Tang

A Programming Language for Arduino

Group:
sw402f17

Supervisor:
Lone Leth Thomsen

May 28, 2017

Department of Computer Science
Aalborg University
http://cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Tang

Theme:
A Programming Language for Arduino

Project Period:
Spring 2017

Group:
sw402f17

Participants:
Morten Rask Andersen
Anton Christensen
Christian Mønsted Grünberg
Steffan Riemann Hansen
Mathias Ibsen
Mathias Rohde Pihl

Supervisor:
Lone Leth Thomsen

Pages:
129

Date of Completion:
May 28, 2017

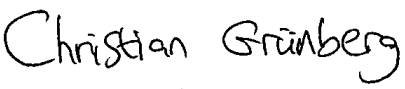
Number of Copies:
1

Abstract:

The focus of this project is to design, formalise, implement, and test a programming language, named Tang, with the purpose of programming an Arduino. The language should allow the programmers to control hardware directly in a readable way. To do so, language design criteria and language features are prioritised in regards to the target group and time limitation of the project. To describe the syntactical rules of Tang, regular expressions and a context free grammar were used. The semantic meaning of the language has been defined using structural operational semantics and a type system.
For the development of Tang, a compiler generator was created with the purpose of allowing dynamic changes in the design of Tang, without having to manually update the compiler each time. The parser, alongside the other compiler phases, are individually and collectively tested and Tang is also evaluated by two computer science students. The result of the project is an imperative language that allows programmers, new to embedded programming, to write programs for the Arduino. This is done by using the features of Tang to abstract over and control hardware components without having to learn complex operations like bit-wise operators.

Morten Rask Andersen

Anton Christensen



Christian Mønsted Grünberg

Steffan Riemann Hansen

Mathias Ibsen

Mathias Rohde Pihl

Preface

This project has been produced by the Software group sw402f17 consisting of six members attending Aalborg University. The project is a fourth semester project spanning the time from February to May 2017. In addition to this report, the code from this project can be found in the electronic appendix which is uploaded alongside the report.
We would like to thank Lone Leth Thomsen for her consultations and supervision of the project.

Reading Guide

Unless otherwise stated, the illustrations and figures used in this report have been made by the group sw402f17. In some of the code listings shown in the report, only the essential parts have been included. Removed sections of the code have been replaced by an ellipsis [...]. For citation, we will be using a number system where citation numbers appear inside brackets where the number is a reference to the sources in the bibliography. In the bibliography, sources have the following format: author, title, language, URL or ISBN as well as an optional date of when the URL/book was written or edited. An example is [1, p. 100] which is a reference to the book Crafting a Compiler by Fisher et al. page 100.

Abbreviations

Listed below are abbreviations used in the project along with their meaning.

- AST = Abstract Syntax Tree
- CST = Concrete Syntax Tree
- GPIO = General Purpose Input Output
- CFG = Context Free Grammar
- BNF = Backus-Naur Form
- IDE = Integrated Development Environment
- IR = Intermediate Representation
- kb = kilo*bit*
- kB = kilo*byte*
- MCU = MicroController Unit
- RAM = Random Access Memory
- PCB = Printed Circuit Board
- PWM = Pulse Width Modulation
- I/O = Input Output
- MIPS = Computing speed (Million Instructions Per Second)
- MHz = Mega Hertz
- LED = Light Emitting Diode

- VM = Virtual Machine
- JVM = Java Virtual Machine
- ADC = Analogue to Digital Converter
- DDR = Data Direction Register

Contents

Chapter 1

Introduction

Programming languages are the means by which software developers write programs. There exist a wide range of them, ranging from high-level languages such as C# and Java, to low-level languages like machine code.
In this project, we will develop a programming language according to the project proposal in [2] which suggests developing a new programming language for the Arduino. The reason for this choice is that the project group is familiar with writing programs for the Arduino platform, and through this experience, members of the group have experienced problems understanding concepts such as bit-shifting and pointers.

We therefore establish the following preliminary problem for creating a new language for Arduino:

> *How can a programming language for Arduino be developed for programmers, who have no previous experience with embedded programming?*

The programming language developed in this project will be referred to as Tang. According to the study-curriculum, seen in [3], we will, in addition to creating a language, also implement either an interpreter or compiler for Tang. Until we choose between using an interpreter or compiler, we will refer to interpreters and compilers by the common word translator.

The target group in this project is programmers with no previous experience in embedded programming. To be able to make choices in relation to this target group, we state the following that defines our target group. Programmers who ...

- know about basic arithmetical operations, primitive types, variables, control structures, classes, loops, and functions and have experience using them.
- have little or no experience using bit-wise operators (left/right shift, not, or, xor, and).
- do not have any previous experience with development boards like Arduino.

In order to develop the Tang programming language for our intended target group, we will, in chapter 2, describe the methodology used throughout the project including the constraints, development method, and test plan for this project.
In chapter 3, we will describe the Arduino platform and look at issues regarding different programming languages for the Arduino platform.
The analysis conducted in chapter 3 will be used along with the study curriculum to establish the problem statement in chapter 4.
In chapter 5, the theory necessary for understanding the development of Tang will be described

which includes theory on the following topics: programming languages, paradigms, syntax, semantics, translators, parsing techniques, and the visitor pattern.
After the description of these topics, we will begin to design Tang in chapter 6 which includes prioritisation of language evaluation criteria and choice of paradigm for Tang. Furthermore, we will in this chapter choose the language features that are included in Tang and how each of these language features can be written in Tang.
As a final part of the design chapter we will, in section 6.7.1, make the choice of translator used to develop Tang.
Chapter 7 will give a formal description of the syntax and semantics of Tang which includes a description of the tokens, grammar, operational semantics, and type system of Tang.
After having described the language specifications for Tang, we will, in chapter 8, describe the implementation of Tang including a discussion of different approaches for implementing Tang.
Chapter 8 proceeds by presenting the different parts of the implementation of Tang. We will then conduct different tests for Tang for evaluation purposes in chapter 9 by using the test plan we established in section 2.6. Finally, we will discuss and conclude the development of Tang in chapter 10 and 11, respectively. The conclusion will be drawn based on the problem statement in relation to the design and implementation of the programming language, Tang.

Chapter 2

Methodology

The purpose of this chapter is to describe, discuss, and choose the development methodology and test plan used throughout this project based on the project conditions. To choose the development method we describe methodologies ranging from predictive to adaptive methodologies which are the waterfall, iterative-waterfall, and iterative method.

2.1 Project Conditions

The goal of this project is to design, formalise, and implement a programming language. The workload of the project is 15 ECTS for each group member. Furthermore, the project group has not previously developed a programming language and must therefore in relation to the development methodology take this into account.

2.2 Waterfall

In the waterfall method all preceding phases are completed before the next phase is begun. This ensures a complete understanding before further development is done. Generally, phases are not revisited once done, but there can be deviations from this rule, see section 2.3.

Documentation and analysis are in focus in the waterfall method, where the supporting argument is, that problems found earlier in a large project are far less costly in terms of both time and money, compared to finding the problem at a later stage. This is one of the reasons why a significant amount of the development period is spent on analysing the problem before coding begins [4].

2.3 Iterative Waterfall

It can be beneficial to adapt any methodology to the specific project in question [5]. Specifically, this can be done by introducing iterative elements into the waterfall approach, so it is possible to change earlier phases, based on the knowledge gained in the later phases [4]. This is called the iterative waterfall model, and is well suited for teams with less experience on the given subject [6]. In most projects which rely on the waterfall method, some deviations are made from the pure waterfall phases described as "In practice, these stages overlap and feed information to each other." [7]. This means that the waterfall approach used in practice is in most projects similar to the iterative waterfall model. The iterative waterfall model is visualised in figure 2.1.

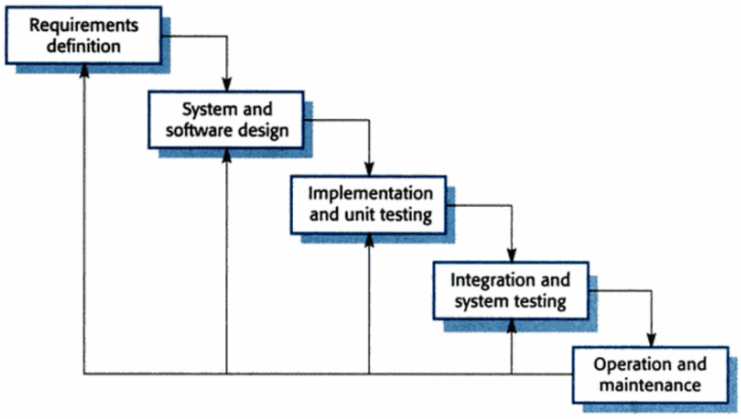


Figure 2.1: Iterative Waterfall Phases [7]

2.4 Iterative

The iterative approach to software development is characterised by having multiple repetitions of the development process. The product is reevaluated at the end of each iteration, and all the individual development steps can be revisited throughout the development. By starting the actual coding phase as early as possible, it leaves room for change at a later stage, which may require more resources when using a more traditional method [5].

With the lack of documentation needed in an iterative method, compared to the waterfall method in section 2.2, an iterative approach is well-suited for projects, where getting the product developed fast is important [8]. In smaller teams, there is less need for structure and well-documented phases. Therefore, the iterative model suits these teams well. The iterative method also suits teams with a more ambiguous problem, where vast amounts of documentation is not an option [8].

2.5 Discussion

As the group does not have any experience with design and implementation of a programming language and the development of a compiler in relation to the programming language, it is difficult to organise and know which phases are more important and should have more time allocated, and which are less important. Therefore, the waterfall method is not ideal for this project, as our lack of experience impedes planning ahead.
A more suitable way of planning is to follow the courses spanning most of the semester. This enables us to learn about a specific subject and thereafter implement it in our own project. It also helps us organise our schedule. However, as the courses span most of the semester, using the iterative development method would constrain us to making the first iteration last until the courses are finished.
Therefore, we choose the iterative waterfall method as the development method for this project. The iterative waterfall method fits our project as we do not need to follow a strict plan, it aids a project like ours, where the next phase can be somewhat unknown to the group, and it makes us able to iterate through earlier phases to revise them once we obtain new knowledge.

2.6 Test Plan

In this section, we define the test plan used in this project for testing Tang. The test plan consists of four types of tests: unit tests, integration tests, verification tests, and language evaluation tests. The goal of the test plan is to document the testing approach used throughout this project to test programming languages.

Unit Test

To ensure that the code written for the implementation of Tang works as intended, we will be conducting unit tests on the written code across all the different phases of the translator. The purpose of unit tests is to test small units of the program such as classes and methods. The focus of the unit tests will thus be to evaluate the different units of the compiler such as the lexer. If the tests do not comply with the expected results, informative and precise exceptions should be cast.

Integration Test

In the second phase of testing, we combine the modules of the translator to see if they work together by conducting integration tests. To do this programs are written in Tang and translated. The translation is then tested by writing tests that checks if the result of the translation is as expected.

Verification Test

The third phase of testing is the verification test where we test if the program written in Tang is able to run as intended on an Arduino and is meeting the requirements set by the group. An example of such a test is writing a program in Tang that makes an LED blink, and then testing if we can translate this program, run it on the Arduino, and observe the LED is blinking.

Language Evaluation

In the language evaluation test we will compare Tang to the following five different languages for the Arduino platform: the Arduino language, AVR C, Java, JavaScript, and AVR Assembler. To evaluate the languages, we will implement a blink program in each language where an Arduino is programmed to make an LED blink with a given time interval as described in [9]. We will evaluate the languages in accordance to four language evaluation criteria: writability, readability, reliability, and cost, which are further elaborated on in section 5.1.1.

For a general evaluation of the Tang language, we will perform a user test based on the overall overall steps in the Discount Method for Programming Language Evaluation [10]. We will not use it for the iterative evaluation purposes it was designed for. The primary focus of this test is to evaluate if the target group can understand programs written in Tang, and the secondary purpose is to evaluate how intuitively our target group can write programs in Tang.

The Discount Method [10] specifies steps to evaluate a programming language. The steps used for language evaluation are briefly defined below:

- `Create tasks` - create language specific problems for the test users to solve.
- `Create a sample sheet` - create a cheat sheet with code examples etc. to help the test users with e.g. the syntax of the language.
- `Estimate the task duration` - estimate by solving the tasks yourself.

- `Prepare setup` - choose the setup relevant for your test. Recording the test will help save data.
- `Gather participants` - the ideal number of participants is five.
- `Start the experiment` - explain to the test person that it is the language being tested and not him.
- `Keep the participants talking` - making the participants think out loud can be helpful.
- `Interview the participant` - after the experiment is conducted, ask the participant for his/her thoughts. Optionally, questions could be prepared beforehand.
- `Analyse data` - analyse the data collected during the test, this could e.g. be done by categorising problems based on their importance.

Chapter 3

Problem Analysis

In this chapter, we describe and analyse Arduino hardware and its constraints. In relation to this we analyse how an Arduino board can be programmed to control a LED in five different programming languages. From this analysis we discuss problems of the tested programming languages and derive which problems we will address in this project.

3.1 Arduino

The Arduino is described as: "[...] *an open-source electronics platform* [...]" [11] that includes PCB designs and firmware development tools like an IDE based on the processing IDE[12] and a bootloader which makes programming the Arduino possible without specialised tools.
In this project, we choose to focus on the Arduino Uno and Arduino Nano boards because these development boards are available to us and because these are categorised as entry level boards described as: "[...] *easy to use and ready to power your first creative projects.*" [13]. This makes these boards suited for beginners like the target group in this project as described in section 1. Another argument for choosing them, is they both use the ATmega328P which means code will be compatible between them as described in section 3.1.1.

3.1.1 Hardware Platform

In this section, we introduce the hardware associated with the Arduino and some of the similar alternatives to some of the components in the Arduino.

Microcontrollers

The Arduino is a development board for the ATmega328P microcontroller. The Arduino is therefore different from the typical micro*processor* in most modern PCs. In this section we will discuss the pros and cons and differences of each technology.

A microcontroller unit (MCU), or *microcontroller* for short, is a single chip on which all the basic components a computer system need reside. These basic components are permanent storage on which it stores program code and constants, processing units that evaluate instructions, RAM, and I/O interfaces in the form of physical pins with different functions. Though it has some limitations, microcontrollers are typically less expensive and smaller than a processor in modern computers. The offical Arduino Uno and Nano can be bought for 20-22 EUR [14] [15]. Unofficial boards resembling the official, but for a fraction of the price exist [16], even though it is possible to buy the

ATmega328 chip by itself relatively cheap [17].

Micro*processor* Units (MPU) are different in several regards. A microcontroller will have flash storage on the chip to store the application firmware needed to run the chip whereas a microprocessor, like the Intel or AMD processors in modern PCs, need external data storage like a hard-drive which is comparably slow and therefore it is necessary to load program code and data into faster RAM storage [18]. Microcontrollers also include other systems-on-chip like internal RAM and input/output, which a microprocessor in general does not. In most computers these tasks are handled by the motherboard and connected components [19]. With that said, a microcontroller is limited to its flash storage and memory, where a microprocessor can have these expanded depending on which components are connected to it [18].

Pin

Pins are the physical connection between a microcontroller like the Arduino and external devices. Each pin has one or more functions like power input and ground pins through which the microcontroller receives the power necessary to operate. Other pins can be configured through I/O registers to detect a change in voltage and trigger an interrupt function in software, or generate PWM signals that can control the speed of motors or intensity of light. A common type of pin is a GPIO pin which can either be controlled to output a high or low voltage or be configured as input to read a digital value if a voltage is supplied by an external source.

I/O registers

Microcontrollers are controlled through software. The software controls the hardware by controlling input and output functions of the microcontroller by reading input registers or writing to output registers. A register has an address in memory and setting bits can control functions like setting the voltage level of a pin either high (usually 3.3 or 5 Volts) or low (0 Volts). Registers might also, configure which events should trigger an interrupt.

Interrupts

Interrupts are used to signal to the MCU when a certain event like a change in voltage on a pin occurs. Interrupts could e.g. occur when a button is pressed or a hardware failure occurs. Interrupts on the ATmega328P can be categorised as either internal and external [20].
Internal interrupts are triggered by an internal event in the chip, like a timer or the analogue comparator [21].
External interrupts are triggered by external events like a rising edge on a pin.
Interrupts are triggered by a rising edge, falling edge or any edge and is called no matter what the Arduino is doing when the interrupt is triggered, which also allows the Arduino to react quickly to whatever caused the interrupt. A rising edge is when a signal goes from low to high, and falling edge is the reverse, from high to low. Any edge is one of the two.

The ATmega328P MCU

The ATmega328P microcontroller (or ATmega328P) is used in both the Arduino Uno and Nano development boards. Some of the overall specifications of the ATmega328P is shown in table 3.1.

