and disadvantages. The purpose of the analysis is to get inspiration for the definition of the syntax of the new language.

The languages chosen are Python, Java, and Pascal. Python was chosen mainly because of the minimal amount of code needed to create a working program. Java has high readability and is a widely used programming language and was therefore also chosen. Pascal was chosen because it is a language that was designed for learning programming. The concepts in the syntax of Pascal are interesting to analyse.

2.5.1 Python

Python is an interpreted and object-oriented programming language. The reason for analysing Python is because of the possibility to write applications for both small and large tasks in a clear way [15]. This is a substantial feature to look into when it comes to designing a programming language.

Python has a simple and consistent syntax [16]. Furthermore, it features a dynamic type sys-

tem as well as automatic memory management e.g. garbage collection. The Python Standard library is also very extensive and offers large functionality for the programmer.

Considering these features, Python's own philosophy is that their language gives beginners the ability to concentrate on the important skills in programming and learn them quicker. In that way students will be able to: *"be assigned programming projects very early in the course that do something."* [16]. In other words the language allows for quicker learning and implementation of basic programming concepts.

A substantial thing to note when looking at the syntax of Python is its use of English keywords instead of operators and no need for a main function.

```
for x in sequence:
    print(x)
```

Figure 2.3: Syntax for creating a for-loop in Python

Figure 2.3 depicts an example of a for-loop written in Python. Python's for-loop is a powerful collection iterator and is easily read. Compared to C's notation:

```
for (int i = 0; i < sizeof(collection); i++) {
    printf("%s", collection[i]);
}
```

The Python code can be read almost entirely in words in a simple way. The same applies to the logical operators. Python uses English keywords for them as well: *and*, *or*, and *not*. This feature increases the readability as it allows the reader to read expressions like a sentence.

Another interesting part of the Python syntax is its way of structuring code blocks. Rather than curly brackets, Python uses indentation to delimit code blocks. This creates a connection between the visual structure and the semantic structure. It encourages the programmer to always write structured code without affecting writability of the language. This structure also enhances readability.

Python's dynamic type system allows for use of variables without declaring their type. The type of variables are inferred by the value assigned to a variable.

```
name = "Jack"
age = 25
weight = 85.4
```

Figure 2.4: Syntax for assinging values to variables in Python

The example on **Figure 2.4** shows how the values "Jack", 25 and 85.4 are assigned to the *name*, *age*, and *weight* variables, respectively. Each of these values are of different types namely

string, integer and float. Python lets you assign all these values to a variable without a type declaration. This feature increases the writability of the language but also in turn decreases readability.

To sum up, Python is a good example of a language that lets it's users code in a simple way while keeping away from statically typed languages like C, Java and C++. It also does not have a main function like the aforementioned languages.

2.5.2 Java

Java is a general-purpose, concurrent, object oriented programming language similar to C++. It also possesses features which make the language beginner friendly. These features will be discussed in the following.

In the Java programming language, complexities such as pointers, operator overloading, which appear in C++ and various other languages, have been removed or have been implemented on an abstract level which makes it easier for beginners to use. Java is also platform independent and thereby *portable*. After writing a simple program, it is possible to run the program on another machine as long as the machine has the Java runtime environment [17]. However, coding in Java requires plenty of code for a relatively small task. For example a simple program which prints "Hello World!" would require about 5 lines of code, while an equivalent program in e.g. Python would only require one line.

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Figure 2.5: Syntax for a "Hello World"- program in Java

In **Figure 2.5** a simple *Hello World* - program is shown, displaying the amount of code a simple program requires. In the same figure, there are details which can be abstracted, which could be helpful for novices who would not have any understanding of the details.

In Java, the primitives are explicitly declared. An example of how the primitives are declared in Java is shown in **Figure 2.6**. This increases the readability of the code as the type of the variable is visible in the code. However, it would also increase the difficulty of writeability as the developer would have to explicitly declare the variable type every time a new variable is introduced.

```
int x;
double y;
float z;
char q;
```

Figure 2.6: Syntax for a data types in Java

Another rather unique iterative control structure in Java, compared to the other langauges presented in this section, is the do-while loop which is presented in **Figure 2.7**.

```
1    do {
2            ...
3    } while(boolean expression);
```

Figure 2.7: Syntax for do-while loop in Java

In the do-while control structure in Java, the body or the statement of the loop will be executed a single time before evaluating the continue expression (line 3 on **Figure 2.7**). Depending on this evaluation, the loop will either continue or stop. This type of iterative control structure can be useful when the body of the loop has to be executed at least once before evaluating whether the loop should continue.

As a result, Java has a number of features and syntax constructions which could be valuable to consider when developing a beginner friendly language. This includes that details should be abstracted from the developer and explicitly declared data types increase the readability of the language.

2.5.3 Pascal

Pascal is an imperative and procedural programming language developed by Niklaus Wirth with the purpose of education [18].

It was designed for use in a teaching environment in an era where computers were extremely limited in terms of access. The interest in Pascal arises from the fact that it was designed for teaching students programming, although before the big multi paradigm of current programming languages. An analysis of Pascal's syntax will provide insight and act as an inspiration to concepts that can be brought into a new programming language.

On **Figure 2.8** is an example of a small program in Pascal. First is an overview of the variables used in the program and their types. It is easy to identify scopes with the *begin* and *end* keywords as indicators. A program written in Pascal reads like a sentence where end finishes with a dot.

```
1  program repeatUntilLoop;
2  var
3     a: integer;
4
5  begin
6     a := 10;
7     (* repeat until loop execution *)
8     repeat
9        writeln('value of a: ', a);
10        a := a + 1
11     until a = 20
12 end.
```

Figure 2.8: Syntax for a repeat until- program in Pascal[19].

The iterative structure on **Figure 2.8** (lines 8-11) can be easily read and understood. Iterative structures are achievable in most programming languages but this structure in Pascal is particular noteworthy.

In Pascal global variables must be declared after the *var* keyword. Also every time a scope is entered local variables must be declared before any statements with the *var* keyword. This gives the language structure when typing and reading a program [20].

2.5.4 Evaluation

When evaluating the aforementioned languages and their approaches to syntax, the emphasis should be put on their ability to be read and written by inexperienced programmers.

Python's syntax is similar to the English language when typed. This can be helpful for beginners to better understand programming. The incentive of making the user write structured code is also an aspect to consider when designing the language. Another beginner friendly aspect of Python is its ability to create programs with minimal amount of code compared to the other languages.

The dynamic type system that Python offers has both advantages and disadvantages. For an experienced programmer, a dynamic type system can be an improvement to the ease of use, while for inexperienced programmers, an explicit type system might yield a deeper understanding of what operations are permitted on the different data types.

Java requires more code to create simple programs, however, the language offers higher levels of abstractions over complex concepts. The language is statically typed which makes it more readable.

Pascal offers readable iterative control structures that are worth considering in a beginner

friendly programming language. Also the begin and end keywords makes it clear where a scope starts and ends.

The terms used to express keywords and functionality within the program should be simple to understand for young students and therefore also avoid abbreviations of words (e.g. var, int). Abstract concepts of programming will not be familiar for beginners and therefore the code should be as reader friendly as possible. For reason readability has is emphasised more than writability features of the aforementioned programming languages are taken into account when designing the syntax for the targeted language.

2.6 Visual Feedback for Code

Having visual feedback when executing the code enhances the understanding of a program. A software and a hardware related concept is analysed for a suitable decision.

2.6.1 Swift Playground

Swift Playground is an example of a development environment created for the language Swift with visual representation of the code being written. The language is meant to be used for teaching beginners coding concepts.

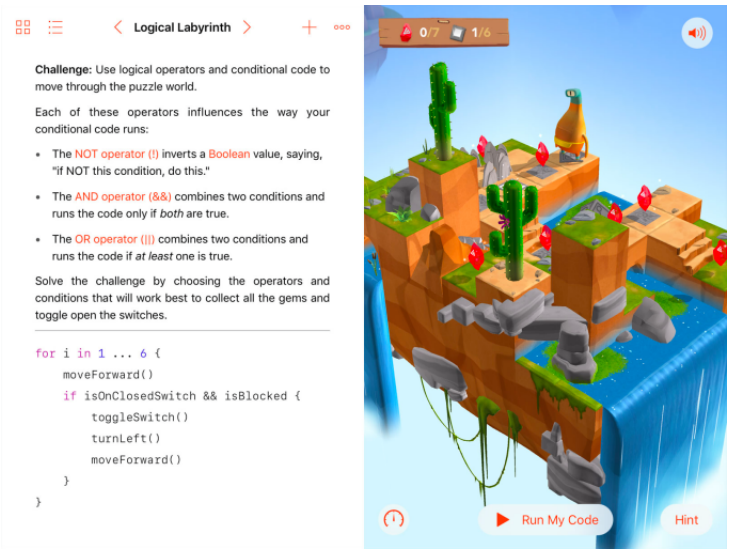

Figure 2.9: Code snippet and preview of Swift Playground [21]

Figure 2.9 is a screenshot of the Swift development environment. On the left of the screenshot are the instructions for a puzzle that the user has solved with coding. As the user is writing code, the graphics on the right executes the code, thereby giving the coder direct response to the written program. In that way the user becomes familiar with programming concepts through a concrete representation. This strongly contributes to the users understanding of code in a quick and fun way.

2.6.2 Arduino as visual feedback

A visual representation of code does not have to be virtual as with Swift Playground. The visual feedback can also come through hardware, in this case an Arduino. To support a physical representation of code a simple Arduino car can be built.

Section 2.4 mentioned the difficulties of programming in the Arduino language for beginners. A language that includes specific constructs that targets the Arduino car can be designed and implemented. In this way, a different visualalisation of the code could be achieved through the Arduino hardware.

Figure 2.10 is an example of how the Arduino car can be build. In the rest of the report the Arduino car will be referred to as the Trun car.

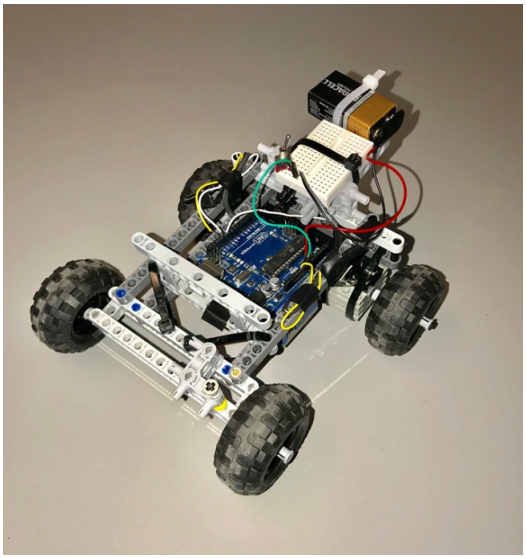

Figure 2.10: Preview of the car build to accommodate the Arduino

The Trun car could use two LEGO Technic Mini dc motors. The motors would be connected to digital pin 12 and 13 on the Arduino and it would be powered by a 9V battery.

2.7 Programming Paradigms

A programming paradigm is a style of programming. Several paradigms exists with different assets, making certain paradigms more suitable for particular programming situations. The paradigms can therefore also be used to categorise the different programming languages [22]. Though there are many programming paradigms, this section will mention the most commonly used paradigms.

Imperative paradigm

The first paradigm, imperative programming, follows a sequence of commands explicitly.

"*First do this and next do that*" [22]

This is reflected in the order in which the functions are declared as the compiler can not look ahead in the program for a function.

This programming paradigm typically offers basic commands such as assignment, IO and procedure calls, which could be useful when teaching programming to beginners. Imperative programming also relates to defining what the system must do, which could also be suitable for learning the basics of Human-Computer Interaction [22].

Examples of imperative programming languages: C, Pascal, Fortran.

Functional paradigm

The functional paradigm is found quite popular due to its way of handling mathematical functions.

"*Evaluate an expression and use the resulting value for something*" [22]

Unlike the imperative paradigm, the functional paradigm allows for functions to be executed in a non-sequential matter. This could potentially make it easier for a user to create functions, but having to follow a certain program flow, like in the imperative paradigm, could be easier for an inexperienced users to understand. Much like the imperative paradigm, the functional paradigm offers basic commands, which is relevant for teaching [23].

Examples of functional programming languages: Scala, Haskell, Earlang.

Logic paradigm

The logic paradigm is based largely on formal logic, and therefore requires a different mindset than the other paradigms described [24].

"*Answer a question via search for a solution*" [22]

Logical programming consists of a set of instructions in logical form, which are used as facts and rules. Therefore a programming language for this paradigm can be described as rule-based. The advantages of using logical programming is that the programs are easy to understand, assuming the user is moderately experienced with the language. The programs can also easily be modified or structured in any order [24]. The language implementation system chooses the order for the rules to provide the desired result [11, p. 45].

Examples of logic programming languages: Prolog, Datalog.

Object-oriented paradigm

The last commonly used paradigm, the object-oriented, focuses on objects and the interaction between them.

"*Send messages between objects to simulate the temporal evolution of a set of real world phenomena*" [22]

The objects/classes are often modelled after real-world objects and contain data fields with belonging information and methods to create interactions between them. This interaction could be complicated for a beginner to learn. The object-oriented paradigm is connected with concepts like encapsulation, inheritance and polymorphism, which could be relevant in a learning scenario for students [23].

Examples of object-oriented programming languages: Java, Python, C++, C#.

Evaluation

Different paradigms are useful for certain situations. The imperative programming paradigm is deemed most suitable of the four when teaching programming to beginners, due to it encouraging sequential code. Logical programming requires a different mindset, which would not be efficient when teaching programming to beginners. The object-oriented paradigm would also be a suitable candidate to chose as it contains more concepts than the imperative paradigm. They might however be too complicated for a beginner to learn. Therefore the imperative programming paradigm is deemed most suitable.

3 | Problem Definition

In this chapter, conclusions are drawn from the problem analysis in Chapter 2. This will be followed by the problem definition for the project. Afterwards, the project's requirements will be defined using the MoSCoW priorities to categorise them.

When evaluating the Arduino programming language, it does not seem suitable in a learning environment for the chosen target group due to the required knowledge regarding both the Arduino libraries and the Arduino hardware.

To gain a better understanding of programming languages, three existing languages' syntax and features were studied to uncover what could be relevant for developing a programming language for the target group. Visual tools for learning programming via software were analysed through the Swift development environment.

Different paradigms were also analysed, whereto the imperative paradigm was deemed most suitable when learning programming. The target group could also benefit from a physical visual representation of code.

The problem statement is as follows:

A new programming language should help the target group learn programming by providing abstraction for challenging elements of the Arduino programming language, and utilising visual representation in the form of the Trun car.

3.1 Requirement specification

This section will include a list of requirements for the Trun-language using the MoSCoW priorities. The requirements are classified in four different categories. Essentially the MoSCoW method is a more natural way of assigning priority to requirements, rather than rating them high, medium or low. Note that if the time to fulfil all requirements is insufficient, those that have a lower priority, namely Could have and Want to have, will be ignored.

Below are the four categories in which a requirement can be prioritised [25, p. 140]:

- The *Must have* (M)
- The *Should have* (S)
- The *Could have* (C)
- The *Want to have but Won't be this time around* (W)

In **Table 3.1**, the requirement specification is presented. There are three columns specifying details of each requirement. The reference number column denotes a unique reference number for each requirement. Whether the requirement is functional (f) or non functional (n) is also encoded in this column. The priority column denotes the prioritisation category. Each category has its own letter referring to the category list above.

Reference no.	Requirement	Priority
1.f	The programming language Trun must be able to compile to Arduino code.	M
2.f	The language must have iterative control structures.	M
3.f	The language must have selective control structures.	M
4.f	The language must implement basic data types namely integer, float, boolean, and string.	M
5.f	The language must implement arrays as a collection of the basic data types.	M
6.f	The language must implement basic arithmetic relational, and logical operations	M
7.f	The Trun-language must follow the principles of the imperative programming paradigm.	M
8.f	Descriptive error messages must be shown to the user.	S
9.n	The programming language should be easier for beginners to use compared to the Arduino language.	S
10.f	The language should implement functionality for driving the Trun car forward.	S
11.f	The language should implement functionality for turning the Trun car.	S
12.f	A subset of the language could be translated directly to AVR Assembly code.	C

Table 3.1: Table of requirements

4 Compiler Theory

In this chapter the elemental theory of a compiler is presented. An overview of the compiler phases will briefly be presented, followed by a more extensive explanation of the theory behind a compiler's individual parts. All theory described is inspired by Fischer et. al.'s "Crafting a Compiler".

4.1 Phases of a compiler

When creating a new programming language, the code should be executable on a target machine. A high level programming language can not be directly executed on a machine before it has been translated to something the target machine understands. This is the compilers task. In this section the compiler and its different components are described on a general level to outline what is necessary in order to craft a working compiler.

A compiler is a piece of software that translates code from a certain programming language (*the source program*) into another (*the target language*) as seen on **Figure 4.1**. Typically, compilers are distinguished between their generated type of machine code and the format of their target code.

Figure 4.1: The basic stages of a compiler [26, p. 2]

This is a simplified description of a compiler. The process of compiling can be divided into several components as seen on **Figure 4.2**. The figure illustrates a syntax-directed compiler, which is the most common type of modern compilers [26, p. 15]. These components are furthermore divided into three different phases, namely the *syntax analysis*, *contextual analysis*, and *code generation* each of which will be elaborated on below.

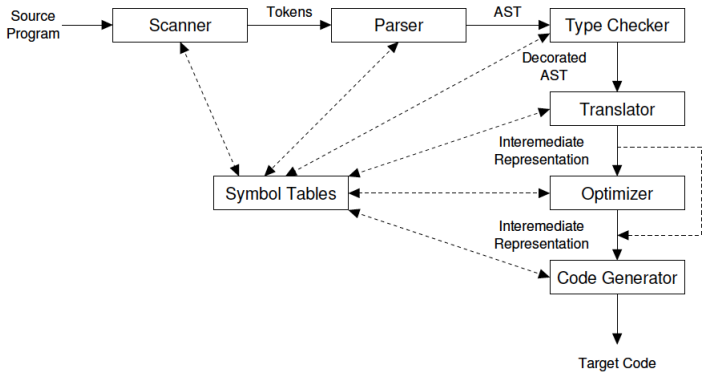

Figure 4.2: A syntax-directed compiler [26, p. 15]

4.1.1 Syntactical Analysis

The syntax analysis encapsulates the first components of the compiler: the *Lexer* and the *Parser*. The lexer looks through the source program code and creates a stream of *tokens* for the parser. Tokens can be identifiers, data-types, and reserved words. They each have two components: a token type and a token value [26, p. 16].

The parser reads the tokens and checks for syntax errors. In the case of a syntax error, the parser might be able to safely correct the error or provide a syntax error message. The parser can achieve this by comparing the token stream to the Context Free Grammar (CFG)[1]. In the end a parse tree representing the source program is created [26, p. 17].

Lastly in the syntactical analysis, depending on the compiler, an Abstract Syntax Tree (AST) is constructed. The AST is an abstract representation of the source program and is the basis for the semantic analysis [26, p. 17].

4.1.2 Contextual Analysis

The second phase of the compiler consists of the contextual analysis. The components in this phase are the *type checker*, *translator* and *optimizer*.

The type checker examines the semantics of the AST. When a node is visited and checked for type errors. If no type error is found the node is decorated, which means type information is added to the AST [26, p. 17].

The translator transforms the AST to an Immediate Representation (IR). The IR carries the

[1] the definition of a CFG is given in **Section 4.3**

meaning of the code with it, in other words the semantics [26, p. 17].

The optimizer is not implemented in every compiler. This component creates an operationally faster version of the IR while still maintaining equivalency. An example of optimization may be multiple lines of code that can be simplified to a simple instruction [26, p. 18].

4.1.3 Code Generation

Code generation is conducted by the *code generator*, and is the last step in the process. It generates the target code to be run on the target machine. Further optimization is done at this stage by considering specific information about the target machine. By the end of this process, executable code is generated for the target machine. In some cases the target code is in another language that needs to compile again to be able to be executed by the target machine [26, p. 19].

4.2 Lexer

The principles of the compiler's lexer component will be presented. The lexical analysis is the first component of a compiler and the syntactical analysis.

The purpose of a lexer, also commonly referred to as the *scanner*, is to put a program into a compact and uniform format. The lexer's job is to translate an input stream of characters to a stream of tokens [26, p. 58].

All lexers should perform the same task where the only variation is the tokens produced. Therefore many generator tools exist for producing a lexer. When generating a lexer with a tool, a description of tokens is sufficient. The regular expression notation is an effective and often used approach to specify the tokens [26, p. 69]. Strings defined by a regular expression is called a regular set, and a regular set is a token class. An instance of a token class is called a lexeme.

An example of the lexer's job would be, when encountering the keyword *is*, to create a token for assign. The lexer reads a stream of characters, in this case *is*, and determines that the characters corresponds to a terminal symbol. After determining the corresponding terminal symbol, the lexer should create a token and add it to the token stream for that said terminal symbol which is specified in the token specifications in **Chapter 5**.

4.3 Context-Free Grammar

The concept of Context-Free Grammars (CFG) was first described by Noam Chomsky in the mid-1950s. The original idea was to find a method for describing common languages, but this type of grammar turned out to be a useful method for describing programming language syntax [11, p. 137].

A short time after Chomsky finished his work on describing language syntax, John Backus began formalising the syntax of the programming language ALGOL 58. Shortly after the notation was altered by Peter Naur for ALGOL 60. The method was similar to CFGs and became known as Backus-Naur Form (BNF).

For the explanation of CFGs consider **Table 4.1**.

1	$S \rightarrow A\ B\ C\ \$$
2	$A \rightarrow a$
3	$B \rightarrow b$
4	$C \rightarrow c$

Table 4.1: A simple example of a CFG with four production rules.

The grammar in **Figure 4.1** has four *production rules*. Each production rule has a Left-hand side (LHS) and a Right-hand side (RHS). The LHS can produce the RHS of a rule. The grammar rules begin from the start symbol (S) and an input string is always appended with the symbol $. The first rule is the start rule that is always applied first in a derivation.

A grammar consists of *terminals* and *non-terminals*. Non-terminals are the set of symbols appearing on the LHS of a grammars production rules. Terminals are symbols that only appear on the RHS of all production rules [26, p. 114]. Note that non-terminals can also appear on the RHS of a production rule allowing creation of a series of *derivations*. It is common for non-terminals to begin with a capitalised letter. However, the grammar syntax used in this report utilises capitalised terminals and non-capitalised non-terminals. This will briefly be discussed towards the end of this section.

The example grammar on **Table 4.1** only accepts one string: *abc*. The grammar can be extended further to accept more strings as seen on **Table 4.2**.

1	$S \rightarrow A\ B\ C\ \$$
2	$A \rightarrow a$
3	$\mid \lambda$
4	$B \rightarrow b$
5	$\mid \lambda$
6	$C \rightarrow C\ c$
7	$\mid \lambda$

Table 4.2: CFG from **Table 4.1** extended

The grammar has been extended with three additional rules, namely rule 3, 5 and 7. The symbol '|' allows for different productions for the same non-terminal. This creates the possibility for the non-terminals A, B and C to produce the empty string denoted by the lambda symbol (λ). Additional changes have also been made to rule 6, which can now create a series of the terminal c until the non-terminal C produce the empty string.

Extended Backus-Naur Form (EBNF) is an extension to BNF's functionality, removing the need for long series of derivations when representing repeating or optional productions [11, p. 149]. EBNF provides notations for describing these occurrences, as shown below:

Notation	Usage
{ ... }	Repetition (0 to many)
[...]	Optional (0 or 1)

Table 4.3: Notation for repeating or optional productions EBNF

Any production enclosed by the curly brackets can be omitted or repeated, hence the notation's name as seen on **Table 4.3**, and anything enclosed by the hard brackets can be omitted or included once, therefore named Optional.

The source language CFG in **Chapter 5** will be presented in EBNF using ANTLR's syntax.

ANTLR's EBNF syntax represents the aforementioned notations with operators found in regular expressions, namely the asterisk **(*)** and the question mark **(?)** .

Notation	Usage
...*	Repetition (0 to many)
...?	Optional (0 or 1)
...+	Mandatory (1 to many)

Table 4.4: ANTLR's EBNF syntax notation

In addition, the plus sign notation **(+)** is also available, denoting that something is included at least once, which is why it is called Mandatory.

Converting the CFG shown in **Table 4.2** to EBNF with ANTLR's syntax would look like this:

Table 4.5: BNF Syntax

1	$S \rightarrow A\ B\ C\ \$$
2	$A \rightarrow a$
3	$\mid \lambda$
4	$B \rightarrow b$
5	$\mid \lambda$
6	$C \rightarrow C\ c$
7	$\mid \lambda$

Table 4.6: EBNF (ANTLR) Syntax

1	$s \rightarrow A?\ B?\ C*$

Table 4.7: CFG in BNF converted to EBNF (ANTLR)

Using ANTLR's powerful EBNF syntax, the CFG is reduced to a single line, while maintaining the same meaning.

4.4 Parser

The parsing phase follows the Lexer in the syntactical analysis. A parser is also typically built using a parser generator tool, since the parsing procedure is the same algorithmic task, where the only variation is the grammar rules. In the same manner as regular sets can be used to automatically generate a lexer, a grammar specification of a programming language's syntax can be used to generate a parser [26, pp. 113-114].

The parser checks whether the syntax conforms with the specified grammar rules (CFGs). The result of the parsing phase should be a structured representation of the tokens received from the lexer/scanning phase. The structured representation could be a parse tree but is typically represented with an abstract syntax tree (AST). If a program's source code does not follow the rules defined by the CFG, it is syntactically incorrect and should cause a syntax error to be generated in the parsing phase.

4.4.1 First and Follow sets

When creating a deterministic parser it is necessary to determine the first and follow set of each grammar rule [26, p. 130]. In this section the sets are briefly explained.

First Sets

The first set is denoted as First(α), where α is a string of grammar symbols. The first set is the set of terminals that can appear first on the RHS of a grammar rule derived from α [26, pp. 130-134]. Consider the following cases:

- If α begins with a terminal, only that terminal will be in the first set.
- If α begins with a non-terminal, the first set will be all possible terminal symbols derivable from the non-terminal that comes first.

The empty string λ is excluded from the first set.

Follow Sets

The follow set is denoted as Follow(A), where A is a non-terminal. The follow set is the set of terminals that can follow the non-terminal A. In other words the follow set is the set of terminals that is possible to come after A in a token stream. Again the empty string λ is excluded from the set [26, pp. 134-137].

4.4.2 Top-Down parsing

Top-Down parsers are also commonly referred to as LL(x) parsers (e.g. LL(1), LL(2)). The first L describes that the token sequence is processed from left to right. The second L denotes that a leftmost parse is produced [26, p. 145]. Replacing x with a number specifies the amount of lookahead the parser needs in order to be decisive [26, p. 146]. In a Top-Down parse the parse tree grows top-down and from left to right [26, p. 145].

A Top-Down parser begins with the start symbol and uses a predict set to apply grammar rules and to arrive at the input tokens. The predict set is the union of the first set and the follow set [26, p. 145]. There are three cases for the predict set:

- Predict set is empty. No rule can therefore be applied to derive the following token sequence. This means there is a syntax error in the input string.
- The predict set has more than one element. In this case multiple production rules can be applied. This parse would require non-determinism to apply the correct rule.
- The predict set contains one element. Only one rule can be applied to continue.

For a grammar to be LL(1) every terminal should allow for only one prediction of a non-terminal. In other words each non-terminal productions must have disjoint predict sets for one token lookahead. If the non-terminals predict sets are not disjoint either the grammar should be changed or more lookahead could be used since a deterministic parser is preferred [26, pp. 146-147].

4.4.3 Bottom-Up parsing

Bottom-Up parsers are also known as LR parsers. Here the second letter R denotes that the parse is a rightmost parse deriving non-terminals from the right to left. A bottom-up parser works its way from the leaves toward the root node [26, p. 180]. Adding a number in parenthesis LR(1) has a slightly different meaning than with top-down parsers. Here the number 1 denotes that the parser has one token of lookahead when a parse table is being constructed [26, pp. 187-189].

The generic form of a bottom-up parser is called a shift-reduce parser. The shift-reduce parser is a table driven parser, meaning the parser can do a lookup in a table to figure out which action to do next in the current state [26, p. 181].

There are two actions a shift-reduce parser can perform: shift or reduce. This form of bottom-up parser uses a stack and the input in form of a token stream. When a shift action is performed the topmost element of the token stream is pushed to the stack. When a reduce action is performed a grammar rule $A \rightarrow \gamma$ is applied and the non-terminal A takes the place of γ in the token stream. This means grammar rules are applied in a reversed order [26, p. 182].