Property	Value
Max-throughput	20 MIPS
Program storage	32 Kilo bytes
RAM	2 Kilo bytes
EEPROM storage	1 Kilo byte
I/O pins	23

Table 3.1: Select ATmega328p hardware specifications [20, p. 1-2].

Constraints

In this section we will get an understanding of the constraints that the programmer of an Arduino is working under. We will do this by analysing the constraints of the ATmega328P.

Memory There are 32 kiloBytes of program memory available on the ATmega328P chip. As applications grow larger you will eventually run out of program space. As such a compiler should be efficient as any reduction of compiled code will allow the programmer more space on the chip without having to buy a larger chip.

Speed The ATmega328P has a maximum throughput of 20 MIPS which, compared to modern computers, is slow. Computers with large operating systems will, on the other hand, have to share execution time among many processes, whereas with embedded development you will often be in control of exactly what code is executing. When using an Arduino the ATmega328P is running with a 16 MHz and therefore the maximum throughput is decreased to 16 MIPS.

Debugging When developing software for a microcontroller it can be difficult to debug run-time errors because not all microcontrollers and corresponding development environments support debugging. Therefore, it is difficult to step through the program to inspect values of different variables at run-time to discover where in the source program an error has occurred. Simple debugging of variable values can be accomplished on the Arduino platform by printing values via USB to the connected computer using a statement like *Serial.println(i)*. The problem with this debugging strategy and break point based debugging is that inserting extra statements or breaking at a statement in the program might not reflect the original behaviour of the running program as communication with other connected components might be interrupted by the extra print statements or breakpoints.

3.1.2 The Arduino Language

A program, or *sketch* as it is called in the Arduino environment, consists first and foremost of two functions: `void setup()` and `void loop()` (see listing 3.1). `setup` is called first whereafter `loop` is called repeatedly. The Arduino language is based on C++ [22], but the IDE makes some small changes to the files before compiling with the avr-g++ compiler. Arduino generates function prototypes for all functions in a file and also adds an include statement that includes Arduino.h, see listing 3.2. Arduino.h defines constants like `HIGH`, and `LOW`. Arduino.h also adds prototypes for functions like `pinMode` and `digitalWrite` which are defined in the Arduino standard libraries. There are several libraries for the Arduino language which implement some of the interaction with the hardware platform as well as interactions with external modules like motors and LCDs [23].

```
1  int led = 13;
2
3  void setup() {
4    // initialize the digital pin as an output.
5    pinMode(led, OUTPUT);
6  }
7
8  // the loop routine runs over and over again forever:
9  void loop() {
10   digitalWrite(led, HIGH);   // turn the LED on
11   delay(1000);               // wait for a second
12   digitalWrite(led, LOW);    // turn the LED off
13   delay(1000);               // wait for a second
14 }
```

Listing 3.1: The Arduino example that blinks an LED

```
1  #line 1 "Blink.ino"
2  #include "Arduino.h"
3  void setup();
4  void loop();
5  #line 1
6  int led = 13;
7
8  void setup() {
9    // initialize the digital pin as an output.
10   pinMode(led, OUTPUT);
11 }
12
13 // the loop routine runs over and over again forever:
14 void loop() {
15   digitalWrite(led, HIGH);   // turn the LED on
16   delay(1000);               // wait for a second
17   digitalWrite(led, LOW);    // turn the LED off
18   delay(1000);               // wait for a second
19 }
```

Listing 3.2: An Arduino example that blinks an LED, after being processed by the arduino "compiler". Notice that prototypes for each function is added on line 3 and 4, and Arduino.h is also included on line 2.

3.1.3 Using Classes as Abstractions of Hardware Components

Because the Arduino language supports classes it is possible to create and use classes as abstractions for hardware components. An example of this is shown in listing 3.3 which is an example for the liquid crystal library from [24].

```
1  [. . . ]
2  // include the library code:
3  #include <LiquidCrystal.h>
4
5  // initialize the library with the numbers of the interface pins
6  LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
7
8  void setup() {
9    // set up the LCD's number of columns and rows:
10   lcd.begin(16, 2);
11   // Print a message to the LCD.
12   lcd.print("hello, world!");
13 }
14
15 void loop() {
16   // set the cursor to column 0, line 1
17   // (note: line 1 is the second row, since counting begins with 0):
18   lcd.setCursor(0, 1);
19   // print the number of seconds since reset:
20   lcd.print(millis() / 1000);
21 }
```

Listing 3.3: A *Hello World!* example for the liquid crystal library [24].

Using an object oriented approach as in listing 3.3 makes it possible for beginners to control hardware components without knowing the underlying implementation of the interaction with these components. In listing 3.3 this works by initialising an instance of the class in line 12 and use instance methods of this class to perform different actions on the LCD. Line 6 demonstrates a difficulty when using a constructor without named parameters to describe the mapping between the Arduino and the LCD. The pin numbers passed to the constructor must be in the correct order and correspond to the physical connections. To help with this, the example provided by Arduino comes with a drawing showing how to connect the Arduino with the LCD [24].

3.2 Alternative Programming Languages for Arduino

Besides the official Arduino programming language described in 3.1.2, there are alternative programming languages that can be used as source language for Arduino. In this section, we test and analyse how the languages: AVR assembler, AVR C, Java, and JavaScript, at different levels of abstraction, support interaction with the Arduino hardware platform. To compare these languages, we implement a simple LED blinking program for each language, like we did for the Arduino language in section 3.1.2.

3.2.1 Low Level Programming in AVR Assembler

From section 3.1, we know that Arduino is based on the ATmega328P MCU. The ATmega328P is part of the AVR family MCUs [20] and we can therefore develop software for Arduino, using the AVR assembler language. The Assembler languages offer words called mnemonics for machine code instructions to make it easier for the programmer to remember instructions at a low level of abstraction [25]. Most mnemonics abbreviate the action performed by the instruction like `brne` for

branch if not equal and `sbci` for *subtract immediate with carry*. There are about 120 AVR assembler instructions in the instruction set [26].

Based on the AVR assembler instruction set in [26], we can implement the blink example program in AVR assembler as shown in listing 3.4. The shown AVR assembler program in 3.4 is inspired by the disassemble in appendix A. The disassemble in appendix A is based on the binary machine code for the blink example program in C, shown in listing 3.8.

```
1     .global main
2  main:
3     sbi     0x04, 5     ; set led pin as output
4  loop:
5     sbi     0x05, 5     ; turn led on
6     call    delay       ; wait 1 second
7     cbi     0x05, 5     ; turn led off
8     call    delay       ; wait 1 second
9     rjmp    loop        ; loop again
10 delay:
11    ldi     r18, 0x00   ; initialise counter to 3.2M
12    ldi     r19, 0xD4
13    ldi     r20, 0x30
14    subi    r18, 0x01   ; decrement counter by 1
15    sbci    r19, 0x00
16    sbci    r20, 0x00
17    brne    .-8         ; repeat decrement until counter == 0
18    ret
```

Listing 3.4: Shows the blink example program for Arduino in AVR assembler. Compiled flash memory usage: 164 bytes.

I/O Registers and Digital Output

Analysing the blink example written in assembly yields facts about how to control the hardware pins at a low level of abstraction. At line 3 in listing 3.4, we use the `sbi` instruction to set the bit at index 5 to 1 in the `DDRB` register at address 4. The `DDRB` register is a data direction register where the bit at index 5 determines if the pin `PB5` acts as input or output [27, p. 91]. We want to control pin `PB5` as it is connected to the LED on the Arduino Nano [28] [29]. In this example, we want pin `PB5` to act as output by setting the bit value to 1. The next instruction at line 5 we use `sbi` again to set the bit at index 5 in the `PORTB` data register at address 5 to turn on the LED. The value of the bit at index 5 in the `PORTB` data register determines the output voltage of pin `PB5` [27, p. 82 and 91]. If the bit is set to 1, the output voltage is 5. If the bit is set to 0 then there is no output voltage. We use `PORTB` data register again in line 7 to clear the bit to turn off the LED using the `cbi` instruction.

Loop and Delay

Inspired by the disassembly in appendix A, we create the loop construct by using the `rjmp` instruction at line 9, in listing 3.4, to repeat turning the LED on and off by doing a relative jump to the second instruction at line 5. To implement the 1 second delay between the LED blinking we refactor the delay implementation in appendix A by creating a subroutine called delay shown at

lines 10-18. From section 3.1, we know that the Arduino executes 16 million cycles per second and we can therefore construct a delay of 1 second, by executing instructions with a total of 16 million cycles. We do this by storing a counting value starting at 3.2 million using the three registers r18, r19 and r20. At line 14-17, we then repeatedly decrement the counting value by 1 while the counting value does not equal 0. This means that lines 14-17 are executed 3.2 million times. From [26], we know that the subtract instructions in line 14-16 execute in 1 clock cycle each and the branch instruction in line 17 executes in two clock cycles when the delay counter is greater than 0. Therefore, the total clock cycles of execution at lines 14-17 after entering the delay subroutine are the following.

$$\begin{aligned} n_{cycles} = &\quad 3.2 * 10^6 * 5 \\ clock\ cycles = &\quad 16 * 10^6\ clock\ cycles \end{aligned}$$

From this we can calculate and show the that the delay is 1 second.

$$t_{delay} = \frac{n_{cycles}}{f_{cycles}} = \frac{16 * 10^6\ clock\ cycles}{\frac{16 * 10^6\ clock\ cycles}{1\ second}} = 1\ second$$

In these calculations, we do not include clock cycles for instructions at lines 11-13 as well as the `call` and `ret` instructions for the delay subroutine, because the number of clock cycles in these instructions are negligible in this example as they are only executed once per delay call.

Compiling and Uploading the Blink Example Program

Saving the blink example program in listing 3.4 as `blink.s`, enables us to use the avr-gcc compiler to assemble the program with the command in listing 3.5.

```
1 avr-gcc -mmcu=atmega328p .\blink.s -o .\blink.out
```

Listing 3.5: Shows the command for assembling the blink example program for Arduino [30]

This generates a binary `blink.out` file including the binary machine code for the blink example. To upload this program to the Arduino, we first create an ASCII encoded hex file using the command as shown in listing 3.6.

```
1 avr-objcopy -O ihex -R .eeprom .\blink.out .\blink.hex
```

Listing 3.6: Shows the command for converting to hex file [31]

This hex file is then passed to AVRDUDE that writes the machine code to the Arduino via USB communication with a command as shown in listing 3.7.

```
1 avrdude -C "C:\Program Files
    ↪ (x86)\Arduino\hardware\tools\avr/etc/avrdude.conf" -v
    ↪ -patmega328p -carduino -PCOM3 -b115200 -D -Uflash:w:blink.hex:i
```

Listing 3.7: Shows the command for writing the hex file to the Arduino [32]

Using AVR assembler as Source Language for Arduino

There are reasons to use AVR assembler as source language for Arduino. One reason is to use assembler code to ensure which machine instructions are executed. An example of this is the implementation of the 1 second delay in the blink example where we need to know exactly which instructions are executed to wait 16 million clock cycles. A similar reason to use assembler is in situations where a higher level language like C does not support the same executions as assembler code [33]. A disadvantage of using AVR assembler code as the source language is that the programmer needs to remember mnemonics for the available assembler instructions to write and read programs. Remembering 122 assembler instructions and the available operands can be a difficult task. Another disadvantage is that AVR assembler instructions for the Arduino mainly support arithmetical instructions `add`, `subtract`, and `multiply` on operands with 8 bits of precision except the instructions `adiw` and `sbiw`, which can add or subtract a constant value from a pair of registers representing a 16 bit integer [26]. Another limitation of AVR assembler is that it does not support floating point number arithmetic [26]. To do arithmetical operations on larger values we can use multiple registers and perform arithmetical operations with carry, like we do in the blink example program at figure 3.4 line 14-16. In the next section we will see how these disadvantages can be addressed using the C programming language.

3.2.2 The C Programming Language

The C programming language has several data types like `char`, `short`, `int`, and `long`, as well as control structures like `if`, `if-else`, `for`, and `while`. These constructs, and the ability to create arrays, structs, and functions enable the programmer to develop software at a more abstract level than assembler. Listing 3.8 shows how the blink example program can be implemented in C using AVR Libc libraries for AVR-GCC.

```
1  #define F_CPU 16000000L     //define clock frequency
2  #include <avr/io.h>
3  #include <util/delay.h>
4
5  void main() {
6    DDRB |= 1<<PIN5;          //set led pin as output
7    while(1) {                //loop
8      PORTB |= 1<<PIN5;       //turn led on
9      _delay_ms(1000);        //wait 1 second
10     PORTB &= ~(1<<PIN5);    //turn led off
11     _delay_ms(1000);        //wait 1 second
12   }
13 }
```

Listing 3.8: Blink program in AVR C. Compiled flash memory usage: 176 bytes.

Libraries and Preprocessor Directives

In the first three lines of the blink example program in C, as seen in listing 3.8, we use what is called preprocessor directives, which are identified by the octothorpe prefix. Preprocessor directives are used by the C preprocessor to make text replacements before compilation of the program [34]. At line 1, we use `#define` to create a symbolic constant for the clock frequency used in the `util/delay.h` standard library that implements the `_delay_ms` function used in line 9 and 11. To interact with I/O

registers, we include the `avr/io.h` standard library that defines the symbols `DDRB`, `PORTB`, and `PIN5`. To understand what these symbols represent we can invoke the preprocessor with the following command in listing 3.9.

```
avr-gcc -mmcu=atmega328p -E blink.c
```

Listing 3.9: Shows the command for preprocessing the blink example program in C [30]

Using the command in listing 3.9 we get the output shown in listing 3.10.

```
[...]
void main() {
  (*(volatile uint8_t *)((0x04) + 0x20)) |= 1<<5;
  while(1) {
    (*(volatile uint8_t *)((0x05) + 0x20)) |= 1<<5;
    _delay_ms(1000);
    (*(volatile uint8_t *)((0x05) + 0x20)) &= ~(1<<5);
    _delay_ms(1000);
  }
}
```

Listing 3.10: Shows the main function of the blink example program in C after preprocessing

Addressing I/O Registers In C

As shown in listing 3.10, the `DDRB` and `PORTB` symbols are replaced by dereferenced pointers to the memory addresses 0x24 and 0x25 respectively. This is different from the addresses 0x04 and 0x05 used in the AVR assembler version of the blink example in listing 3.4. The reason for this can be found in the data-sheet for the ATmega328P: *"I/O Registers within the address range 0x00-0x1F are directly bita-ccessible using the SBI and CBI instructions. [...] When using the I/O specific commands IN and OUT, the I/O addresses 0x00-0x3F must be used. When addressing I/O Registers as data space using LD and ST instructions, 0x20 must be added to these addresses."* [27, p. 21]. This means that the compiler needs to consider which address to use for the different I/O registers and add the 0x20 offset depending on how the I/O registers are used.

Bitwise Operations In C

The preprocessed blink example in listing 3.10 reveals how to change the bit at index 5 in `DDRB` and `PORTB`. To set the bit at index 5 we use the compound assignment $|=$ as shown in line 3 and 5 with the expression $1 << 5$ as right hand side. $1 << 5$ is another way of writing literals 32, $0x20$, or $0b00100000$, which are all evaluated to the integer represented in binary as all zeros except the bit at index 5 which is 1. Listing 3.11 shows how *binary or* between an I/O register like `PORTB` and $1 << 5$ sets the bit at index 5 in DDRB without changing other bits.

```
1<<5:             00100000
PORTB:            abcdefgh
PORTB | 1<<5:     ab1defgh
```

Listing 3.11: Shows how the binary or (|) operation can set a single bit in a register where a, b, ... , h represents the values of the 8th, 7th, ... , 1st bit in `PORTB`

To clear a bit in an I/O register we can use a combination of the *left shift* («), *binary not* (~) and *binary and* (&) as shown in listing 3.12.

```
1<<5:             00100000
~(1<<5):          11011111
PORTB:            abcdefgh
PORTB & ~(1<<5):  ab0defgh
```

Listing 3.12: Shows how the binary and (&) operation can clear a single bit in a register where a, b, ... , h represents the values of the 8th, 7th, ... , 1st bit in `PORTB`

Even though the blink example in C shown in listing 3.8 uses multiple bitwise operations to modify a bit in an I/O register, the machine code generated by avr-gcc might be a single set bit instruction `sbi` or clear bit instruction `cbi` as shown in appendix A line 44 and 54.

Conditions and Expressions

If we look at the while statement in listing 3.8 line 7, we see that the condition for the while statement is the expression '1'. This is a valid condition because the conditions for iterative control structures in C are expressions which are considered true if the value is not 0. If it is 0, it is considered false [35, p. 150]. This means that an expression like $5 - 10 * 42$ used as a condition evaluates to true because $-415 \neq 0$. Because assignments in C are also expressions we can use assignments as conditions for control structures. Listing 3.13 shows an example of an assignment condition for an if statement in C.

```
1 void main(void) {
2     int a = 0;
3     if(a = 0) {
4         [. . . ]
5     }
6 }
```

Listing 3.13: Shows assignment as condition for if statement.

Listing 3.13 shows an example of a condition that might confuse some programmers. A programmer might read the condition in line 3 intuitively as *a equals 0* and because we initialised a to 0 in line 2 the programmer might think that the condition is true. But because the condition is actually an assignment, and the value of an assignment is the value of the right hand side, which is 0, then the condition is false.

Using C as Source Language for Arduino

An advantage of using C as source language for Arduino instead of AVR assembler is that it supports more data types and control structures and thereby enable the programmer to develop software at a more abstract level while still maintaining the ability to control the hardware. A disadvantage of using C for Arduino is that if you do not have any previous knowledge about bitwise operations on hardware registers, it might be difficult to understand that line 10, in listing 3.8, turns off the LED by clearing the bit at index 5 in the I/O register `PORTB`. Using three bitwise operations to clear a single bit in a register seems complex, when it might be compiled down to a single `cbi` assembler instruction, as shown in appendix A line 54.

3.2.3 Java

In the previous sections, we have analysed the imperative languages AVR assembler and C. In this section we will now look at the object oriented programming language Java, which enables the programmer to create more abstract code using classes. There are several projects that attempt to make microcontrollers programmable with Java [36] [37] [38]. To test Java for Arduino we use HaikuVM. HaikuVM is described as *"[...] a JAVA VM for micro controllers. It implements all JAVA byte codes (without 'invokedynamic')."* [37]. Listing 3.14 shows the blink example for Arduino which comes with the HaikuVM tools.

```
1  package processing.examples._01_Basics;
2
3  import static processing.hardware.arduino.cores.arduino.Arduino.*;
4  [...]
5  public class Blink {
6      // The status LED of the ARDUINO will blink.
7
8      static byte ledPin = 13;            // LED connected to digital
           ↪ pin 13
9
10     public static void setup() {
11         pinMode(ledPin, OUTPUT);        // sets the digital pin as
               ↪ output
12     }
13
14     public static void loop()           // run over and over again
15     {
16         digitalWrite(ledPin, HIGH);     // sets the LED on
17         delay(1000);                    // waits for a second
18         digitalWrite(ledPin, LOW);      // sets the LED off
19         delay(1000);                    // waits for a second
20     }
21 }
```

Listing 3.14: Shows the blink example program in Java the comes with HaikuVM [37]. Compiled flash memory usage: 5794 bytes.

At line 3, we import a Java port of the Arduino.h library to be able to use functions like `pinMode` and `digitalWrite`. The blink example in listing 3.14 uses a similar structure as the blink example in the Arduino language, shown in listing 3.1. Both use the `setup` and `loop` functions.

Classes and Threads

We will now show some of the capabilities that Java offers by creating a more object oriented version of the blink example. Instead of using `digitalWrite` directly from the class `Blink`, we can instead implement a new `Led` class as shown in appendix B, and use this class to implement the program shown in listing 3.15.

```
1  [. . . ]
2  public class Blink {
3      static Led led;
4
5      public static void setup() {
6          Serial.begin(9600);
7          led = new Led(13);
8          led.blink(1000); //Start blinking thread with interval of 1
               ↪ second
9      }
10
11     public static void loop() {
12         //Do other things while Led is blinking on a separate thread
13         Serial.println("Led is " + (led.isOn() ? "on" : "off"));
14         delay(1000);
15     }
16 }
```

Listing 3.15: Shows an extended blink example program in Java using the Led class in appendix B. Compiled flash memory usage: 8760 bytes.

Because the `blink` method of the `Led` class is using a separate thread to turn the LED on and off, as shown in appendix B, it is possible to execute other statements concurrently. In this example we send messages about the LEDs current state to the connected PC while the LED is blinking. A sample of the messages received from the Arduino is shown in figure 3.1.

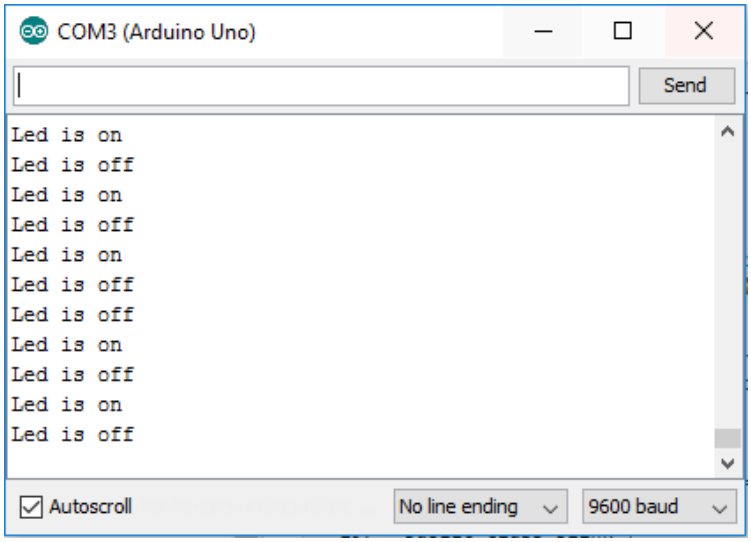


Figure 3.1: The figure shows the Arduino serial monitor that communicates with the program in listing 3.15.

As shown in figure 3.1, the messages received from the Arduino are shifting between *"Led is on"* and *"Led is off"*. This is also expected because we insert a delay of 1000 milliseconds after each `Serial.println` call, in line 13, which is equal to the interval we pass to the `blink` method in line 8. But from figure 3.1, we can also observe that there is an occurrence of two equal messages in a row. This can occur if the execution time of each loop iteration, at line 11-15, is not equal to the execution time of each loop iteration in the `BlinkThread` of the `Led` class, shown in appendix B. Another reason could be that the execution is interleaved between the main thread and the `BlinkThread` in such a manner, that the `BlinkThread` executes instructions, that both turn the led on and off in between the time that two messages are send. This will result in two equal messages being sent.

Using Java as Source Language for Arduino

An advantage of using an object oriented language like Java for Arduino, is that the programmer is able to use classes as abstractions for a hardware components like we did with the `Led` class in listing 3.15. Another advantage is that if the programmer needs to run code concurrently, it is possible to use multi-threading. A disadvantage of using Java for Arduino with a JVM, like HaikuVM, is the speed of execution. Based on a comparison by HaikuVM between Java and C, using the calculation of numbers in the Fibonacci sequence, the following was concluded: *"This (specific) plain C program is 35 (or 41) times faster than the corresponding JAVA program running with HaikuVM."* [37]. Therefore it is worth to consider if fast execution speed is a requirement for the project before choosing Java as source language for Arduino.

3.2.4 Interfacing with Higher Level Languages

In the previous sections, we have described languages that compile to a program that is running on the Arduino. In this section, we will describe how to develop software for Arduino using high level languages like JavaScript, C#, and Python by interfacing with an Arduino from a PC.

Interfacing with higher level languages works by having some firmware running on the Arduino, like Firmata [39], which basically enslaves the Arduino and uses its input-output capabilities. This

means that we can have a large and complex program running on a faster PC and only use the Arduino for input and output. This also means that any language can be run if it can make calls to the Firmata API [39].
In this sectionm JavaScript is used as an example of development software for Arduino in a dynamic typed language. In the example, we use Johnny-five.IO [40], which works as an interface between JavaScript and Firmata and also provides some libraries for the Arduino hardware platform.

```
1 var five = require("johnny-five");
2 var board = new five.Board();
3
4 board.on("ready", function() {
5   var led = new five.Led(13); //initialise led
6   led.blink(1000);            //toggle led every 1 second
7 });
```

Listing 3.16: Shows the blink example program for interfacing with Arduino in JavaScript using Johnny-Five

The JavaScript example starts by getting an instance of the Johnny-Five API. In line 2, It connects the Arduino and we are ready to interface with it. In line 4, we define the "ready" function in the Arduino, which works the same way as in the Arduino IDE, but still runs on the PC. We tell it to initialise the LED on pin 13, and then we tell it to blink/toggle every 1000 ms.

3.2.5 Comparing Languages for Arduino

In the previous sections 3.2.1, 3.2.2, 3.2.3, and 3.2.4, we implemented and analysed programs which turn the Arduino's LED on and off with an interval of one second. From the analysis of these programs we can now compare how AVR assembler, C, Java, and JavaScript can be used to develop software for Arduino.

Level of Abstraction

A key difference between the tested languages is the level of abstraction. Starting at a low level of abstraction, is the AVR assembler, where the machine instructions executed are known and therefore we can do precise timing like in the implementation of the delay subroutine in listing 3.4. Next is the C programming language, where we do not know exactly which machine instructions that are executed, because it depends on how the compiler translates the different higher level constructs into machine code. Even though C supports `struct` and union data types, it is not possible to use the same object oriented abstractions, which can be achieved in Java and JavaScript, shown in section 3.2.3 and 3.2.4.
Using an abstraction like the `Led` class in Java instead of bitwise operations on I/O registers in C, makes development easier for the target group in this project, because they are familiar with object-oriented programming but not bitwise operations on registers. Therefore, if we focus on designing a new programming language for Arduino, it makes it possible for beginners to control the hardware platform. It would be a solution to design an object-oriented language, like Java, or the current Arduino language and just add standard libraries. These could include classes and functions for different hardware components and hide operations on registers from the programmer. Seen from an educational point of view, this approach introduces some problems. If the programmer is only able to control the hardware at a high level of abstraction by using classes made by experts, the programmer does not learn how to control the hardware at a concrete level i.e. doing operations

on registers. Therefore, the programmer is challenged if he encounters a situation where classes or libraries have not yet been implemented to fit his project.
To make it easier to understand how to control hardware at a concrete level, a solution could for instance be to find a simpler way to change bits in a register, without the need to use two or three bitwise operators like in C. In the AVR assembler and C blink programs we refer to the LED pin as the bit at index 5 in `DDRB` and `PORTB` whereas in the Arduino and Java implementation we refer to the more abstract physical label of the pin i.e. the number 13 which the on-board LED is connected to. From a programmers perspective it is easier to use the pin number labelled on the Arduino than bits in registers, because labelled pin numbers are directly visible to the programmer.

Memory Usage

Because the Arduino is constrained to 32 kB of programmable flash memory, we will now compare the memory usage of programs in the tested languages. The JavaScript blink example is not included because it is running on the PC and not directly on the Arduino. Table 3.2 shows the flash memory usage for the blink program and the bare minimum programs in AVR assembler, C, Arduino, and Java.

Language	Blink program	Bare minimum program
AVR Assembler	164 bytes	132 bytes
C	176 bytes	134 bytes
Arduino	928 bytes	444 bytes
Java	5794 bytes	4622 bytes

Table 3.2: Shows the flash memory usage for the blink program and the bare minimum program. (see appendix M for how these numbers are obtained).

As shown in table 3.2, there are significant differences in flash memory usage of the blink program depending on the source language. We have included the memory usage of the bare minimum program i.e. a program without any statements and included libraries as a reference. The reason that the bare minimum in AVR assembler and C uses 132 and 134 bytes when no assembler instructions or statements are compiled, is primarily that avr-gcc inserts an interrupt vector, as shown in appendix A, to ensure that hardware interrupts are jumping to a valid instruction, even if an interrupt function is not defined. The memory usage of the bare minimum program in Arduino can be explained by listing 3.2, which shows how Arduino code is compiled to C++ code including the Arduino.h library. Finally is the bare minimum program in Java which uses at least ten times more memory than in the other languages. A reason for the relatively large memory usage, is that the Java Virtual Machine HaikuVM is included.

Assuming that the memory usage of the bare minimum program is the offset for all other programs' memory usage, we can subtract this offset from the memory usage of the blink program, to better compare the specific memory usage for the blink program, as shown in table 3.3.

Language	Blink program
AVR assembler	32 bytes
C	42 bytes
Arduino	484 bytes
Java	1172 bytes

Table 3.3: shows the flash memory usage for the blink program minus the bare minimum program's memory usage in the languages: AVR assembler, C, Arduino, and Java.

Comparing the specific memory usage for the blink program, we see in 3.3 that the object-oriented languages Arduino and Java both uses the more abstract representation of the led pin as the number 13, but uses significantly more memory than the imperative languages AVR assembler and C, which use registers and bitwise operations. Another observation is that the difference between AVR assembler and C is relatively small compared to the difference between AVR assembler and Arduino. Therefore choosing C as source language instead of AVR assembler may not have a significant impact on the memory usage compared to using Arduino or Java.

3.3 Problem Considerations

Based on the analysis of the Arduino hardware platform and the languages for Arduino that have been tested and analysed previously in this chapter, we now describe different considerations of creating a new programming language for Arduino. From section 3.1.1, we know that some of the major constraints in Arduino development is memory, execution speed, and debugging capabilities, which are relevant to take into account.
Another consideration is the way the language interacts with the hardware. From section 3.2.1, we know that, at hardware level, the hardware pins can be controlled e.g. by setting or clearing bits in the I/O registers using the AVR instructions `sbi` (set bit) and `cbi` (clear bit). A new language must decide how to represent low level instructions and at what abstraction level. In C, one can represent instructions on I/O registers as bitwise operations on registers located at specific memory addresses, whereas in the Arduino and Java example program, a more abstract function `digitalWrite` is available to control pins referred by the number labelled beside the pin on the Arduino board. When creating a new language for Arduino we must also consider how the programmer is able to use and implement abstractions of hardware like an LED or an LCD while still considering the memory usage of programs written in the language in order to respect the memory constraint of the Arduino.
In this project we want to make a programming language for programmers new to embedded programming and for that reason we want programmers to understand how to interact with hardware but without having to use unnecessary complex operations. The main focus of this project is therefore how we can make a programming language that allows interacting with hardware components and how to represent these hardware interactions in the language. These considerations will form the basis for a problem statement and for the design of a new language for Arduino.

Chapter 4

Problem Statement

Based on the considerations of the problem, described in section 3.3, and the study curriculum for this project, defined in [3, p. 26-27], we can now define the problem statement for this project. *How can a programming language for Arduino be developed for programmers with no previous experience in embedded programming?*

- How can the programmer control hardware components on an Arduino board?
- How can the programming language support abstractions of hardware components?
- How can syntax and semantics be defined for the programming language?
- How can a translator be implemented for the programming language to enable execution on an Arduino?

Chapter 5

Theory

The aim of this chapter is to explain the theory used throughout this project to make the Tang programming language. The theory will cover some aspects of: the main four programming language paradigms, syntax, semantics, compilers, interpreters, and the visitor pattern. The purpose of covering these topics is to provide the reader with the theory needed for understanding the remainder of this paper.

5.1 Programming Languages

In this section, we describe what a programming language is and explain four criteria for evaluating a programming languages. This theory will be used in chapter 6 as a basis for discussing which design criteria are most relevant for the design of the Tang programming language.

Generally speaking, a language consists of sentences, which are a series of symbols over an alphabet Σ [41, p. 136]. The syntax rules define which of these strings is part of the language [41, p. 136]. Consider a natural language like the English over the alphabet $\Sigma = \{a, b, c...z\}$ thus we can define the string, w, to be part of the English language if: $\{w \in \Sigma^* |\ w$ complies with English syntax$\}$. A programming language is "a technical language in which computer programs are written." [42].

When evaluating a programming language, certain criteria may be considered. The following section about language evaluation criteria will, based on [41], describe four language evaluation criterias readability, writability, reliabity and cost.

5.1.1 Language Evaluation Criteria

This section about language evaluation criteria is based on the book Concepts of Programming Languages by Robert W. Sebesta [41, ch. 1.3] and will, in accordance with this book, therefore describe four different criteria: readability, writability, reliability, and cost. The theory in this section will be used later in section 9.4 to evaluate Tang in comparison with other languages for Arduino.

5.1.2 Readability

Readability is about *"[...] the ease with which programs can be read and understood."* [41, p. 31]. If readability is high, the maintainability of a program tends to be higher. For a programming language to have readability it must have some of the following characteristics:

Overall Simplicity If a certain programming language has a large number of basic constructs, it can be hard to know all of its features. Readability problems may occur, when one programmer writes a program using one subset of features in a programming language, and another programmer who knows another subset has to read and understand that program.
Other characteristics that contribute to the readability of a program could be feature multiplicity, i.e. being able to do some operation in multiple different ways.
Operator overloading, if used incorrectly or illogically can also cause problems when reading a program.