When doing shift-reduce actions the parser is looking for a handle. A handle is the exact sequence of tokens that are to be replaced by a non-terminal next. The parser will do shift actions until a handle is found on the stack. Then a reduce action is performed. This continues until the start symbol is on the stack [26, p. 185].

The real challenge in creating a bottom-up parser lies in generating the parse table. The different techniques for creating a parse table will not be discussed.

4.4.4 Parse Tree Example

A parse tree for an input string is presented as an example of a top-down parse. The example is taken from [26, p. 117]. A parse tree consists of a root, nodes, and leaves. Non-terminals are the nodes, terminals are the leaves, and the root is always the start symbol.

Consider the following grammar:

```
1  E      → Prefix ( E )
2         | v Tail
3  Prefix → f
4         | λ
5  Tail   → + E
6         | λ
```

Figure 4.3: CFG for the language used in the parse tree example below [26, p. 116]

A parse tree for the input stream f(v+v) is shown on **Figure 4.4**. A top-down parse has been used to construct the tree.

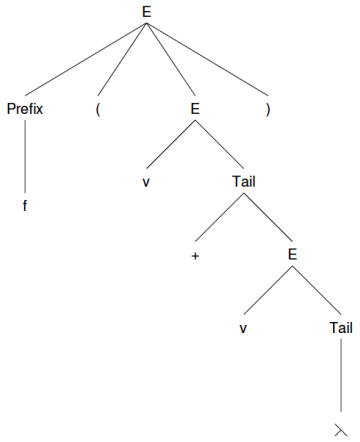

Figure 4.4: The parse tree for f(v+v) [26, p. 118]

The parse tree is first build from the root which is the start symbol E. Rule 1 is used to get Prefix (E). Using a top-down parse a leftmost derivation is used. Therefore rule 3 is applied next to get the first leaf. The entire derivation process is as follows:

$$
\begin{aligned}
E &\Rightarrow_{lm} Prefix(E) \\
&\Rightarrow_{lm} f(E) \\
&\Rightarrow_{lm} f(vTail) \\
&\Rightarrow_{lm} f(v+E) \\
&\Rightarrow_{lm} f(v+vTail) \\
&\Rightarrow_{lm} f(v+v)
\end{aligned}
$$

The difference when doing a top-down parse or a bottom-up parse is the order of rules applied. A rightmost derivation would therefore apply the rules in reverse order in the above example.

In order to be able to construct a deterministic parser, the grammar of the source language must be unambiguous. A grammar is ambiguous if a terminal string does not have a unique parse tree. There exists no algorithm that can check if a given CFG is ambiguous because this problem is undecidable [26, p. 121].

4.4.5 Abstract Syntax Tree

The concrete parse tree can be simplified to what is called an Abstract Syntax Tree (AST). The Abstract Syntax Tree (AST) is the central data structure for all activities happening after the parsing if this representation is chosen for the program. This includes the contextual analysis and code generation phase. Generally speaking, the goal of the syntax-directed translation can generally be simplified to the construction of an AST. The AST is the final product of the syntax analysis phase of the compiler [26, p. 250].

The compiler needs a structural representation of the program that has to be compiled. The parse tree explained previously would be sufficient for this. However, the parse tree also captures details that is unnecessary for the rest of the compiler. This is why the AST is needed. It omits the unwanted information such as parenthesis and single successor nodes and thereby makes the tree more compact and simpler to use.

As the AST has such a major role in the majority of the compilers activities it is important to take precautions when designing and constructing it. When designing an efficient AST data structure the following things should be taken into account:

- The construction of the AST typically happens in a bottom-up fashion starting from the leaves at the bottom and moving upwards to the root.
- A list of siblings are generated which is later adopted by a parent. These lists are often generated by recursive rules which means a simplification of the adoption of siblings should be made.
- Finally the AST should also support tree nodes with an arbitrary number of children. This is because some constructs may need more than a fixed number of children such as a function list of parameters or arguments as it can contain zero to many children [26, p. 252].

4.5 Symbol Table

The symbol table is responsible for storing all of the AST node's relevant information [26, p. 279]. When the AST has been constructed a traversal of the AST can be done to insert node information into the symbol table.

There are two common ways to implement symbol tables in a compiler: One big symbol table for the entire program scope or multiple tables where each table is responsible for each scope. If a single table is used the table should have a way to differentiate two variables with the same name when they belong to different scopes [26, p. 285].

Multiple symbol tables can be implemented using a LIFO structure. Most commonly this is done with a stack. The stack 'level' controls which scope variables belong in. Every time a nested scope begins the stack will be pushed with a reference to an empty symbol table. When the scope is left the stack is popped back to the previous scope. When searching for a variable in multiple symbol tables the current table is searched first. If the variable is not found the outer scope table is searched. This creates the need of searching multiple symbol tables. Most lookups of symbols in block-structured languages return symbols in the inner or outermost scopes [26, p. 285]. Therefore a single symbol table can be faster than using multiple symbol tables.

In the construction of the symbol table each variable declaration is checked to see if the same variable id has already been used to describe another value. If the declarations does not correspond to the language's scope rules the error should be caught in this part of the compiler. In this way the symbol table enforces the scope rules of a language.

4.6 Type checking

Type checking has two aspects to it: Static type checking and dynamic type checking. The static type checking is performed during compile time. It checks whether any requirements, that are deemed necessary for a program, are imposed to compile successfully. The dynamic type checking is performed during run time instead [11, pp. 311-312]. As type checking is performed after the parsing process, the AST can be traversed using a visitor pattern. When visiting each node in the AST their parents and potential children will be type checked in the context they are placed. This means based on the semantic rules of the language only certain types can be matched with each other.

Errors will be produced if the type checker finds type errors in a traversal. In the end, the type checking produces the decorated AST required later for the code generation.

4.7 Code Generation

Code generation is the part of the compiler process where the source code, often represented in an intermediate form such as an AST, is translated into executable code. There are three different types of machine code that compilers can generate: Pure machine code, Augmented machine code, and Virtual machine code [26, pp. 4-5].

Pure machine code is generated for a specific machine's instruction set. It is called pure code because it does not depend on any other software to run. This type of compiled code is typically used for system implementation languages such as operating systems or other integrated systems [26, p. 4].

Augmented machine code is generated code that make use of system routines and runtime language support. To be able to run the generated code the target machine should also have an operating system present. The functionality made available is typically for I/O operations [26, p. 5].

Virtual machine code is generated code that only consists of virtual instructions. Generating this type of code will enable the possibility for a program to be run on multiple machines with different architectures. Thus virtual machine code has a lot of pros considering portability [26, p. 5].

It is possible to use Intermediate Languages, such as other programming languages, and thereafter use an already existing compiler for those languages to translate the code to machine code. When implementing the code generation part of a compiler an AST traversal can be done to visit each node class and specify what code should be generated for each node class. The target code can be generated by an *emit* method which in practice writes target code to a file.

Part II

Design & Implementation

5 | Syntax Specification

The syntax of the language will be defined informally through the English language. The syntax definitions will cover the primitives, operations, selective control structures etc. Each definition will consist of an example written in Trun language followed by a more formal syntactical definition in the form of a Context-Free Grammar and finally a token specification.

A language can be divided into two parts: its syntax and its semantics. The syntax of a language is the visual part of the language. It defines how the language is allowed to be typed. While syntax describes the form of the language the semantics describes the meaning of the language.

5.1 Syntax Justification

pIn this section the ideas for the syntax will be presented and inspiration will be drawn from the evaluation of the existing programming languages in **Section 2.5.4**.

The Trun language is created with the purpose of teaching the target group of 7-9th graders (**Section 2.2**) programming with the help of visual representation. The goal is to create a language that emphasises the use of natural language with familiar terminology to what the target group is experienced with. This goes in hand with the conclusion from the criteria **Section 2.3.4** where having a programming language that is readable is easier to learn.

To understand the fundamentals of programming, primitive types are essential. Therefore, based on the criteria, it is decided to have everything regarding the primitives statically typed as it will increase the readability for the user and hopefully decrease the errors in matching types.

Furthermore, the language should keep the amount of overloaded operators at a minimum by creating distinctions. An example is the assign operator, which is going to be denoted by '*is*'. Compared to the languages that use the '=' symbol it is a more natural way of writing it. This allows the equal symbol to define equality between two operands, which is similar to what

the target group will be familiar with from mathematics.

In line with keeping to simple mathematics, having the index in arrays start at 1 and also not worrying about having to declare the size of arrays would be helpful to the users. When accessing the elements in the array, it should be readable and clearly indicate what is being selected.

Overall the language is inspired by the preexisting languages evaluated in **Section 2.5.4**, drawing from each the best they have to offer. The syntax will follow conventions known from the imperative paradigm, as it was deemed the most suitable in **Section 2.7**.

Preview of complete program structure and syntax

To introduce some of the concepts in the Trun language syntax, a code example of a small program is displayed on **Figure 5.1**. Following the Trun structure, the variables should be declared first, then the declaration of arrays followed by declaration of functions and statements.

```
integer decider is 1
decimal pi is 3.14
integer array runTime

function calculateRunTime returns integer (integer leftHand, decimal rightHand) {

    if (leftHand > rightHand) then {
        return leftHand / 2
    } else if (leftHand * 2 > rightHand) then {
        return leftHand * leftHand
    } else then {
        return 1
    }
}

#Fills up array, decider is counted up to 10
from(decider upto 10) {
    runTime element decider is calculateRunTime(decider, pi)
}

repeat {
    if(11 > decider AND decider > 7) then {
        drive(runTime element decider)

    } else if (8 > decider AND decider > 4) then{
        pause(runTime element decider)
        turnright((runTime element decider) + 1)

    } else then {
        turnleft(runTime element decider)
        drive((runTime element decider) + decider)
    }
    decider is decider - 1
} until (decider = 0)
```

Figure 5.1: Example of a complete program in Trun

5.2 Primitives

The primitives of the language will be presented in this section. In total, four primitive data types will be available in the programming language, those being *integer*, *decimal*, *text* & *truth*.

The primitive *text* will denote strings. The *text* must be encapsulated by quotation marks (" "). *text* is allowed to be empty. The primitive *truth* will contain the truth values, which can either be *true* or *false*. Furthermore, a data type for collection of primitives will be implemented in the programming language, and it will be denoted as *array*.

In order to declare a variable, its specific primitive data type has to be written in front of the variable identifier, which can be used to reference the declared variable. The allowed formatting of an identifier is shown with a regular expression in **Table 5.1**. To end the declaration, it has to be followed by the EOL (end of line) or the EOF (end of file) character. An example would be:

```
data-type identifier
```

It will not be allowed to make multiple declarations on a single line.

5.2.1 Collection of primitives

The *array* will be a collection of a single type of primitives. The declaration is shown below:

```
data-type array identifier
```

In the array declaration above, *data-type* specifies which type of data should be contained in the array. The keyword *array* is used to specify that the identifier is a collection, and again the identifier specifies the variable id (the name of the array).

5.2.2 CFG and token specification

As mentioned in **Section 4.3**, the CFG will be presented using ANTLR's EBNF syntax. The CFG and token specification for the primitive data types and collection are presented here. On **Figure 5.2**, there are non-terminals presented later in **Section 5.3.7**, and the non-terminal *type* is defined in **Section 5.6.1**.

```
dcl
        : INTDCL ID dclValue?
        | FLOATDCL ID dclValue?
        | TEXTDCL ID dclValue?
        | TRUTHDCL ID (ASSIGN truthexpr)?
        ;

arrdcl  :
          type ARRDCL  ID
        ;
```

Figure 5.2: CFG in EBNF for declarations of primitives and arrays

The CFG shown on **Figure 5.2** depicts the rules for writing the declarations of the primitives and collection. A *dcl* can produce five different productions, INTDCL, FLOATDCL, TEXTDCL, TRUTHDCL and ARRDCL. It needs an ID to be declared and declarations for variables can be initialised. Exactly what an assignment is, will be explained in **Section 5.3.7**. After a declaration has been made, it has to be followed by EOL or EOF.

Token specification for primitives can be seen on **Table 5.1** with their corresponding lexemes.

Token	**Lexeme**
INTDCL	integer
FLOATDCL	decimal
TRUTHDCL	truth
TEXTDCL	text
ARRDCL	array
ID	$[\_\|a-z\|A-Z]^{+}[\_\|0-9\|a-z\|A-Z]^{*}$ \Keyword $\cup$ truthval

Table 5.1: Token table for primitive data-types

All type tokens only have a single lexeme: integer, decimal, truth, text, and array. See **Appendix A** to see a full list of keywords.

5.3 Operators

Operations available in Trun can be put into three categories:

- Arithmetic operators
- Assignment operators
- Logical & Relational operators

5.3.1 Arithmetic Operators

An *arithmetic operation* will be created with the four basic arithmetic operators +, -, *, /. An arithmetic operation will contain an operator with an operand on both sides.

The - operator is also possible to use on a single operand. Using it on the left hand side of an operand will make it a unary minus operand.

Example of basic arithmetic operations are shown below:

```
a + b
a / b
 -a
```

The above examples shows valid syntactical use of the arithmetic operators *a* and *b* are used to denote operands.

Another operator similar to the arithmetic operators is the *append* operator. It is used to concatenate two strings. When the append operator is used on operands of type *text* the right operand will be concatenated on the left operand, as shown in the following example:

```
"text" append "text"
```

5.3.2 Assignment Operator

In Trun, the assignment operator is denoted with the keyword *is*. An assign operation starts with the identifier of the variable followed by the keyword *is* and then an operand.

An example of an assign operation in Trun is:

```
identifier is a
```

It should also be possible to combine a declaration operation with an assignment operation. An example is:

```
integer identifier is a
```

Arrays

When dealing with arrays a different set of operations can be executed. Each of them will be specified below.

```
data-type array identifier
```

Above is the valid way of declaring an array. The size of the array will not be specified in the declaration. In other words, this means that the user does not have to think about allocating memory for the size of the array. It is only allowed to create an array of one of the four primitive data-types. An array can not contain mixed data-types. It is furthermore not allowed to create multidimensional arrays.

Arrays are one indexed. This means that the first element in the array will be accessed as index 1 contrary to many programming languages using index 0.

```
identifier element a is b
```

In order to assign values to an array the assignment operator has to be used like in the example above. To specify the exact element in the array (a) to place the new value the keyword *element* is used followed by a numeral which should be an integer value. However, the integer must be greater than 0 to accommodate for the 1-index rule.

```
identifier is array-identifier element b
```

The *element* keyword is also used to access elements of a declared array and use it as an operand. The above example shows a declared variable being assigned element b of an array.

5.3.3 Logical Operators

The logical operators in Trun are displayed next to their Arduino C counterparts in **Table 5.2** below.

Trun	Arduino C equivalent
NOT	!
OR	||
AND	&&

Table 5.2: The logical operators in the TRUN language

The logical operators OR and AND is syntactical equivalent to the arithmetic operators with two operands on each side as it is shown below:

```
a OR b
a AND b
```

The logical NOT operator acts syntactically like the unary minus operator with a single operand on the right hand side:

```
NOT a
```

5.3.4 Relational Operators

The relational operators are syntactical equivalent to the aforementioned operators. Trun incorporates three relational operators: =, <, >. They are displayed next to their C counterparts in **Table 5.3** below.

Trun	Arduino C equivalent
=	==
>	>
<	<

Table 5.3: The relational operators in the TRUN

Valid use of the relational operators are shown below. *a* and *b* are used as operands:

```
a = b
a > b
a < b
```

5.3.5 Predefined operations

In this section operations related to interacting with the Trun car will be defined. These are included to provide a visual connection between Trun and the hardware. **Table 5.4** gives an overview of the special operators in the language.

Operations	**Action on car**
drive()	Drive forward
pause()	Pauses movement
turnright()	Turn right
turnleft()	Turn left

Table 5.4: The Special operators used for the Trun car

Each of these operators takes a single operand inside the parenthesis as shown below:

```
drive(a)
pause(a)
turnright(a)
turnleft(a)
```

5.3.6 Precedence Table

When evaluating the different operators, the order is defined by the precedence and associativity of each operator. The associativity of an operator defines whether the operator evaluates to the right or left first. The precedence describes which operators are evaluated first.

Level	Type	Operators	Associativity
9	Parentheses	( )	left to right
8	Logical Operator	NOT	right to left
7	Arithmetic Operator	- (unary)	right to left
6	Arithmetic Operator	*, /	left to right
5	Arithmetic Operator	+, -	left to right
4	Append Operator	append	left to right
3	Relational Operator	>, <, =	left to right
2	Logical Operator	AND, OR	left to right
1	Assignment Operator	is	right to left

Table 5.5: Precedence table for the operators in the TRUN

As seen in **Table 5.5**, the precedence is to prioritise the highest level, which is the parentheses in level 9. Opposite, level 1 is lastly evaluated. Even though the predefined operations are defined syntactically as operators they are not influenced by precedence. The precedence can also be seen in the CFG for operations.

5.3.7 CFG for Operations

The CFG, followed by the token specification is displayed on **Figure 5.3** & **Figure 5.4**, and **Table 5.6** respectively. The precedence has been assured in the syntax by having the operation with a higher precedence deeper in the CFG and thereby evaluated first by the parser. E.g. in **Figure 5.3**, a plus/minus expression (arithmexpr) produces a number of multiplication/division expressions (multexpr) seperated by either a plus or a minus. This means that multiplication/-division expressions is deeper in the parse tree compared to plus/minus expression and will therefore be evaluated before plus/minus expression.

```
assignment
    : ID ( ASSIGN (value | functioncall | TEXT)
    |  ASSIGN expr );

value
    : arithmexpr
    | arrindex
    | ID ;

expr
    : arithmexpr
    | truthexpr
    | append;

arithmexpr
     : multexpr ((PLUS | MINUS ) multexpr)* ;
multexpr
     : unaryminus ((TIMES | DIVIDES) unaryminus)* ;
unaryminus
    :  (MINUS)? parexpr;
parexpr
     : nums
     | functioncall
     | arrindex
     | LPAR arithmexpr RPAR ;

nums
    : INUM
    | FNUM
    | ID ;
```

Figure 5.3: 1st part CFG for operations, containing assignment, value, expr, arithmexpr, multexpr, unary, parexpr and nums

```
truthexpr
    : logicalexpr ;

logicalexpr
    :  NOT? relationalexpr ((OR| AND) NOT? relationalexpr)*;

relationalexpr
    :   (value | functioncall | TEXT) ((EQUALS  | GRTHAN | LESSTHAN)  (value | functioncall | TEXT))
    |   LPAR logicalexpr RPAR
    |   truth
    |   functioncall;

append
    : (TEXT | ID | arrindex) APPEND (TEXT | ID  arrindex) ;

arrindex
    :  ID ELEMENT arithmexpr;

arradd
    : ID ELEMENT arithmexpr ASSIGN expr;

drive
    : DRIVE LPAR (value | functioncall | TEXT) RPAR;

turnleft
    : TURNLEFT LPAR (value | functioncall | TEXT) RPAR;

turnright
    : TURNRIGHT LPAR (value | functioncall | TEXT) RPAR;

pause
    : PAUSE LPAR (value | functioncall | TEXT) RPAR;
```

Figure 5.4: 2nd part CFG for operations, containing productions for truth expressions, array operations and library functions

The token specification for operations can be seen on **Table 5.6** with their corresponding lexemes.

Token	Lexeme
ASSIGN	is
INUM	$[0-9] \mid [1-9][0-9]^+$
FNUM	$[0].[0-9]^+ \mid [1-9]^+.[0-9]^+$
TRUTHVAL	[true | false]
TEXT	ASCII characters
LCB	{
RCB	}
LPAR	(
RPAR	)
PLUS	+
MINUS	-
TIMES	*
DIVIDES	/
OR	OR
AND	AND
NOT	NOT
EQUALS	=
GRTHAN	>
LESSTHAN	<
APPEND	append
DRIVE	drive()
PAUSE	pause()
TURNLEFT	turnleft()
TURNRIGHT	turnright()

Table 5.6: Token table for operations

5.4 Selective Control Structures

In Trun, there is one selective control structure, namely the If statement. It is possible to append additional else if statements to the original if statement, as well as an else statement. However, the else if's and else clauses are optional. The scope for the if statement is dictated by the curly brackets also making it possible to nest the if statement structure.

```
if (truth-expr) then {
    # body
} else if (truth-expr) then {
    # else if body
} else then {
    # else body
}
```

Figure 5.5: Example of the if statement

5.4.1 CFG for the Selective Control Structure

The formal definition of the selective control structure will follow. The formal grammar allows for the if statement to be followed by an else if statement or an if statement alone. Both cases can also be followed by an else in which it enters if the truth-expression evaluates incorrect.

The CFG for the if statement syntax shown on **Figure 5.5**:

```
ifstmt
    : IF LPAR truthexpr RPAR THEN stmtblock EOL*
      (ELSE IF LPAR truthexpr RPAR THEN stmtblock EOL*)*
      (ELSE THEN stmtblock)? ;

stmtblock
    : LCB EOL* dclblock stmtstartblock RCB ;
```

Figure 5.6: CFG for the if statement

The token specification for selective control structures can be seen on **Table 5.7**.

Token	Lexeme
IF	if
ELSE	else
THEN	then

Table 5.7: Token table for selective control structures.

5.5 Iterative Control Structures

In this section, the iterative control structures, the *while do*-loop, *repeat-until*, and the *from*-loop, supported by the source language will be presented.

5.5.1 *While do*-loop

The syntax of the *while do*-loop is shown on **Figure 5.7**:

```
while (truth-expr) do {
    # body of code
}
```

Figure 5.7: Example of the *while do*-loop

Two keywords *while* and *do* are used in the syntax and the the condition is encapsulated in parenthesis. The body of the loop is encapsulated in curly brackets.

5.5.2 *Repeat until*-loop

The *repeat until*-loop makes use of two new keywords: *repeat* and *until*. Just like the *while do*-loop the condition is encapsulated with parenthesis and the body with curly brackets. An example is shown on **Figure 5.8**:

```
repeat {
    # body of code
} until (truth-expr)
```

Figure 5.8: Example of the repeat until-loop

5.5.3 *From*-loop (*upto* and *downto*)

The third iterative control structure is the *from*-loop. The *from*-loop comes in two variations using two different keywords: *upto* and *downto* in the loop condition. Both variants are shown in the example shown on **Figure 5.9**.

```
from (a upto x){
    # body of code
}

from (b downto y){
    # body of code
}
```

Figure 5.9: Examples of the *from*-loop

5.5.4 CFG for Iterative Control Structures

The formal definition for the iterative control structures will be described individually by presenting their CFG and thereafter be followed by a description.

```
whilestmt
    : WHILE LPAR truthexpr RPAR DO stmtblock ;
```

Figure 5.10: CFG for the *while do*-loop

On **Figure 5.10**, the CFG for a *while-do* statement is presented. This type of statement will begin with the keyword *while*, followed by a truth expression encapsulated in parentheses. While the truth expression returns the value true, the body of the loop will be executed. The loop's body is a sequence of statements which is denoted by *stmtblock*.

```
repeatuntilstmt
    : REPEAT stmtblock UNTIL LPAR truthexpr RPAR ;
```

Figure 5.11: CFG for the *repeat until*-loop

In **Figure 5.11**, the formal definition of a *repeat-until* statement is given. A *repeat-until* statement will start with the keyword *repeat*. The *repeat* keyword will be followed by the body of the loop which should be encapsulated in curly brackets ({ }). The loop body is a *stmtblock* as in the *while-do* statement. As the last part of the *repeat-until* statement, the keyword *until* followed by a truth expression encapsulated with parentheses should be written. The truth expression will determine whether the loop continues. If the truth expression returns true the loop will be terminated.

```
fromstmt
    : FROM LPAR (value | functioncall | TEXT)  (UPTO | DOWNTO) (value | functioncall | TEXT) RPAR stmtblock ;
```

Figure 5.12: CFG for the *from*-loop

In **Figure 5.12**, the formal definition of the from statement is given. The from statement will be initiated through the keyword *from*. After *from*, the next part is specifying how many iterations should be executed. This is done by setting a start value, which will be incremented/decremented by one for each loop, until it is equal to the limit value. The start value and limit value are separated by the keyword *upto* or *downto*. The last part is the loop's body encapsulated in curly brackets.

The token specification for iterative control structures can be seen on **Table 5.8**:

Token	Lexeme
WHILE	while
DO	do
REPEAT	repeat
UNTIL	until
FROM	from
UPTO	upto
DOWNTO	downto

Table 5.8: Token table for iterative control structures

5.6 Functions

In this section the two different function variations supported in the source language is presented. The first variant is a function which returns a value and the other a non-returning function (procedure). With regards to C, they can be described as a function and a procedure.

```
function identifier returns data-type (parameters) {
    # body of function

    return a
}
```

Figure 5.13: Example of a function that returns a data-type

Figure 5.13 shows the syntax for declaration of a function with a *returns data-type*. The declaration in both variants start with the *function* keyword. The return variation needs the *returns* keyword followed of the desired data type to return. The parameter(s) are encapsulated in parenthesis and needs to be comma separated. The body of the function is encapsulated in curly brackets.

Below is the syntax for declaration of a function with no return type. It is identical to the return function but without the *returns* keyword and data-type.

```
function identifier (parameters) {
    # body of function
}
```

Figure 5.14: Example of the syntax for a function without return value

A function call is made by writing the function name (identifier) followed by argument(s) enclosed in parentheses. If the function does not require any arguments empty parentheses

should follow the function name.

The following is an example of a function call:

```
identifier(arguments)
```

Figure 5.15: Example of the syntax for a function call

The function with that returns a data type is allowed to be assigned to a variable.

5.6.1 CFG for Functions

In the CFG for functions each parameter in the parameter list is a type followed by an identifier.

```
functiondcl
    : FUNCTION ID RETURNS type LPAR (param (COMMA param)*)? RPAR stmtblock
    | FUNCTION ID LPAR (param (COMMA param)*)? RPAR stmtblock ;

type
    : INTDCL
    | FLOATDCL
    | TRUTHDCL
    | TEXTDCL ;

param
    : INTDCL ID
    | FLOATDCL ID
    | TEXTDCL ID
    | TRUTHDCL ID ;

functioncall
    : ID LPAR (arg (COMMA arg)*)? RPAR;

arg
    : nums
    | expr
    | TEXT
    | TRUTHVAL ;
```

Figure 5.16: CFG for Functions

On **Figure 5.16** the CFG required for creating functions and making function calls are depicted.

The token specification for functions can be seen on **Table 5.9**:

Tokens	Lexeme
FUNCTION	function
RETURNS	returns
RETURN	return

Table 5.9: Token table for functions

5.7 Comments

To create comments in the source language, the desired comment text can either be encapsulated by /# and #/ or preceded by a # for multi lined or single lined comments, respectively. An example is shown in **Figure 5.17**:

```
# single lined comment here
/# multi lined
   comment here #/
```

Figure 5.17: Example of syntax for comments

The CFG for the comments is implemented as rules in the lexer, consisting of regular expressions:

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

The first regular expression matches all characters in a non-greedy way between a forward-slash followed by a pound-sign and a pound sign followed by a forward-slash. The second expression matches all characters between a pound sign and a newline character, also lazily. They are implemented in the lexer, meaning the comment is scanned and discarded as part of the token creation process, before parsing, as comments have no semantic value in Trun.

5.8 Scoping

The scoping rules of variables are defined by curly brackets within the program. Curly brackets create a new block with its own scope but the brackets are allowed to be written in connection with a control structure. This means that control structures are what defines the scope of the variables. The innermost nested curly brackets have the information from all the outer scopes available to it, however the outer scopes cannot read information from the innermost scopes. Whenever a variable is created within a scope, it stays in that scope and cannot be used in an outer layer of scoping.

Here is an example to further demonstrate it.

```
1    integer a is 5
2    if (a = 5) then {
3        integer b is a
4    }
5    a is a + b
```

Figure 5.18: Example of scoping that causes an error

On line 5 on **Figure 5.18**, a semantic error would be thrown as b is declared within the scope of the if statement's curly brackets.

5.9 Structure for a valid program in Trun

In the following paragraph, the overall structure of a program written in Trun will be described. The CFG for a legitimate program in Trun is presented in **Figure 5.19**.

```
start
    : dclblock arrdclblock functiondclblock stmtstartblock;

dclblock:
    (dcl (EOL dclblock)*) EOF?
    | EOF?;
arrdclblock:
    (arrdcl (EOL arrdclblock)*) EOF?
    | EOF?;
functiondclblock:
    (functiondcl (EOL functiondclblock)*) EOF?
    |EOF?;
stmtstartblock:
    (stmt (EOL stmtstartblock)*) EOF?
    | EOF?;

stmt
    : ifstmt
    | whilestmt
    | returnstmt
    | functioncall
    | repeatuntilstmt
    | fromstmt
    | assignment
    | arradd
    | arrindex
    | drive
    | turnleft
    | turnright
    | pause;
```

Figure 5.19: CFG for a program in TRUN

Figure 5.19 describes a program written in the Trun langauge. The structure of a program is essentially split up into four parts. At the very beginning of a program is declarations of primitive variables, followed by declaration of arrays, function declarations, these are followed by the statements of the language.

The syntax of the Trun language has been defined in this chapter and next follows the semantic theory and specification.

6 | Semantic Specification

In this chapter an introduction to structural operational semantics will be presented. Structural operational semantics will be utilised to describe the formal semantics of Trun. First the abstract syntax will be described followed by theory and definitions used for the semantic rules. Lastly, the structural operational semantics for Trun will be described.

6.1 Abstract Syntax

Trun has an operational semantic, as its core elements consists of variables, expressions, statements and declarations. These were all syntactically described in **Chapter 5**. An abstract syntax is utilised to only describes the linguistic constructions of the language. The abstract syntax for Trun consists of two parts; the specification of the syntactical categories and formation rules for each of the syntactical categories.

x	$\in$ **Var**	Variables
e	$\in$ **Exp**	Expressions
S	$\in$ **Stmt**	Statements
p	$\in$ **Prog**	Program Structure
b	$\in$ **Block**	Block Structure
T	$\in$ **Type**	Types
D_V	$\in$ **DclV**	Declaration of variables
D_F	$\in$ **DclF**	Declaration of functions
D_A	$\in$ **DclA**	Declaration of arrays
a	$\in$ **Aname**	Array names
f	$\in$ **Fname**	Function names

Table 6.1: Collection of syntactical categories

e	:=	x | e_1 + e_2 | e_1 append e_2 | e_1 - e_2 | e_1 * e_2 | e_1 / e_2 | (e) | $-e$ | $e_1 = e_2$ | $e_1 > e_2$ | $e_1 < e_2$ | NOT e | e_1 AND e_2 | e_1 OR e_2 | $f(\overrightarrow{e})$ | a element e
S	:=	x is e | a element e_1 is e_2 | S_1 S_2 | $\overrightarrow{if\ \{e\ :\ b\}}$ | $\overrightarrow{if\ \{e\ :\ b_1\}}$ else b_2 | skip | while e do b | repeat b until e | from (e_1 upto e_2) b | from (e_1 downto e_2) b | drive(e) | pause(e) | turnleft(e) | turnright(e) | $f(\overrightarrow{e})$ | active S end | return e
p	:=	begin D_V D_A D_F S end
b	:=	begin D_V S end
D_V	:=	T x is e D_V | T x D_V | ϵ
D_F	:=	function f returns T $(\overrightarrow{x})$ b D_F | function f $(\overrightarrow{x})$ b D_F | ϵ
D_A	:=	T array a D_A | ϵ

Table 6.2: Formation rules for each of the syntactical categories in the abstract syntax for Trun

In **Table 6.1** the different syntactical categories are shown, where the left side column displays the meta variables, which are being put to use in the formation rules for the syntactical categories. The formation rules of each of these categories are described in **Table 6.2**. The arrow displayed over some of the meta variables in the formation rules implicate a sequence of that given meta variable. This can be viewed in the formation rules for function call and function declaration, where the parameters of the function is described as a sequence of expressions or variables. The notation for describing a sequence is also applied in the formation rules for the selective control structure, *if-else*. In the first formation rule, $\overrightarrow{if\ \{e\ :\ b\}}$, the arrow is used to define a chain of *if* statements. The rule briefly defines the chain to be a sequence of conditions (e) with each a belonging body (b). The second formation rule describes the same chain, but in the scenario where the selective control structure ends with the '*else*' keyword following no condition. The syntactical category for Types are exploited as a general data type in the section regarding the formal semantics, whereto the use of Types are defined in the **Chapter 7**.

6.2 Transition systems

The definition of an operational semantic can be described with a transition system, which can be seen as an oriented graph. This transition system contains vertices and edges. The vertices are called *configurations* and represents the specific state of a program at a specific time much like a snapshot. The edges of the graph are called *transitions* and corresponds to the steps of the program. Configurations without any outgoing transitions are called *terminal configurations*. In short the transition system describes the valid evaluations that is a transition that goes through a number of vertices and edges to finally end in a terminal configuration.

Formally a transition is a pair of configurations (γ, γ') where the γ' is reachable from γ. Transition system can be described as a triplex consisting of $\{\Gamma, \rightarrow, T\}$. Γ denotes the set of all configurations in the transition system. The $\rightarrow$ denotes the set of pairs of transition relations

within the transition system. $T \subseteq \Gamma$ is the set of terminal configurations. These transition systems are therefore used to define the operational semantics. [27, p. 27] To define the transition relations (i.e. *transition rules*) certain techniques are used, these are defined in the next sections.

6.3 Big-step and Small-step semantics

When it comes to describing set of transition rules (i.e. *the* $\rightarrow$ *in the transition system*), two main approaches are big-step and small-step semantics. Big-step semantics describes the fully evaluation, which implicates that in the case of $\gamma \rightarrow \gamma'$, the transition will start in a configuration (γ), and with transition relations ($\rightarrow$) always end in a terminal configuration (γ'). [27, p. 38] This can also be viewed in below.

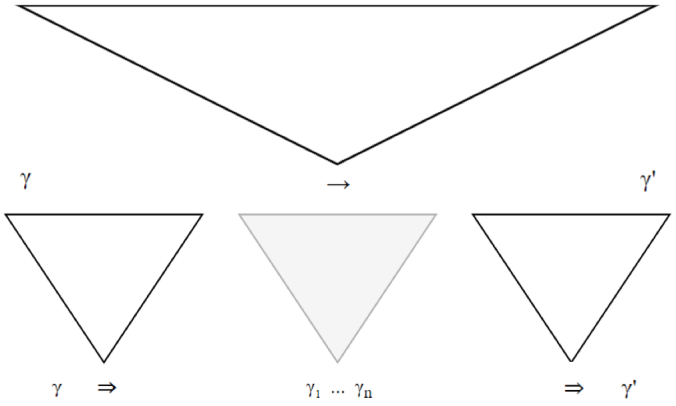

Figure 6.1: The difference illustrated between big-step and small-step semantics [27, p. 39]

In **Figure 6.1**, the big-step semantic is displayed at the top with the large triangle, which stretches over the entire process from the start configuration (γ) to the terminal configuration (γ'). Below the triangle, the small-step semantics are shown with several smaller triangles. This is to illustrate the different configurations that is reached before the terminal configuration. In a *small-step semantic* the same transition can end in the terminal configuration (γ') immediately, but can also contain multiple steps inbetween start (γ) and end (γ') configurations. The number of transitions that leads from γ to γ' is denoted as $\gamma \Rightarrow \gamma_1 ... \gamma_n \Rightarrow \gamma'$ [27, p. 38].

For the formal specification of the transition rules a small-step semantic has been chosen. The reason for using a small-step semantic is to describe the transitions in greater detail. In theory a big-step semantic could be used since the Trun language does not have parallelism. The small-step semantic could be useful for the implementation and debugging of the compiler though.

6.4 Environment-Store-Model

The Environment-Store-Model is used to describe the states of a configuration in a transition relation. The model was first suggested by Dana Scott and Christopher Strachey through their work with denotational semantic for programming languages. [27, p. 78]

In the Environment-Store-Model, two functions are used to model the state of an element. The partial function env_V finds the location of a variable, and the partial function *sto* finds the value of a location. So through these partial functions the value of a variable can be found. In this theory section, the set of environments for variables, denoted *envV*, are defined as below. [27, p. 79]

$$\mathbf{EnvV} = \mathbf{Var} \cup \{\text{next}\} \rightharpoonup \mathbf{Loc}$$

The set of variable environments, *EnvV*, are here defined to be all the partial functions from variables and *next* to locations. Where *next* is included to always point to the next available location. The set of all partial functions of *sto*, denoted *Sto*, are defined as below. [27, p. 79]

$$\mathbf{Sto} = \mathbf{Loc} \rightharpoonup \mathbb{Z}$$

Here the *Sto* is defined to contain all partial functions, *sto*, from location to integer. The overall model therefore describes how variables are bound to memory locations and how the values are stored in the memory locations. An example of the entire model can be seen in **Figure 6.2**.

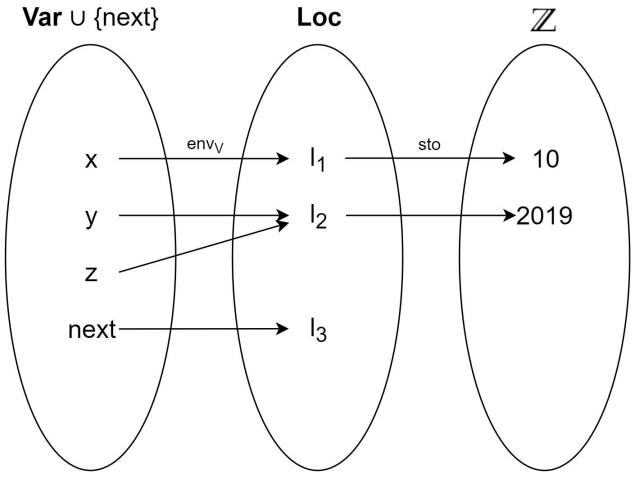

Figure 6.2: Example of Environment-Store-Model for variables [27, p. 80]

In the example above, env_V maps the variable x to the location l_1 while *sto* maps the location of l_1 to the value 10. It is also possible for two variables to map to the same location as it is shown with y and z that both map to the same location l_2 [27, p. 80].

6.4.1 Expanding the Environment-Store-Model

When using a small-step semantic the Environment-Store-Model is expanded.

Since the language supports arrays a new environment for arrays is introduced. The set of array environments is defined as:

$$\mathbf{EnvA} = \mathbf{Aname} \cup \{next\} \rightharpoonup \mathbf{Loc}$$

An element from the set of array environments maps from a variable to a location (l). The location mapped to the first element of the array environment represents an array. To access an element from the array *following* can be used. The function *following* is defined to give the next location from a given location.

$$following(l) \; = \; l \; + \; 1$$

Therefore, the third location of the array should be accessed, the location of the array should be found first. Then *following* is used three times on that location to get the third location of the array. This can be written as:

$$following^3(env_A(a))$$

Where a is the name of an array.

Trun supports functions and therefore, a function environment should be defined. The function environment definition depends on the scope rules of the language. Since the language uses fully static scope rules a function is bound to the block body, parameters, variable environment, function environment, and array environment:

$$\mathbf{EnvF} = \mathbf{Fname} \rightharpoonup \mathbf{Block} \times \mathbf{Var} \times \mathbf{EnvV} \times \mathbf{EnvA} \times \mathbf{EnvF}$$

For the Trun language an updated store is also necessary. Now the stores map from locations to all primitives allowed in the language. The stores are defined as:

$$\textbf{Sto} = \textbf{Loc} \rightharpoonup \textbf{Values}$$

where

$$\textbf{Values} = \mathbb{R} \cup \{true,\ false\} \cup \Sigma^*$$

In the above definition, Σ is the alphabet over all ASCII characters.

In a small-step semantic the configurations for statements are snapshots of the execution of a statement. To describe a rule, it should be possible to keep track of the states, which here is represented with bindings. At a given point it is necessary to know the bindings for precisely that point. To keep track of bindings a runtime stack is introduced. The runtime stack can be viewed as a list of three-tuple where the tuple consists of a variable environment a function environment, and an array environment (env_V, env_F, env_A). An element in the list is denoted as *envl* and the set of runtime stacks is defined as:

$$\textbf{Envl} = (\textbf{EnvV} \times \textbf{EnvF} \times \textbf{EnvA})^*$$

The top element of a runtime stack contains the current bindings for variables, arrays and functions.

With this model it is necessary to denote which part of the program has the current bindings. This is done with evaluating contexts. The rule:

$$\text{active } S \text{ end}$$

is an evaluating context added to the abstract syntax to denote the current active piece of code in the program where the current bindings from the runtime stack is valid.

6.5 Formal Semantics

The section regarding formal semantics will be structured after the syntactical categories, as shown in **Figure 6.1**. Only the syntactical categories with belonging formation rules will be described here, which include the Expressions, Statements and Declarations of Variables, Functions and Arrays.

The different syntactical categories will be based on the format ($\Gamma, \Rightarrow$, T) as described in **Section 6.2**. This will be expanded further on in the section regarding that particular syntactical category. Firstly, the different configurations for both Γ and T will be defined, then the $\Rightarrow$ will be described with small-step transition rules thereafter. The choice of using small-step semantics is for an expanded view of what each operation do.

Throughout the formal semantics the reference to Values (v_1, v_2, ..., v_n) will be referring the set **Values** from section 6.4.1. This allows the set **Values** to fit all the data types, which are described in the abstract syntax.

6.5.1 Expressions

The syntactical category *Expressions* describes numerals, variables, arithmetic expressions, relational expressions, logical expressions and the way to access arrays. The transition system are defined for this syntactical category as ($\Gamma, \Rightarrow$, T), where the configurations in Γ is defined:

$$\mathbf{\Gamma_{Exp}} = Exp \times Envl \times Sto \cup Values \times Envl \times Sto$$

Where the end configurations in T is defined as:

$$\mathbf{T_{Exp}} = Values \times Envl \times Sto$$

For translations of the numerals and variables to values, the two transition rules below are used.

		where $envl = (env_V,\ env_F,\ env_A) : envl$
[VAR]	$\langle x,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl,\ sto\rangle$	and $env_V x = l$
		and $sto\ l = v$

Table 6.3: Small-step transition rules for numerals and variables

In **Table 6.3**, the value of a variable is found with the Environment-Store-Model. Where the function env_V is used to find the location of a variable x, which is denoted as l in the rules above. Then the function *sto* is exploited to find the value stored at that location, which is the value v.

Arithmetic Expressions

In the language, there are several arithmetic expressions, which include the operators for addition, append, subtraction, multiplication and division. For this section only the small-step

transition rules for addition is explained due to the similarity between the rules. The rest of the rules are defined in **Appendix C**. Below the transition rules for the addition operator is defined.

[PLUS-1] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle e_1',\ envl',\ sto'\rangle}{\langle e_1\ +\ e_2,\ envl,\ sto\rangle \Rightarrow \langle e_1'\ +\ e_2,\ envl',\ sto'\rangle}$$

[PLUS-2] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle v_1,\ envl',\ sto'\rangle}{\langle e_1\ +\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ +\ e_2,\ envl',\ sto'\rangle}$$

[PLUS-3] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle e_2',\ envl',\ sto'\rangle}{\langle v_1\ +\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ +\ e_2',\ envl',\ sto'\rangle}$$

[PLUS-4] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle v_2,\ envl',\ sto'\rangle}{\langle v_1\ +\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ +\ v_2,\ envl',\ sto'\rangle}$$

[PLUS-5] $\langle v_1\ +\ v_2,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl,\ sto\rangle$ where $v = v_1 + v_2$

Table 6.4: Small-step transition rules for addition

In **Table 6.4**, the first rule evaluates the first operand partially, and then evaluates it fully in the second rule. The same applies to the second operand in rule three and four. The last rule applies only in the case, where both operands are evaluated fully to values. Lastly, the two values can be added, which is described in the side condition. The operator in the rule is syntax of the language, whereto the side condition holds the mathematical symbol for addition, this difference will be shown more clearly in later rules.

Due to determinism in the language, it is enforced to evaluated the left side operand fully first, before evaluating the right side operand. Therefore the right side operand can only be evaluated, when the left side operand is a value. This also corresponds to the associativity described in the precedence table in **Figure 5.5**. The same structure of transition rules apply to subtraction, multiplication and division, where each operand are evaluated partially and fully to values, and lastly the values can be calculated with that particular operator.

Relational and Logical Expressions

There are also the three relational operators for describing equality, greater than and less than. There are also three logical operators for describing negation, logical or and logical and. For this section only the operator for equality will be described due to the similarity between the transition rules for the different operators. All the transition rules for the relational and logical expressions can be viewed in **Appendix C**. Below the small-step transition rules for equality are defined.

[EQUALS-1]	$\dfrac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle e_1',\ envl',\ sto'\rangle}{\langle e_1\ =\ e_2,\ envl,\ sto\rangle \Rightarrow \langle e_1'\ =\ e_2,\ envl',\ sto'\rangle}$	
[EQUALS-2]	$\dfrac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle v_1,\ envl',\ sto'\rangle}{\langle e_1\ =\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ =\ e_2,\ envl',\ sto'\rangle}$	
[EQUALS-3]	$\dfrac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle e_2',\ envl',\ sto'\rangle}{\langle v_1\ =\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ =\ e_2',\ envl',\ sto'\rangle}$	
[EQUALS-4]	$\dfrac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle v_2,\ envl',\ sto'\rangle}{\langle v_1\ =\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ =\ v_2,\ envl',\ sto'\rangle}$	
[EQUALS-TRUE]	$\langle v_1\ =\ v_2,\ envl,\ sto\rangle \Rightarrow \langle true,\ envl,\ sto\rangle$	where $v_1 = v_2$
[EQUALS-FALSE]	$\langle v_1\ =\ v_2,\ envl,\ sto\rangle \Rightarrow \langle false,\ envl,\ sto\rangle$	where $v_1 \neq v_2$

Table 6.5: Small-step transition rules for the equality operator

Similarly to the arithmetic expressions, the first and second rule in **Table 6.5** evaluates the first operand, and the third and fourth rule evaluates the second operator. Once again, the second operator can only be evaluated, when then first is already evaluated due to determinism and the associativity of the operators. The last two rules describe the two different outcomes of a relational or logical expression, which are either *true* or *false*.

In the transition rules the operators of the syntax in the language is used, whereto the mathematical symbol is used in the side condition to describe the operation. The five other relational or logical operators, greater than, less than, negation, logical or and logical and, are described similarly with evaluating the operands and thereafter the two outcomes.

Accessing Array

In the language, there are also the Arrays, which need a couple of ways to be interacted with. Under this syntactical category, there is the expression to access an element in the array. Further interaction with the Arrays will be covered in the syntactical categories for Statements and Declarations of Arrays. Below are the transition rules for accessing elements in arrays.

[ARR-IND-1]
$$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle a\ element\ e,\ envl,\ sto\rangle \Rightarrow \langle a\ element\ e',\ envl',\ sto'\rangle}$$

[ARR-IND-2]
$$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle a\ element\ e,\ envl,\ sto\rangle \Rightarrow \langle a\ element\ v,\ envl',\ sto'\rangle}$$

where $envl = (env'_V,\ env_F,\ env'_A)\ :\ envl'$
and $env_V\ e\ =\ l$
and $v\ =\ sto\ l$

[ARR-IND-3] $\langle a\ element\ v_1,\ envl,\ sto\rangle \Rightarrow \langle v_2,\ envl,\ sto\rangle$

where $envl = (env_V,\ env_F,\ env_A)\ :\ envl$
and $env_A\ a\ =\ l$
and $l'\ =\ following^{v_1}\ l$
and $v_2\ =\ sto\ l'$
and $1\ \leq\ v_1\ \leq\ k$

Table 6.6: Small-step transition rules for accessing an Array

The first rule in **Table 6.6** evaluates the expression (e) on the index partially, whereto the second rule evaluates it fully. The third rule examines the result of the expression, which is the value (v_2) which lies on the chosen index (v_1) in the Array (a). As written in the side condition, the location (l) of the Array is found using the env_A function from the list of environments ($envl$), which describes the environment for arrays. To locate the different elements in an array, the *new* function to the power of index value (v_1) is used. This function finds the next location and to the power of index will locate the location for that particular index (l'). The *sto* function is used again to find the value (v_2) of that particular location (l'). Lastly, the index value (v_1) has to be larger than or equal to 1, because the arrays are 1-indexed. The size of the arrays are static, though this will not be viewed to the user. Indirectly the arrays will therefore not be able to be larger than 128 (k).

6.5.2 Statements

The syntactical category *Statements* describes assignment, compound statements, control structures, predefined statements, function calls and code blocks. The transition system are defined for this syntactical category as ($\Gamma, \Rightarrow$, T), where the configurations Γ is defined:

$$\mathbf{\Gamma_{Stmt}} = Stmt \times Envl \times Sto\ \cup Envl \times Sto$$

Where T, the end configurations, is defined as:

$$\mathbf{T_{Stmt}} = Envl \times Sto$$

Assignment

There are two ways of using assignment in this language, which are the assignment of variables and the assignment to an element of an array. Below the transition rules for the assignment of variables is defined. The transition rules for assigning to arrays can be viewed in **Appendix C**.

[ASS-VAR-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle x\ is\ e,\ envl,\ sto\rangle \Rightarrow \langle x\ is\ e',\ envl',\ sto'\rangle}$$

[ASS-VAR-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle x\ is\ e,\ envl,\ sto\rangle \Rightarrow \langle x\ is\ v,\ envl',\ sto'\rangle}$$

[ASS-VAR-3] $\langle x\ is\ v,\ envl,\ sto\rangle \Rightarrow \langle envl,\ sto[l \mapsto v]\rangle$

where $envl = (env_V,\ env_F,\ env_A)\ :\ envl$
and $env_V\ x\ = l$

Table 6.7: Small-step transition rules for assignment

Firstly in **Table 6.7**, the expression (e) being assigned to the variable (x) is evaluated partially and fully. When the expression is evaluated to a value (v), the third rule can be applied. Here the env_V is found under the list of environments ($envl$) in the side condition, thereafter the env_V function is used to find the location (l) of the variable (x). In the transition rule, it can be seen that sto function is used to assign the value (v) to the location (l) of variable (x).

For the arrays, the assignment is quite similar (*a element* e_1 *is* e_2). This is a composition of transition rules for accessing Arrays and for assignment. Both expressions (e_1 & e_2) are evaluated partially and fully to the values (v_1 & v_2). Where the value (v_2) is bound to the location of the index (v_1) within the Array (a).

Compound Statements

To create multiple following *Statements* there are small-step transition rules for the compound statements.

[COMP-1] $$\frac{\langle S_1,\ envl,\ sto\rangle \Rightarrow \langle S_1',\ envl',\ sto'\rangle}{\langle S_1\ S_2,\ envl,\ sto\rangle \Rightarrow \langle S_1'\ S_2,\ envl',\ sto'\rangle}$$

[COMP-2] $$\frac{\langle S_1,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle S_1\ S_2,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}$$

[COMP-3] $$\frac{\langle S_1,\ envl,\ sto\rangle \Rightarrow \langle \epsilon,\ envl',\ sto'\rangle}{\langle S_1\ S_2,\ envl,\ sto\rangle \Rightarrow \langle S_2,\ envl',\ sto'\rangle}$$

Table 6.8: Small-step transition rules for compound statements

The first transition rule evaluates the first statement (S_1) partially, which can be seen on **Table 6.8**. The second and third transition rules describe the full evaluation of the first statement (S_1), whereto the second rule describes when the first statement (S_1) return a value (v), where the second statement (S_2) will never be run. The third rule describes the scenario, where the first statement (S_1) does not return a value, so the second statement (S_2) will be run.

Control Structures

In the language, there are one selective control structure and four iterative control structures. First, the transition rules for the selective control structure, *if-then-else-then*, will be defined and then the different iterative structures will be defined from this.

[IF-E-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle if\ e\ \ldots,\ envl,\ sto\rangle \Rightarrow \langle if\ e'\ \ldots,\ envl',\ sto'\rangle}$$

[IF-E-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle if\ e\ \ldots,\ envl,\ sto\rangle \Rightarrow \langle if\ v\ \ldots,\ envl',\ sto'\rangle}$$

[IF-2-TRUE] $\langle if\ v\ then\ b_1\ else\ then\ b_2,\ envl,\ sto\rangle \Rightarrow \langle b_1,\ envl,\ sto\rangle$

where $v\ =\ true$

[IF-2-FALSE] $\langle if\ v\ then\ b_1\ else\ then\ b_2,\ envl,\ sto\rangle \Rightarrow \langle b_2,\ envl,\ sto\rangle$

where $v\ =\ false$

Table 6.9: Small-step transition rules for the selective control structure, *if-then-else-then*

Displayed above, in **Table 6.9**, is the rules for evaluating the expression e partially and fully and the second scenario of the selective control structure with the two different outcomes.

When the expression (e) is evaluated to *true*, then the first block (b_1) will be run. Whereto when the expression (e) evaluates to *false*, then the second block will be run (b_2).

The three dots (. . .) in **Table 6.9** represent all the different scenarios for the selective control structure. This is due to the different ways an *if-then-else-then* can be written. Shortly described, there are two versions of the selective control structure. The first version of the *if/if-else* statement, which does not have end in the else statement, whereto the other version does. For both of these two scenarios, it is possible to have a single statement, where there will only be evaluated one condition. It is also possible to have a chain of *if/if-else* statements, where several conditions will have to be evaluated. This creates four different ways the *if/if-else* statement can evaluate both *true* and *false*. All the different transition rules can be seen in **Appendix C**.

This scenario will also be used to evaluate the different iterative control structures. The control structures are the *repeat-until*, *while-do*, *from-upto* and *from-downto*. The transition rules for these are defined below.

[SKIP]	$\langle skip,\ envl,\ sto\rangle \Rightarrow \langle envl,\ sto\rangle$
[REPEAT]	$\langle repeat\ b\ until\ e,\ envl,\ sto\rangle$ $\Rightarrow \langle b\ if\ NOT\ e\ then\ (repeat\ b\ until\ e)\ else\ then\ skip,\ envl,\ sto\rangle$
[WHILE]	$\langle while\ e\ do\ b,\ envl,\ sto\rangle \Rightarrow \langle if\ e\ then\ (b\ while\ e\ do\ b)\ else\ skip,\ envl,\ sto\rangle$
[UPTO]	$\langle from\ e_1\ upto\ e_2\ b,\ envl,\ sto\rangle$ $\Rightarrow \langle if\ e_1 < e_n\ then\ (b;\ e_2 = e_1 + 1;\ from\ e_2\ upto\ e_n\ b)\ else\ skip,\ envl,\ sto\rangle$
[DOWNTO]	$\langle from\ e_1\ downto\ e_2\ b,\ envl,\ sto\rangle$ $\Rightarrow \langle if\ e_1 > e_n\ then\ (b;\ e_2 = e_1 - 1;\ from\ e_2\ downto\ e_n\ b)\ else\ skip,\ envl,\ sto\rangle$

Table 6.10: Small-step transition rules for the iterative control structures

The different iterative control structures in **Table 6.10** are transformed to an *if-then-else-then* statement, which can be evaluated with the transition rules for the selective control structure. The first rule defined is for the keyword *skip*, which is just evaluated to the environment. This is used in the transition rules above to end the loop.

The second rule, which is for the *repeat-until* control structure, will call the block (b). Thereafter, the expression (e) will be evaluated, if it returns *false* the first block (*repeat b until e*) of the selective control structure will be run due to the negation. If the expression (e) evaluates *true* then *skip* and the loop terminates.

The third rule, reflects the *while-do* loop. If the expression (e) evaluates *true* the first block (b) will run, else it will use skip and terminate.

The rule for the *from-upto* control structure, checks with the less than operator that the first expression (e_1) is smaller than the second expression (e_2), and if so runs the first block (b; $e_2 = e_1 + 1$; $from\ e_2\ upto\ e_n\ b$) else it will skip and terminate the control structure. The last rule, for *from-downto* control structure, checks with the greater than operator that the first expression (e_1) is larger than the second expression (e_2), and if so runs the first block (b; $e_2 = e_1 - 1$; $from\ e_2\ downto\ e_n\ b$) else it will skip and terminate the control structure.

Predefined Statements

There are four predefined statements in the language to make the Trun car move, which is *drive*(), *pause*(), *turnleft*() and *turnright*(). In this section only the *drive* statement will be described due to the similarities to the other predefined statements, which can be seen in **Appendix C**. Below the small-step transition rules for the *drive* statement is defined.