Orthogonality *"Orthogonality in a programming language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language."* [41, p. 33]. In addition to this, it should be possible to combine every primitive type in a valid way. Lack of orthogonality leads to exceptions of language rules [41, p. 33]. However, too much orthogonality can also lead to unnecessary complexity, since it allows a large number of combinations.

Data Types Having sufficient data types for describing the variables in a suitable manner also contributes to the readability of a computer program. An example of a lack of sufficient data types could be when having to describe a Boolean type with an integer, where 0 is `false` and 1 is `true`. The statement `isActive = 1` is not as meaningful as `isActive = true`.

Syntax Design Syntactic design can also affect the readability of a programming language. Followingly described are syntactic design choices that influence readability.

- Special words. Having a long list of nested special words, e.g. **while**, **if**, and **for** may cause readability problems, when the statements are 'closed', if they are closed in the same way, e.g. by 'end' or '}'. A solution for this could be by using 'end if', 'end while' etc.
- Having statements somewhat describe their meaning in their name can also be helpful in regards of the readability of a program. When two language constructs are indistinguishable from each other, it may cause problems understanding them.

5.1.3 Writability

Writability is a measure of how easily a programming language can be used by a computer programmer to write programs. Readability and writability go hand in hand for many of the characteristics, since, when writing a program, you often go back and re-read some of the earlier parts of the program [41, p. 37].

Simplicity and Orthogonality As explained earlier, it can be a problem if a certain programming language has a large amount of constructs, since most programmers will only learn a subset of them, if there are too many. The right amount of orthogonality will help programmers solve complex problems only knowing a smaller set of constructs. Too much orthogonality, i.e. being able to combine a large number of primitive constructs, can lead to undiscoverable errors.

Expressivity Expressivity *"[...] means that a language has relatively convenient, rather than cumbersome, ways of specifying computations."* [41, p. 37]. This could e.g. be in C, where $count++$ is more convenient than $count = count + 1$.

5.1.4 Reliability

"program is said to be reliable if it performs to its specifications under all conditions." [41, p. 37].
Related to the reliability of a language is: type checking, exception handling, and aliasing.
A reliable program language has some kind of type checking, either by the compiler or at run-time. Failure to type check has often been the cause of program errors as it was with an earlier version of C where parameters given to a function was not checked whether it corresponded to the formal parameters of the function or not [41, p. 38].
Another aspect of the reliability of a programming language is the ability to catch and handle exceptions at run-time. This phenomenon is called exception-handling [41, p. 38]. In some programming languages it is not possible or impractical to detect run-time errors which means that the particular program will be terminated and control will be given back to the operating system in case of a run-time error that could have been fixed by the program [41, p. 622]. Exceptions allow the programmer to catch and handle these exceptions and thus not terminate the program which increases the reliability of a programming language because it, at run-time, can be ensured that the program performs to its specifications.
A programming language that supports aliasing is a language that allows two or more different variables to point to the same place in memory [41, p. 38]. Many languages have support for pointers which allows the programmer to have variables that refer to the same place in memory. In languages that support aliasing the programmer must be aware of side effects, e.g. changing a value in memory that is pointed to by multiple variables. Aliasing is considered dangerous and affects the reliability of a programming language if not used carefully.

5.1.5 Cost

Adding to the evaluation of a language is the total cost of using a language which according to [41, p. 39] is measured according to the following characteristics:

- Cost of training programmers to use the language (Readability)
- Cost of writing programs in the language (Writability)
- Cost of compiling programs in the language (Performance of compiler)
- Cost of executing programs written in the language (Execution speed)
- Cost of the language implementation system (Language implementation system)
- Cost of poor reliability (Reliability)
- Cost of maintaining a program written in the language (Maintainability)cpc

The cost of training a programmer to write programs in a particular language is related to how orthogonal and readable a language is [41, p. 39]. In order for a programmer to learn to program in a particular language the programmer must be able to read and understand his own code. A program that is orthogonal and readable thus reduces the cost of training a programmer to write programs in a particular language.
The cost of writing programs in a language is a function of the writability of a language and is therefore also related to the expressivity and simplicity of a language. The original intention behind the design of some of the first high-level languages was to reduce the cost of writing a program in a language and to make code machine independent [41, p. 39].
A language with an inefficient compiler adds to the cost of using a particular programming language [41, p. 39].
Another desirable feature of a language is related to the execution of a language which, in large,

is influenced by the design choices of a language, thus languages that require many run-time type checks will, to some degree, hinder fast execution of a language.
The early interests for Java were helped by the fact that free compilers and interpreters were available for the language, and thus the language implementation system also plays a role in the total cost of using a programming language [41, p. 40]. For some programs, reliability is a key factor e.g. on aeroplane software, where the cost of not having a reliable programming language can be severe.
The last characteristic of a language is the cost of maintaining a language which is closely related to how readable the language is.

5.2 Programming Paradigms

In this section, we will describe the four main programming paradigms: the imperative, the object-oriented, the functional, and the logical programming paradigms [43]. These will be used in the design of Tang in chapter 6 to help decide which kind of language Tang is and thus which single or multiple paradigms Tang fits within.

A programming paradigm is *´´A pattern that serves as a school of thoughts for programming of computers"* [44] meaning that a programming paradigm is a specific way of thinking of computer programming and how to solve problems.

5.2.1 The Object-Oriented Paradigm

The object-oriented paradigm spans the languages that follow the concepts of object-oriented programming which according to [45, p. 281] is:

- Encapsulation and abstraction
- Subtypes
- Inheritance
- Dynamic method selecting

In object-oriented languages, objects can be created that encapsulate some data, and is only accessible within or through that class. Information hiding is also used as way to abstract over data, some languages support information hiding through private and public access modifiers [45, p. 281].
Subtypes are the principle of which we define a new type (the set of object instances for a class [45, p. 287]) `T` based on another type `S`. More specifically a class `T` is said to be a subtype of `S` when *"[. . .] every message understood (that is which can be received without generating an error) from objects of S is also understood by the objects of T."* [45, p.287].
Inheritance is the definition of new objects, which extend upon another object and thus inherit the behaviour of that object. Dynamic look-up is related to the fact that methods in object-oriented languages can be overridden and thus there can be multiple versions of a a method on a specific object, the dynamic look-up will ensure that the right method is called at run-time [45, p. 281].
Consider an example with two classes: `Rectangle` and `Square`, that both extend upon the `Geometric` class and override its method `CalculateArea`. Then the method can either be the area of a square or a rectangle depending on how it is declared (see listing 5.1).

```
1 Geometric s = new Square(2, 2);
2 Geometric r = new Rectangle(2, 5);
3 System.out.print(s.CalculateArea()); // printing 4 (square)
4 System.out.print(r.CalculateArea()); // printing 10 (rectangle)
```

Listing 5.1:

5.2.2 The Functional Paradigm

Functional programming languages are based on mathematical functions, and differ a lot from the imperative paradigm. It is said that "*[. . .] purely functional programs are easier to understand, both during and after development, largely because the meanings of expressions are independent of their context [. . .]*"[41, p. 658], making them more readable and reliable. In purely functional programming no variables or assignment statements are used. Instead programs consist of function definitions and function applications [41, p. 662]. Functional languages are said to be referential transparent meaning that given the same parameters to a function, the function always produces the same result [41, p. 662]. Many functional programming languages are not purely functional, but have imperative-style variables, assignments etc. [41, p. 662]. Because of language characteristics like lazy evaluation where expressions are not evaluated before its use [41, p. 74], programs written in a functional programming language will on average, according to [41, p. 704], roughly have a execution speed twice as high as that of the same program written in a imperative programming language.

5.2.3 The Logic Paradigm

"*Logic programming is characterized by programming with relations and inference.*" [46, p.35]. A logic program describes a problem to be solved and consists of axioms, rules and a goal statement [46, p.35]. Furthermore, logic programming is said to be non-procedural and thus does not define a procedure for how computations are made, but instead defines the structure of the result. Because of problems with efficiency and to avoid non-terminating loops programmers must sometimes, in current logical programming languages, use control statements [41, p. 745]. This approach breaks with the pure logical way of thinking where we only specify what we need to accomplish the result. Not how or in which order.

5.2.4 The Imperative Paradigm

The design of imperative languages is based on the Von Neumann architecture in computers. " *This is reflected in the design of the imperative languages, with*

- *states - representing memory values with changing values,*
- *sequential orders - reflecting the single sequential CPU, and*
- *assignment statements - reflecting piping." [46, p. 69].*

One of the major differences between the imperative paradigm and e.g. the functional paradigm is referential transparency. The idea of referential transparency does not hold for imperative languages due to side effects [47].
Another important idea in imperative languages is sequencing: *"Since the order of the bindings affects the value of expressions, an important issue is the proper sequencing of bindings."* [46, p. 69].

5.3 Syntax

The purpose of this section is to describe the theory relevant for defining the syntax of a language, more specifically what a language, tokens, and regular expressions are, which we use to make the lexer for the Tang language. We will then look at context free grammars which will be used in chapter 6 to formally specify the syntactical rules of the Tang language. Furthermore, we look into different grammar classes and their properties in order to determine which parsing technique will be used in the implementation of the Tang language.

5.3.1 Tokens

"[...] a token of a language is a category of its lexemes." [41, p. 135]. Given a string `s`, `s` consists of different lexemes which is the smallest syntactical unit for whom it is true that the unit is part of a language and thus complies with the syntax of the language [41, p. 134]. A token is thus a grouping of lexemes for which a syntactic rule can be defined. Such a rule can be described formally by e.g. regular expressions.

Regular Expressions

Regular expressions are expressions that are built up by regular operations to describe languages. Formally, given an expression `R`, it is said to be regular if it satisfies one of the following conditions, which are explained further below [48, p. 63-64]:

1. "`a` for some `a` in the alphabet Σ"
2. "ε"
3. "$\emptyset$"
4. "$(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions"
5. "$(R_1 \ o \ R_2)$, where R_1 and R_2 are regular expressions"
6. "(R_1^*), where R_1 is a regular expression"

Item 1 represents the language for some {a} in the language. Item 2 represents the language containing the empty string $\{\varepsilon\}$, and item 3 the empty language. Item 4-5 represents the languages that can be obtained by applying the regular operations concatenation or union to the languages R_1 and R_2. Item 6 represents the languages that can be obtained by using the Kleene star regular operator on a language R [48, p. 64]. In this project we will use the notation for regular operations defined in [49].

5.3.2 Context-Free Grammars

In this section, we will describe context free grammars (CFG) and Backus-Naur Form (BNF). BNF is a notation used for CFG's, which are notions used to specify the rules for strings within a language using the syntax of a specific language. We do this in order to provide the theory necessary to understand the notation for CFGs used in chapter 7, where we define the syntax for Tang.
A CFG is a way of defining the structure of sentences in a language and thus how they are built [1, p. 114]. CFGs are often used in programming language specifications as a formal way to define the syntax of a language, but are also used in various of parsing techniques and algorithms. A CFG, G, can formally be defined by the elements represented in the following 4-tuple: $G = (N, \Sigma, P, S)$ [1, p. 114].

- Σ represents a finite set of terminal symbols, which is found during the lexical analysis phase of the compiler.
- N represents a finite set of non-terminal variables in the grammar. The non-terminal symbols and terminal symbols must be disjoint: $N \cap \Sigma = \emptyset$.
- P represents a finite set of productions, which is a set of rewriting rules of the form A $\rightarrow$ $X_1 ... X_m$ where $A \in N$, $X_i \in N \cup \Sigma$, $1 \leq i \leq m$ and m ≥ 0.
- S represents the start symbol.

To distinguish terminals and non-terminals, we use the notation given in [1, p. 115]. This means terminal symbols are all lower case and names of non-terminals starts with an uppercase letter.

5.3.3 Backus-Naur Form

Instead of writing all terminals, non-terminals, and rules explicitly in a 4-tuple to define a CFG, we can instead simplify this by using a BNF. A BNF is a meta-language, and is used to denote the syntax of a programming language [41, p. 137-138].

A BNF consists of a set of productions of the form $V \rightarrow \alpha$ where V is a variable and α is an arbitrary number of non-terminals and/or terminals. Each production has a right-hand side (RHS) and a left-hand side (LHS) which are the symbols placed of the right-hand and the left-hand side of the $\rightarrow$ symbol.

Consider a simple example:

1. $S \rightarrow aS|\varepsilon$

In this example, S is the start symbol for the grammar and the LHS of the production. On the RHS of the production we find two terminal symbols contained in the alphabet Σ which for this simple language are "a", "ε". ε denotes the empty string, which will be denoted as *EPSILON* later in the report. Furthermore, we find the variable S on the RHS and since S is also the symbol on the LHS this rule is recursive. The '|' symbol denotes the meaning of 'or' stating that S can either derive $\Rightarrow$ 'a S' or ε. Alternatively this can be written as:

1. $S \rightarrow aS$
2. $S \rightarrow \varepsilon$

5.3.4 Grammar Classes

In this section, we describe LL and LR grammars used in the top-down and bottom-up parsing techniques respectively which are described in section 5.5.1. Furthermore, the properties of the grammars will be used in section 6.3 to determine which grammar is most suitable for the choices we make in this project.

LL Grammars

Given a grammar, G, then G is LL(k) if it is possible to make an LL(k) parser that uses a lookahead of k symbols to parse a string in the language described by G [1, p. 145]. Furthermore, a grammar is LL(k) if a parser uses a lookahead of k to determine which productions to apply [1, p. 146]. LL(1) grammars are a subset of the LL(k) grammars where $k = 1$ [1, p. 146]. An LL(1) grammar

has the property that the grammar is unambiguous because if the grammar was ambiguous, there would exist a string S such that S has two or more distinct left most derivations that can be used to derive the string which cannot be the case since we can predict which rule to apply using a lookahead of only one token [1, p. 161-162].

LR Grammars

"A Grammar is an LR Grammar if it can be parsed by an LR parsing algorithm" [50, p. 66], whereas a LR(k) parser given the next k symbols of input and the symbols to be replaced (the handle) will make the correct reduction [1, p. 190]. As seen in figure 5.1 all LL(K) grammars are a subset of the LR(K) grammars meaning that if a given parsing algorithm can parse a LR(K) grammar it can also parse an LL(K) grammar.

LALR As seen in figure 5.1 LALR grammars are between the LR(0) and LR(1) grammars. This means that a parser that can parse LALR grammars can also parse LR(0) grammars but not LR(1) grammars. LALR grammars are useful, since LR(1) parsers can have a large overhead of states it can transition to. *"With LALR (lookahead LR) parsing, we attempt to reduce the number of states in an LR(1) parser by merging similar states. This reduces the number of states to the same as SLR(1), but still retains some of the power of the LR(1) lookaheads."* [51].

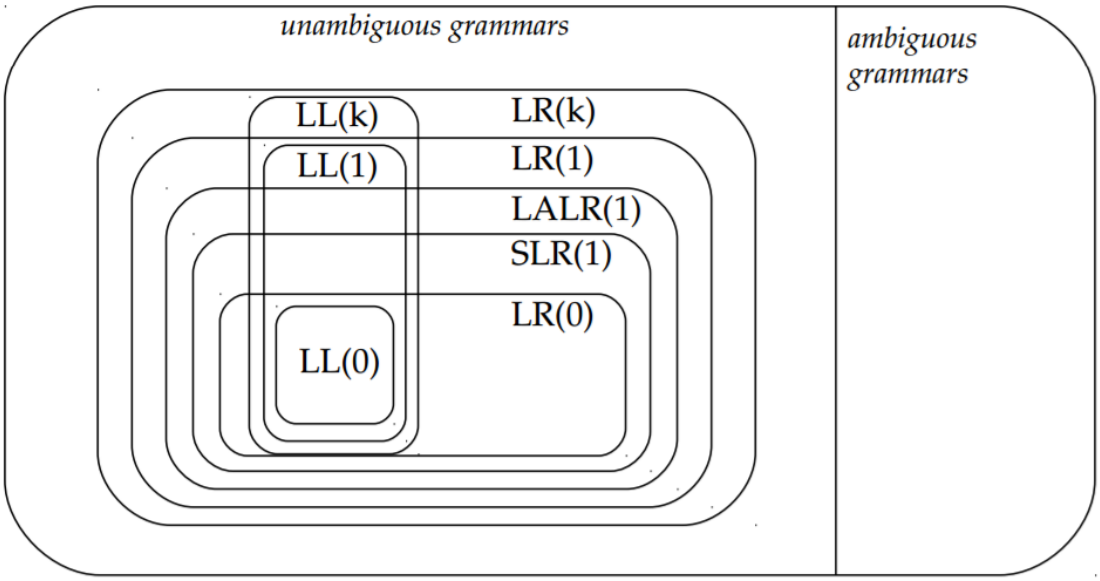


Figure 5.1: Hierarchy over different parsing techniques [52, p. 4].