[DRIVE-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle drive(e),\ envl,\ sto\rangle \Rightarrow \langle drive(e'),\ envl',\ sto'\rangle}$$

[DRIVE-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl,\ sto\rangle}{\langle drive(e),\ envl,\ sto\rangle \Rightarrow \langle drive(v),\ envl,\ sto\rangle}$$

[DRIVE-3] $$\langle drive(v),\ envl,\ sto\rangle \Rightarrow \langle envl,\ sto[l_{out} \mapsto (v,\ forward)\ :\ vl]\rangle$$

Table 6.11: Small-step transition rules for the predefined statements

In the first rule in **Table 6.11**, the expression (e) is partially evaluated. The second rule describes when the expression (e) is fully evaluated. Whereto the third rule describes how the value is stored. The value (v) is stored with a direction (*forward, stop, left, right*) according to which predefined statement is chosen. This is stored as an output (l_{out}) and added to the value list (vl). The value list (vl) is used as a driving manual for the Trun car.

Function Calls

Function call is described in this section, but is implemented in the syntactical category for expressions, which are the expressions and the statements. The first three call rules evaluates a function call with parameters and the last call rule evaluates a call without parameters.

[RETURN-3] $\langle \textit{active return } v \textit{ end}, \textit{ envl}, \textit{ sto} \rangle \Rightarrow \langle v, \textit{ envl}', \textit{ sto}' \rangle$

[CALL-1]
$$\frac{\langle e, \textit{ envl}, \textit{ sto} \rangle \Rightarrow \langle e', \textit{ envl}', \textit{ sto}' \rangle}{\langle f(\overrightarrow{e}), \textit{ envl}, \textit{ sto} \rangle \Rightarrow \langle f(\overrightarrow{e'}), \textit{ envl}', \textit{ sto}' \rangle}$$

where $\overrightarrow{e} = \{e_1, e_2, \ldots e_n\}$
and $\overrightarrow{e'} = \{e'_1, e'_2, \ldots e'_n\}$

[CALL-2]
$$\frac{\langle e, \textit{ envl}, \textit{ sto} \rangle \Rightarrow \langle v, \textit{ envl}', \textit{ sto}' \rangle}{\langle f(\overrightarrow{e}), \textit{ envl}, \textit{ sto} \rangle \Rightarrow \langle f(\overrightarrow{v}), \textit{ envl}', \textit{ sto}' \rangle}$$

where $\overrightarrow{e} = \{e_1, e_2, \ldots e_n\}$
and $\overrightarrow{v} = \{v_1, v_2, \ldots v_n\}$

[CALL-3]
$$\frac{\begin{array}{c}\langle f(\overrightarrow{v}), \textit{ envl}, \textit{ sto} \rangle \\ \Rightarrow \langle b, (\textit{env}'_V[\overrightarrow{x} \mapsto \overrightarrow{l}][\textit{next} \mapsto \textit{following } l], \textit{env}'_F, \textit{env}'_A) : \textit{envl}', \textit{sto}'[\overrightarrow{l} \mapsto \overrightarrow{v}] \rangle\end{array}}{\langle f(\overrightarrow{v}), \textit{envl}, \textit{sto} \rangle \Rightarrow \langle b, \textit{envl}', \textit{sto}' \rangle}$$

where $\textit{env}_F\ f = \langle b, \overrightarrow{x}, \textit{env}'_V, \textit{env}'_F, \textit{env}'_A \rangle$
and $\textit{envl} = (\textit{env}'_V, \textit{env}'_F, \textit{env}'_A) : \textit{envl}'$
and $\overrightarrow{x} = \{x_1, x_2, \ldots x_n\}$
and $\overrightarrow{v} = \{v_1, v_2, \ldots v_n\}$
and $l = \textit{env}_V\ \textit{next}$

[CALL-4]
$$\frac{\langle f(), \textit{ envl}, \textit{ sto} \rangle \Rightarrow \langle b, (\textit{env}'_V, \textit{env}'_F, \textit{env}'_A) : \textit{envl}', \textit{sto} \rangle}{\langle f(), \textit{envl}, \textit{sto} \rangle \Rightarrow \langle b, \textit{envl}', \textit{sto} \rangle}$$

where $\textit{env}_F\ f = \langle b, x, \textit{env}'_V, \textit{env}'_F, \textit{env}'_A \rangle$
and $\textit{envl} = (\textit{env}'_V, \textit{env}'_F, \textit{env}'_A) : \textit{envl}'$

Table 6.12: Small-step transition rules for function call

In **Table 6.12** above, the rules regarding function calls and return statements is described (The other transition rules for the *return* statement can be found in the **Appendix C**). The first rule, the **RETURN-3** rule, contains an evaluating context to ensure the right current bindings. The return rule will evaluate the evaluation context to a returning the value (v). The rest of the rules in the table concerns function calls.

The second rule, **CALL-1** rule, is used for evaluating the parameters partially. The third rule, **CALL-2**, is used to evaluate a function's parameters fully to values. The fourth rule, **CALL-3**, is used when evaluating a function call with parameters. It evaluates to the evaluating

to a block (b), which is the body of the function. To show the call-by-value parameters, the sequence of formal parameters ($\overrightarrow{x}$) maps to a sequence of locations ($\overrightarrow{l}$) in the environment. Whereto the sequence of actual parameters ($\overrightarrow{v}$) will be mapped to the location's of the formal parameters through store. The fifth rule, **CALL-4**, describes the transition for a function call without parameters. The difference between this rule and **CALL-3** is that no parameters is known in the variable environment.

6.5.3 Program & Block Structure

The programs in this language is determined to follow the structure *begin* D_V D_A D_F S *end*. The small-step transition rules for this structure is defined below.

[PROG-1]
$$\frac{\langle D_V,\ envl,\ sto\rangle \Rightarrow \langle D'_V,\ envl',\ sto'\rangle}{\langle begin\ D_V\ D_A\ D_F\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle begin\ D'_V\ D_A\ D_F\ S\ end,\ envl',\ sto'\rangle}$$

[PROG-2]
$$\frac{\begin{array}{c}\langle D_V,\ env_V,\ sto\rangle \Rightarrow \langle env'_V,\ sto'\rangle \\ env'_V,\ sto' \vdash \langle D_A,\ env_A,\ sto'\rangle \Rightarrow \langle env'_A,\ sto''\rangle \\ env'_V,\ env'_A \vdash \langle D_F,\ env_F\rangle \Rightarrow \langle env'_F\rangle \\ \langle S,\ envl',\ sto''\rangle \Rightarrow \langle active\ S\ end,\ envl',\ sto''\rangle\end{array}}{\langle begin\ D_V\ D_A\ D_F\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle active\ S\ end,\ envl',\ sto''\rangle}$$

where $envl = (env'_V,\ env'_F,\ env'_A) : envl'$

Table 6.13: Small-step transition rules for the program structure

Due to the assignment in expressions the declarations of variables (D_V) can be evaluated in several steps as displayed in **Table 6.13**. Therefore the first rule is to partially evaluate the declaration of variables (D_V). In the second rule the entire structure is evaluated to the evaluation context (*active S end*). As seen in the premise of the second rule, the declaration of the variables (D_V), arrays (D_A) and functions (D_F) are evaluated respectively and in context to the already evaluated declarations. Lastly the evaluation context is described in the changed state of the environment list, which is defined in the side condition to correspond to the changes from the different declarations. Within certain statements, there are a block (b) called, which is described below.

[BLOCK-1] $$\frac{\langle D_V,\ envl,\ sto\rangle \Rightarrow \langle D'_V,\ envl',\ sto'\rangle}{\langle begin\ D_V\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle begin\ D'_V\ S\ end,\ envl',\ sto'\rangle}$$

[BLOCK-2] $$\frac{\begin{array}{c}\langle D_V,\ envl,\ sto\rangle \Rightarrow \langle envl',\ sto'\rangle \\ \langle S,\ envl',\ sto'\rangle \Rightarrow \langle active\ S\ end,\ envl',\ sto'\rangle\end{array}}{\langle begin\ D_V\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle active\ S\ end,\ envl',\ sto'\rangle}$$

Table 6.14: Small-step transition rules for the block structure

Like the program structure, there are also the block structure, which is used with control structures and functions to create the possibility of having variable declarations within these. The two rules for this block is displayed as above in **Table 6.14**. The first rule, describes the partial evaluation of the declaration of variables. The second rule describes when the declarations of variables are fully evaluated, so the statements can be evaluated in the evaluation context. For both the program structure and the block structure, they evaluate to an evaluation context. The transition rules for these are described below.

[ACTIVE-1] $$\frac{\langle S,\ envl,\ sto\rangle \Rightarrow \langle S',\ envl',\ sto'\rangle}{\langle active\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle active\ S'\ end,\ envl',\ sto'\rangle}$$

[ACTIVE-2] $$\frac{\langle S,\ envl,\ sto\rangle \Rightarrow \langle envl',\ sto'\rangle}{\langle active\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle envl',\ sto'\rangle}$$

[ACTIVE-3] $$\frac{\langle S,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle active\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}$$

Table 6.15: Small-step transition rules evaluating context

The transition rules for evaluating context in **Table 6.15** are used to evaluate the statements in the program. The first rule partially evaluates the statement (S). Whereto the second evaluates the statement (S) fully without returning a value and the third rule when returning a value.

6.5.4 Declaration of Variables

The transition system are defined for this syntactical category as ($\Gamma, \Rightarrow$, T), where Γ is defined:

$$\mathbf{\Gamma_{DclV}} = DclV \times Var \times Envl \times Sto\ \cup Envl \times Sto$$

T is defined as:

$$\mathbf{T_{DclV}} = Envl \times Sto$$

The first rule describes the declaration of a variable, whereto the second rule describes the initialisation of a variable. Small-step transition rules for declaration of variables:

[VAR-DCL] $$\frac{\langle D_V, (env_V[x \mapsto l][next \mapsto following\ l], env_F, env_A) : envl,\ sto\rangle \Rightarrow \langle envl', sto\rangle}{\langle T\ x\ D_V, envl, sto\rangle \Rightarrow \langle envl', sto\rangle}$$

where $l = env_V\ next$

[VAR-INI-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle T\ x\ is\ e,\ envl,\ sto\rangle \Rightarrow \langle T\ x\ is\ e',\ envl',\ sto'\rangle}$$

[VAR-INI-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle T\ x\ is\ e,\ envl,\ sto\rangle \Rightarrow \langle T\ x\ is\ v,\ envl',\ sto'\rangle}$$

[VAR-INI-3] $$\frac{\langle D_V, (env_V[x \mapsto l][next \mapsto following\ l], env_F, env_A) : envl,\ sto[l \mapsto v]\rangle \Rightarrow (envl', sto')}{\langle T\ x\ is\ v\ D_V, envl, sto\rangle \Rightarrow (envl', sto')}$$

where $l = env_V\ next$

[EMPTY-DCL] $\langle \epsilon,\ envl,\ sto\rangle \Rightarrow \langle envl,\ sto\rangle$

Table 6.16: Small-step transition rules for declaration of variables

In **Table 6.16**, **VAR-DCL** is the rule for declaring variables. The rule does not change store but the environment is changed due to the new variable being declared, which needs to have a location. In the condition new bindings are updated in the variable environment while the other environments remain the same. The next variable is also updated to the next location by using the new function on the location l.

VAR-INI-1 is the rule for a middle configuration when an initialisation is done at the same time as a declaration. In this rule only part of the expression *e* is evaluated. In rule **VAR-INI-2** the expression *e* is evaluated to a value.

VAR-INI-3 is the rule that updates the bindings in the variable environment. the variable x is bound to location l in the condition. The variable next is updated to a new location. The value *v* is bound to location l. Therefore the rule updates both the environment list and store.

EMPTY-DCL is an empty rule that virtually does nothing. This rule is necessary in order for stopping the sequence of variable declarations.

6.5.5 Declaration of Functions

The transition system are defined for this syntactical category as ($\Gamma, \Rightarrow$, T), where Γ is defined:

$$\mathbf{\Gamma_{DclF}} = DclF \times Fname \times Envl \times Sto \cup Envl \times Sto$$

T is defined as:

$$\mathbf{T_{DclF}} = Envl \times Sto$$

Small-step transition rules for declaration of functions:

[T-FUNC-DCL]
$$\frac{\langle D_F, (env_V, env_F[f \mapsto (b, \overrightarrow{x}, env_V, env_F, env_A)], env_A) : envl,\ sto\rangle \rightarrow \langle envl',\ sto\rangle}{\langle function\ f\ returns\ T\ (\overrightarrow{x})\ b\ D_F, envl, sto\rangle \rightarrow \langle envl',\ sto\rangle}$$

where $env_F\ f = \langle b,\ \overrightarrow{x},\ env_V,\ env_F,\ env_A\rangle$
and $\overrightarrow{x}\ =\ \{x_1,\ x_2,\ \ldots\ x_n\}$

[FUNC-DCL]
$$\frac{\langle D_F, (env_V, env_F[f \mapsto (b, \overrightarrow{x}, env_V, env_F, env_A)], env_A) : envl,\ sto\rangle \rightarrow \langle envl',\ sto\rangle}{\langle function\ f\ (\overrightarrow{x})\ b\ D_F, envl, sto\rangle \rightarrow \langle envl',\ sto\rangle}$$

where $env_F\ f = \langle b,\ \overrightarrow{x},\ env_V,\ env_F,\ env_A\rangle$
and $\overrightarrow{x}\ =\ \{x_1,\ x_2,\ \ldots\ x_n\}$

[EMPTY-DCL] $\langle \epsilon,\ envl,\ sto\rangle \rightarrow \langle envl,\ sto\rangle$

Table 6.17: Small-step transition rules for declaration of functions

In **Table 6.17**, **T-FUNC-DCL** is the rule for functions with a return type. The condition updates the function environment where the function name is bound to the function body, the formal parameters, and the environments at the time of declaration.

FUNC-DCL is similar except the syntax is different in the rule. In these two rules the sequences of x also describes the case of no parameters due to the similarity in the rules.

6.5.6 Declaration of Arrays

The transition system for array declarations are defined as $(\Gamma, \Rightarrow, \mathrm{T})$, where Γ is defined:

$$\mathbf{\Gamma_{DclA}} = DclA \times Aname \times Envl \times Sto \cup Envl \times Sto$$

T is defined as:

$$\mathbf{T_{DclA}} = Envl \times Sto$$

Transitions for array declarations therefore describe how the environment list and sto is updated when declaring an array.

Definition of small-step transition rules for declaration of arrays:

[ARR-DCL]
$$\frac{\langle D_A, (env_V, env_F, env_A[a \mapsto l][next \mapsto following\ l]) : envl,\ sto\rangle \rightarrow \langle envl',\ sto\rangle}{\langle T\ array\ a\ D_A,\ envl,\ sto\rangle \rightarrow \langle envl',\ sto\rangle}$$

where $l = env_A\ next$

[EMPTY-DCL] $\langle \epsilon,\ envl,\ sto\rangle \rightarrow \langle envl,\ sto\rangle$

Table 6.18: Small-step transition rules for declaration of Arrays

The rule **ARR-DCL** in **Table 6.18** shows how the runtime stack and sto is updated when declaring an array. From the condition it can be seen how the variable a is bound to a location in the array environment and next variable updated to the next unused location.

7 | Type System Specification

The type system of the Trun language will be defined by a type judgement for each syntactical category of the abstract syntax. Each type judgement will cover a number of type rules. The formal procedure to describe these type rules will be introduced together with a new environment model for types. All type rules will not be included in this chapter. A complete list can instead be found in ***Appendix D****.*

7.1 Types in Trun

The Trun-language has a static typed system, meaning that for every variable, array, and function declaration the specific type must syntactically be specified. The type system makes it possible to do so. The use of types in general helps to prevent run-time errors when translating a program. The definition of a type system is similar to the definition of a structural operational semantic as both are syntax-directed [27, p. 184].

```
A   :=   integer | decimal | text | truth
N   :=   integer | decimal
T   :=   A | (x : T) → ok | (x : T) → T'
```

Table 7.1: Abstract syntax of Types

Table 7.1 elaborates on the syntactical category Type with meta variable (T). A subset of this category are the *basic types*, which are the same primitive types introduced back in **Section 5.2**, namely *integer, decimal, text* and *truth*.

The compound type $((x : T) \rightarrow ok)$ denotes that the formal parameters of a function has to be of type T. This applies to functions without a return type. $((x : T) \rightarrow T')$ is used for functions with a return type. The function has to specify its return type T'.

7.2 Type environments

For the type system another environment concept, similar to the environment-store model from **Section 6.4**, is needed to keep track of the type of each declared variable, array, or function. This model is called the type environment and is denoted with E [27, p. 187].

The type environment E is a partial function from the set of variables, function names, and array names to types:

$$E : \mathbf{Var} \cup \mathbf{Aname} \cup \mathbf{Fname} \rightharpoonup \mathbf{Type}.$$

E can be used to look up the type of every known declared variable, array, and function. A single update to the type environment is denoted by $E[x \mapsto T]$. This means that in E the variable x is now bound to type T. The same rule applies to arrays (a) and functions (f).

To be able to update the existing type environment after multiple declarations four update functions are introduced. $E(D_V, E)$ for variable declarations, $E(D_A, E)$ for array declarations, and $E(D_F, E)$ for function declarations. The definitions are shown below in **Table 7.2**:

$$E(T\ x;\ D_V, E) = E(D_V, E[x \mapsto T]) \tag{7.1}$$

$$E(T \text{ array } a;\ D_A, E) = E(D_A, E[a \mapsto T]) \tag{7.2}$$

$$E(\text{function } f(\overrightarrow{D_V}\ S;\ D_F, E) = E(D_F, E[f \mapsto (\overrightarrow{D_V} : \overrightarrow{T}) \to ok]) \tag{7.3}$$

$$E(\text{function } f(\overrightarrow{D_V} \text{ returns } T'\ S;\ D_F, E) = E(D_F, E[f \mapsto ((\overrightarrow{D_V} : \overrightarrow{T}) \to ok,\ (S : T') \to ok])) \tag{7.4}$$

Table 7.2: Functions used to update the type environment after declarations

Each function takes a declaration of a specific kind and updates the input type environment E with the new type bindings. As a result a new type environment is returned with the types of the new declarations. E(D_F, E) includes both (7.3) and (7.4), which is used to update the type environment with declarations of functions with and without a return type.

7.3 Type judgements

The type judgements defined in **Table 7.3** shows the valid type for each of the syntactical categories.

$$
\begin{aligned}
&E \vdash e : \mathbf{T} \\
&E \vdash S : \mathbf{ok} \\
&E \vdash p : \mathbf{ok} \\
&E \vdash b : \mathbf{ok} \\
&E \vdash D_V : \mathbf{ok} \\
&E \vdash D_A : \mathbf{ok} \\
&E \vdash D_P : \mathbf{ok}
\end{aligned}
$$

Table 7.3: List of type judgements for the syntactical categories

Expressions will be of one of the basic types: $E \vdash e : \mathbf{T}$ means that the expression e has type $\mathbf{T}$ when looking at the type bindings in the type environment E. The two Block-categories, Statements, and Declarations will all be of type **ok**. This means that all the type evaluations are valid.

7.4 Expressions

All type rules that define the type judgement $E \vdash e : \mathbf{T}$ are presented. These rules will ensure that an expression always will have a type that is a basic type. The expressions are split into three subcategories: General expressions, arithmetic expressions and truth expressions.

General expressions

[PAREN$_{exp}$]	$\dfrac{E \vdash e : \mathbf{T}}{E \vdash (e) : \mathbf{T}}$
[ARR-ELE]$_{exp}$]	$\dfrac{E(a) = \mathbf{T} \quad E \vdash e : \mathbf{integer}}{E \vdash a \text{ element } e : \mathbf{T}}$

Table 7.4: Type rules for general expressions

[ARR-ELE] uses the type environment E to find the type of the specific array to make sure the type of entire expression corresponds to it.

Arithmetic expressions

[PLUS$_{exp}$]	$\dfrac{E \vdash e_1 : \mathbf{N''} \quad E \vdash e_2 : \mathbf{N'}}{E \vdash e_1 + e_2 : \mathbf{N}}$
[U-MINUS$_{exp}$]	$\dfrac{E \vdash e : \mathbf{N}}{E \vdash -e : \mathbf{N}}$
[APPEND$_{exp}$]	$\dfrac{E \vdash e_1 : \mathbf{text''} \quad E \vdash e_2 : \mathbf{text'}}{E \vdash e_1 \text{ append } e_2 : \mathbf{text}}$

Table 7.5: Type rules for arithmetic expressions

In **Table 7.5** the [PLUS] and [U-MINUS] rule are chosen as they describe the pattern of the rest of the type rules in the arithmetic expressions. All type rules has to be of type N (integer or decimal). The [APPEND] rule is exclusively used for the text type.

Operator	Expression	Evaluates to
+	integer + integer	integer
	decimal + decimal	decimal
	integer + decimal	decimal
	decimal + integer	decimal
append	text append text	text
-	integer - integer	integer
	decimal - decimal	decimal
	decimal - integer	decimal
	integer - decimal	decimal
unary	- integer	integer
unary	- decimal	decimal
/	integer / integer	integer
	decimal / decimal	decimal
	decimal / integer	decimal
	integer / decimal	decimal
*	integer * integer	integer
	decimal * decimal	decimal
	decimal * integer	decimal
	integer * decimal	decimal

Table 7.6: Evaluation of types in arithmetic expressions

Table 7.6 shows the specific evaluations of the arithmetic operators. Expressions with operands of different types (integer and decimal) will always evaluate to decimal.

Truth expressions

[EQUALS$_{exp}$]	$\dfrac{E \vdash e_1 : \mathbf{T} \quad E \vdash e_2 : \mathbf{T}}{E \vdash e_1 = e_2 : \mathbf{truth}}$
[GREATER-THAN$_{exp}$]	$\dfrac{E \vdash e_1 : \mathbf{N'} \quad E \vdash e_2 : \mathbf{N}}{E \vdash e_1 > e_2 : \mathbf{truth}}$
[LESS-THAN$_{exp}$]	$\dfrac{E \vdash e_1 : \mathbf{N'} \quad E \vdash e_2 : \mathbf{N}}{E \vdash e_1 < e_2 : \mathbf{truth}}$
[NOT$_{exp}$]	$\dfrac{E \vdash e_1 : \mathbf{truth}}{E \vdash \text{NOT } e : \mathbf{truth}}$
[AND$_{exp}$]	$\dfrac{E \vdash e_1 : \mathbf{truth'} \quad E \vdash e_2 : \mathbf{truth''}}{E \vdash e_1 \text{ AND } e_2 : \mathbf{truth}}$
[OR$_{exp}$]	$\dfrac{E \vdash e_1 : \mathbf{truth'} \quad E \vdash e_2 : \mathbf{truth''}}{E \vdash e_1 \text{ OR } e_2 : \mathbf{truth}}$

Table 7.7: Type rules for relational and logical expressions

[EQUALS] from **Table 7.7** is possible to use on all types (T). It is however not possible to evaluate two different types against each other. [GREATER-THAN] and [LESS-THAN] are able to evaluate the two types integer and decimal (N) against each other. [NOT], [AND], and [OR] are logical operators and can therefore only be of type truth.

7.5 Block structures

Here the two type rules that defines the type judgements for the two syntactical categories (p) and (b): $E \vdash p$: **ok** and $E \vdash b$: **ok** will be explained.

[PROG-BLOCK]$_{block}$	$\dfrac{E \vdash D_V : \textbf{ok} \quad E_1 \vdash D_A : \textbf{ok} \quad E_2 \vdash D_F : \textbf{ok} \quad E_3 \vdash S : \textbf{ok}}{E \vdash \text{begin } D_V\ D_A\ D_F\ S \text{ end} : \textbf{ok}}$ where $E_1 = E(D_V, E)$ and $E_2 = E(D_A, E_1)$ and $E_3 = E(D_F, E_2)$
[BLOCK]$_{block}$	$\dfrac{E \vdash D_V : \textbf{ok} \quad E_1 \vdash S : \textbf{ok}}{E \vdash \text{begin } D_V\ S \text{ end} : \textbf{ok}}$ where $E_1 = E(D_V, E)$

Table 7.8: Type rules for the block structures

For the [PROG-BLOCK] rule the functions from **Table 7.2** are used to update the type environment with the declarations inside the block that in this case is the entire program. As declarations are made in a certain order starting with variables D_V, D_A arrays second, and functions D_F last, the update functions has to use the newly updated type environments. The update function for D_V creates a new type environment, with the new type bindings of the variables, denoted E_1. That means the update function for D_A has to update E_1 making E_2. The update function for function declarations is then used with E_2 to create type environment E_3. The statements are evaluated in E_3.

7.6 Statements

This section presents all type rules that together defines the type judgement $E \vdash S : \textbf{ok}$. The type of a statement will always be ok. Just like expressions, the statements will also be split into subcategories: general statements and control structures. The predefined statements is excluded from this chapter but can instead be found in **Appendix D**.

General statements

[VAR-ASS$_{stmt}$]	$\dfrac{E \vdash x : \mathbf{T} \quad E \vdash e : \mathbf{T}}{E \vdash x \text{ is } e : \mathbf{ok}}$
[ARR-ASS$_{stmt}$]	$\dfrac{E \vdash e_2 : \mathbf{T} \quad E(a) = \mathbf{T} \vdash \mathbf{T} \quad E \vdash e_1 : \mathbf{integer}}{E \vdash a \text{ element } e_1 \text{ is } e_2 : \mathbf{ok}}$
[COMP$_{stmt}$]	$\dfrac{E \vdash S_1 : \mathbf{ok} \quad E \vdash S_2 : \mathbf{ok}}{E \vdash S_1\ S_2 : \mathbf{ok}}$
[RETURN]$_{stmt}$]	$\dfrac{E(f) = ((\overrightarrow{x} : \overrightarrow{T}) \rightarrow \mathbf{ok}, (S : T')) \vdash \mathbf{T'} \quad E \vdash e : \mathbf{T'}}{E \vdash \text{return } e : \mathbf{ok}}$
[CALL-RETURN$_{stmt}$]	$\dfrac{E(f) = ((\overrightarrow{x} : \overrightarrow{T}) \rightarrow \mathbf{ok}, (S : T')) \vdash \overrightarrow{\mathbf{T}}, \mathbf{T'} \quad E \vdash \overrightarrow{e} : \overrightarrow{\mathbf{T}}}{E \vdash f(\overrightarrow{e}) : \mathbf{T'}}$
[CALL$_{stmt}$]	$\dfrac{E(f) = ((\overrightarrow{x} : \overrightarrow{T}) \rightarrow \mathbf{ok}, (S : T')) \vdash \overrightarrow{\mathbf{T}} \quad E \vdash \overrightarrow{e} : \overrightarrow{\mathbf{T}}}{E \vdash f(\overrightarrow{e}) : \mathbf{ok}}$

Table 7.9: Type rules for the general statements

[ARR-ASS] uses the type environment E to make sure the type of the expression e_2 is evaluated to the same type as the array it self.

In the [RETURN] the type environment E is also used to get the type of the function in which the return statement is used. This information is added to the to the function identifier f from the declaration rules which are described later. With knowledge of the function's type is can be checked if the type of the expression to return is the same type as the function that holds the return statement.

The [CALL-RETURN] rule also needs the type of the function it self: T'. This type has to correspond to the type that is returned from the entire call statement. The rule also uses the type environment to check if the type of the argument(s) ($\overrightarrow{e}$) within the call are the same type as the corresponding formal parameter(s) from the declaration.

The [CALL] rule is identical to [CALL-RETURN] except for it does not need no evaluate the type of the function called to the type of the call statement it self.

Control structures

[IF$_{stmt}$]	$\dfrac{E \vdash e : \textbf{truth} \quad E \vdash b : \textbf{ok}}{E \vdash \text{if } \overrightarrow{\{e \text{ then} : b\}} : \textbf{ok}}$
[IF-ELSE$_{stmt}$]	$\dfrac{E \vdash e : \textbf{truth} \quad E \vdash b_1 : \textbf{ok} \quad E \vdash b_2 : \textbf{ok}}{E \vdash \text{if } \overrightarrow{\{e \text{ then} : b_1\}} \text{ else } b_2 : \textbf{ok}}$
[WHILE$_{stmt}$]	$\dfrac{E \vdash e : \textbf{truth} \quad E \vdash b : \textbf{ok}}{E \vdash \text{while } e \text{ do } b : \textbf{ok}}$
[REPEAT$_{stmt}$]	$\dfrac{E \vdash b : \textbf{ok} \quad E \vdash e : \textbf{truth}}{E \vdash \text{repeat } b \text{ until } e : \textbf{ok}}$
[FROM-UPTO]$_{stmt}$]	$\dfrac{E \vdash e_1 : \textbf{integer} \quad E \vdash e_2 : \textbf{integer} \quad E \vdash b : \textbf{ok}}{E \vdash \text{from } (e_1 \text{ upto } e_2)\ b : \textbf{ok}}$
[FROM-DOWNTO]$_{stmt}$]	$\dfrac{E \vdash e_1 : \textbf{integer} \quad E \vdash e_2 : \textbf{integer} \quad E \vdash b : \textbf{ok}}{E \vdash \text{from } (e_1 \text{ downto } e_2)\ b : \textbf{ok}}$

Table 7.10: Type rules for the control structures in statements

In **Table 7.10**, the type rules for the control structures are presented. For both [IF] and [IF-ELSE] e is the conditional expression for the control structure. It can only be of type truth. All control structures has a block as the body. The [BLOCK] denoted with (b) is explained in **Table 7.8**.

7.7 Declarations

This section presents all type rules that together defines the type judgements $E \vdash D_V : \textbf{ok}$, $E \vdash D_P : \textbf{ok}$, and $E \vdash D_A : \textbf{ok}$.

[EMPTY$_{dec}$]	$E \vdash \epsilon : \mathbf{ok}$
[VAR-ASS$_{dec}$]	$\dfrac{E[x \mapsto T] \vdash D_V : \mathbf{ok} \quad E \vdash e : \mathbf{T}}{E \vdash T\ x \text{ is } e\ D_V : \mathbf{ok}}$
[VAR$_{dec}$]	$\dfrac{E[x \mapsto T] \vdash D_V : \mathbf{ok}}{E \vdash T\ \text{x}\ D_V : \mathbf{ok}}$
[FUNC-RETURN$_{dec}$]	$\dfrac{E \vdash b : \mathbf{ok} \quad E[f \mapsto ((\overrightarrow{D_V} : \overrightarrow{T} \rightarrow \mathbf{ok}),\ T')] \vdash D_F : \mathbf{ok}}{E \vdash \text{function } f \text{ returns } T'\ (\overrightarrow{D_V})\ b\ D_F : \mathbf{T'}}$
[FUNC$_{dec}$]	$\dfrac{E \vdash b : \mathbf{ok} \quad E[f \mapsto (\overrightarrow{D_V} : \overrightarrow{T} \rightarrow \mathbf{ok})] \vdash D_F : \mathbf{ok}}{E \vdash \text{function } f\ (\overrightarrow{D_V})\ b\ D_F : \mathbf{ok}}$
[ARR$_{dec}$]	$\dfrac{E[a \mapsto T] \vdash D_A : \mathbf{ok}}{E \vdash T \text{ array } a\ D_A : \mathbf{ok}}$

Table 7.11: Type rules for variable, array, and function declarations

In type rule [VAR-ASS] in **Table 7.11** the notation $[x \mapsto T]$ means that variable x is given a type binding to T. Expression e has to correspond to that specific type.

[FUNC-RETURN] and [FUNC] are very similar. Each type rule says that in type environment E function f's formal parameter(s) $(\overrightarrow{D_V})$ have to be of type $\overrightarrow{T}$. [FUNC-RETURN] adds another type binding to the function identifier f. f has to be of type T'. This is done in order to be able to check if the type of the return statement within the body of the function corresponds to the type it returns. This was also explained with the [RETURN] rule from **Table 7.9**.

8 | Implementation

The implementation of the compiler is described throughout this chapter with code snippets from the source code. Different tools for generating the lexer and parser are discussed to choose the most appropriate tool for the project. From this the implementation of the abstract syntax tree, symbol table, type checker and code generation phase is described in further detail.

8.1 Tools

Three parsing tools have been analysed relative to their strengths and weaknesses. The three parsing tools presented are ANTLR, JavaCC, and SableCC. These tools have been chosen as they were presented in the language and compiler course. Each parsing tool also comes with an integrated lexer. The purpose of researching the tools capabilities is to choose an appropriate tool for implementation.

8.1.1 ANTLR

ANTLR (ANother Tool for Language Recognition) is a *powerful parser generator for reading, processing, executing, or translating structured text or binary files* [28]. It is widely used for educational purposes and in the industry [29]. ANTLR is capable of generating parsers in multiple programming languages such as Java, Python, C#, and JavaScript and includes support (via plugins) for popular IDE's (JetBrain's IntelliJ, NetBeans, Eclipse etc.) [30] [31]. The plugin provides the advantages of an IDE when utilising ANTLR, such as syntax highlighting, syntax error checking, code auto completion, lexer and parser generation.

The grammar supports EBNF notations such as optional, repetition, grouping etc., as well as using the vertical bar (|) to denote alternative productions. ANTLR can generate the lexer, parser and concrete syntax tree from a grammar file [32] [29].

8.1.2 JavaCC

Java Compiler Compiler (JavaCC) is a popular parser generator exclusively for writing the parser in Java. This generator builds top-down parsers. The grammar specification is given using EBNF.When generating the parser, the lexical specifications, such as regular expression and the grammar specification (the CFG), are gathered into one file making it easier to maintain. [33].

Furthermore, JavaCC can handle all Unicode characters. It is possible to define special tokens which is ignored during parsing which can be useful when processing comments. This tool, in addition to generating the parser, also includes the JJTree tool which helps with tree building [33].

8.1.3 SableCC

SableCC is another compiler compiler for Java. SableCC handles LALR(1) grammars and generates bottom-up parsers, unlike JavaCC and ANTLR which are both top-down parsers. Therefore, SableCC has the advantages that LALR(1) parsers provide, e.g. left recursion is possible [34].

A difference between SableCC and the other tools is that it is possible to specify an abstract syntax and how it relates to the concrete syntax in the grammar specification [35]. The parser can return an AST instead of a CST and save the developers the trouble of converting the CST to an AST.

8.1.4 Evaluation

The chosen tool for generating the lexer and the parser is ANTLR. ANTLR is well documented with a book to describe every aspect of the tool. Furthermore, the plugin ANTLRWorks can be used together with Jetbrain's IntelliJ. The plugin will be helpful when testing and debugging the grammar.

The ANTLR tool generates a parse tree for an input string. The parse tree can be used in the following phases but an AST is preferable. It is necessary to manually construct an AST from the CST.

8.2 AST implementation

As an AST cannot be generated by ANTLR. It has to be manually implemented. The reasoning for implementing an AST is that it removes redundant information from the CST that is not needed in the later phases of the compiler. This means that the compiler has to visit less nodes and the implementation using the AST will be simpler when implementing visit methods for each node.

8.2.1 CST

The ANTLR generated concrete syntax tree contains a great deal of redundant information. Every terminal and non-terminal symbol is represented in the CST. To visualise this behaviour, a sample of the Trun language is presented in **Figure 8.1**. The code represents an integer declaration and an if-else control structure.

```
integer d
if (true) then {
    integer s
} else if (false) then {
    d is 4
} else then {
    d is 5
}
```

Figure 8.1: Sample code in Trun language

An example of a CST generated by ANTLR is shown on **Figure 8.2**.

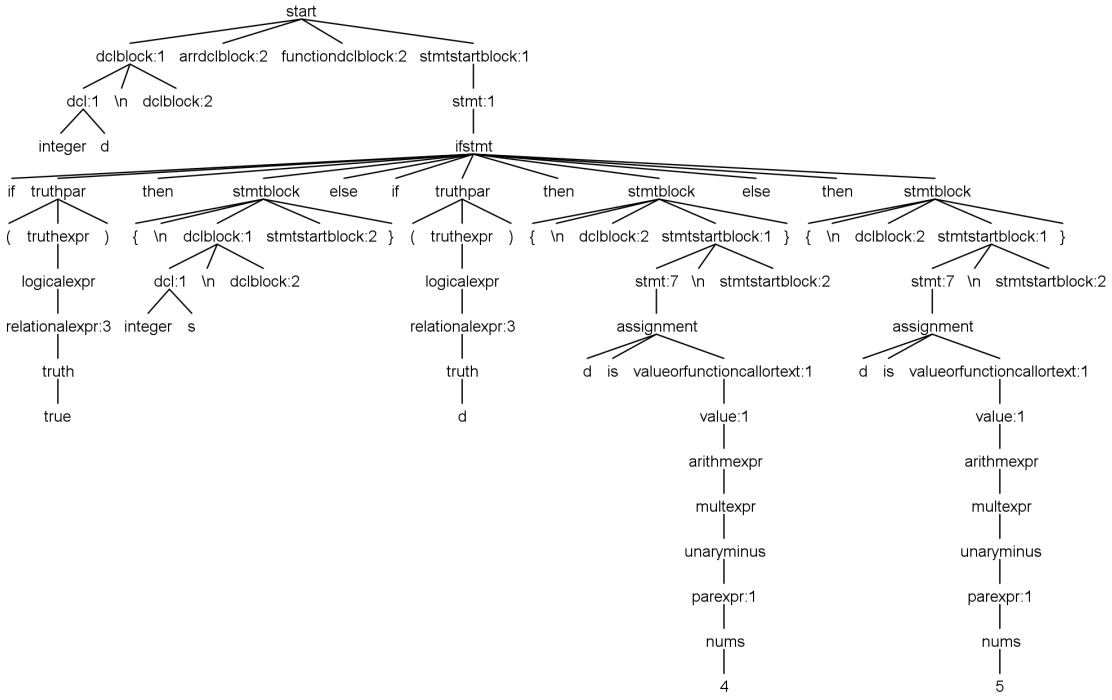

Figure 8.2: Concrete syntax tree for code in **Figure 8.1**

In **Figure 8.2**, there is information regarding terminal symbols such as parenthesis, newline characters, and *then* appears, which are redundant syntactical details. Unnecessary non-terminals are also presented. E.g. the non-terminals truthexpr, logicalexpr & relationalexpr are only present due to the implementation of precedence in the syntax. When designing the AST node classes, the unnecessary details should not be included.

8.2.2 Node classes

Before implementing the AST, it is necessary to determine which node classes the AST structure should consist of. When designing the AST node hierarchy, it is important to only include the necessary details for the later phases. This is important as more nodes will require longer time when traversing the AST.

In the AST classes, there will be an AbstractNode class which all other nodes will either inherit from or will be a field in a node which inherits from AbstractNode. The class diagram for the AST classes is attached in **Appendix E**. In order to show the difference in the amount of nodes

between the AST and CST, the AST in **Figure 8.3** and the CST in **Figure 8.2** can be compared.

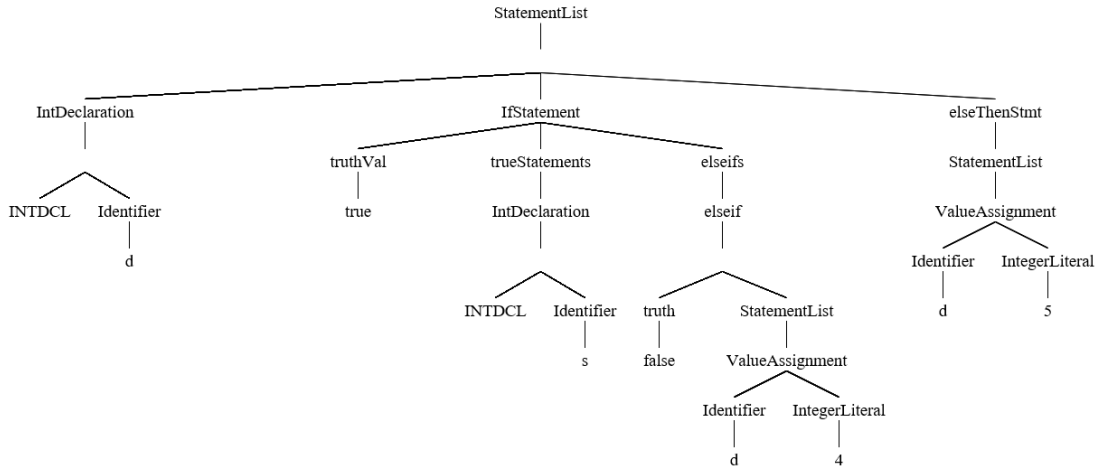

Figure 8.3: Abstract syntax tree for code in **Figure 8.1**

8.2.3 Conversion from CST to AST

In order to generate an AST from a CST, it is necessary to traverse the CST. ANTLR provides two approaches to traverse the CST. One of the methods is through the visitor pattern while the other is a listener. The main difference between a visitor and a listener is that the developer can decide which children to visit in a visitor implementation, while in the listener it is possible to react to the tree walker generated by ANTLR. Due to this difference, the visitor has been chosen as it is not necessary to traverse all nodes in the CST.

In the implementation of the visitor pattern, a visit method for each node in the CST has been implemented. The classes generated by ANTLR contain a field for each terminal or non-terminal of the production rules. E.g. it can be seen from the CFG that the while statement contains the symbols: WHILE, truthpar, DO & stmtblock. As such the WhilestmtContext class generated for the while statement by ANTLR will contain the fields of the aforementioned symbols. When visiting the while statement, the non-terminals have to be visited in most cases. The necessary results from terminals and visited non-terminals have to be used to construct AbstactNode instances. In the case with the while statement, the visit method for the while statement should return a WhileStatement (a subclass of AbstactNode) object with the relevant information from the parse class. The implementation of the visit method for a while statement is presented on **Figure 8.4**.

```
1 @Override
2 public AbstractNode visitWhilestmt(PyTrun.WhilestmtContext ctx) {
3         return new WhileStatement((Value) visitTruthpar(ctx.truthpar()),
4                         (StatementList) visitStmtblock(ctx.stmtblock()),
5                         ctx.getStart().getLine());
6 }
```

Figure 8.4: Visit method for While statement

In the while statement, the noteworthy symbols are truthpar & stmtblock and therefore has been included in the *WhileStatement* object in visit method for the while statement. On the other hand, the symbols WHILE & DO are unnecessary details and therefore have not been included. In cases where a non-terminal has more than one production, it is necessary to check whether a symbol which is unique to one of the production rules is null. If it is different from null then the production rule applied will be the rule with the unique symbol. An example of multiple productions is the production rule for statement in **Figure 5.19**. The visit method for statement is shown in **Figure 8.5**.

```
1 @Override
2 public AbstractNode visitStmt(PyTrun.StmtContext ctx) {
3     Statement stmt = null;
4     if(null != ctx.ifstmt()){
5             stmt = (Statement) visitIfstmt(ctx.ifstmt());
6     }
7     //else ifs omitted
8     return stmt;
9 }
```

Figure 8.5: visit method for Statement

The StmtContext class generated for statement contains a unique field for each statement. In **Figure 8.5** on the lines 4 - 6, an example of how the visitor is implemented for a production rule with more than one production.

8.2.4 Visitor for AST

The visitor for the AST can be developed using different approaches. The chosen approach is the reflective visitor pattern. The main reason for choosing the reflective visitor pattern rather than the traditional visitor pattern is due to the implementation language (Java) does not support multiple dispatch. This means that each of the the AST nodes would require accept methods which call the visit methods of the input visitor on the nodes. As Java supports reflection, it is possible to find the most appropriate method for an Abstract node through reflection rather than with visit methods in every concrete node (multiple dispatch).

In the traditional visitor pattern, it is also necessary to implement a visit method for each concrete node in the visitor, even if no action is required on the visitor part. This is not necessary in the reflective visitor pattern. The reflective visitor pattern requires an abstract node visitor with a visit method which finds the most appropriate visit method in a visitor and invokes it when called on an Abstract node object. The most appropriate visit method is the method with the input parameter of the closest ancestor in the visitor. **Figure 8.6** shows the visit method.

```
1 public Object visit(visitable v) throws NoSuchMethodException{
2     Method m = findMethod(v);
3     try {
4        return m.invoke(this, new Object[] { v });
5     }
6     catch ( InvocationTargetException e2 ) {
7         throw e2.getTargetException();
8     }
9     return null;
10 }
```

Figure 8.6: The visit method implemented in Java with reflection

The *findMethod()* for finding the most appropriate method is shown in **Figure 8.7**.

```
1  private Method findMethod(visitable o) throws NoSuchMethodException{
2      Class<? extends Object> nodeClass = o.getClass();
3      Method answer = null;
4
5      for (Class<? extends Object> c = nodeClass; c != Object.class
6              && answer == null; c = c.getSuperclass()) {
7          answer = getClass().getMethod("visit", new Class[] { c });
8      }
9
10     Class<? extends Object> iClass = nodeClass;
11     while (answer == null && iClass != Object.class) {
12         Class<? extends Object>[] interfaces = iClass.getInterfaces();
13         for (int i = 0; i < interfaces.length; i++) {
14             try {
15                 answer = getClass().getMethod("visit",
16                         new Class[] { interfaces[i] });
17             } catch (NoSuchMethodException e) {
18             }
19         }
20         iClass = iClass.getSuperclass();
21     }
22
23     if (answer == null) {
24         try {
25             answer = getClass().getMethod("defaultVisit",
26                     new Class[] { (new Object()).getClass() });
27         } catch (NoSuchMethodException e) {
28         }
29     }
30     return ans;
31 }
```

Figure 8.7: The findMethod which finds the most appropriate visit method in a visitor.

In **Figure 8.7**, the implementation behind *findMethod* is shown. It is used to find the most appropriate method. The method takes a *visitable* object as an input parameter. *visitable* is an interface which every class which should be visitable by the visitor implements. In the lines 5-8, it checks whether a visit method for the input parameter exists. If it does not then the for loop will check whether a visit method for an ancestor class exists in the next iteration. If no such method exists, answer will still be null and whether a visit method for one of interfaces implemented by the visitable object o is checked on the lines 10-21. If no such method exists, then the *defaultVisit* method will be assigned to answer on the lines 23 to 30, and if the *defaultVisit* is not implemented then an exception will be thrown to notify that no implementation of visit is available for object o.

8.3 Symbol table

For the implementation of the symbol table, a visitor is implemented to visit all nodes with variable, array, and function declarations. The symbols are saved in a single symbol table for the entire program containing all the scopes. This is an alternative to creating a symbol table for each scope as explained in **Section 4.5**. This symbol table is constructed using a HashMap containing a String object as key and *Symbol* object as the value. The Symbol class contains fields for the depth of the scope and the type.

The symbol table has methods to open and close scopes to ensure the correct variables are available. A new scope is opened upon reaching a control structure node or a function node, hereby the depth level is increased and variables within the scope gain access to the declarations which has the same depth value or lower. Closing the scopes after its use is important to remove the entries in the symbol table as variables created within scopes do not persist afterwards.

When adding symbols to the symbol table, it checks whether they already exist. If the symbol already exists in the table a duplication error message is shown stating which variable is previously already declared, as displayed on **Figure 8.8**.

```
DuplicateDeclaration line: 4 --  Variable a is already declared
```

Figure 8.8: The exception given when a identifier is already in use.

If the symbol does not exist in the table it enters the symbol table, which is completed by the method shown on **Figure 8.9**. It shows an example of visiting an integer declaration (*IntDeclaration* node) that has yet to be added to the table.

```
1  @Override
2  public Object visit(IntDeclaration node) throws NoSuchMethodException {
3      if (!(isInSymbolTable(node.getId().getSpelling()))) {
4          symbolTable.put(node.getId().getSpelling(), new Symbol(node,
5          symbolTable.getDepth(), new IntegerLiteral(
6          node.getId().getSpelling())));
7      } else {
8          errorCallDuplicateDeclaration(node.getId().getSpelling(),
9          node.getLineNumber());
10     }
11      //Type checking and semantic check omitted
12
13     return new IntegerLiteral(node.getId().getSpelling());
14 }
```

Figure 8.9: Implementation for integer declaration being added to the symbol table

isInSymbolTable checks if an identifier name already exists in the HashMap. If the symbol is already present an error is thrown on lines 8-9. If the identifier name is not found in the HashMap the node is inserted into the HashMap with the ID as key and a symbol object containing the node, depth, and type as value (lines 4-6).

Consider the example on **Figure 8.10** of a particular program in the Trun language.

```
1 integer a is 5
2
3 function func returns integer(){
4     integer x is 6 + a
5     return x
6 }
```

Figure 8.10: Sample code in source language

In this example the variables *a*, *func* and *x* are all entered into the symbol table as the AST is traversed. The symbol contains the information regarding each variable from the node type whether it is initialised or not.

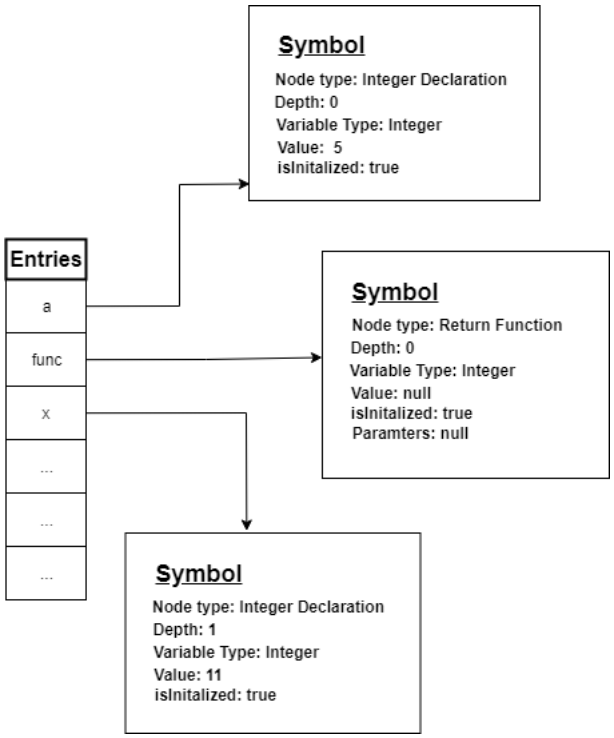

Figure 8.11: Symbol table for a specific program

As displayed in the symbol table on **Figure 8.11**, the symbols are *a*, *func* and *x*. The field *isInitialized* of function is always set to true. To use a function all the symbols' *isInitialized* field of the parameters must also be true as it will give an error message otherwise.

The symbol of *x* stores the value 11, meaning it can see the value of a. A supporting visitor is used to calculate the values of an expression to save it in the symbol table.

8.4 Type Checking

The type checking implementation is based on the rules stated formally in **Chapter 7**. The visitor for the type checker is implemented in the same visitor as the symbol table's visitor.

Figure 8.12 depicts an example of type checking of the *less than* operator. The left-hand and right-hand side is of type integer. Based on the type rules, these can be compared by the *less than* operator and will thereby evaluate to a truth type.

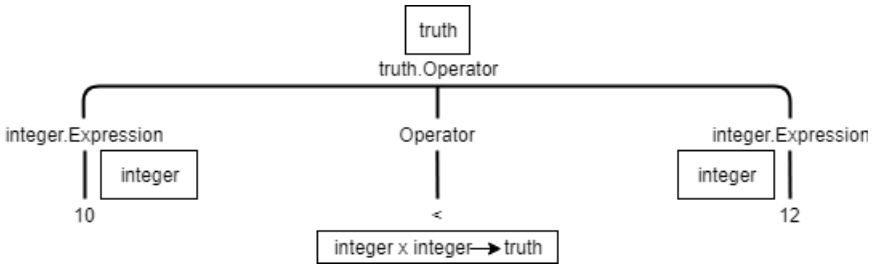

Figure 8.12: Example of the truth operator, less than

In the implementation of this particular example in **Figure 8.13**, the left and right-hand side's values are visited in order to determine their type. Afterwards, each value is put into the *evaluateTruth* method that determines if the types are allowed on the specific operation. An exception is thrown if the type rules do not allow the comparison.

```
1  @Override
2 public Object visit(LessThan node) throws NoSuchMethodException {
3     Value leftValue = (Value) visit(node.getLhs());
4     Value rightValue = (Value) visit(node.getRhs());
5
6     return evaluateTruth(node.getClass().getName(), leftValue, rightValue,
7     node.getLineNumber());
8 }
```

Figure 8.13: Code implementation of the LessThan node.

As stated in **Section 7.1** the comparisons are not limited to only the left and right-hand side being the same type. E.g some operators allows comparison between integer and decimal.

8.4.1 Exceptions in Type Checking

Type checking introduces custom exceptions in the semantic analysis that give feedback for the user at compile time. These exceptions are the following:

- IncompatibleTypes
- IncorrectOperatorUse
- FunctionCallParameter

IncompatibleTypes exception is given when two different types of literals are used in a context in which it is not allowed. This could mean trying to multiply a text type with an integer, or simply attempting to store a decimal number in an integer declaration.

```
IncompatibleTypes line: 1 -- Decimal cannot be assigned to b
```

Figure 8.14: The exception given when the types used cannot be mixed

IncorrectOperatorUse takes care of exceptions when the user attempts to use an operator in an incorrect context. An example of this would be using the *NOT*-keyword on an integer.

```
IncorrectOperatorUse line: 2 --  Operator Not cannot be applied to IntegerLiteral
```

Figure 8.15: The exception given when the operator cannot be used with the type

The last exception related to type checking is regarding the function calls. When a user makes a function call with arguments of types not corresponding to the type of the formally declared parameters in the function declaration the exception **Figure 8.16** is thrown.

```
FunctionCallParameter line: 7 -- the parameter types of functionName are incorrect
```

Figure 8.16: The exception given when a functioncall has incorrect parameter types

Another variation of the same exception exists. This exception is thrown if the amount of parameters are incorrect with the formally declared function.

8.5 Additional Contextual Analysis

Additional semantic analysis has been performed to prevent errors not handled prior in the implementation. The purpose of the additional analysis is to make a compiler that considers as much as possible at compile time.

The additional contextual analysis consists of error handling of the following:

- Missing variables
- Uninitialised Variables
- Array out of bounds
- Missing return statement in functions

All of these issues are checked and exceptions are thrown giving the user feedback specifically stating why and where the issue lies. These exceptions are very similar to the ones previously displayed in **Section 8.4.1**.

Missing variables is an issue whenever a user is attempting to use a variable or function that has not previously been declared. This is checked in the context of the scope in which the variable is attempted to be used.

Uninitialised variables are checked because attempting to use variables without being initialised can cause errors and is semantically incorrect. It is helpful for debugging as well.

While all out of bounds errors with arrays cannot be checked at compile time, the ones where the user attempts to index directly outside the array are possible to catch. Instead of an error message, a warning is given as it does not necessarily stop the program from executing.

Functions of the return type are required to have return statements in them to be valid. Whether this is in another control structure or at the very end within the function, is up to the user, however without a return statement a successful compile is not possible.

8.6 Code Generation

Code generation utilises yet another visitor. By visiting the AST, appropriate code is generated for each node visited. The code is compiled to C which can be executed by an Arduino Uno.

Three classes are used during code generation. The first class is called *GenSetup*. The *GenSetup* class implements the method *getInitialCode()*, which can be seen on **Figure 8.17**.

```
1 String getInitialCode(){
2     return
3         "const int leftMotor = 12;\n" +
4         "const int rightMotor = 13;\n" +
5         "\n" +
6         "void setup() {\n" +
7         "  pinMode(leftMotor, OUTPUT);\n" +
8         "  pinMode(rightMotor, OUTPUT);\n";
9 }
```

Figure 8.17: Method getInitialCode() in GenSetup class

getInitialCode() will return a formatted string containing the functions and specific constants needed for the Trun car in C. As mentioned in **Section 2.4**, an Arduino program requires *setup()* and *loop()* functions in order to properly operate. Access to pins on the Arduino is achieved through initialising constant variables to the corresponding pin numbers (lines 2-3), and specifying the pin mode in the *setup()* function (lines 6-7).

The second class is called Emitter. The Emitter class implements a method *emit()* for writing generated code to a file. The method and the constructor for the Emitter class can be seen on **Figure 8.18**.

```
1 public Emitter(String filepath) {
2     try {
3         writer = new FileWriter(filepath);
4     } catch (IOException e) {
5         e.printStackTrace();
6     }
7 }
8
9 void emit(String emitString) {
10     try {
11         writer.write(emitString);
12     } catch (IOException e) {
13         e.printStackTrace();
14     }
15 }
```

Figure 8.18: Method emit() in Emitter class

When an instance of the Emitter class is created, a Writer object is created through the Emitter class' constructor (line 3). Every time a node is visited, the method *emit()* will be called with a string as a parameter and write to the file. The string passed to the emit method is the generated code.