5.4 Semantics

According to [53, p. 1] semantics *"[...] is the study of mathematical models of and methods for describing and reasoning about the behaviour of programs."*. When looking into the semantics of a programming language, we are only concerned with the behaviour of a running program and not with external things like if the language is user-friendly [53, p. 1]. In order to describe the behaviour of the program, an abstract syntax for a legal program must be defined. Such a grammar could be

a CFG which is described in section 5.3.2. The theory defined in this section will be used as a basis for understanding the semantic definitions for the Tang language in section 6.3.

5.4.1 Structural Operational Semantics

The structural operational semantics define the behaviour of a program by using a transition system [53, p. 1].
A transition system is a directed graph with a configuration that corresponds to different states in a program and with transitions between the states. If a configuration has no outgoing transitions it is called a terminal configuration. Furthermore, given two configurations a and b, there is a transition from a to b (a, b), if there is a direct edge from the a configuration to the b configuration.
Formerly, *"A transition system is a triple* $(\Gamma, \rightarrow, T)$ *where* Γ *is a set of configurations,* $\rightarrow$ *is the transition relation, which is a subset of* $\Gamma \times \Gamma$*, and* $T \subseteq \Gamma$ *is a set of terminal configurations."* [53, p. 30].

In [53, p. 30] two kinds of operational semantics are described: big-step and small-step semantics.

Big-step semantics transitions (α, β) are computations that start in the α configuration state and end in the β configuration state, where β is a terminal configuration [53, p. 31].

Small-step semantics transitions (α, β) describe single steps of computations that are part of a larger computation and as such the β configuration is not always a terminal configuration.

A transition relation denoted by $\rightarrow$ is defined by a finite set of transition rules, which are rules defined by a small-step or big-step semantic notation [53, p. 40-41]. A transition rule can consist of two parts: premises and a conclusion. The rules are often written using fraction notation where the numerator is the premise to the rule (claims) and the denominator is the conclusion of a rule. A rule that has no premise is called an axiom [53, p. 40-41].
Consider a simple example from [41, p. 136] of a big-step transition rule for a plus operation:
"$\dfrac{s \vdash a_1 \rightarrow_a v_1 s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 + a_2 \rightarrow_a v}$ *where* $v = v_1 + v_2$" [41, p. 136]

Here the premises of the rule are given by the semantics in the numerator: $s \vdash a_1 \rightarrow_a v_1 s \vdash a_2 \rightarrow_a v_2$ which says 'given a state, s, the expression a_1 evaluates to v_1 and given a state s, the expression a_2 evaluates to v_2'. The conclusion is the denominator of the fraction: $s \vdash a_1 + a_2 \rightarrow_a v$ which together with the side note reads *"given a state* s*, the expression* a_1 *added with* a_2 *evaluates to* v *where* $v = v_1 + v_2$*.*

An execution of a statement might change the value of a variable, the following notation represents a change of state where statement S *is executed in state* s *which leads to state* s'*":* $\langle S, s \rangle \rightarrow s'$

5.4.2 Type System

The type system defines, for each syntactical category, in a abstract grammar for the language, a definition of the type judgement as well as a finite set of type rules that defines which type judgements are valid [53, p. 186]. Given a syntactical category for integer expressions, a type judgement could be defined as follows: $E \vdash a : int$, which states that a is of type `int` given the type bindings of the type environment, E [53, p. 186]. A rule for numerals could then be: $E \vdash n : int$ which states that a numeral n is of type `integer`. Like with the operational semantics the rules of a type

system does not have to be an axiom.

To keep track of the types of identifiers like variables and procedure names, a type environment, denoted E, is used. E is a partial function $E : \mathbf{Var} \cup \mathbf{Pnames} \rightharpoonup Types$ where E is the domain and the union of procedure names and variables is the range of the function [53, p. 190] [53, p. 23]. A type rule could for instance be: $E \vdash a : T$, which states that a is of type T given the type bindings of E [53, p. 186].
The type environment and environment-store model is related in the sense that if a variable x has type T then the location of x should contain a value of type T.
Type systems are used in section 7.2.3 to define the type system for Tang.

5.5 Compilers and Interpreters

In this section we will describe the theory behind two different types of translators: interpreters and compilers. The reason for doing so is to give an overview of the advantages and disadvantages of each, which will be used in section 6.7.1 to discuss whether an interpreter or a compiler is most suitable for Tang.

5.5.1 Compiler

A compiler translates a program written in a high-level languages such as C and Java into an object program written in a low-level language such as x86 machine code [54, p. 27]. The program in which the compiler itself is written is referred to as the implementation language [54, p. 27]. In order to translate a program into its semantic equivalent object-code the compiler goes through a series of steps as illustrated in figure 5.2. Each of these steps will be shortly described in order to get an understanding of the phases involved in crafting a compiler.

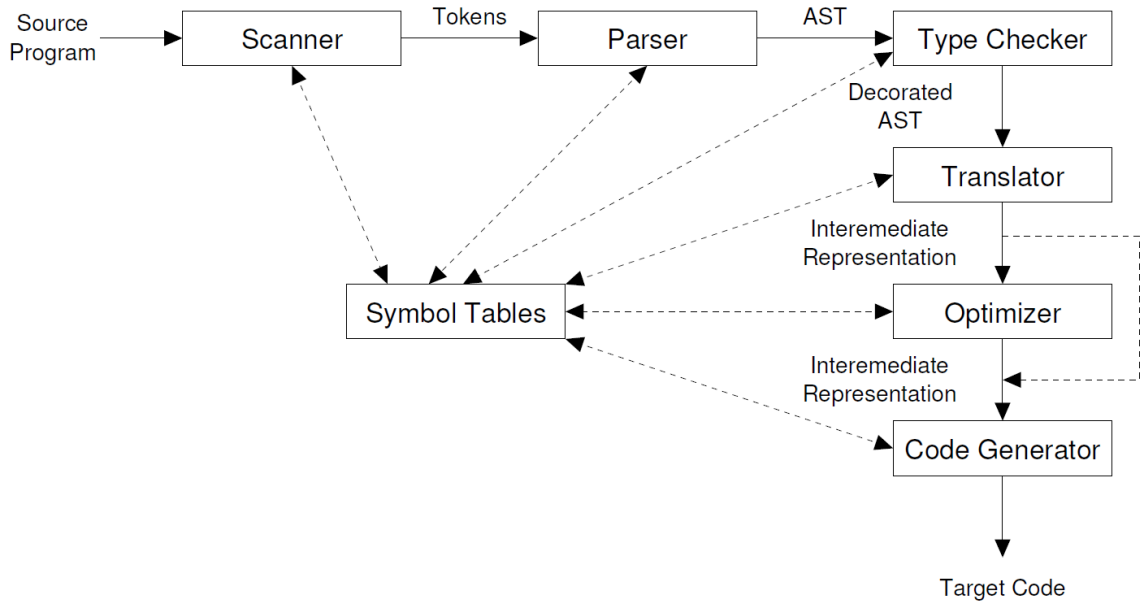


Figure 5.2: Organisation of a syntax-directed compiler [55, p. 53]

Lexer

The lexer is the part of the compiler that reads the source code and produces a string of tokens (see section 5.3.1 for description of tokens) [1, p. 58]

Parser

A parser takes the tokens generated by the lexer and parses them according to the formal syntax specification of the language [1, p. 16-17], which could be formally expressed through a CFG as described in section 5.3.2. In other words the parser checks if the input can be derived from the rules of the language. The result is shown in the form of a parse tree [1, p. 37]. The last process of the parser is to build an AST [1, p. 37], which is a semantically equivalent stripped down version of the initial parse tree where all unnecessary information in the parse tree has been removed [1, p. 45-46].

Type Checker

In the type checker, each AST is traversed in order to check that each construct in the tree is semantically correct e.g. that variables are declared before they are used (assuming this is a semantic rule of the language) [1, p. 17]. If the tree traversal does not produce any errors the AST is decorated with 'types'. If it does produce errors an appropriate error message can be shown [1, p. 17].

Translator

Assuming that the type checker has traversed the AST and correctly checked that each node is semantically valid, each node in the AST can be translated into an Intermediate Representation (IR) [1, p. 17]. The IR not only describes the overall structure of the program, but also captures the meaning of constructs and thus an IR of a 'while-loop' will express that such a construct in fact loops [1, p. 17].

Symbol Tables

The symbol table is a mechanism which can be accessed by all phases of the compiler. It contains information about the code being compiled, such as types, variables, procedures, and labels [1, p. 18].

Optimiser

In the optimiser phase, the IR code is optimised in order to increase the execution speed of the compiled program [1, p. 18]. A compiler might for instance eliminate unused computations.
Optimising can also take place after the code generation phase [1, p. 18]. An example of this is *peephole optimisation* [1, p. 18]. This form of optimisation looks at small samples of the code at a time and eliminates e.g. multiplication by 1 or addition by 0. Many compilers allow the user to skip optimisation, as it is not needed to generate target code, but it often speeds up programs significantly.

Code Generator

The code generator maps the IR onto the target language for the target platform. The complexity of the code generator can become high, as it requires consideration of many special cases [1, p. 19]. Besides having compilers that compile code which are run on the same machine on which the code is compiled, there exist a specific type of compilers known as cross-compilers. Cross-compilers run on what is called a host-machine and generate code for another machine, the target machine [54,

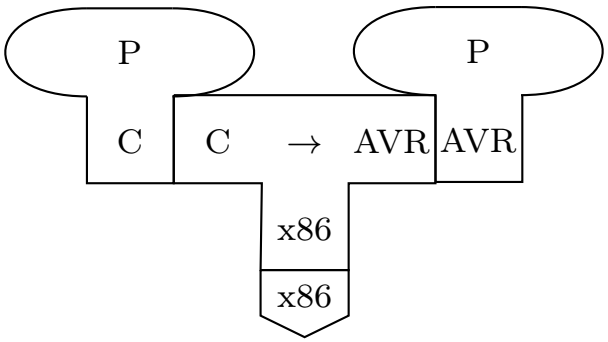


Figure 5.3: Tombstone diagram illustrating the process from C code compiled to AVR.

p. 31]. Cross-compilers are especially useful if the target machine has small memory capabilities [54, p. 31]. The tombstone diagram in figure 5.3 shows an example of a cross-compiler where C code is compiled to AVR machine code by a compiler written in x86 machine code and run on an x86 machine. The result of this compilation is a program written in AVR code that can run on an AVR machine.

5.5.2 Interpreter

An interpreter is, like a compiler, a language processor. Even though the end result of executing a program by compiling or interpreting is mostly the same, the actions behind the two language processors are undoubtedly different. This results in various advantages and disadvantages, which will be described below.

One of the major differences between compilers and interpreters is their way of translation. Compilers translate the entire input program to machine code, which will then be run at a speed that is determined by the machine on which it is run.
Interpreters, on the other hand, do not perform much translation. The interpreter executes the source program by fetching, analysing, and executing it immediately one line at a time [54, p. 34]. This allows the debugging process to pause at certain points in the code and read variable values which can be a big help. However, it may also bring "*[. . .] significant overhead. As execution proceeds, program text must be continuously reexamined.*" [1, p. 10]. In languages with fluid bindings, this can mean interpretation becomes almost 100 times slower than compiled code [1, p. 10].

In code where each instruction in a source program will not be executed multiple times it can be favourable to use an interpreter [54, p. 34]. This is because the overhead will be much reduced, and since it will only be executed once execution speed is not essential [54, p. 34]. It can also be favourable to use an interpreter, if each instruction can be analysed efficiently [54, p. 34]. If a program of high speed is of importance, if each instruction will be executed multiple times, or if instructions are inefficient to analyse, an interpreter may not be the ideal choice [54, p. 34].

5.6 Parsing Techniques

When creating a parser, some choices have to be made in advance. One of these choices regard parsing techniques.

Some of the most prominent and most frequently used are the parsers making use of the top-down and bottom-up techniques. These are called LL and LR, respectively. The first letter of both, *L*, denotes the reading and writing direction, left-to-right whereas the alternative is right-to-left. The second letter denotes if the parser makes use of a leftmost or rightmost parse.
The names of the techniques can also describe a number of lookahead symbols, which describe how many symbols should be read in addition to the current token, e.g. *LL(1)* describes a left-to-right, leftmost parse with a lookahead of one symbol [1].
The following sections will describe the two parsing techniques: *LL* and *LR*.

5.6.1 Top-down Parsing

As mentioned above, top-down parsers make use of left-most derivations.
"*Top-down parsers are in theory not as powerful as the bottom-up parsers [...] However, because of their simplicity, performance, and excellent error diagnostics, top-down parsers have been constructed for many programming languages, almost always using the recursive-descent approach.*" [1, p. 143].
Top-down parsers can be further categorised into recursive decent parsers and table-driven LL parsers.

Recursive decent parsers Recursive-descent parsers are parsers that "*[...] contain a set of mutually recursive procedures that cooperate to parse a string.*" [1, p. 144]. Furthermore, they are called recursive decent parsers because they are implemented by a collection of recursive procedures based on a specific CFG.

Table-driven LL Parsers A table-driven LL parser uses a table to determine which rule to apply. The table maps each pair of a non-terminal and a terminal symbol to a rule.

5.6.2 Bottom-up Parsing

"*[...] bottom-up parsers can handle the largest class of grammars that allow parsing to proceed deterministically (i.e. without backtracking).*" [1, p. 179]. This is illustrated in figure 5.1.
Bottom-up parsers are also called shift-reduce, because "*[...] the two most prevalent actions taken by the parser are to **shift** symbols onto the parse stack and to **reduce** a string of such symbols located at the top-of-stack to one of the grammar's non-terminals.*" [1, p. 180].

5.7 Visitor Pattern

In this section we describe the visitor pattern which is used throughout the implementation in chapter 8 to traverse trees.
As the size of a compiler grows, so does the number of node types. Because of this, a large number of classes have to be created which makes it difficult to grasp each node's functionality. According to [1, p. 264]: "*[...] modern software engineering principles dictate that the code for a phase should be written in a single class, and not distributed among the various node types [...]*". The visitor pattern is applicable when "*[...] you want to perform operations on these objects that depend on their concrete classes.*" [56, p. 333].

The visitor pattern works by implementing a general class that defines the visit method for the nodes in the tree [56, p. 334]. A concrete implementation of the general class is also created, this class defines a type of nodes in the tree and includes attributes valuable for the type of node as well as making sure the accept method is used in each type of node [56, p. 334].

Each type of concrete element needs its own visit method, this can be implemented as an interface or can be generated automatically based on a grammar.
The implementation of the visitors defines what each type of node has to do when being visited.
Finally, to execute the visitor, we have to enumerate through all nodes by visiting each [56, p. 335].
The visitor pattern makes use of double dispatching when using its accept method. This means, that invoking a method on an object with an abstract declared parameter makes the compiler know which two types of objects it is dealing with at run-time. "*Its meaning depends on two types: the Visitor's and the Element's.*" [56, p. 338].

Chapter 6

Design

This chapter will specify the design of the Tang language by first analysing the language design criteria and then prioritise them according to which are important for Tang in section 6.1. After this, the language paradigm will be asserted in section 6.2 followed by a specification of the features included in Tang in section 6.3. The syntax of the selected features is designed in section 6.4. In continuation of this, scope and type rules are specified in section 6.5.
The previously mentioned choices of features to be implemented in Tang need to be prioritised according to which features are the most essential. This is done by prioritising them according to a MoSCoW analysis which can be seen in section 6.6.
The last section of the design chapter describes the compilation process including the choice of translator as well as the target and implementation language which can be seen in section 6.7.

6.1 Language Design Criteria

Based on the language evaluation criteria described in section 5.1.1, we will, in this section discuss the importance of each criteria in the context of this project. The resulting prioritisation of the criteria will be used throughout this chapter when making decisions that will shape the Tang programming language.

6.1.1 Readability

We prioritise readability as being very important because it can be difficult to debug errors at run-time when developing software for microcontrollers, as described in section 3.1. By making the readability high, errors will be easier to find directly by reading the source program. In a language with high readability, reading the program to find errors is also viable since the programs are relatively small in size due to the memory constraint, explained in section 3.1. Readability is also very important because novices to embedded programming need to be able to easily read and understand a program they have not written themselves.

6.1.2 Writability

As mentioned in section 5.1.1, writability is concerned with the ease of which a programmer can write programs in a particular language. In correlation with the intended target group of Tang, we have decided to prioritise writability as being important, since programmers that have not previously programmed embedded systems should be able to easily write programs for the Arduino hardware platform. However, as stated in [41, p. 45], increased writability in form of expressivity comes at the cost of readability. Because of this, we have decided that increased writability of the

programming language should not happen at the expense of decreased readability and thus decided to prioritise readability over writability.