The third class is the visitor class *CodeGenVisitor*. This class implements all necessary visitor methods for each node. The implementation for the *drive()* statement's visitor can be seen on **Figure 8.19**.

```
1 public Object visit(DriveStatement driveStatement) {
2     emitter.emit("\ndigitalWrite(leftMotor, HIGH);\n
3     digitalWrite(rightMotor, HIGH);
4     \ndelay(1000*");
5     visit(driveStatement.getVal());
6     emitter.emit(");\ndigitalWrite(leftMotor, LOW);\n
7     digitalWrite(rightMotor, LOW)");
8     return null;
9 }
```

Figure 8.19: Example of code generation visitor

Visitors for all other constructs in the Trun language are also implemented in a similar fashion. When the *visit()* method call in **Figure 8.19** on line 5 is called with *driveStatement*'s *'Val'*, it will call the appropriate visit() method. This will call the *visit()* that takes an *IntegerLiteral* object as an input parameter (syntactically, the *drive()* statement only accepts an integer as a parameter). This *visit()* method can be seen on **Figure 8.20** and simply uses the emitter to emit the Integer in question.

```
1 public Object visit(IntegerLiteral integerLiteral) {
2         emitter.emit(integerLiteral.getSpelling());
3         return null;
4     }
```

Figure 8.20: IntegerLiteral visitor in CodeGenVisitor.java

As code segments to be executed on the Arduino need to be put in either the *setup()* or *loop()* function body, a small complication arises. It is required to place function definitions outside of the *setup()* and *loop()* body, as one cannot define a function within a function (nested functions) in standard C. Another complication that arises from this is the visibility or scope of declared variables. If a variable is used in a function without being an actual parameter, it needs to be a global variable in C, e.g. declared in the outermost scope. The generated code that is not a declaration or function is put in the *setup()* function, as the Arduino only executes this block once. Furthermore, it is possible run the compiler with *–loop* as an argument to have the code generated inside the *loop()* function instead of *setup()*, if the user would like the code to repeat continuously.

This is achieved by adding a boolean field in the CodeGenVisitor class, `boolean isDeclaration`, to denote whether it should generate code inside or outside of the setup() body. Creating two objects of the CodeGenVisitor class each with a different boolean value, as depicted on **Figure**

8.21, allows us to determine which nodes to visit before and after printing the formatted string from **Figure 8.17**.

```
1 Emitter emitter = new Emitter(args[1]);
2 CodeGenVisitor codeGenDclVisitor = new CodeGenVisitor(emitter, true);
3 codeGenDclVisitor.visit(ast);
4
5 CodeGenVisitor codeGenVisitor = new CodeGenVisitor(emitter, false);
6 codeGenVisitor.setup();
7 codeGenVisitor.visit(ast);
8
9 codeGenDclVisitor.closeEmitter();
```

Figure 8.21: Code Generation Code in Main.java

Figure 8.22 depicts the for loop that iterates over the list of all statements in the AST. The aforementioned boolean *isDeclaration* (line 2) will decide if the CodeGenVisitor object will visit instances of all declarations (lines 3-5), or everything besides declarations (line 9).

```
1 for (Statement s : statementList.getStmts()) {
2     if (isDeclaration) {
3         if (s instanceof FunctionDeclaration || s instanceof IntDeclaration
4             || s instanceof TextDeclaration  || s instanceof FloatDeclaration
5             || s instanceof TruthDeclaration || s instanceof ArrayDeclaration) {
6             visit(s);
7         }
8     } else {
9         if (omitted) //The same statement as the if on line 3, but negated
10             visit(s);
11     }
12 }
```

Figure 8.22: Control Structures Responsible For Distributing Code In The Generated File

Below is an example of how code from the Trun Language (**Figure 8.23**) is split up when generating the code file (**Figure 8.24**).

```
integer a is 5
decimal b is 5.5

function turnLeftOrRight(truth t, integer c){
    if(t) then{
        turnleft(c)
    } else then{
        turnright(c)
    }
}

from (a upto 20) {
    turnLeftOrRight(b > a, 3)
    b is b + 0.7
}
```

Figure 8.23: Program in Trun

```
int a = 5;
float b = 5.5;
void turnLeftOrRight(int t,int c) {
    if(t){
        //turnleft
    }
    else{
        //turnright
    }
}
const int leftMotor = 12;
const int rightMotor = 13;

void setup() {
    pinMode(leftMotor, OUTPUT);
    pinMode(rightMotor, OUTPUT);

    for (a; a < 20; a++) {
        turnLeftOrRight(b > a, 3);
        b = b + 0.7;
    }
}

void loop() {
}
```

Figure 8.24: Generated Code in C

9 | Testing

The testing in the project is described in this chapter. Three different testing methods has been used to ensure the correctness of the implementation. The first method is unit testing by use of the JUnit framework, second is integration testing, and the third is usability testing.

9.1 Testing of Implementation

JUnit testing of the implemented visitors was performed using JUnit 4.12. Besides using JUnit, white box testing in the form of the integration testing has also been performed on the compiler to see if the input corresponds with the expected output of the compiler[36].

The purpose of unit testing is to test a specific part of the code. If the result of the test corresponds with the expected result it is considered passed, otherwise failed and the implementation is likely not behaving as intended.

9.1.1 JUnit Testing

The first code snippet of the JUnit testing is from the type checking implementation. From **Table 7.7**, [AND] is tested in the code snippet **Figure 9.1**

```
@org.junit.Test
public void visitAndTest() {
    And node = new And("And node", new TruthLiteral("false"),
        new TruthLiteral("true"), 2);
    try {
        assertTrue(symbolTableVisitor.visit(node) instanceof TruthLiteral);
        assertFalse(symbolTableVisitor.visit(node) instanceof TextLiteral);
        assertFalse(symbolTableVisitor.visit(node) instanceof DecimalLiteral);
        assertFalse(symbolTableVisitor.visit(node) instanceof IntegerLiteral);
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    }
}
```

Figure 9.1: AND node JUnit testing

By following the rules of **Table 7.7** the expression should only evaluate to truth when a truth is placed on each side of the AND node.

This type of JUnit testing is performed on every visit method of visitor as every method should have a meaningful return. Hereby it is ensured that every visit behaves as it should.

9.1.2 Integration Testing

Integration tests are when individual units are tested in a group. The purpose of this test is to find errors when combining the different units or phases in the implementation. The integration tests performed are manual tests where a Trun language program is written. Afterwards, the compiled file is examined to ensure it acts accordingly[36].

The expected layout of the compiled file is variable declaration, array declaration and function declarations at the top, in the stated order, followed by the statements.

```
integer i is 0
integer array arr

function driveFunc(integer sec){
    i is sec + i
    drive(i)
}

repeat{
    driveFunc(10)
} until(i = 200)
```

a: Trun integration test code

```
const int leftMotor = 12;
const int rightMotor = 13;
int i = 0;
int *arr = (int*) calloc(128,sizeof(int));
void driveFunc(int sec) {
    i = sec + i;

    digitalWrite(leftMotor, HIGH);
    digitalWrite(rightMotor, HIGH);
    delay(1000*(i));
    digitalWrite(leftMotor, LOW);
    digitalWrite(rightMotor, LOW);
}

void setup() {
  pinMode(leftMotor, OUTPUT);
  pinMode(rightMotor, OUTPUT);
    do {
        driveFunc(10);
        } while (!(i == 200));
    }

void loop() {
}
```

b: The compiled Arduino C code

Figure 9.2: Comparison between Trun and Arduino C

When comparing the generated code to the source code, displayed on **Figure 9.2**, it is important for the semantics to be properly implemented. Besides checking if the drive function does what it is supposed to. As seen on **Figure 9.2b** the corresponding code to *drive(i)* is both digital pins being set to high before being delayed the amount of time of *i*. The statements following the declarations should be printed in the setup body of Arduino C as it should only be executed fully once.

It is concluded based on this test and multiple other integration tests that the compiler behaves as it should when testing multiple components.

9.2 Usability Testing

A usability test is conducted to compare the Trun language to the existing Arduino language and to examine whether Trun is helpful for beginners compared to the exiting language[37].

9.2.1 Purpose

The purpose of the usability test is to determine whether the source language is easier to learn than the existing Arduino language. The test subject used for this usability test has prior experience with Arduino and programming which is not the ideal case for this test. However due to time and resources constraints, the test has been conducted on the test subject.

9.2.2 Method

The method used to conduct the usability test is the instant data analysis. The benefit of using this approach is that the analysis phase is short compared to the traditional usability method. To compare the Trun language to the existing Arduino language, time on task will be noted.

Before the test, the tasks will be prepared and during the test the computer screen will be recorded to measure the time taken to perform tasks which will be the main metric used to evaluate the usability test. During the test, a test moderator will present the background information and tasks necessary to complete the test, while data loggers will note observations of the test subject. When the tasks are completed, the test moderator will conduct a short debriefing. After the test, the notes from the data loggers and the time for each task will be gathered and evaluated.

9.2.3 Tasks

The test subject would need selected parts of the syntax and semantics for Trun and Arduino C language to complete the tasks. The notes on the required information to complete the task is present in **Appendix F.2**. The following is the tasks the test subject was presented:

1. Make the car drive forward for 2 seconds, then pause for 2 seconds. Repeat this 10 times using a from loop.
2. Make a declaration of integer named i and initialise it with the value 5. Then another integer declaration named j and initialise it with the value 2. Create an if statement with condition i > j. In the body of the if-statement make it turn right (turnright()).

9.2.4 Observations

The observations noted while testing the Trun language will be presented first followed by the observations for the existing Arduino C language. They will be categorised into cosmetic, serious and critical. If the test subject spends less than a few minutes to complete a task and an issue is noted, the observation will be classified as *cosmetic*. If the noted issue causes the test subject to use several minutes on a task, it will be classified as *serious*. If the usability test has to end prematurely due to an issue, then the observation will be classified as *critical*. Other than the observations, the time on task metric will also be evaluated.

Trun

The observations made for the Trun language will be presented in a table. Afterwards, a more comprehensive description of the issues will be written.

Reference nr	Description	Category
t.1	In From loop, the test subject was confused	cosmetic
t.2	The test subject assumed that the operand in predefined statements is seconds	cosmetic

Table 9.1: Observations from the test of Trun

The **t.1** was observed when the test subject created a from loop. He assumed that when running from 1 upto 10, it would run ten times. However in reality, it will run for 9 times. The test subject experienced **t.2** when using the predefined statements such as drive(), turnright() or turnleft. He assumed that the operands should be of the type seconds. However, he remarked that it would also make sense if the operand was a distance. When the test subject created an if statement, he remarked that the if construct has similarities to other programming languages such as C. Other than the tasks presented in this section, a third task has also been performed in Trun with the purpose of testing the error messages. The details is in **Appendix F.2**. The test subject remarked that the error message was good, and only in bigger programs would a more specific message be useful.

Arduino C

The observations noted for the existing Arduino C language are presented in the table below.

Reference nr	Description	Category
a.1	Structure of Arduino C was confusing	cosmetic
a.2	Arduino commands were hard to use and required an information on the hardware	cosmetic

When the test subject had to write declarations, he was confused where in the file to write the

variable declarations and statements (**a.1**), as it can be written in **loop()** or **setup()**. The test subject had to repeatedly look at the notes in order to make the Trun car drive or turn and had to think before choosing the right pin when trying to make the car turn (**a.2**).

Time on Task

The time spent by the test subject has been calculated by subtracting the time when the test subject began typing (to solve the task) from the time when the task is completed.

	Task 1	**Task 2**	**Total**
Trun	58 s	66 s	124 s
Arduino C	230 s	87 s	317 s
Percentage deviation	74,78%	24,14%	60,88%

Table 9.2: Table over time of completion for the tasks in the two languages

In **Table 9.2**, it is clear that the test subject spent a considerable amount of time on completing the tasks in the existing Arduino language compared to Trun. The deviation in **Table 9.2** is calculated with the time spent in the Arduino C as the time benchmark. The test subject used 61% more time on tasks in Arduino C compared to Trun.

9.2.5 Evaluation

The cosmetic issues mentioned in **Section 9.2.4** are minor issues which might be taken into consideration if changing the semantics is necessary. It is also clear that Trun is more efficient to write compared to the existing Arduino C language as the test subject used 61% less time in Trun than Arduino C. The abstraction of the complicated functionality in the existing Arduino C language is also helpful for beginners. This is indicated by the fact that the test subject spent $\approx$ 75% time more on Arduino C in Task 1 compared to Trun and due to the fact that the complicated functionality such as making the car drive forward or turn is prevalent in Task 1.

Part III

Evaluation

10 | Discussion

*The discussion and evaluation of this project will take place in this chapter. The requirements stated in **Section 3.1** will be evaluated first followed by the language criteria stated in **Section 2.3**. Lastly, a discussion of the major challenges faced during the creation of Trun will be made.*

10.1 Evaluating Requirements

The requirements that was stated in **Section 3.1** will be evaluated. Each requirement are shown in **Table 10.1**. Two columns have been added at the very right to indicate if the requirement has been deemed fulfilled or partially fulfilled. The column contains a check mark if the requirement has been fulfilled. A description of why the requirement was deemed fulfilled or not is included below the requirements table in the form of a short statement.

Reference no.	Requirement	Priority	Fulfilled	Partially fulfilled
1.f	The programming language Trun must be able to compile to Arduino code.	M	✓	
2.f	The language must have iterative control structures.	M	✓	
3.f	The language must have selective control structures.	M	✓	
4.f	The language must implement basic data types namely integer, float, boolean, and string.	M	✓	
5.f	The language must implement arrays as a collection of the basic data types.	M	✓	
6.f	The language must implement basic arithmetic relational, and logical operations.	M	✓	
7.f	The Trun language must follow the principles of the imperative programming paradigm.	M	✓	
8.f	Descriptive error messages must be shown to the user.	S		✓
9.n	The programming language should be easier for beginners to use compared to the Arduino language.	S	✓	
10.f	The language should implement functionality for driving the Trun car forward.	S	✓	
11.f	The language should implement functionality for turning the Trun car.	S	✓	
12.f	A subset of the language could be translated directly to AVR Assembly code.	C		

Table 10.1: Fulfilment of requirements

Deemed Fulfilled

1.f: A functioning compiler for the Trun language has been implemented. JUnit testing has been implemented and integration testing has been conducted to secure major parts of the key functionality.

2.f: Trun has three iterative structures: the *While do*-loop, the *Repeat until*-loop, and the *From*-loop (*upto* and *downto*).

3.f: Trun contains an iterative control structure: the *if-else* structure.

4.f: Data-types have been implemented in the form of integer, decimal (float), truth (boolean), and text (string).

5.f: Arrays have been added in the Trun language. Arrays can contain one of the four primitive types in the Trun language.

6.f: The basic arithmetic, relational, and logical operators (+, -, *, /, >, <, =, AND, OR) has been implemented in Trun.

7.f: Trun follows the standard imperative paradigm program structure e.g. functions are created before statements.

9.n: Many statements and primitive type's syntax has been changed to a more natural language. The success of this is reflected in the usability test, where the test subject had an easier time with Trun's syntax in contrast to the Arduino programming language.

10.f: Trun implements a statement for making the car drive forward.

11.f: Trun implements statements for making the car turn left and right.

Partially or Not Fulfilled

8.f: Multiple exceptions have been made to show the user a printed message if an error occurs regarding types, use of operators and function calls. These were described in **Section 8.4.1**. The compiler catches errors in a source program and even specifies which line and character the error occurs on. However, syntax errors are in some cases reported without addition information about where the error takes place. The requirement is therefore deemed partially fulfilled.

12.f: Due to time constraints, generating a subset of Trun to AVR assembly code has not been completed. However, it was already clear at the beginning of the project that time was limited, and therefore it was not prioritised compared to the other requirements. Hence, this requirement is not fulfilled.

To conclude the requirement evaluation all Must have and Should have requirements have been deemed fulfilled or partially fulfilled, which means the project as a whole can be seen as complete.

10.2 Evaluation of Language Criteria

In **Section 2.3** the *Language Evaluation Criteria* was introduced, clarifying the importance of which considerations to make when creating a language. In this section, we'll discuss how the different criteria have been met in Trun.

Sebesta highlights characteristics that each affect the overarching criteria, which will be used in the discussion to evaluate Trun[11, Section 1.3.1]. The conclusion drawn in **Section 2.3.4** is that readability and writability is important for Trun, with less priority on reliability.

The characteristics that define readability and writability are *simplicity*, *orthogonality*, *data types* and *syntax design*. To create simplicity within the language, operator overloading has been avoided to the largest extend possible. The only existing operator that has overloading is + and - between integers and decimal, however this is deemed necessary to sustain a natural language and it would be confusing for the target group otherwise.

Trun does not have high orthogonality as the ability to describe higher level of abstract structures is not possible in the current version of the language. Having the ability to create multi dimensional arrays or have arrays containing functions would be a way of increasing orthogonality. However, this was not deemed suited for the target group and was therefore not implemented.

Data type names should be a clear indication of which type of data is stored in a variable. Therefore, a truth data type was included in the language instead of using an integer with values 0 (false) or any integer other than 0 (true).

To accommodate the criteria of reliability, type checking has been an area of focus as run-time error handling has not been implemented. Therefore, it is important to catch as many errors as possible at compile time.

10.3 Challenges

This section discusses the challenges met throughout the development of Trun and the changes they entailed.

10.3.1 Big-step vs Small-step semantic

The Trun language has been decided to be an imperative non-parallel language at an early stage in the project. It is sufficient to make a big-step semantic for the language to define all the semantic actions. Despite that, a small-step semantic has been used. It would be easier to implement the code generation when implemented by the rules of a small-step semantic and also give a full picture of the semantic rules of the language.

Structure of the language

Making a small-step semantic had some obstacles. As we couldn't find a considerate approach to describe the allowance of declarations being spread out in the code semantically, we had to limit the language. A choice had to be made between having the semantic rules fully describe the language or implement the intended program structure but with lacking semantic rules.

Having the semantic rules correspond fully to the language was deemed most important, which meant a specific program structure had to be implemented. This structure forced each declarations to be at the very top of the program in the specific order of variables first, arrays second, and functions third. The initial intention of allowing the user to make declarations wherever in the program was thereby unreachable.

The benefit of the small step semantic has not been revealed as the generated code is for a relatively high level language, C. In a low level language like AVR assembly, the benefit of small step semantic would be more apparent as the semantics would cover every outcome an implementation would need to accommodate for.

Taking all these things into consideration it can be discussed whether or not it was beneficial to spend a significant amount of resources to define a small-step semantic over a less demanding big-step semantic. We however believe that a small-step semantic was the right choice as it leaves minimal doubt to the reader as to what the capabilities of the Trun-language are.

10.3.2 Implementation of arrays

As Trun is meant as a beginner language, we wanted to implement dynamically sized arrays as to not complicate an already challenging concept. However, as our target code is C this is

not achievable, *per se*. To circumvent the need for memory allocation when declaring arrays in Trun, any array is generated as an array of size 128 (of the declared type) when generating C code.

10.4 Symbol table

For the symbol table implementation we decided to go with the version of having a single large table containing all information. This means that whenever a new scope is created and furthermore closed, everything within the scope has to be deleted.

Since we perform semantic checks in the same visitor as the symbol table entries are made, there will be no doubt as to where to find the correct symbol table containing the needed information. The reason additional semantics and type checking are performed in the same visitor is out of convenience as we're already accessing the variables and values within the AST. An example is the check about uninitialised variables, when type checking an expression we're visiting the identifiers and checking their type and whether they're initialised is checked through the symbol table.

10.5 Data abstraction

Trun only supports a single data abstraction in form of the arrays. This is a limiting factor in terms of possibilities. However, we feel Trun manages to succeed in introducing the target group to data abstraction. It is not overly complicated as the language is not very orthogonal. The simplicity of Trun's syntax makes it apparent how arrays can be utilised.

11 | Conclusion

We have proposed the Trun language to support the current development public schools have regarding technology related classes, and as a solution to the problem statement, defined in **Chapter 3**, which is as follows:

A new programming language should help the target group learn programming by providing abstraction for challenging elements of the Arduino programming language, and utilising visual representation in the form of the Trun car.

In the process of creating the Trun language, a set of requirements has been established to create a sufficient solution for the problem statement. Afterwards, the syntax has been described with a CFG. The visual part of the language is defined to fit the target group with inspiration from the existing languages, analysed in **Chapter 2**. After defining the syntax, the meaning of the language was specified in **Chapter 6** through structural operational semantics and in **Chapter 7** through a type system.

The implementation of the compiler for Trun language is described in **Chapter 8** with a reflective visitor pattern being the foundation for the later phases. Visitors have been implemented to reflect the semantic and type system rules.

To ensure correctness of the implementation's structural parts, a combination of integration and unit tests have been performed. To evaluate if the Trun language is easier to learn than Arduino C, a usability test has been conducted. All aforementioned tests are described in **Chapter 9**.

In the discussion **Chapter 10**, the requirements have been evaluated, where the greater part is deemed fulfilled. The error messaging is partially fulfilled, and no subset of the language is translated directly to AVR Assembly code making the last requirement not fulfilled. Despite this, the fulfilled Must-have requirements are sufficient to deem the project fulfilled. The Trun language provides an easy-to-understand syntax in addition to a visual representation in form of the Trun car to encourage the learning process.

12 Future Works

The final chapter is dedicated to features that were not implemented in the project due to time restrictions or design choices.

12.1 Extension of functionality of the Trun language

The purpose of the language is to teach the target group programming with visualisation of the Trun car. The Trun language has fulfilled the requirements of having different control structures and statements for interacting with the Trun car. Additions and improvements can always be made and the section will cover some of them. The additions could potentially give the users approaches to create more creative solutions when programming.

12.1.1 Extensions for the Trun car

The goal of Trun was never to give the user full access to every detail of the Arduino hardware, as many aspects were abstracted to keep the language beginner friendly.

The Trun language only has four specific commands to interact with the Trun car. Future versions of the language could include more statements for controlling the car. The car could also be extended with for example a display to have more options to output. This would also give text in the language more relevance. This would let users utilise the full potential of the Arduino board instead of only two pins used in the predefined statements implemented in Trun.

12.1.2 Improvement of the program structure

As mentioned in the discussion, we had to limit ourselves to a certain program structure that only allowed declarations in the very top of the program in the specific order of variables first,

arrays second, and functions third.

This was done in order to make the language correspond to the semantic rules. Future works therefore include changes to the small-step semantic that allows for declarations of variables, arrays, and functions in a non-specified order.

12.1.3 Improvement of arrays

We would like to calculate the exact size of arrays used in the program. To do so we would have to do a loop bounds analysis measuring how large the arrays could potentially be, another solution would be to not allow indefinite loops. Hereby, arrays would become truly dynamic and potentially save memory.

12.2 Compiling to AVR assembly

Undergoing the task of translating directly to the Arduino requires extensive knowledge of the AVR instruction set [38]. Furthermore, knowledge about the physical mapping of pins on the chip, how they are connected to the pins on the Arduino, and the interfaces provided. **Figure 12.1** shows the mapping between the ATmega chip pins and the physical pins of the Arduino. The physical Arduino pins 12 and 13, which are used for the Trun car, are mapped to pin 18 and 19 on the ATmega called PB5 and PB4.

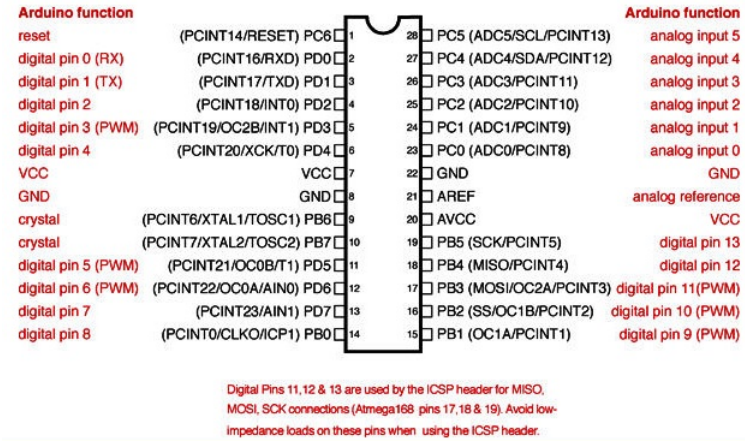

Figure 12.1: Mapping between ATmega328 pins and Arduino physical pins [39]

The ATmega328 has three ports, namely B, C, and D. Each digital I/O port has three dedicated registers to manage the ports. Each port is an 8-bit register where each bit controls a specific

pin (see **Figure 12.2**). The first register is the Data Direction Register (DDRx) where x is the port e.g. B, C, or D. This register is used for managing if a pin is used as input or output [40, p. 84].

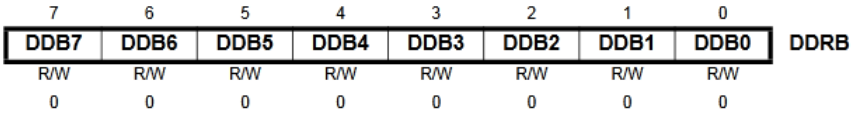

Figure 12.2: DDRB register. Numbers on top specifies the pin number, numbers on the bottom specifies a binary value of that pin [40, p. 100].

The second register is the Port Input Pins register (PINx). This register is a read only register that is used to read the pin states.

The third register is the Data Register (PORTx). This register is used to specify if a pin should output high or low voltage. By writing a value to this register it is possible to change the state of the pins if the pin is specified as output in the DDRx register.

12.2.1 Code for drive statement

The assembly code for the drive statement could look something like the code on **Figure 12.3**.

```
.include "m328def.inc"
        ldi     r16, 0x30     ; Load 0b00110000 in r16
        out     ddrb, r16     ; Set pin 4 and 5 as output
        out     pinb, r16     ; Set pin 4 and 5 HIGH
```

Figure 12.3: Code example in assembly for drive statement

Line 1 includes the file m328def, which contains definitions for register aliases. Line 2 loads an immediate value 0x30 to register 16. Line 3 writes to the DDRB register to set pin 4 and 5 as output pins. Finally, on line 4 the PINB register is written to, setting the output of pin 4 and 5 to HIGH.

12.3 Integrated Development Environment

Developing an Integrated Development Environment (IDE) could be beneficial for users of Trun when programming. The IDE should provide the user with an interactive graphical user interface including a text editor.

The text editor should be able to identify code mistakes and various typos and thereby give the user real-time feedback. Minimising these errors would prevent many potential compile-time errors. The text editor should also include a colour scheme to highlight the keywords used in Trun to make it easier to get an overview of the code when programming.

The IDE should also provide the necessary tools in order to test the code by integrating an interpreter. This would detect run-time errors. Furthermore, it could make it possible for the user to transfer the code to the Arduino without having to go through the terminal.

12.4 Compiler Optimisation

A valuable improvement for the compiler would be to optimise for efficiency within it. While efficiency was not a priority for the project, improvements could still be made. The optimisation that would be easiest to implement in the current iteration of the compiler would be to reduce dead-weight of code. If there is code after a return statement it should be ignored or at the very least mentioned to the user.

Another example would be to analyse whether it is possible to reach certain steps in an if - else if chain, determining it if is even reachable. If the code body is not reachable it should be skipped for the sake of efficiency.

Bibliography

[1] Steen A. Jørgenssen (Jyllands-posten). *Undervisningsministeren vil lære alle børn at kode*. URL: https://jyllands-posten.dk/politik/ECE10086478/undervisningsministeren-vil-laere-alle-boern-at-kode/.

[2] Steen A. Jørgenssen (Jyllands-posten). *Undervisningsministeren vil gøre undervisning i teknologiforståelse obligatorisk i folkeskolen*. URL: https://jyllands-posten.dk/politik/ECE10243855/undervisningsministeren-vil-goere-undervisning-i-teknologiforstaaelse-obligatorisk-i-folkeskolen/.

[3] Undervisnings Ministeriet. *Teknologiforståelse*. URL: https://arkiv.emu.dk/modul/teknologiforst%C3%A5else.

[4] Webopedia. *Definition of Language*. URL: https://www.webopedia.com/TERM/P/programming_language.html.

[5] Jordan Hudgens. *developer learning curve*. URL: https://www.crondose.com/2016/09/developer-learning-curve/.

[6] Anabela Gomes A. J. Mendes. *Learning to program - difficulties and solutions*. URL: http://icee2007.dei.uc.pt/proceedings/papers/411.pdf.

[7] Arduino. *Arduino Introduction*. URL: https://www.arduino.cc/en/guide/introduction.

[8] nongnu. *AVR LIBC FAQ*. URL: http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_cplusplus.

[9] Arduino. *Arduino FAQ*. URL: https://www.arduino.cc/en/main/FAQ.

[10] Arduino. *Arduino Uno Rev3*. URL: https://store.arduino.cc/arduino-uno-rev3#.UxNpBk2YZuG.

[11] Robert W. Sebesta. *Concepts of Programming Languages*. PEARSON, 2016.

[12] Vangie Beal. *C*. URL: https://www.webopedia.com/TERM/C/C.html.

[13] Vangie Beal. *C++*. URL: https://www.webopedia.com/TERM/C/C_plus_plus.html.

[14] Olympia Circuits. *The SETUP() And LOOP() Blocks*. URL: http://learn.olympiacircuits.com/setup-and-loop-blocks.html.

[15] Dave Kuhlman. *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. URL: https://web.archive.org/web/20120623165941/http://cutter.rexx.com/~dkuhlman/python_book_01.html#introduction-python-101-beginning-python.

[16] Python Software Foundation. *General Python FAQ*. URL: https://docs.python.org/3/faq/general.html#general-python-faq.

[17] Aayushi Johari. *What Is Java? A Beginner's Guide to Java and Its Evolution*. URL: https://www.edureka.co/blog/what-is-java/?utm_source=quora&utm_medium=crosspost&utm_campaign=social-media-edureka-aug-aj.

[18] Marco Cantù. *Marco Cantù's Essential Pascal*. URL: http://www.marcocantu.com/epascal/English/ch01hist.htm.

[19] Tutorialspoint. *Pascal tutorialspoint*. URL: https://www.tutorialspoint.com/pascal/pascal_repeat_until_loop.htm.

[20] WikiBooks. *Pascal Programming/Syntax and functions*. URL: https://en.wikibooks.org/wiki/Pascal_Programming/Syntax_and_functions.

[21] Tech Crunch. *Tech Crunch*. URL: https://www.lifewire.com/apples-swift-playgrounds-help-kids-code-4053170.

[22] Kurt Nørmark. *Overview of the four main programming paradigms*. URL: http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html.

[23] Rambabu Posa. *Compare Functional Programming, Imperative Programming and Object Oriented Programming*. URL: https://www.journaldev.com/8693/functional-imperative-object-oriented-programming-comparison.

[24] Magnus Madsen. *Logic Programming*. Mar. 18, 2019. URL: http://cs.au.dk/~magnusm/courses/ilp/slides/Lesson%5C%201.pdf.

[25] David Benyon. *Designing Interactive Systems - A comprehensive guide to HCI, UX and interaction design*. PEARSON, 2014.

[26] Richard J. LeBlanc Charles N. Fischer Ron K. Cytron. *Crafting A Compiler*. Addison Wesley, 2009.

[27] Hans Hüttel. *Pilen ved træets rod*. BOOKS on DEMAND, 2019.

[28] ANTLR / Terence Parr. *ANTLR Website / Index*. URL: https://www.antlr.org/index.html.

[29] ANTLR / Terence Parr. *ANTLR Website / About*. URL: https://www.antlr.org/about.html.

[30] Terence Parr. *Runtime Libraries and Code Generation Targets*. URL: https://github.com/antlr/antlr4/blob/master/doc/targets.md.

[31] ANTLR / Terence Parr. *ANTLR Website / Development Tools*. URL: https://www.antlr.org/tools.html.

[32] Brett Crawley. *Parser Generators: ANTLR vs JavaCC*. URL: https://dzone.com/articles/antlr-and-javacc-parser-generators.

[33] JavaCC. *Features*. URL: https://javacc.org/features.

[34] SableCC. *SableCC Features*. URL: http://sablecc.sourceforge.net/features.html.

[35] Nat Pryce. *Concrete to Abstract Syntax Transformations with SableCC*. URL: http://www.natpryce.com/articles/000531.html.

[36] STF. *Integration Testing*. URL: http://softwaretestingfundamentals.com/integration-testing/.

[37] David Benyon. *Designing Interactive Systems*. 3. edition. Pearson, 2018, p. 140.

[38] Atmel. *Atmel AVR 8-bit Instruction Set*. URL: https://cdn.instructables.com/ORIG/FIH/E7GU/I301K81S/FIHE7GUI301K81S.pdf.

[39] Nearbus. *Atmega 328 Pinout*. URL: http://nearbus.net/wiki/index.php?title=Atmega_328_Pinout.

[40] Microchip. *megaAVR® Data Sheet*. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061A.pdf.

Part IV

Appendices

A Table of Keywords

Data types	Operators	Control structures	Functions
integer	NOT	if	function
decimal	OR	then	returns
text	AND	else	return
truth	is	while	drive
array	append	do	pause
		repeat	turnright
		until	turnleft
		from	
		upto	
		downto	

B | CFG

```
parser grammar Trun;

options { tokenVocab=TrunLexer; }

start
        : dclblock arrdclblock functiondclblock stmtstartblock;

dclblock:
        (dcl (EOL dclblock)*) EOF?
        | EOF?;

arrdclblock:
        (arrdcl (EOL arrdclblock)*) EOF?
        | EOF?;
functiondclblock:

        (functiondcl (EOL functiondclblock)*) EOF?
        |EOF?;

stmtstartblock:
        (stmt (EOL stmtstartblock)*) EOF?
        | EOF?;
stmt
        : ifstmt
        | whilestmt
        | returnstmt
        | functioncall
        | repeatuntilstmt
        | fromstmt
        | assignment
        | arradd
```

```
        | arrindex
        | drive
        | turnleft
        | turnright
        | pause
        ;

functiondcl
        : FUNCTION ID RETURNS type LPAR (param (COMMA param)*)? RPAR stmtblock
        | FUNCTION ID LPAR (param (COMMA param)*)? RPAR stmtblock ;


dcl
        : INTDCL ID dclValue?
        |  FLOATDCL ID dclValue?
        |  TEXTDCL ID dclValue?
        |  TRUTHDCL ID (ASSIGN truthexpr)?
        ;

arrdcl :
        type ARRDCL  ID
       ;

dclValue
        :( ASSIGN value
        |  ASSIGN TEXT
        |  ASSIGN expr);

param
        : INTDCL ID
        | FLOATDCL ID
        | TEXTDCL ID
        | TRUTHDCL ID ;

functioncall
        : ID LPAR (arg (COMMA arg)*)? RPAR;

ifstmt
        : IF truthpar THEN stmtblock EOL*
          (ELSE IF truthpar THEN stmtblock EOL*)*
          (ELSE THEN stmtblock)? ;

whilestmt
        : WHILE truthpar DO stmtblock ;
```

```
repeatuntilstmt
        : REPEAT stmtblock UNTIL truthpar ;

fromstmt
        : FROM LPAR valueorfunctioncallortext  (UPTO | DOWNTO) valueorfunctioncallortext RPAR stmtbloc

returnstmt
        : RETURN (valueorfunctioncallortext | truthexpr) EOL* ;

assignment
        : ID ( ASSIGN valueorfunctioncallortext
        |  ASSIGN TEXT
        |  ASSIGN expr );
value
        : arithmexpr
        | arrindex
        | ID ;

valueorfunctioncallortext
        : value
        | functioncall
        | TEXT;

expr
        : arithmexpr
        | truthexpr
        | append;

arithmexpr
         : multexpr ((PLUS | MINUS ) multexpr)* ;
multexpr
         : unaryminus ((TIMES | DIVIDES) unaryminus)* ;
unaryminus
        :  (MINUS)? parexpr;
parexpr
         : nums
         | functioncall
         | arrindex
         | LPAR arithmexpr RPAR ;

truthexpr
        : logicalexpr;
```

```
logicalexpr
        :  NOT? relationalexpr ((OR| AND) NOT? relationalexpr)*
        ;

relationalexpr
        :   valueorfunctioncallortext ((EQUALS  | GRTHAN | LESSTHAN)  valueorfunctioncallortext)
        |   LPAR logicalexpr RPAR
        |   truth
        |   functioncall
        ;
append
        : textorid APPEND textorid ;
textorid
        : arrindex | TEXT | ID;

arrindex
        :  ID ELEMENT arithmexpr;

arradd
        : ID ELEMENT arithmexpr ASSIGN (expr | TEXT);

drive   : DRIVE LPAR valueorfunctioncallortext RPAR;

turnleft   : TURNLEFT LPAR valueorfunctioncallortext RPAR;

turnright   : TURNRIGHT LPAR valueorfunctioncallortext RPAR;

pause    : PAUSE LPAR valueorfunctioncallortext RPAR;

nums
        : INUM
        | FNUM
        | ID;

stmtblock
        : LCB EOL* dclblock stmtstartblock RCB ;

truthpar
        : LPAR truthexpr RPAR ;

truth
        : TRUTHVAL
        | ID ;
```

```
type
        : INTDCL
        | FLOATDCL
        | TRUTHDCL
        | TEXTDCL ;

arg
        : nums
        | expr
        | TEXT
        | TRUTHVAL ;

lexer grammar TrunLexer;

INTDCL   : 'integer' ;
FLOATDCL : 'decimal' ;
TRUTHDCL : 'truth' ;
TEXTDCL  : 'text' ;
ARRDCL   : 'array' ;
LPAR     : '(' ;
RPAR     : ')' ;
TIMES    : '*' ;
DIVIDES  : '/' ;
PLUS     : '+' ;
MINUS    : '-' ;
AND      : 'AND' ;
OR       : 'OR' ;
NOT      : 'NOT' ;
TRUTHVAL  : 'true' | 'false' ;
EQUALS   : '=' ;
GRTHAN   : '>' ;
LESSTHAN : '<' ;
IF       : 'if' ;
ELSE     : 'else' ;
WHILE    : 'while' ;
DO       : 'do' ;
FROM     : 'from' ;
REPEAT   : 'repeat' ;
UNTIL    : 'until' ;
THEN     : 'then' ;
UPTO     : 'upto' ;
DOWNTO   : 'downto' ;
COMMA    : ',' ;
RETURN   : 'return' ;
```

```
RETURNS  : 'returns' ;
FUNCTION : 'function' ;
ELEMENT  : 'element' ;
ASSIGN   : 'is' ;
LCB      : '{' ;
RCB      : '}' ;
DRIVE    : 'drive';
TURNLEFT : 'turnleft';
TURNRIGHT: 'turnright';
PAUSE    : 'pause';
APPEND   : 'append';
EOL      : NEWLINE+ ;
FNUM     : ([0-9])+ '.' ([0-9])+ ;
INUM     : ([0-9])+ ;
ID       : (([_A-Za-z])+([_0-9A-Za-z])*) ;
TEXT
   : '"' ~ ["\r\n]* '"'
   | '\'' ~ ['\r\n]* '\'' ;

WS : [ \t\u000C]+ -> skip ;

COMMENT       : '/#' .*? '#/' -> skip ;
LINE_COMMENT  : '#' .*? NEWLINE -> skip ;

fragment UNICODE  :  '\u0021'..'\u007E' ;
fragment NEWLINE   : '\r' '\n' | '\n' | '\r' ;
```

C | Formal Semantics

[VAR] $\langle x,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl,\ sto\rangle$ where $env_V\, x = l$ and $sto\ l = v$ and $envl = (env_V,\ env_F,\ env_A) : envl'$

[PAR-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle (e),\ envl,\ sto\rangle \Rightarrow \langle (e'),\ envl',\ sto'\rangle}$$

[PAR-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle (e),\ envl,\ sto\rangle \Rightarrow \langle (v),\ envl',\ sto'\rangle}$$

[PAR-3] $\langle (v),\ envl,\ sto\rangle \Rightarrow \langle v,\ envl,\ sto\rangle$

[PLUS-1] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle e_1',\ envl',\ sto'\rangle}{\langle e_1\ +\ e_2,\ envl,\ sto\rangle \Rightarrow \langle e_1'\ +\ e_2,\ envl',\ sto'\rangle}$$

[PLUS-2] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle v_1,\ envl',\ sto'\rangle}{\langle e_1\ +\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ +\ e_2,\ envl',\ sto'\rangle}$$

[PLUS-3] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle e_2',\ envl',\ sto'\rangle}{\langle v_1\ +\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ +\ e_2',\ envl',\ sto'\rangle}$$

[PLUS-4] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle v_2,\ envl',\ sto'\rangle}{\langle v_1\ +\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ +\ v_2,\ envl',\ sto'\rangle}$$

[PLUS-5] $\langle v_1\ +\ v_2,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl,\ sto\rangle$ where $v = v_1 + v_2$

[APP-1] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle e_1',\ envl',\ sto'\rangle}{\langle e_1\ append\ e_2,\ envl,\ sto\rangle \Rightarrow \langle e_1'\ append\ e_2,\ envl',\ sto'\rangle}$$

[APP-2] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle v_1,\ envl',\ sto'\rangle}{\langle e_1\ append\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ append\ e_2,\ envl',\ sto'\rangle}$$

[APP-3] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle e_2',\ envl',\ sto'\rangle}{\langle v_1\ append\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ append\ e_2',\ envl',\ sto'\rangle}$$

[APP-4] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle v_2,\ envl',\ sto'\rangle}{\langle v_1\ append\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ append\ v_2,\ envl',\ sto'\rangle}$$

[APP-5] $\langle v_1\ append\ v_2,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl,\ sto\rangle$ where $v = v_1 v_2$

[MINUS-1] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle e_1',\ envl',\ sto'\rangle}{\langle e_1\ -\ e_2,\ envl,\ sto\rangle \Rightarrow \langle e_1'\ -\ e_2,\ envl',\ sto'\rangle}$$

[MINUS-2] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle v_1,\ envl',\ sto'\rangle}{\langle e_1\ -\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ -\ e_2,\ envl',\ sto'\rangle}$$

[MINUS-3] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle e_2',\ envl',\ sto'\rangle}{\langle v_1\ -\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ -\ e_2',\ envl',\ sto'\rangle}$$

[MINUS-4] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle v_2,\ envl',\ sto'\rangle}{\langle v_1\ -\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ -\ v_2,\ envl',\ sto'\rangle}$$

[MINUS-5] $\langle v_1\ -\ v_2,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl,\ sto\rangle$ where $v = v_1 - v_2$

[U-MINUS-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle -e,\ envl,\ sto\rangle \Rightarrow \langle -e',\ envl',\ sto'\rangle}$$

[U-MINUS-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle -e,\ envl,\ sto\rangle \Rightarrow \langle -v,\ envl',\ sto'\rangle}$$

[U-MINUS-3] $\langle -v,\ envl,\ sto\rangle \Rightarrow \langle -v,\ envl,\ sto\rangle$

[MULT-1] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle e_1',\ envl',\ sto'\rangle}{\langle e_1\ *\ e_2,\ envl,\ sto\rangle \Rightarrow \langle e_1'\ *\ e_2,\ envl',\ sto'\rangle}$$

[MULT-2] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle v_1,\ envl',\ sto'\rangle}{\langle e_1\ *\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ *\ e_2,\ envl',\ sto'\rangle}$$

[MULT-3] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle e_2',\ envl',\ sto'\rangle}{\langle v_1\ *\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ *\ e_2',\ envl',\ sto'\rangle}$$

[MULT-4] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle v_2,\ envl',\ sto'\rangle}{\langle v_1\ *\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ *\ v_2,\ envl',\ sto'\rangle}$$

[MULT-5] $\langle v_1 * v_2, envl, sto\rangle \Rightarrow \langle v, envl, sto\rangle$ where $v = v_1 * v_2$

[DIV-1] $$\frac{\langle e_1, envl, sto\rangle \Rightarrow \langle e_1', envl', sto'\rangle}{\langle e_1 \ / \ e_2, envl, sto\rangle \Rightarrow \langle e_1' \ / \ e_2, envl', sto'\rangle}$$

[DIV-2] $$\frac{\langle e_1, envl, sto\rangle \Rightarrow \langle v_1, envl', sto'\rangle}{\langle e_1 \ / \ e_2, envl, sto\rangle \Rightarrow \langle v_1 \ / \ e_2, envl', sto'\rangle}$$

[DIV-3] $$\frac{\langle e_2, envl, sto\rangle \Rightarrow \langle e_2', envl', sto'\rangle}{\langle v_1 \ / \ e_2, envl, sto\rangle \Rightarrow \langle v_1 \ / \ e_2', envl', sto'\rangle}$$

[DIV-4] $$\frac{\langle e_2, envl, sto\rangle \Rightarrow \langle v_2, envl', sto'\rangle}{\langle v_1 \ / \ e_2, envl, sto\rangle \Rightarrow \langle v_1 \ / \ v_2, envl', sto'\rangle}$$

[DIV-5] $\langle v_1 \ / \ v_2, envl, sto\rangle \Rightarrow \langle v, envl, sto\rangle$ where $v = v_1 \ / \ v_2$ and $v_2 \neq 0$

[EQUALS-1] $$\frac{\langle e_1, envl, sto\rangle \Rightarrow \langle e_1', envl', sto'\rangle}{\langle e_1 \ = \ e_2, envl, sto\rangle \Rightarrow \langle e_1' \ = \ e_2, envl', sto'\rangle}$$

[EQUALS-2] $$\frac{\langle e_1, envl, sto\rangle \Rightarrow \langle v_1, envl', sto'\rangle}{\langle e_1 \ = \ e_2, envl, sto\rangle \Rightarrow \langle v_1 \ = \ e_2, envl', sto'\rangle}$$

[EQUALS-3] $$\frac{\langle e_2, envl, sto\rangle \Rightarrow \langle e_2', envl', sto'\rangle}{\langle v_1 \ = \ e_2, envl, sto\rangle \Rightarrow \langle v_1 \ = \ e_2', envl', sto'\rangle}$$

[EQUALS-4] $$\frac{\langle e_2, envl, sto\rangle \Rightarrow \langle v_2, envl', sto'\rangle}{\langle v_1 \ = \ e_2, envl, sto\rangle \Rightarrow \langle v_1 \ = \ v_2, envl', sto'\rangle}$$

[EQUALS-TRUE] $\langle v_1 \ = \ v_2, envl, sto\rangle \Rightarrow \langle true, envl, sto\rangle$ where $v_1 = v_2$

[EQUALS-FALSE] $\langle v_1 \ = \ v_2, envl, sto\rangle \Rightarrow \langle false, envl, sto\rangle$ where $v_1 \neq v_2$

[GRT-THAN-1] $$\frac{\langle e_1, envl, sto\rangle \Rightarrow \langle e_1', envl', sto'\rangle}{\langle e_1 \ > \ e_2, envl, sto\rangle \Rightarrow \langle e_1' \ > \ e_2, envl', sto'\rangle}$$

[GRT-THAN-2] $$\frac{\langle e_1, envl, sto\rangle \Rightarrow \langle v_1, envl', sto'\rangle}{\langle e_1 \ > \ e_2, envl, sto\rangle \Rightarrow \langle v_1 \ > \ e_2, envl', sto'\rangle}$$

[GRT-THAN-3] $$\frac{\langle e_2, envl, sto\rangle \Rightarrow \langle e_2', envl', sto'\rangle}{\langle v_1 \ > \ e_2, envl, sto\rangle \Rightarrow \langle v_1 \ > \ e_2', envl', sto'\rangle}$$

[GRT-THAN-4] $$\frac{\langle e_2, envl, sto\rangle \Rightarrow \langle v_2, envl', sto'\rangle}{\langle v_1 \ > \ e_2, envl, sto\rangle \Rightarrow \langle v_1 \ > \ v_2, envl', sto'\rangle}$$

[GRT-THAN-TRUE] $\langle v_1 \ > \ v_2,\ envl,\ sto\rangle \Rightarrow \langle true,\ envl,\ sto\rangle$ where $v_1 > v_2$

[GRT-THAN-FALSE] $\langle v_1 \ > \ v_2,\ envl,\ sto\rangle \Rightarrow \langle false,\ envl,\ sto\rangle$ where $v_1 \not> v_2$

[LESS-THAN-1] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle e_1',\ envl',\ sto'\rangle}{\langle e_1 \ < \ e_2,\ envl,\ sto\rangle \Rightarrow \langle e_1' \ < \ e_2,\ envl',\ sto'\rangle}$$

[LESS-THAN-2] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle v_1,\ envl',\ sto'\rangle}{\langle e_1 \ < \ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1 \ < \ e_2,\ envl',\ sto'\rangle}$$

[LESS-THAN-3] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle e_2',\ envl',\ sto'\rangle}{\langle v_1 \ < \ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1 \ < \ e_2',\ envl',\ sto'\rangle}$$

[LESS-THAN-4] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle v_2,\ envl',\ sto'\rangle}{\langle v_1 \ < \ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1 \ < \ v_2,\ envl',\ sto'\rangle}$$

[LESS-THAN-TRUE] $\langle v_1 \ < \ v_2,\ envl,\ sto\rangle \Rightarrow \langle true,\ envl,\ sto\rangle$ where $v_1 < v_2$

[LESS-THAN-FALSE] $\langle v_1 \ < \ v_2,\ envl,\ sto\rangle \Rightarrow \langle false,\ envl,\ sto\rangle$ where $v_1 \not< v_2$

[NOT-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle NOT\ e,\ envl,\ sto\rangle \Rightarrow \langle NOT\ e',\ envl',\ sto'\rangle}$$

[NOT-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle NOT\ e,\ envl,\ sto\rangle \Rightarrow \langle NOT\ v,\ envl',\ sto'\rangle}$$

[NOT-TRUE] $\langle NOT\ v,\ envl,\ sto\rangle \Rightarrow \langle true,\ envl,\ sto\rangle$ where $v = false$

[NOT-FALSE] $\langle NOT\ v,\ envl,\ sto\rangle \Rightarrow \langle false,\ envl,\ sto\rangle$ where $v = true$

[AND-1] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle e_1',\ envl',\ sto'\rangle}{\langle e_1\ AND\ e_2,\ envl,\ sto\rangle \Rightarrow \langle e_1'\ AND\ e_2,\ envl',\ sto'\rangle}$$

[AND-2] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle v_1,\ envl',\ sto'\rangle}{\langle e_1\ AND\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ AND\ e_2,\ envl',\ sto'\rangle}$$

[AND-3] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle e_2',\ envl',\ sto'\rangle}{\langle v_1\ AND\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ AND\ e_2',\ envl',\ sto'\rangle}$$

[AND-4] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle v_2,\ envl',\ sto'\rangle}{\langle v_1\ AND\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ AND\ v_2,\ envl',\ sto'\rangle}$$

[AND-TRUE] $\langle v_1\ AND\ v_2,\ envl,\ sto\rangle \Rightarrow \langle true,\ envl,\ sto\rangle$ where $v_1 \wedge v_2$

[AND-FALSE] $\langle v_1\ AND\ v_2,\ envl,\ sto\rangle \Rightarrow \langle false,\ envl,\ sto\rangle$ where $\neg(v_1 \wedge v_2)$

[OR-1] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle e_1',\ envl',\ sto'\rangle}{\langle e_1\ OR\ e_2,\ envl,\ sto\rangle \Rightarrow \langle e_1'\ OR\ e_2,\ envl',\ sto'\rangle}$$

[OR-2] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle v_1,\ envl',\ sto'\rangle}{\langle e_1\ OR\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ OR\ e_2,\ envl',\ sto'\rangle}$$

[OR-3] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle e_2',\ envl',\ sto'\rangle}{\langle v_1\ OR\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ OR\ e_2',\ envl',\ sto'\rangle}$$

[OR-4] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle v_2,\ envl',\ sto'\rangle}{\langle v_1\ OR\ e_2,\ envl,\ sto\rangle \Rightarrow \langle v_1\ OR\ v_2,\ envl',\ sto'\rangle}$$

[OR-TRUE] $\langle v_1\ OR\ v_2,\ envl,\ sto\rangle \Rightarrow \langle true,\ envl,\ sto\rangle$ where $v_1 \vee v_2$

[OR-FALSE] $\langle v_1\ OR\ v_2,\ envl,\ sto\rangle \Rightarrow \langle false,\ envl,\ sto\rangle$ where $\neg(v_1 \vee v_2)$

[ARR-IND-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle a\ element\ e,\ envl,\ sto\rangle \Rightarrow \langle a\ element\ e',\ envl',\ sto'\rangle}$$

[ARR-IND-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle a\ element\ e,\ envl,\ sto\rangle \Rightarrow \langle a\ element\ v,\ envl',\ sto'\rangle}$$

where $envl = (env_V',\ env_F,\ env_A')\ :\ envl'$
and $env_V\ e\ =\ l$
and $v\ =\ sto\ l$

[ARR-IND-3] $\langle a\ element\ v_1,\ envl,\ sto\rangle \Rightarrow \langle v_2,\ envl,\ sto\rangle$

where $envl = (env_V,\ env_F,\ env_A)\ :\ envl$
and $env_A\ a\ =\ l$
and $l'\ =\ following^{v_1}\ l$
and $v_2\ =\ sto\ l'$
and $1\ \leq\ v_1\ \leq\ k$

[ASS-VAR-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle x\ is\ e,\ envl,\ sto\rangle \Rightarrow \langle x\ is\ e',\ envl',\ sto'\rangle}$$

[ASS-VAR-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle x\ is\ e,\ envl,\ sto\rangle \Rightarrow \langle x\ is\ v,\ envl',\ sto'\rangle}$$

[ASS-VAR-3] $\langle x \text{ is } v,\ envl,\ sto\rangle \Rightarrow \langle envl,\ sto[l \mapsto v]\rangle$

where $envl = (env_V,\ env_F,\ env_A)\ :\ envl'$
and $env_V\ x\ = l$

[ASS-ARR-1] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle e_2',\ envl',\ sto'\rangle}{\langle a \text{ element } e_1 \text{ is } e_2,\ envl,\ sto\rangle \Rightarrow \langle a \text{ element } e_1 \text{ is } e_2',\ envl',\ sto'\rangle}$$

[ASS-ARR-2] $$\frac{\langle e_2,\ envl,\ sto\rangle \Rightarrow \langle v_2,\ envl',\ sto'\rangle}{\langle a \text{ element } e_1 \text{ is } e_2,\ envl,\ sto\rangle \Rightarrow \langle a \text{ element } e_1 \text{ is } v_2,\ envl',\ sto'\rangle}$$

[ASS-ARR-3] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle e_1',\ envl',\ sto'\rangle}{\langle a \text{ element } e_1 \text{ is } v_2,\ envl,\ sto\rangle \Rightarrow \langle a \text{ element } e_1' \text{ is } v_2,\ envl',\ sto'\rangle}$$

[ASS-ARR-4] $$\frac{\langle e_1,\ envl,\ sto\rangle \Rightarrow \langle v_1,\ envl',\ sto'\rangle}{\langle a \text{ element } e_1 \text{ is } v_2,\ envl,\ sto\rangle \Rightarrow \langle a \text{ element } v_1 \text{ is } v_2,\ envl',\ sto'\rangle}$$

[ASS-ARR-5] $\langle a \text{ element } v_1 \text{ is } v_2,\ envl,\ sto\rangle \Rightarrow \langle envl,\ sto[l' \mapsto v_2]\rangle$

where $envl = (env_V,\ env_F,\ env_A)\ :\ envl$
and $env_A\ a\ =\ l$
and $l'\ =\ following^{v_1}\ l$
and $1\ \leq\ v_1\ \leq\ k$

[COMP-1] $$\frac{\langle S_1,\ envl,\ sto\rangle \Rightarrow \langle S_1',\ envl',\ sto'\rangle}{\langle S_1\ S_2,\ envl,\ sto\rangle \Rightarrow \langle S_1'\ S_2,\ envl',\ sto'\rangle}$$

[COMP-2] $$\frac{\langle S_1,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle S_1\ S_2,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}$$

[COMP-3] $$\frac{\langle S_1,\ envl,\ sto\rangle \Rightarrow \langle \epsilon,\ envl',\ sto'\rangle}{\langle S_1\ S_2,\ envl,\ sto\rangle \Rightarrow \langle S_2,\ envl',\ sto'\rangle}$$

[IF-E-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle if\ e\ ...,\ envl,\ sto\rangle \Rightarrow \langle if\ e'\ ...,\ envl',\ sto'\rangle}$$

[IF-E-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle if\ e\ ...,\ envl,\ sto\rangle \Rightarrow \langle if\ v\ ...,\ envl',\ sto'\rangle}$$