6.1.3 Reliability

When developing software for microcontrollers it is important to limit errors as they can be difficult to find at run-time. Furthermore, since programmers have full control of the hardware, an error could potentially break the physical hardware, if for example the Arduino controls a large current, a motor, or an otherwise safety-critical system. Because of this, we prioritised reliability as important in the design and implementation of the programming language developed in this project.

6.1.4 Cost

Tang's target group consists of experienced programmers that want to learn embedded programming, as described in section 1. This means that the cost of training programmers to use the language is important to keep to a minimum in order to move the focus from learning a new language to learning how to program embedded systems. Writability in our language is of concern since programmers that are new to embedded programming need to be able to use Tang without a lot of training. The main focus of this project is not for programmers to learn to use Tang, but rather use the language to program embedded systems. Programmers in the target group for Tang need to be able to read programs written in the language in order to learn how to program embedded and to understand how the language can be used to interact with underlying hardware components. For this reason, we prioritise readability over writability.
Compilation cost is not an important concern for this project, since the Arduino's flash-memory limits software size, as described in section 3.1, and flashing the software to the hardware will often take even longer than the actual compilation process. Apart from that, Tang will not be created to replace the Arduino language or AVR-C, but will be designed to be a bridge between low-level embedded programming and high-level general purpose languages. This means that Tang will not be practical for big and complicated projects.
Cost of executing programs is very important, since the Arduino is slow, compared to what most programmers are used to in a modern PC.
Reliability cost is a concern, as a failure at this level can cause critical mistakes as software updates are usually impossible or expensive to do and a failure will likely cause the product to be unusable or even dangerous or it could destroy expensive hardware. But since the language is aimed for programmers new to embedded programming we do not expect anyone to create critical systems in Tang.
The cost of the language implementation system is less important, because the price of the Arduino and compiler along with the availability of the compiler is not a concern of this project. Since readability and writability is high, maintainability automatically becomes high, but this is not a priority for us, just a side-effect.

Minimising cost of ...	Very important	Important	less important
Training programmers	x		
Executing programs	x		
Writing programs		x	
Reliability		x	
Compiling programs			x
Maintaining programs			x
Implementation System			x

Table 6.1: Shows prioritisation of the cost using a language.

6.1.5 Prioritisation

Based on the discussion in this section the following table 6.2 shows the final prioritisation of the four main language criteria which we can use to make language decisions in section 6.3.

	Very important	Important	Less important
Readability	x		
Writability		x	
Reliability		x	
Cost		x	

Table 6.2: Shows prioritisation of the four main language criteria.

6.2 Language Paradigm

Based on the theory of the four main programming paradigms in section 5.2, we will in this section discuss and evaluate which paradigm(s) is/are more suitable for the development of the Tang language which will be used in section 6.3 to discuss the features included in Tang.
According to 6.1 reliability and readability are highly prioritised language criteria for the Tang language and therefore choosing to develop a language that has characteristics of the functional programming language might seem like the obvious choice, since according to the theory in 5.2 functional programming languages tend to be both readable and reliable. However a major drawback of the functional programming language is that it requires a lot of memory and runs relatively slow compared to e.g. languages within the imperative paradigm due to reasons explained in section 5.2.2.
This is a problem due to the speed and memory limitations of the AVR ATmega328p MCU introduced in section 3.1.1. The lack of side-effects in purely functional programming languages also somewhat contradict the need to change the state of a system which might be relevant for microcontrollers, if they are to react on some input from the real world.
As with functional programming languages, current logical programming languages suffer in lack of efficiency, especially for larger problems [41, p. 721].
Another drawback of using the characteristics of logical programming in this project is that we, in this project, want programmers to fully understand what they do and how it works in order to debug conveniently. Because the goal is to make a language for programmers that already know software programming and want to learn hardware programming, we do not want a declarative approach, but rather a procedural approach.
The idea of classes in the object-oriented paradigm can be used in the Tang language for making libraries and abstractions of hardware components like we did in Arduino and Java in section 3.1 and 3.2.3. However due to the hardware limitations of the Arduino processor, as explained in 3.1, programs are relatively small and thus adding other object oriented principles like polymorphism, inheritance, and dynamic bindings might cause higher memory usage as shown in section 3.2.5.

Based on the previously described disadvantages of the functional, logical, and object-oriented paradigms, in relation to this project, we have chosen the imperative paradigm for the Tang language. The reasons for this choice is that the imperative languages do not have any common characteristics which limit the creation of an efficient programming language in terms of speed of execution.
Furthermore, the imperative paradigm allows for a sequential execution of instructions with side effects which we think are most suitable for hardware programming as it is the way the execution of machine code instructions works on the hardware level.

According to [41, p. 704] in imperative languages *"[...] the details of dealing with variables obscure the logic of the program."* which affects the readability of the language. Therefore, focus on readability is especially important in the syntax design of the Tang language which is discussed in section 6.4.

6.3 Language Features

In this section we will discuss different language features and select which of these features should be included in Tang.
We select these features based on a discussion of each feature and the prioritised language criteria established in section 6.1.
We use AVR C-syntax as a reference for recognising which features are included in Tang, thus, for example, the % operator is a reference to the modulo operator in AVR C, but does not state the syntax of the operators included in the Tang. The syntax of the operators included in Tang is instead discussed in section 6.4.
The symbol x indicates that there is no AVR C syntax for a given language feature.

Data Types and Literals

Based on the different primitive data types shown in table 6.3, we will discuss which primitive data types should be included in Tang.

There is a need to represent integer values in our language since integers can be used to make iteration variables, represent analogue input/output values, and make mathematical expressions. We choose to exclude unsigned integers because of casting problems between unsigned and signed integers and the dangers of overflow associated with casting.
If an unsigned integer is used in an expression in C, C will cast everything to unsigned and execute the expression, which might cause unexpected results which the code in listing 6.1 illustrates. This could however be avoided in a language if unsigned and signed integers were seen as two distinct types. We choose not to do this because of the similarities that signed and unsigned integers have and that it therefore can be confusing for a programmer that they e.g. cannot multiply a signed integer with an unsigned.

```c
#include <stdio.h>

int main(void)
{
  unsigned int plus_one = 1;
  int minus_one = -1;

  if(plus_one < minus_one)
    printf("1 < -1"); // Gets executed
  else
    printf("boring");

  return 0;
}
```

Listing 6.1: Demonstration of casting an unsigned integer to an expression in C. Program outputs "1 < -1"

Larger bit lengths than 32 has been omitted for the sake of the simplicity of the language. This means that numbers over 2^{31} will not be natively supported, but since the Arduino is not designed for doing heavy math, we do not expect this to be an issue in most cases. Binary values were chosen to be represented through a boolean type instead of a bit type in order to enhance readability and to ensure that boolean expressions are used for control structure conditions [41, p.35]. In contrast to C that represents registers as integer pointers, we instead choose to introduce separate types for registers in Tang without introducing pointer arithmetic. We include register types to represent hardware registers like `DDRB` and `PORTB`, to easily enable the programmer to interface with the hardware like pins, ADC, and timers.

Type	AVR C syntax	Included in Tang
Unsigned integer	unsigned int	No
Real numbers	float / double	No
Boolean	_Bool	Yes
String	char*	No
Integer	int	No
Signed integer 8 bit	char	Yes
Signed integer 16 bit	short / int	Yes
Signed integer 32 bit	long	Yes
No type	void	Yes
Register 8 bit	x	Yes
Register 16 bit	x	Yes
Pin	x	No
Char	char	No

Table 6.3: Shows selected primitive data types for the Tang language.

Bitwise Operators

Line 10 in the blink example for AVR C seen in listing 3.8 shows how bitwise operations are used to turn off an LED by setting the fifth pin on PORTB to 0: `PORTB &= ~ (1<<PIN5)`. To simplify the operations of modifying bits in registers we choose to include a bit index operator for changing

the value of a bit based on an integer index. The syntax of the bit index operator is described in section 6.4. We do not include bitwise *and*, *or*, *exclusive or*, *left shift*, *right shift*, and *not*. This is done to simplify the language for programmers who are not familiar with bitwise operators.

Bitwise operators	AVR C syntax	Included in Tang
And	&	No
Or	|	No
Exclusive or	^	No
Left Shift	«	No
Right shift	»	No
Not	~	No
Bit index operator	x	Yes

Table 6.4: Bitwise operators, their AVR C syntax, and if they are included in Tang.

Logical Operators

Logical operators provide a way of expressing logic in a language and have therefore been included in the Tang language. The binary logic operators `and` and `or` can be used to connect boolean expressions while the `not` operator can invert the boolean value of an expression.

Logical operators	AVR C syntax	Included in Tang
And	&&	Yes
Or	||	Yes
Not	!	Yes

Table 6.5: Logical operators, their semantic meaning, and if they are included in Tang.

Comparison Operators

The six comparison operators in table 6.6 have all been included in the Tang language as they provide the opportunity for comparisons between different expressions, which for instance can be useful in control structure conditions for stating when a condition is `true` or `false`.

Comparison operators	AVR C syntax	Included in Tang
Equality	==	Yes
Greater than	>	Yes
Less than	<	Yes
Bigger than or equals	>=	Yes
Less than or equals	<=	Yes
Not equals	!=	Yes

Table 6.6: Comparison operators, their description, and if they are included in Tang.

Arithmetical Operators

Table 6.7 shows different arithmetical operators which have been included in Tang as they can be used to evaluate numerical expressions. Addition, subtraction, multiplication, and division are all basic arithmetical operations. Power extends these basic arithmetical operations and the modulo operator is used to get the remainder of an integer division. The modulo operator is useful whenever integers are included in a language which is the case for this project as previously described.

Arithmetical operators	AVR C syntax	Included natively in Tang
Addition	+	Yes
Subtraction	-	Yes
Multiplication	*	Yes
Division	/	Yes
Modulo	%	Yes
Power	x	Yes

Table 6.7: Arithmetical operators, their description, and if they are included in Tang.

Pointer Operators

Avoiding visible pointers and pointer operators will give a more reliable programming language, as explained in section 5.1.4, which we previously determined to be important for the Tang language. Furthermore, from the group members own experiences, we know that pointers can be difficult to work with and can be dangerous because you can overwrite important data, which are a reason why we choose to exclude pointers in the language and thus omitting pointer operators.

Pointer operators	AVR C syntax	Included in Tang
De-reference	*	No
Address of	&	No

Table 6.8: Pointer operators, their description, and if they are included in Tang.

Assignment Operators

Compound assignments such as the five operators: `+=', '-=', '*=','/=', '%='` showed in table 6.9 illustrates a short syntax for computing the result of an expression where the variable that is assigned to is used as the left operand in an operation. For instance `a += 2` is a shorter version of `a = a + 2` and are thus two ways of expressing the same computation.

Having multiple ways of performing the same action reduces the simplicity and therefore also the readability of the language. Furthermore increased expressivity can decrease the readability of a program [41, p. 45]
As determined in section 6.1, readability is prioritised over writability and therefore we choose not to include any compound assignment in the Tang language.
The assignment operator is included as this is essential for saving the result of a computation.

Assignment operators	AVR C syntax	Included in Tang
Assign	=	Yes
Add and assign	+=	No
Subtract and assign	-=	No
Multiply and assign	*=	No
Divide and assign	/=	No
Modulo and assign	%=	No

Table 6.9: Assignment operators, their AVR C syntax, and if they are included in Tang.

Control Statements

The control statements listed in table 6.10 are constructs commonly seen in imperative languages such as C and Python and are convenient ways of iterating and controlling the state of a program. However, some of these statements have many common usages and thus both if-statements and switch-statements are conditional statements where the execution of code depends on the conditional expression in the statements.
Switch cases generally have some limitations in their usage. For instance, switch cases cannot evaluate logical expressions, only `int` or `char` types in languages such as C. Since if-statements can do all these, this means that every switch-statement can be rewritten using `if`-statements, whereas the opposite is not the case.
Because we prioritise readability over writeability, we choose to exclude switches in the language for the sake of the simplicity, which according to the theory in section 5.1.1, makes a programming language easier to learn.
Likewise, while loops have been favoured over do while loops.

Control statements	AVR C syntax	Included in Tang
If then	if(){}	Yes
If then else	if(){} else{}	Yes
Switch	switch(){case 1 : break;}	No
For loop	for(; ;){}	Yes
While loop	while(){}	Yes
Do while loop	do {} while()	No

Table 6.10: Control statements, their AVR C syntax, and if they are included in Tang.

Additional Constructs

Table 6.10 shows other language features we have discussed as possible features for the Tang language.
We choose to include comments in the Tang language as it can increase the readability of a program by explicitly stating and explaining thoughts, choices, and meaning throughout the source code. Multiple and single line comments have both been included to increase the writability of writing comments as this choice in most cases will not affect the readability of the source program due to syntax highlighting in most modern text editors.
As a way of adding libraries and splitting source code into multiple files, we choose to add a feature for file inclusion in the Tang language. However, we choose not to have access modifiers and namespaces in the Tang language as these features are more relevant for big and complex programs, whereas the Arduino has limited memory (see section 3.1). Therefore, it is unsuited for bigger and more complicated projects and thus we choose to omit these features to enhance the simplicity of the language.
Furthermore, arrays will be included in Tang as they are useful for encapsulating values of the same type into a simple data structure. Arrays can also increase the readability of a program, in contrast to not having a data structure that e.g. allows a programmer to iterate through a number of variables multiple times.
Another feature is interrupts which will also be included in Tang to enable full utilisation of the microcontroller's capabilities.
A variation of classes as we know them from the object oriented paradigm will be included in Tang such that a class in Tang is an encapsulation of methods and variables, but will not contain constructors, access modifiers etc. because of the selected paradigm of Tang and for the sake of the simplicity.

As a last feature we choose to include functions in Tang as it allows programmers to define algorithms which encourages code reuse and makse more structured code and thus increases readability and writability. We do not allow nested functions, as this can be simulated with classes and could decrease the readability of the program.

Additional Constructs	AVR C syntax	Included in Tang
Single line comment	//	Yes
Multi line comment	/* */	Yes
File inclusion	#include	Yes
Access modifiers	*x*	No
Namespaces	*x*	No
Arrays	type name[]	Yes
Classes	*x*	Yes
Interrupts	void INT0_vect (void) { ... }	Yes
Functions	type name() {}	Yes

Table 6.11: Control statements, their AVR C syntax, and if they are included in Tang.

6.4 Syntax Design

This section will describe the syntax of the included language features of Tang that were established in section 6.3.
We will discuss the choices we make in regards to the syntax of Tang based on the prioritised language criteria in section 6.1. Code snippets and tables are included in this section to show the syntax of Tang in relation to the names used in section 6.3. The syntactical choices made in this section are primarily based on a scientific paper regarding programming language syntax where different syntax is rated in regards to intuitiveness by programmers and non-programmers [57].

Types

Tang is chosen to be statically typed. This means, that for every declaration, a type is required and all types are known at compile-time. The type of a declaration is always located as the left-most keyword in a statement. This applies to declarations of variables as well as functions, to increase consistency and readability.
Integers in Tang are required to have the length defined as described in section 6.3. As described in section 3.1, microcontrollers have relatively low memory sizes compared to e.g. a PC. In these situations, it can be important to know the bit length to ensure that variables of a certain type can hold the values assigned to these variables. Because of this, we choose to name the integer data types with a post-fix number indicating the number of bits used to store the data type in memory to enable the programmer to read and determine which integer type is suitable in specific situations. By doing so, the programmer can also read and identify where integer types with lower bit lengths can be used to reduce memory usage. This decreases the writability of the language in comparison to always using the same integer type without considering the bit length, but as mentioned in section 6.1, we prioritise readability over writability and therefore require the programmer to explicitly state the bit length for integer types.
To represent the bit length of integers, we choose three lengths of signed integers with a precision of 8, 16, and 32 bits. The reason for this is that even though 16 bits are sufficient to hold numbers up to $2^{15} = 32768$ and then a sign bit indicating if the number is positive or negative. There might be a need for making applications where larger values are needed e.g. counting milliseconds on a clock, where a 16 bit integer will only last about 32 seconds.

To represent integers of a particular bit-length the syntax of integers is denoted by the *int* keyword followed by a bit length of 8, 16, or 32, e.g. `int8`. More examples of the different possible integers are shown in table 6.12. The type *register* could be, as in the case with the integer type, reduced to *reg*. the reason for this, is that among programmers, who are new to embedded programming, having this type explicitly written out, increases readability. Like integers, registers will also contain a bit length of either 8 and 16. We do not include a bit length of 32 for registers as there is no single value that can be read from a register on the ATmega328P that is longer than 16 bits but there are few that are longer than 8, such as the 10 bit value that indicates the result of the latest ADC reading [20, p. 305].
According to [57, p. 11], *boolean* was rated as being slightly more intuitive to read than bool, but also with a higher standard deviation. Because of this, the two options are essentially equal in regards to intuitiveness, and therefore, we choose *bool* based on writability.
char was not rated in [57], but as our target group has programming experience and knows about primitive types, we choose to increase writability and use *char* over a more readable statement to non-programmers as *character*.