[IF-1-TRUE] $\langle if\ v\ then\ b,\ envl,\ sto\rangle \Rightarrow \langle b,\ envl,\ sto\rangle$

where $v\ =\ true$

[IF-1-FALSE] $\langle if\ v\ then\ b,\ envl,\ sto\rangle \Rightarrow \langle skip,\ envl,\ sto\rangle$

where $v\ =\ false$

[IF-2-TRUE] $\langle if\ v\ then\ b_1\ else\ then\ b_2,\ envl,\ sto\rangle \Rightarrow \langle b_1,\ envl,\ sto\rangle$

where $v\ =\ true$

[IF-2-FALSE] $\langle if\ v\ then\ b_1\ else\ then\ b_2,\ envl,\ sto\rangle \Rightarrow \langle b_2,\ envl,\ sto\rangle$

where $v\ =\ false$

[IF-3-TRUE] $\langle if\ v\ then\ b_1\ else\ if\ e_1\ then\ b_2,\ envl,\ sto\rangle \Rightarrow \langle b_1,\ envl,\ sto\rangle$

where $v\ =\ true$

[IF-3-FALSE] $\langle if\ v\ then\ b_1\ else\ if\ e_1\ then\ b_2,\ envl,\ sto\rangle \Rightarrow \langle if\ e_1\ then\ b_2,\ envl,\ sto\rangle$

where $v\ =\ false$

[IF-4-TRUE] $\langle if\ v\ then\ b_1\ else\ if\ e_1\ then\ b_2,\ else\ then\ b_3,\ envl,\ sto\rangle \Rightarrow \langle b_1,\ envl,\ sto\rangle$

where $v\ =\ true$

[IF-4-FALSE] $\langle if\ v\ then\ b_1\ else\ if\ e_1\ then\ b_2\ else\ then\ b_3,\ envl,\ sto\rangle$
$\Rightarrow \langle if\ e_1\ then\ b_2\ else\ then\ b_3,\ envl,\ sto\rangle$

where $v\ =\ false$

[SKIP] $\langle skip,\ envl,\ sto\rangle \Rightarrow \langle envl,\ sto\rangle$

[REPEAT] $\langle repeat\ b\ until\ e,\ envl,\ sto\rangle$
$\Rightarrow \langle b;\ if\ NOT\ e\ then\ (repeat\ b\ until\ e)\ else\ then\ skip\ ,\ envl,\ sto\rangle$

[WHILE] $\langle while\ e\ do\ b,\ envl,\ sto\rangle \Rightarrow \langle if\ e\ then\ (b;\ while\ e\ do\ b)\ else\ skip,\ envl,\ sto\rangle$

[UPTO] $\langle from\ e_1\ upto\ e_2\ b,\ envl,\ sto\rangle$
$\Rightarrow \langle if\ e_1\ <\ e_n\ then\ (b;\ e_2\ =\ e_1 + 1;\ from\ e_2\ upto\ e_n\ b)\ else\ skip,\ envl,\ sto\rangle$

[DOWNTO] $\langle from\ e_1\ downto\ e_2\ b,\ envl,\ sto\rangle$
$\Rightarrow \langle if\ e_1\ >\ e_n\ then\ (b;\ e_2\ =\ e_1 - 1;\ from\ e_2\ downto\ e_n\ b)\ else\ skip,\ envl,\ sto\rangle$

[DRIVE-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle drive(e),\ envl,\ sto\rangle \Rightarrow \langle drive(e'),\ envl',\ sto'\rangle}$$

[DRIVE-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl,\ sto\rangle}{\langle drive(e),\ envl,\ sto\rangle \Rightarrow \langle drive(v),\ envl,\ sto\rangle}$$

[DRIVE-3] $$\langle drive(v),\ envl,\ sto\rangle \Rightarrow \langle envl,\ sto[l_{out} \mapsto (v,\ forward)\ :\ vl]\rangle$$

[PAUSE-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle pause(e),\ envl,\ sto\rangle \Rightarrow \langle pause(e'),\ envl',\ sto'\rangle}$$

[PAUSE-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl,\ sto\rangle}{\langle pause(e),\ envl,\ sto\rangle \Rightarrow \langle pause(v),\ envl,\ sto\rangle}$$

[PAUSE-3] $$\langle pause(v),\ envl,\ sto\rangle \Rightarrow \langle envl,\ sto[l_{out} \mapsto (v,\ stop)\ :\ vl]\rangle$$

[TURNLEFT-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle turnLeft(e),\ envl,\ sto\rangle \Rightarrow \langle turnLeft(e'),\ envl',\ sto'\rangle}$$

[TURNLEFT-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl,\ sto\rangle}{\langle turnLeft(e),\ envl,\ sto\rangle \Rightarrow \langle turnLeft(v),\ envl,\ sto\rangle}$$

[TURNLEFT-3] $$\langle turnLeft(v),\ envl,\ sto\rangle \Rightarrow \langle envl,\ sto[l_{out} \mapsto (v,\ left)\ :\ vl]\rangle$$

[TURNRIGHT-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle turnRight(e),\ envl,\ sto\rangle \Rightarrow \langle turnRight(e'),\ envl',\ sto'\rangle}$$

[TURNRIGHT-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl,\ sto\rangle}{\langle turnRight(e),\ envl,\ sto\rangle \Rightarrow \langle turnRight(v),\ envl,\ sto\rangle}$$

[TURNRIGHT-3] $$\langle turnRight(v),\ envl,\ sto\rangle \Rightarrow \langle envl,\ sto[l_{out} \mapsto (v,\ right)\ :\ vl]\rangle$$

[RETURN-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl,\ sto\rangle}{\langle active\ return\ e\ end,\ envl,\ sto\rangle \Rightarrow \langle active\ return\ e'\ end,\ envl',\ sto'\rangle}$$

[RETURN-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl,\ sto\rangle}{\langle active\ return\ e\ end,\ envl,\ sto\rangle \Rightarrow \langle active\ return\ v\ end,\ envl',\ sto'\rangle}$$

[RETURN-3] $$\langle active\ return\ e\ end,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle$$

[CALL-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle f(\overrightarrow{e}),\ envl,\ sto\rangle \Rightarrow \langle f(\overrightarrow{e'}),\ envl',\ sto'\rangle}$$

where $\overrightarrow{e} = \{e_1,\ e_2,\ \ldots e_n\}$

and $\overrightarrow{e'} = \{e'_1,\ e'_2,\ \ldots e'_n\}$

[CALL-2]
$$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle f(\overrightarrow{e}),\ envl,\ sto\rangle \Rightarrow \langle f(\overrightarrow{v}),\ envl',\ sto'\rangle}$$

where $\overrightarrow{e} = \{e_1,\ e_2,\ \ldots e_n\}$
and $\overrightarrow{v} = \{v_1,\ v_2,\ \ldots v_n\}$

[CALL-3]
$$\frac{\begin{array}{c}\langle f(\overrightarrow{v}),\ envl,\ sto\rangle \\ \Rightarrow \langle b,\ (env'_V[\overrightarrow{x} \mapsto \overrightarrow{l}][next \mapsto following\ l],\ env'_F,\ env'_A) : envl',\ sto'[\overrightarrow{l} \mapsto \overrightarrow{v}]\rangle\end{array}}{\langle f(\overrightarrow{v}),\ envl,\ sto\rangle \Rightarrow \langle b,\ envl',\ sto'\rangle}$$

where $env_F\ f = \langle b,\ \overrightarrow{x},\ env'_V,\ env'_F,\ env'_A\rangle$
and $envl = (env'_V,\ env'_F,\ env'_A) : envl'$
and $\overrightarrow{x} = \{x_1,\ x_2,\ \ldots x_n\}$
and $\overrightarrow{v} = \{v_1,\ v_2,\ \ldots v_n\}$
and $l = env_V\ next$

[CALL-4]
$$\frac{\langle f(),\ envl,\ sto\rangle \Rightarrow \langle b,\ (env'_V,\ env'_F,\ env'_A) : envl',\ sto\rangle}{\langle f(),\ envl,\ sto\rangle \Rightarrow \langle b,\ envl',\ sto\rangle}$$

where $env_F\ f = \langle b,\ x,\ env'_V,\ env'_F,\ env'_A\rangle$
and $envl = (env'_V,\ env'_F,\ env'_A) : envl'$

[PROG-1]
$$\frac{\langle D_V,\ envl,\ sto\rangle \Rightarrow \langle D'_V,\ envl',\ sto'\rangle}{\langle begin\ D_V\ D_A\ D_F\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle begin\ D'_V\ D_A\ D_F\ S\ end,\ envl',\ sto'\rangle}$$

[PROG-2]
$$\frac{\begin{array}{c}\langle D_V,\ env_V,\ sto\rangle \Rightarrow \langle env'_V,\ sto'\rangle \\ env'_V,\ sto' \vdash \langle D_A,\ env_A,\ sto'\rangle \Rightarrow \langle env'_A,\ sto''\rangle \\ env'_V,\ env'_A \vdash \langle D_F,\ env_F\rangle \Rightarrow \langle env'_F\rangle \\ \langle S,\ envl',\ sto''\rangle \Rightarrow \langle active\ S\ end,\ envl',\ sto''\rangle\end{array}}{\langle begin\ D_V\ D_A\ D_F\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle active\ S\ end,\ envl',\ sto''\rangle}$$

where $envl = (env'_V,\ env'_F,\ env'_A) : envl'$

[BLOCK-1]
$$\frac{\langle D_V,\ envl,\ sto\rangle \Rightarrow \langle D'_V,\ envl',\ sto'\rangle}{\langle begin\ D_V\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle begin\ D'_V\ S\ end,\ envl',\ sto'\rangle}$$

[BLOCK-2]
$$\frac{\begin{array}{c}\langle D_V,\ envl,\ sto\rangle \Rightarrow \langle envl',\ sto'\rangle \\ \langle S,\ envl',\ sto'\rangle \Rightarrow \langle active\ S\ end,\ envl',\ sto'\rangle\end{array}}{\langle begin\ D_V\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle active\ S\ end,\ envl',\ sto'\rangle}$$

[ACTIVE-1]
$$\frac{\langle S,\ envl,\ sto\rangle \Rightarrow \langle S',\ envl',\ sto'\rangle}{\langle active\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle active\ S'\ end,\ envl',\ sto'\rangle}$$

[ACTIVE-2] $$\frac{\langle S,\ envl,\ sto\rangle \Rightarrow \langle envl',\ sto'\rangle}{\langle active\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle envl',\ sto'\rangle}$$

[ACTIVE-3] $$\frac{\langle S,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle active\ S\ end,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}$$

[VAR-DCL] $$\frac{\langle D_V, (env_V[x \mapsto l][next \mapsto following\ l], env_F, env_A) : envl,\ sto\rangle \Rightarrow \langle envl', sto\rangle}{\langle T\ x\ D_V, envl, sto\rangle \Rightarrow \langle envl', sto\rangle}$$

where $l = env_V\ next$

[VAR-INI-1] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle e',\ envl',\ sto'\rangle}{\langle T\ x\ is\ e,\ envl,\ sto\rangle \Rightarrow \langle T\ x\ is\ e',\ envl',\ sto'\rangle}$$

[VAR-INI-2] $$\frac{\langle e,\ envl,\ sto\rangle \Rightarrow \langle v,\ envl',\ sto'\rangle}{\langle T\ x\ is\ e,\ envl,\ sto\rangle \Rightarrow \langle T\ x\ is\ v,\ envl',\ sto'\rangle}$$

[VAR-INI-3] $$\frac{\langle D_V, (env_V[x \mapsto l][next \mapsto following\ l], env_F, env_A) : envl,\ sto[l \mapsto v]\rangle \Rightarrow (envl', sto')}{\langle T\ x\ is\ v\ D_V, envl, sto\rangle \Rightarrow (envl', sto')}$$

where $l = env_V\ next$

[EMPTY-DCL] $\langle \epsilon,\ envl,\ sto\rangle \Rightarrow \langle envl,\ sto\rangle$

[T-FUNC-DCL] $$\frac{\langle D_F, (env_V, env_F[f \mapsto (b, \overrightarrow{x}, env_V, env_F, env_A)], env_A) : envl,\ sto\rangle \rightarrow \langle envl',\ sto\rangle}{\langle function\ f\ returns\ T\ (\overrightarrow{x})\ b\ D_F, envl, sto\rangle \rightarrow \langle envl',\ sto\rangle}$$

where $env_F\ f = \langle b,\ \overrightarrow{x},\ env_V,\ env_F,\ env_A\rangle$
and $\overrightarrow{x}\ =\ \{x_1,\ x_2,\ \ldots\ x_n\}$

[FUNC-DCL] $$\frac{\langle D_F, (env_V, env_F[f \mapsto (b, \overrightarrow{x}, env_V, env_F, env_A)], env_A) : envl,\ sto\rangle \rightarrow \langle envl',\ sto\rangle}{\langle function\ f\ (\overrightarrow{x})\ b\ D_F, envl, sto\rangle \rightarrow \langle envl',\ sto\rangle}$$

where $env_F\ f = \langle b,\ \overrightarrow{x},\ env_V,\ env_F,\ env_A\rangle$
and $\overrightarrow{x}\ =\ \{x_1,\ x_2,\ \ldots\ x_n\}$

[EMPTY-DCL] $\langle \epsilon,\ envl,\ sto\rangle \rightarrow \langle envl,\ sto\rangle$

[ARR-DCL] $$\frac{\langle D_A, (env_V, env_F, env_A[a \mapsto l][next \mapsto following\ l]) : envl,\ sto\rangle \rightarrow \langle envl',\ sto'\rangle}{\langle T\ array\ a\ D_A,\ envl,\ sto\rangle \rightarrow \langle envl',\ sto'\rangle}$$

where $l = env_A\ next$

[EMPTY-DCL] $\langle \epsilon,\ envl,\ sto\rangle \rightarrow \langle envl,\ sto\rangle$

D | Type Rules

[PLUS$_{exp}$] $$\frac{E \vdash e_1 : \mathbf{B''} \quad E \vdash e_2 : \mathbf{B'}}{E \vdash e_1 + e_2 : \mathbf{B}}$$

[MINUS$_{exp}$] $$\frac{E \vdash e_1 : \mathbf{N''} \quad E \vdash e_2 : \mathbf{N'}}{E \vdash e_1 - e_2 : \mathbf{N}}$$

[MULT$_{exp}$] $$\frac{E \vdash e_1 : \mathbf{N''} \quad E \vdash e_2 : \mathbf{N'}}{E \vdash e_1 * e_2 : \mathbf{N}}$$

[DIVIDE$_{exp}$] $$\frac{E \vdash e_1 : \mathbf{N''} \quad E \vdash e_2 : \mathbf{N'}}{E \vdash e_1 / e_2 : \mathbf{N}}$$

[APPEND$_{exp}$] $$\frac{E \vdash e_1 : \mathbf{text''} \quad E \vdash e_2 : \mathbf{text'}}{E \vdash e_1 \text{ append } e_2 : \mathbf{text}}$$

[PAREN$_{exp}$] $$\frac{E \vdash e : \mathbf{T}}{E \vdash (e) : \mathbf{T}}$$

[U-MINUS$_{exp}$] $$\frac{E \vdash e : \mathbf{N}}{E \vdash -e : \mathbf{N}}$$

[EQUALS$_{exp}$] $$\frac{E \vdash e_1 : \mathbf{T} \quad E \vdash e_2 : \mathbf{T}}{E \vdash e_1 = e_2 : \mathbf{truth}}$$

[GREATER-THAN$_{exp}$] $$\frac{E \vdash e_1 : \mathbf{N'} \quad E \vdash e_2 : \mathbf{N}}{E \vdash e_1 > e_2 : \mathbf{truth}}$$

[LESS-THAN$_{exp}$] $$\frac{E \vdash e_1 : \mathbf{N'} \quad E \vdash e_2 : \mathbf{N}}{E \vdash e_1 < e_2 : \mathbf{truth}}$$

[NOT$_{exp}$] $$\frac{E \vdash e_1 : \mathbf{truth}}{E \vdash \text{NOT } e : \mathbf{truth}}$$

$$[\text{AND}_{exp}] \quad \frac{E \vdash e_1 : \textbf{truth'} \quad E \vdash e_2 : \textbf{truth}}{E \vdash e_1 \text{ AND } e_2 : \textbf{truth}}$$

$$[\text{OR}_{exp}] \quad \frac{E \vdash e_1 : \textbf{truth'} \quad E \vdash e_2 : \textbf{truth}}{E \vdash e_1 \text{ OR } e_2 : \textbf{truth}}$$

$$[\text{ARR-ELE}]_{exp}] \quad \frac{E(a) = \mathbf{T} \quad E \vdash e : \textbf{integer}}{E \vdash a \text{ element } e : \mathbf{T}}$$

$$[\text{PROG-BLOCK}]_{block}] \quad \frac{E \vdash D_V : \textbf{ok} \quad E_1 \vdash D_A : \textbf{ok} \quad E_2 \vdash D_F : \textbf{ok} \quad E_3 \vdash S : \textbf{ok}}{E \vdash \text{begin } D_V \ D_A \ D_F \ S \text{ end} : \textbf{ok}}$$

where $E_1 = E(D_V, E)$
and $E_2 = E(D_A, E_1)$
and $E_3 = E(D_F, E_2)$

$$[\text{BLOCK}]_{block}] \quad \frac{E \vdash D_V : \textbf{ok} \quad E_1 \vdash S : \textbf{ok}}{E \vdash \text{begin } D_V \ S \text{ end} : \textbf{ok}}$$

where $E_1 = E(D_V, E)$

$$[\text{VAR-ASS}_{stmt}] \quad \frac{E \vdash x : \mathbf{T} \quad E \vdash e : \mathbf{T}}{E \vdash x \text{ is } e : \textbf{ok}}$$

$$[\text{ARR-ASS}_{stmt}] \quad \frac{E \vdash e_2 : \mathbf{T} \quad E(a) = \mathbf{T} \vdash \mathbf{T} \quad E \vdash e_1 : \textbf{integer}}{E \vdash a \text{ element } e_1 \text{ is } e_2 : \textbf{ok}}$$

$$[\text{COMP}_{stmt}] \quad \frac{E \vdash S_1 : \textbf{ok} \quad E \vdash S_2 : \textbf{ok}}{E \vdash S_1 \ S_2 : \textbf{ok}}$$

$$[\text{IF}_{stmt}] \quad \frac{E \vdash e : \textbf{truth} \quad E \vdash b : \textbf{ok}}{E \vdash \text{if } \overrightarrow{\{e \text{ then} : b\}} : \textbf{ok}}$$

$$[\text{IF-ELSE}_{stmt}] \quad \frac{E \vdash e : \textbf{truth} \quad E \vdash b_1 : \textbf{ok} \quad E \vdash b_2 : \textbf{ok}}{E \vdash \text{if } \overrightarrow{\{e \text{ then} : b_1\}} \text{ else } b_2 : \textbf{ok}}$$

$$[\text{WHILE}_{stmt}] \quad \frac{E \vdash e : \textbf{truth} \quad E \vdash b : \textbf{ok}}{E \vdash \text{while } e \text{ do } b : \textbf{ok}}$$

$$[\text{REPEAT}_{stmt}] \quad \frac{E \vdash b : \textbf{ok} \quad E \vdash e : \textbf{truth}}{E \vdash \text{repeat } b \text{ until } e : \textbf{ok}}$$

$$[\text{FROM-UPTO}]_{stmt}] \quad \frac{E \vdash e_1 : \textbf{integer} \quad E \vdash e_2 : \textbf{integer} \quad E \vdash b : \textbf{ok}}{E \vdash \text{from } (e_1 \text{ upto } e_2) \ b : \textbf{ok}}$$

[FROM-DOWNTO$_{stmt}$] $$\frac{E \vdash e_1 : \textbf{integer} \quad E \vdash e_2 : \textbf{integer} \quad E \vdash b : \textbf{ok}}{E \vdash \text{from } (e_1 \text{ downto } e_2)\ b : \textbf{ok}}$$

[DRIVE$_{stmt}$] $$\frac{E \vdash e : \textbf{integer}}{E \vdash \text{drive}(e) : \textbf{ok}}$$

[PAUSE$_{stmt}$] $$\frac{E \vdash e : \textbf{integer}}{E \vdash \text{wait}(e) : \textbf{ok}}$$

[TURN-LEFT$_{stmt}$] $$\frac{E \vdash e : \textbf{integer}}{E \vdash \text{turnleft}(e) : \textbf{ok}}$$

[TURN-RIGHT$_{stmt}$] $$\frac{E \vdash e : \textbf{integer}}{E \vdash \text{turnright}(e) : \textbf{ok}}$$

[CALL-RETURN$_{stmt}$] $$\frac{E(f) = ((\overrightarrow{x} : \overrightarrow{T}) \rightarrow \textbf{ok}, (S : T')) \vdash \overrightarrow{\textbf{T}}, \textbf{T'} \quad E \vdash \overrightarrow{e} : \overrightarrow{\textbf{T}}}{E \vdash f(\overrightarrow{e}) : \textbf{T'}}$$

[CALL$_{stmt}$] $$\frac{E(f) = ((\overrightarrow{x} : \overrightarrow{T}) \rightarrow \textbf{ok}, (S : T')) \vdash \overrightarrow{\textbf{T}} \quad E \vdash \overrightarrow{e} : \overrightarrow{\textbf{T}}}{E \vdash f(\overrightarrow{e}) : \textbf{ok}}$$

[RETURN$_{stmt}$] $$\frac{E(f) = ((\overrightarrow{x} : \overrightarrow{T}) \rightarrow \textbf{ok}, (S : T')) \vdash \textbf{T'} \quad E \vdash e : \textbf{T'}}{E \vdash \text{return } e : \textbf{ok}}$$

[EMPTY$_{dec}$] $$E \vdash \epsilon : \textbf{ok}$$

[VAR-ASS$_{dec}$] $$\frac{E[x \mapsto T] \vdash D_V : \textbf{ok} \quad E \vdash e : \textbf{T}}{E \vdash T\ x \text{ is } e\ D_V : \textbf{ok}}$$

[VAR$_{dec}$] $$\frac{E[x \mapsto T] \vdash D_V : \textbf{ok}}{E \vdash T\ \text{x}\ D_V : \textbf{ok}}$$

[FUNC-RETURN$_{dec}$] $$\frac{E \vdash b : \textbf{ok} \quad E[f \mapsto ((\overrightarrow{D_V} : \overrightarrow{T} \rightarrow \textbf{ok}),\ T')] \vdash D_F : \textbf{ok}}{E \vdash \text{function } f \text{ returns } T'\ (\overrightarrow{D_V})\ b\ D_F : \textbf{T'}}$$

[FUNC$_{dec}$] $$\frac{E \vdash b : \textbf{ok} \quad E[f \mapsto (\overrightarrow{D_V} : \overrightarrow{T} \rightarrow \textbf{ok})] \vdash D_F : \textbf{ok}}{E \vdash \text{function } f\ (\overrightarrow{D_V})\ b\ D_F : \textbf{ok}}$$

[ARR$_{dec}$] $$\frac{E[a \mapsto T] \vdash D_A : \textbf{ok}}{E \vdash T \text{ array } a\ D_A : \textbf{ok}}$$

E Abstract Nodes diagram

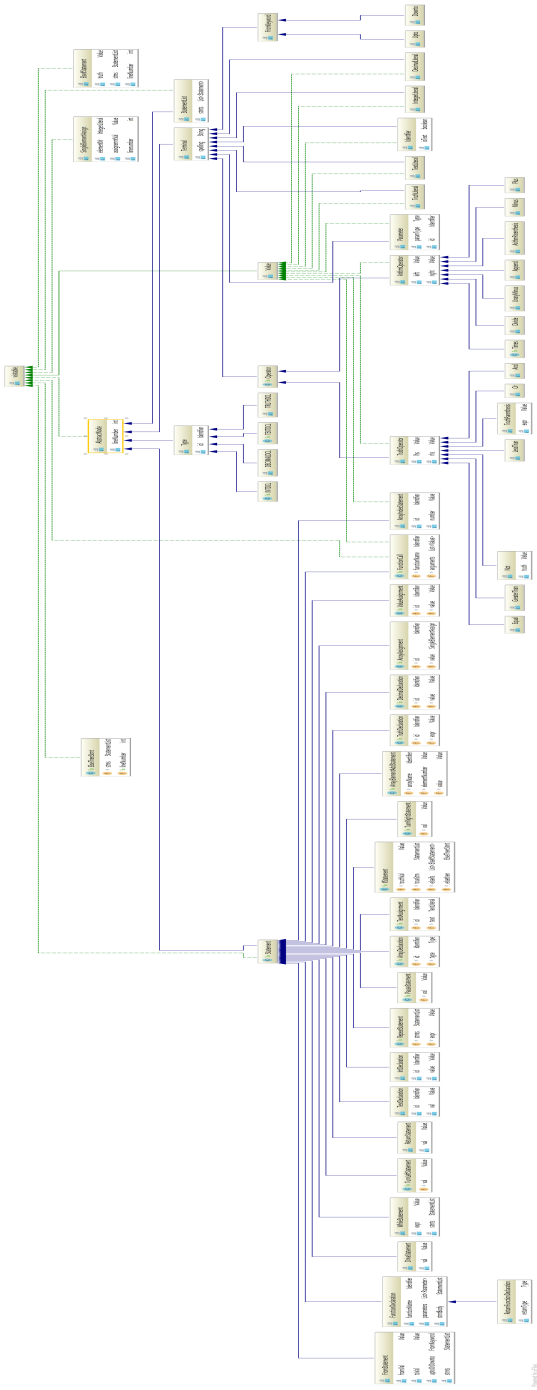

Figure E.1: Class diagram of AST classes

F Usability test

The tasks for the usability test is placed here. The information about the two languages syntax given to the test subject is also included together with the results from the usability test.

F.1 Tasks

1. Make the car drive forward for 2 seconds, then pause for 2 seconds. Repeat this 10 times using a from loop.
2. Make a declaration of integer named i and initialise it with the value 5. Then another integer declaration named j and initialise it with the value 2. Create an if statement with condition i > j. In the body of the if-statement make it turn right for 2 seconds.
3. Initialise an integer with a decimal value. Then compile the program.

F.2 Syntax description for tasks

Trun Language

Language structure:

1. Variable declarations
2. array declarations
3. function declarations
4. statements

Types:

integer, decimal, truth, text

Example of integer initialization:

integer varName is 3

Example of from loop:

from(1 upto 10)

Example of drive, pause, & turn statement:

drive(10)

pause(10)

turnright(10)

Example of if statement:

if(7 > 3) then

Arduino Language

Must always have a setup function and loop function.

The setup function is used to set the mode of the pins (output/input).

Example of setting pin mode:

void setup()

pinMode(12, OUTPUT);

The left motor is connected to pin 12 and the right motor is connected to 13.

Example of integer initialization:

int varName = 2;

Example of loops:

for(int i = 0; i < 8; i++)

while(i < 8)

Example of setting pins high or low:

digitalWrite(pinNumber, HIGH);

digitalWrite(pinNumber, LOW);

High = ON

Low = OFF

When the left motor is ON the car turns right

When the right motor is ON the car turns left.

Example of delay statement:

delay(5000)

The number 5000 is the time in milliseconds.

Example of if statement:

if(7 > 3)

F.3 Briefing Script

Our project is about making a programming language that is suitable for learning beginners to program. Programs written in Trun is able to execute on the Trun car. The trun car has two motors connected to the back wheels of the car.

You are welcome to ask questions at any time.

There are three tasks to complete. First you will complete them in the Trun language and afterwards in the Arduino language.

First i want to ask you to get familiar with the syntax for the Trun language that you will need to complete the tasks.

Then you will complete the tasks using Trun.

Next you will get familiar with the Arduino syntax necessary and complete the tasks using Arduino's syntax.

We will not run any code on the car itself during the test.

F.4 Results

The code written by the test subject can be seen on **Figure F.1**, **Figure F.2**, **Figure F.3**, and **Figure F.4**.

F.4.1 Trun

Task 1:

Time for completion: 0:58

```
from (1 upto 10) {
    drive(2)
    pause(2)
}
```

Figure F.1: Code written for task 1 in Trun

Task 2:

Time for completion: 1:06

```
integer i is 5
integer j is 2

if (i > j) then {
    turnright(2)
}
```

Figure F.2: Code written for task 2 in Trun

F.4.2 Arduino

Task 1:

Time for completion: 3:50

```
void setup(){
    pinMode(12, OUTPUT);
    pinMode(13, OUTPUT);
}

void loop(){
   for (int i = 0; i < 10; i++) {
      digitalWrite(12, HIGH);
      digitalWrite(13, HIGH);
      delay(2000);
      digitalWrite(12, LOW);
      digitalWrite(13, LOW);
      delay(2000);
   }
}
```

Figure F.3: Code written for task 1 in Arduino

Task 2:

Time for completion: 1:27

```
void setup(){
   pinMode(12, OUTPUT);
}

void loop(){
   int i = 5;
   int j = 2;

   if (i > j) {
      digitalWrite(12, HIGH);
      delay(2000);
      digitalWrite(12, LOW);
   }
}
```

Figure F.4: Code written for task 2 in Arduino