Tang syntax	description
bool	boolean
int8	8-bit signed integer
int16	16-bit signed integer
int32	32-bit signed integer
nothing	no type (see section *functions* below for explanation)
register	register type
char	character

Table 6.12: Shows the chosen types for Tang along with the syntactical design in Tang and description of the types.

Arrays

Arrays in Tang can be defined in three ways, either with values, a size, or without. If an array has values, the size does not need to be specified. If an array, when defined, is not initialised to specific values, the size needs to be explicitly stated. Examples of arrays in Tang can be seen in listing 6.2. The index operator for arrays in Tang corresponds to the one, that programmers find the most useful in [57, p. 13] which is `name[indexNumber]`.

```
1 int8[5] emptyArray
2 int8[] nonEmptyArray = [2,3,5,7,11]
3 int8[2] = [1,2,3,4,5] //illegal, size does not fit content
4 int8[] emptyArray2 //illegal, empty array needs explicitly stated size
```

Listing 6.2: A code listing showing arrays in Tang.

Identifiers

Identifiers are necessary for all variables, functions, classes, and arrays in Tang. Identifiers are always to the right of the type when in initialisations and can thereafter stand alone, as seen in listing 6.3.

The reason behind this syntactic design choice of identifiers is that our target group consists of people with programming experience and they already know this syntax from basic languages like C and Java [58] [59]. Valid identifier names consist of only alphanumeric characters and underscore and cannot begin with either a digit or underscore.

```
int8 a
a = 2
```

Listing 6.3: A code listing showing identifiers in Tang.

Separate and Group Statements

A decision we had to make was the syntax of how to group statements into blocks to be used in language constructs such as in the bodies of control structures. We discussed two options for grouping statements; curly braces and indentation.
We chose indentation over curly braces as it forces the programmer to write more readable code (forcing correct indentation) and it increases readability by refraining from getting multiple lines with single closing curly braces. However, this can decrease writability amongst programmers who do not usually indent their code. This choice corresponds with the choices made in section 6.1 where readability was prioritised over writability. Regarding **statement separator-terminators**, we have chosen to use newline to terminate statements. By not having to end each statement with extra symbols, e.g. semi-colon (;), as in programming languages such as C, Java, and C# do, we increase writability. This may also increase readability, as we refrain from making multiple statements in one line and instead force new statements to be on new lines. An example of our implementation of indentation and statement separator-terminators can be seen in listing 6.4 where it is compared with an equal listing written in C, seen in listing 6.5.

```
if(a==5)
    a++
    return x
```

Listing 6.4: A code listing showing indentation and statement separator-terminators in Tang.

```
if(a==5){
    a++;
    return x;
}
```

Listing 6.5: How to group and separate statements in C.

Functions

We now define the syntax for functions in the Tang language. As Tang is a statically typed language, the return type for a function must be defined. In Tang, the type has to be the leftmost lexeme of the function. To the right of the type is the name of the function, as seen in the syntax used for languages like Java, C, and C#. The reason for this structuring is that a syntax already used by

the majority of existing languages makes it easy to migrate both to and from these languages.
A different syntax was discussed, such as having the function name be the left-most token. As declarations in the Tang language have the type to the left of the name, applying the same structure to functions increases the orthogonality of the language (see section 5.1.1).
A specific function keyword is not needed, as it is the parenthesis after the function name, that indicate that it is a function and not a variable declaration. In the same manner, the type indicates that it is a function declaration, and not a function call. An example of such a function can be seen in 6.6. Formal parameters each have a type and are separated by comma.

```
1 int32 max(int8 a, int8 b)
2     if(a>b)
3         return a
4     else
5         return b
```

Listing 6.6: A function i Tang

When the return type of a function is empty, the keyword chosen to indicate this, is *nothing*. The reason this keyword is not chosen to be *void*, as is standard in most languages such as C and Java, is to increase readability of the language. If the function is **not** declared to be of the type *nothing*, a return keyword has to appear inside accessible code in the body of the function. The return statement must include a value matching the return value of the function. The reason for not allowing a return statement to appear alone, as it may in for void functions/methods in C and Java, is to increase the readability of the language, as it is easier to understand what the function returns exactly.

In C, arrays cannot be directly set as parameters to a function, instead a pointer has to be implemented. In Tang, however, pointers are not visible in the same way to the user. Without having the user dealing with pointers, one advantage is, for example, that arrays can be used as parameters in functions, without dealing with setting pointers for it. This increases the overall writability. [57]

Classes

Classes are defined with some of the same syntax as in C#. To simplify classes in Tang, constructors are removed. If the user wants to initialise variables within a class, this has to be done manually. An example of a *Class* in Tang can be seen in 6.7, where a substitute for a constructor, called *init*, can be seen on line 8. The goal of classes in Tang, is to group methods and variables together.

```
1  import uno
2
3  class Led
4      register port
5      register ddr
6      int8 pinIndex
7
8      nothing init(int8 pin)
9          port = uno.getPort(pin)
10         ddr = uno.getDdr(pin)
11         pinIndex = uno.getPinIndex(pin)
12
13     nothing on()
14         port{pinIndex} = true
15
16     nothing off()
17         port{pinIndex} = false
```

Listing 6.7: A class in Tang

Accessing Methods and Attributes in Classes

To access methods in classes in Tang, we have chosen to use the syntax seen in listing 6.8 line 3: an object name is followed by a period followed by the method name to be called on the specific object. After the object name, a period is placed followed by the name of the method and parentheses. If the specific method requests parameters, these will be placed inside the parenthesis separated by a comma.
The attributes that are accessible from an object follow the scope rules which we choose in section 6.5 and define in the operational semantics in section 7.2.

```
1  int8 pin = 2
2  Led led
3  led.init(pin)
4  led.on()
```

Listing 6.8: A class in Tang

Control Structures

if

In Tang, `if` statements are written with the same structure as in C. The only notable difference is the lack of brackets and semicolon, described in section 6.4.

For loops in C, and many other languages, are defined by having three explicit parameters: declaration, range, and increment. This approach can be considered unintuitive, as it does not directly indicate, towards the user, what the expected outcome can be, e.g. in C for(;;) is legal and has the same functionality as a while(true) loop. In contrast to this, the language Quorum has a far more

readable for loop [57, p. 23]. The Quorum language *for loop* cosist of the statement *repeatXtimes*, where the X is the number of times the loop repeats. The downside of increasing the readability to what Quorum has is that some of the functionality is lost from the *for loop*, as the current iteration number cannot be accessed in the scope of the loop, unless this is monitored by the user. In Tang, a *for loop* consists of the keyword *for*, followed by parentheses, in which there are two keywords required: *from* and *to*. In the parenthesis, an integer must first be declared, followed by the range of the loop. The range can be declared as a signed integer, and be either incremental or decremental. An example is shown in listing 6.9, where the loop structure iterates in a decremental order, from 10 to -5. Furthermore, we design the for loops such that the range of the loop is only calculated once, and such that the iteration variable, denoted `i` in listing 6.9, cannot be increased to effect later loop-iterations. For the formal description of the design of for loops see the operational semantic and type system in section 7.2 and 7.2.3, respectively.

```
1 int8 a
2 for(int8 i from 10 to -5)
3     a = a + i
```

Listing 6.9: A *for loop* in Tang

while
For `while` loops, the syntax is based on the C-like languages, as it is considered the most intuitive amongst experienced programmers [57, p. 21].
While loops will therefore have largely the same syntax as if statements and for loops.
The syntactic design of the three control structures explained above can be seen in 6.13.

Tang syntax	AVR C syntax
if(true)	if(1)
for(int8 id from 1 to 10)	for(char id = 1 ; id < 10; id++)
while(false)	while(0)

Table 6.13: Shows examples of control structures in Tang compared to their respective C syntax.

Operators

The *logical operators* included in our language are consistent with the English language with the exception of the logical operator not which is implemented as an exclamation mark (’!’). These choices are based upon which symbol choices in [57, p. 13] that programmers rated the most intuitive.
Similarly, syntactic design of the *comparison operators* is made with programmers in mind. According to [57, p. 13], people with programming experience rate `'=='` the highest in regards to intuitiveness when it comes to an equal operator. The study did not test the compound comparison operators like `=>`. We have chosen to implement compound comparison operators as they are included in most programming languages, e.g. C# [60], which can be seen in table 6.14.
The design of *arithmetical operators* in Tang is based upon standard mathematical notation as most people know this. These operators can also be seen in table 6.14.
As for the *assignment operator*, we choose the most intuitive choice, `'='`, not only from a mathematical point of view, but also from a programming point of view [57, p. 11]. Table 6.14 shows the syntactic design of all operators in Tang.

Tang syntax	name
and	and
or	or
!	not
==	equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
!=	not equal
+	add
-	subtract
*	multiply
/	divide
%	modulo
^	power
=	assign

Table 6.14: Shows the chosen operators for Tang along with their syntactic design and their names on the right.

Additional Decisions

Besides the already defined syntax, we now describe the syntax of the remaining language features that are included in Tang. These features can be seen in 6.15.
To refrain from making bit accessing look like array accessing, we have chosen curly braces instead of square brackets for the bit indexing operator.
Comments are written using the syntax for single line and multiple line comments in listing 6.10 since programmers as well as non-programmers find this the most intuitive [57, p. 17].
For import statements, we have chosen the keyword `import` over e.g. using or include. According to [57, p. 16], programmers find the import statement easier to understand.

```
import io

//Single line comment
/*
 Multiple
 line
 comment
*/

int8 ledPin = 5

//Register type
register8 ledPort = register8(PORTB)

//Interrupt
interrupt(TIMER1_OVERFLOW)
    //Toggle led when timer interval elapsed
    ledPort{ledPin} = !ledPort{ledPin}
```

Listing 6.10: Code listing showing the design choices made in section 6.4 under additional decisions.

Tang syntax	description
DDRB{5}	Access bit at index 5 in register variable DDRB
/* */	Multi line comment
//	Single line comment
import nano	Includes the file nano.tang

Table 6.15: Shows syntactic design of additional decisions in Tang and their description.

Based on the selected operators, defined in this section, table 6.16 shows the operator precedence of operators which are incorporated into the grammar for Tang described in section 7.1.2.

Precedence	Operator	Associativity
1	()	Left to right
	{ }	Left to right
	!	Right to left
2	∧	Left to right
2	*	Left to right
	/	Left to right
	%	Left to right
3	+	Left to right
	-	Left to right
4	>	Left to right
	<	Left to right
	>=	Left to right
	<=	Left to right
5	=	Right to left
	!=	Left to right
6	and	Left to right
7	or	Left to right

Table 6.16: Operators, their precedence and associativity in Tang.

6.5 Selecting Scope and Type Rules

Scope rules describe the available bindings when executing a specific procedure and are therefore important to define. Type rules are also a necessity to be defined, as they define which types can be used together in e.g. addition of integers or in casts. The scope rules and type rules of Tang is formalised in the operational semantics and type system in section 7.2 and 5.4.2, respectively.

Selecting Scope Rules for Tang

As seen in section 7.2, we choose to have variables exist in their own scope and all sub-scopes of that. Furthermore we previously decided that a variable of the same name and type cannot be declared if it already has been declared in the scope.
Another possible solution could be to allow multiple declarations of the same variable, but overwrite it each time it is declared again. This method could introduce problems, if the programmer forgets a variable with a simple name is already declared and then overwrites it.

As seen in listing 6.11, line 1 declares a variable `a` of type `int16` initialised to the value 2. Inside the scope of the for loop we then try to declare the same variable, but as it already exists in an outer scope, this is an illegal action. Control structures and functions declare a new sub-scope.

```
1 int16 a = 2
2 for(int8 i from 10 to 2)
3     int16 a = 10 // illegal declaration, since it has already been
          ↪ declared in the outer scope
4     a = 10 // legal assignment
```

Listing 6.11: An extract of a Tang program showing legal and illegal actions regarding scope.

For function calls we use static scope-rules where the variables we have knowledge about are the variables that were known at the time of the function declaration. Consider the example in listing 6.12 where we, inside the function foo(), call a variable b that is declared after the declaration of foo, but before the call to foo. This is illegal in Tang as we have decided that the language uses static scope rules for variables because it is intuitive that one cannot use a variable that has not yet been defined. However, the code in listing 6.12 would have been legal if Tang had dynamic scope rules for variables.

```
1 int8 a = 2
2 nothing foo()
3     int8 sum = a + b // illegal operation no knowledge of b
4 int8 b = 3
5 foo()
```

Listing 6.12: Scope for variables in Tang

As for the scope rules of function calls you can call a function before it is declared. The reason for this is that functions are different from variables in that, when a function is called, a jump is made to execute the called function. This is possible because functions do not change after they are first declared.
This also allows mutual recursive functions where two functions call each other. Consider a simple example for the scope rules of function calls in listing 6.13 where a function foo calls a function bar before bar is declared which is legal in Tang.

```
1 nothing foo()
2     bar() // legal call
3 nothing bar()
4     foo() // Also legal
```

Listing 6.13: Scope for functions in Tang

Selecting Type Rules for Tang

To avoid confusion and problems that arise when converting between types, we only allow a small number of conversions.
Values can only be compared to values of the same exact type, with integers as an exception, as arithmetic operations and assignments are allowed on integers of different sizes, but not other type-categories.
When two integers are used as operands for an arithmetic operations such as addition, the resulting value will be of the same type-size as the largest of the two values. Examples of type conversions are shown in listing 6.14.

```
1  int8 a = 2
2  int16 b = 5
3  bool c = false
4
5
6  a = c // illegal, as 'a' is in the type-category int
7        // and 'c' is in the type-category bool
8  a = b // illegal, as int16 values do not fit inside types int8
9  b = a // legal, int8 is converted to int16
10 a = a + b // illegal as the resulting type of 'a + b' is int16
11           // and cannot fit in a variable of type int8
12 b = a + b // legal, 'a + b' is of the larger of the operands
13           // type-size which is int16
14 if(c > b) // Illegal, since c is bool and is an int16
15   // Do stuff
```

Listing 6.14: An extract of a Tang program showing legal and illegal actions regarding type conversion.

6.6 MoSCoW Prioritisation of Language Features

Due to the time constraint of this project, we will prioritise the language features established in section 6.3. The prioritised language features will be used to determine which features should be implemented first and also serve as a guideline for establishing the minimum viable product for Tang.

The language features are prioritised using the MoSCoW prioritisation rules, which is a set of rules where requirements are prioritised into one of the four following categories: must have, should have, could have, and will not have [61].

- The **M**ust have requirements are the essential requirements without whom the final product will not be satisfactory [61].
- The **S**hould have requirements are important requirements that should be implemented if there is time for it. However, these requirements are not essential for getting a working product [61].
- The **C**ould have requirements are the nice to have requirements, but not implementing these will not make the final product less usable [61].
- The **W**ill not have requirements are the least important that we would like to have in the final product, but which we will not have time to implement. These requirements are therefore left out to be developed in later software iterations [61].

In the discussion below the language features established in 6.3 serve as the requirements for Tang.

Must Have

The must have features to be implemented in Tang are based on the requirements for the project set in chapter 4 which states that the programmer should be able to control hardware components on the Arduino. This means that the language must be able to do simple tasks like reading and

writing to digital pins, so the language as a minimum can produce a program to run on the Arduino to make a LED blink.

Signed Integer 8	Signed Integer 16
Boolean	Register
Add	Subtract
Multiply	Divide
Not	Equals
Smaller Than	Greater Than
Logic And	Logic Or
While	If-Then
Assignment	Interrupt
Bit index	

Table 6.17: Must have

To fulfil the basics of our language, we choose to include `while` and `if-then` in the must have, since these are basic control structures that can be used to simulate other control structures like `for` and `do-while`. A `for` loop can be simulated with a while-loop, and a `do-while` loop can be simulated using both an `if-then` and a `while`. A `not` operator (`!`) is also included which allows for inverting the condition of `if` statements which can be used to simulate `if-else` statements, as well as `not equals`.
For types, we chose to include a couple of different types suitable for most basic functionality. We have a `register` type which enables the programmer to set and flip single bits in a hardware register. This is especially useful for GPIO to read and write values of the pins of the Arduino.
Another hardware related feature that we choose to include under must have is `interrupt` because it is needed for some applications that need to react on timer intervals or changing pin values. The `bool` type only has two states; `true` and `false`, which `while` and `if` statements use as condition, and all the comparison operators evaluate to one of these.
`signed integer 8` and `signed integer 16` are 8-bit and 16-bit signed integers used for representing whole positive numbers. The reason for both 8 and 16 bits length is that 8 bit can represent a character or just a single byte, while a 16 bit can represent larger numbers that is often dealt with, and since they are very much alike, it will be easier to implement the second, once the first is made. Some 8 bit arithmetic operations are, as described in section 3.2.1, supported by hardware instructions and can therefore be easier to implement, depending on the implementation language.
Basic arithmetic operators are also in the must have group of features. These include addition, subtraction, multiplication, and division. Modulo is not included as this can be simulated with the other operations, although slow. As `bit indexing` is an essential part of programming embedded systems, this is also included in the must have requirements.

Should Have

For-loop	No type
Signed integer 32	If-else
Bigger than or equals	Smaller than or equals
Not equals	$\wedge$ (to the power of)
Arrays	Functions
Modulo	

Table 6.18: Should have

The language should have `for`-loops, which makes iterating over statements both more readable and writable. The language should also support `signed integer 32` to enable bigger numbers. This enables the programmer to have access to larger numbers.
The language should also have **bigger than or equal**, **smaller than or equal** and **not equal** which makes conditions shorter and increases both writability.
The language should also have $\wedge$ (to the power of), **arrays**, **functions**, and % (**modulo**). The $\wedge$ and % provides short syntax for taking the power and modulo of an expression respectively, while the array data structure is for storing a set of values of a specific type, which can e.g. be used in different algorithms.
The functions are useful for abstracting over hardware components, making recursions and allow for code reuse and encapsulation which can enhance readability. Implementing functions help abstract over hardware components, which is one of the research questions listed in section 4.
Common for all these features are that they more or less can be simulated using the features established in the **M**ust have, but they are still important features as they allow for more expressive and readable code as well as providing extra functionality like recursion.

Could Have

Char	Comments
File Inclusion	Classes

Table 6.19: Could have

The language could have the `char` type, **comments**, **file inclusion**, or `class`.
The `char` type will provide an easier to read syntax for displaying characters as they do not have to be specified using integers. However, the char type is not a type that provides any extra functionality to Tang or contain any information that can be used to enhance memory usage like the integer types in the ”should have” requirements.
File inclusion allows for reusable code across multiple files and can be used to include libraries. However, even though file inclusion is a could have feature for Tang it is not essential for making a programming language for programming hardware.
A `Class` are merely an abstraction for encapsulating attributes and functions into objects, which can increase both readability and writability, but is not a basic construct and can be imitated using functions and variables, which is why they are prioritised low to ensure we get the basics implemented first.

Comments are convenient to use to express thoughts and meaning which can enhance readability though they are not essential for hardware programming and are, therefore, a could have requirement.

Will Not Have

The language developed in the project, Tang, will not have floating point variable-types or string variables as this could, although not as intuitively, be emulated with integers and char arrays. In an ideal situation, floats and strings would be implemented in Tang, but as we have a limited time schedule, we do not expect to have the time to implement these features.

Float	Strings

Table 6.20: Will not have

6.7 Compilation Process

We now have a prioritised list of features and we will now look into how we will implement our translator. The purpose of this section is to choose whether an interpreter or a compiler is most suitable for the implementation of Tang based on a discussion of the the advantages and disadvantages that they each have in correlation to this project.

6.7.1 Choice of Using an Interpreter or a Compiler

One of the advantages of using compilation over interpretation is a significant increase of program execution speed gained by compiling in comparison to interpretation as described in 5.5.2. This characteristic is especially important for this project as we are making a program for programming hardware and thus are not interested in any unnecessary performance decreases.
Another advantage of selecting compilation for this project is the ability to reuse the generated object-code if the source-code does not have to be recompiled. However, in section 5.5.2 it was also stated that compilation requires more memory than interpretation which might seem like a concern for this project since section 3.1 highlights that a constraint of the ATmega328P is it has low memory capabilities. To solve this problem while still obtaining the advantages of compilation in relation to interpretation for this project, we chose to make a cross-compiler (5.5.1) which compiles Tang to Arduino machine-code on an x86 system as illustrated on the latter part of the tombstone diagram in figure 5.3. By doing cross compilation, the compilation process is not constraint by the speed and memory constraints of the Arduino.

6.7.2 Target Language

To develop a compiler for Tang we need to decide what language we are translating to.
As described in section 3.1.2 and 3.2 Arduino has support for AVR assembler, C, Arduino language, and Java which all are possible choices of target language.

AVR Assembler is a low-level language and is therefore arguably more difficult to read and write than other high-level languages which makes it a less attractive target-language since we eventually have to generate code for the target-language. However, from a learning perspective it could be interesting to generate code for AVR Assembler as it will give some insight into how different language constructs are represented in low-level programming. Furthermore, it will provide a good insight into the whole compilation process from a high-level language to a low-level language.

Compiling to the Arduino language means that we compile to a language that already has been specially designed for Arduino programming, which means it already have support for different hardware components. However, the Arduino language is merely an abstraction over the C++

language and will just translate into another high-level language which gives a more complex compilation process. A third option is to compile to Java, however the virtual machine for Arduino that we have found and tested has been buggy and not fully developed. The whole setup also took over an hour to make work and was still unstable and would still fail without reason. Furthermore, since Java code is compiled to Java byte code which is then interpreted by the JVM on the Arduino, this process is relatively slow as described in the theory in section 5.5.2. The last option is to generate code for AVR C which is the option that we choose since it enables low memory usage compared to the Arduino language and Java, as shown in section 3.2.5, while still supporting higher level data types and control structures when compared to AVR Assembler.

6.7.3 Implementation Language

The implementation language for this project is going to be C# as it is a language that all group members have experience in while also being an object-oriented language with many features from other paradigms such as lambda-functions, pattern matching and so on which, in our opinion, makes it a good choice for complex projects like this. Another reason is that the book used in the Language and Compiler course is based on another object-oriented language (Java) which makes the mapping between the two languages relatively trivial. Java was not chosen because not all members had previously used written programs in it, and because of small irritations over lack of features like properties.
Another option for implementation language could have been C++, but as with java, few group members has much experience with the language. C++ also requires the developer to manage their own memory, which can be frustrating, especially for larger applications.

6.7.4 Tombstone Diagram

Based on the choice of target and implementation language, we will now give an overview of the compilation process by using tombstone diagrams.
Tombstone diagrams are a way of visualising the compilation process of a program. Figure 6.1 shows a program written in Tang which is inputted to a Tang-to-C compiler implemented in x86 machine code and run directly on a x86 system. This compilation process outputs a C program that is passed as input to a C-to-AVR compiler, like avr-gcc, that runs on a x86 system. The result of this compilation process is AVR macine-code that are able to run on the Arduino's ATmega328P MCU.

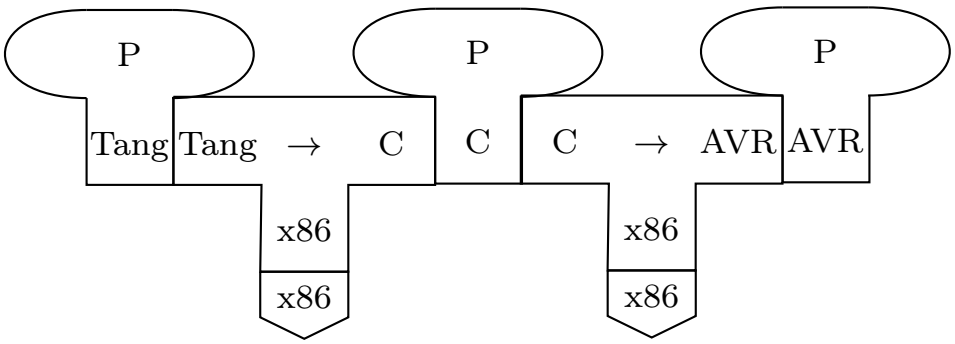


Figure 6.1

Figure 6.2 visualises the process where our compiler written in C# (CS on diagram) is compiled

using a C# to x86 compiler. The result of this process is as shown a compiler implemented in x86 that produces C code.
In the figure 6.2, we visualise the process of compiling our compiler, which compiles Tang to C. Our compiler is implemented in C#, which can be compiled to x86, using the C# to x86 compiler, giving us our compiler in x86, which can be run on our PCs.

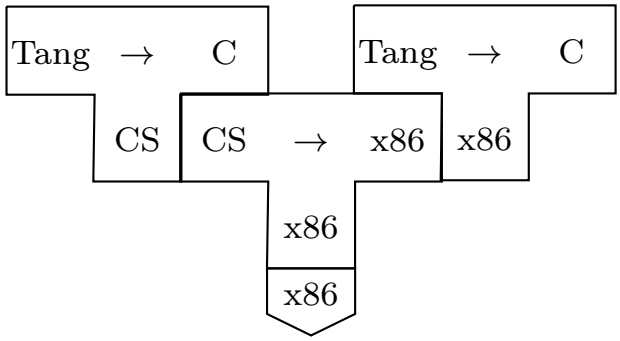


Figure 6.2

Chapter 7

Language Specification

Based on section 6, we will now formalise the Tang language. We will formalise Tang's syntax and semantics so it is clear how the different language constructs work together and what their meaning is.
Note that due to time constrains this section will only describe the language specification for the features that have been implemented in Tang and is thus only a subset of the features selected in section 6.3. A discussion of the constraints of the language specification can be seen in section 7.3.

7.1 Formal Syntax Description

In this section, we describe the formal syntax of Tang based on the theory of tokens, regular expressions, and grammar given in section 6.4. To do so, we use regular expressions to specify the tokens of Tang.
The design choices in chapter 6 and the token specification are then used to define the CFG for Tang. The token specification and the CFG will be used in the implementation chapter for the lexer (section 8.2) and parser (section 8.3), respectively.

7.1.1 Token Specification

Based on the included language features and their syntax, described in 6.3 and 6.4 respectively, we use regular expressions as described in 5.3.1 with the syntax specified in [49] to define the tokens of Tang.

Table 7.1 shows the tokens of Tang including their associated regular expressions and whether or not they are ignored in the lexical analysis of the compiler which will be further described in section 8.2.
Note that the dedent and indent tokens do not have a regular expression. The process of identifying these tokens are instead described in the implementation of the lexer in section 8.2.
Furthermore, the regular expressions marked with (**) indicate that a desired keyword cannot be followed by any letters, numbers, or underscore, which in practice is checked using negative lookahead in the regular expression, however, due to simplicity this has been left out in table 8.2.

Name	Regular Expression	Name	Regular Expression
lineComment (*)	//.*	return (**)	return
blockComment (*)	/*.**/	int8 (**)	int8
whitespace (*)	[\t␣]+	int16 (**)	int16
interrupt (**)	interrupt	int32 (**)	int32
+	\+	enableinterrupts (**)	enableinterrupts
!=	\!=	disableinterrupts (**)	disableinterrupts
!	\!	import	import [a-zA-Z0-9_.]*
%	%	register8 (**)	register8
^	\^	register16 (**)	register16
-	-	bool (**)	bool
*	*	==	==
/	/	<=	<=
(	\(	>=	>=
)	\)	<	<
newline	\r?\n[\r\n\t␣]+	>	>
numeral	([+-]?(0|([1-9][0-9]*)))	=	=
true (**)	true	,	,
false (**)	false	[	\[
if (**)	if	]	\]
else (**)	else	{	\{
for (**)	for	}	\}
while (**)	while	.	\.
and (**)	and	from (**)	from
or (**)	or	to (**)	to
nothing (**)	nothing	identifier	_*[a-zA-Z][a-zA-Z_0-9]

Table 7.1: Shows regular expressions for tokens in Tang.
(*) Ignored
(**) Must not be followed by letters or numbers matched by the regular expression [a-zA-Z_0-9]

The token specifications given in this section are used in the lexical analysis phases of the compiler which is described in section 8.2.

7.1.2 Grammar

The grammar for Tang is shown in listing 7.1 and is based on the theory for BNF described in 5.3.2. Note that all terminal symbols except the empty string symbol, `EPSILON`, start with a small letter. These terminals correspond to the tokens and their associated regular expression in table 7.1.
The start symbol for the grammar is `Program` and the end symbol is `eof`. A string is a part of a language defined by the CFG if it can be derived from the start symbol using 0 or more derivations, which formally is defined as: "$\{w \in \Sigma^* \mid S \Rightarrow^* w\}$" [48, p. 104].
The grammar for Tang is a LL(1) grammar based on the choice of parsing technique that will be described in chapter 8.
Based on the theory in 5.3.4, we know that a grammar is LL(1) if we are able to make a prediction using a look ahead of one. Furthermore, we know that the grammar is not LL(1) if we, given a rule R, have two or more productions with a common left factor.
To remove these in the grammar we use left factorisation. For instance if we have two rules like R -> a c and R -> a b then we would left factorise out the common 'a' terminal by rewriting these rules to R -> a RP and RP -> c | b.

```
1  Program -> GlobalStatements eof
2
3  GlobalStatements -> GlobalStatement GlobalStatements
4                    | EPSILON
5
6  GlobalStatement -> Interrupt
7                  | IdentifierDeclaration
8                  | IdentifierStatement
9                  | RegisterStatement
10                 | IfStatement
11                 | WhileStatement
12                 | ForStatement
13                 | ReturnStatement
14                 | InterruptStatement
15                 | newline
16
17 Interrupt -> interrupt ( numeral ) indent Statements dedent
18
19 Statements -> Statement Statements
20             | EPSILON
21
22 Statement -> IdentifierSimpleDeclaration
23            | IdentifierStatement
24            | RegisterSimpleStatement
25            | IfStatement
26            | WhileStatement
27            | ForStatement
28            | ReturnStatement
29            | InterruptStatement
30            | newline
```

Listing 7.1: Shows the context free grammar for Tang in Backus-Naur Form. Part 1

The purpose of `GlobalStatements` and `Statements`, as seen in listing 7.1 is to make functions inside functions impossible, as stated in section 6.3.
`GlobalStatements` can only exist in the outermost scope, while all other statements are of the form `Statements`.

```
32 RegisterSimpleStatement -> RegisterType RegisterSimpleOperation
33
34 RegisterSimpleOperation -> ( Expression ) { Expression } = Expression
      ↪ newline
35                          | identifier SimpleDefinition
36
37 IdentifierSimpleDeclaration -> IntType identifier SimpleDefinition
38                              | BoolenType identifier SimpleDefinition
39
40 SimpleDefinition -> = Expression newline
41                   | newline
```

```
42
43 IdentifierDeclaration -> IntType identifier Definition
44                       | BooleanType identifier Definition
45                       | nothing identifier ( FormalParameters )
                            ↪ indent Statements dedent
46
47 RegisterStatement -> RegisterType RegisterOperation
48
49 RegisterType -> register8
50               | register16
51
52 RegisterOperation -> ( Expression ) { Expression } = Expression newline
53                    | identifier Definition
54
55 Definition -> = Expression newline
56             | ( FormalParameters ) indent Statements dedent
57             | newline
58
59 InterruptStatement -> disableinterrupts ( )
60                     | enableinterrupts ( )
61
62 ReturnStatement -> return ReturnValue newline
63
64 ReturnValue -> Expression
65              | EPSILON
66
67 FormalParameters -> FormalParameter FormalParametersP
68                   | EPSILON
69
70 FormalParametersP -> , FormalParameter FormalParametersP
71                    | EPSILON
72
73 FormalParameter -> Type identifier
74
75 Type -> IntType
76       | RegisterType
77       | BooleanType
78
79 IdentifierStatement -> identifier IdentifierStatementP
80
81 IdentifierStatementP -> BitSelector = Expression newline
82                       | ( ExpressionList )
83
84 BitSelector -> { Expression }
85              | EPSILON
86
87 IfStatement -> if ( Expression ) indent Statements dedent ElseStatement
88
89 ElseStatement -> else ElseBlock
90                | EPSILON
91
```

```
92  ElseBlock -> IfStatement
93             | indent Statements dedent
94
95  WhileStatement -> while ( Expression ) indent Statements dedent
96
97  ForStatement -> for ( IntType identifier from Expression to Expression
        ↪ ) indent Statements dedent
98
99  IntType -> int8
100            | int16
101            | int32
102
103 BooleanType -> bool
```

Listing 7.2: Shows the context free grammar for Tang in Backus-Naur Form. Part 2

Listing 7.2 shows the possible statements in Tang, such as different control structures, functions, and variable declarations. According to this grammar, a statement such as $true + 5$ is possible, since we do not know the types of variables at this type. These are defined in section 6.5.

Listing 7.3 defines the expressions possible in Tang. The purpose of the hierarchical structure is to define the precedence of the different expressions.

```
108 Expression -> OrExpression
109
110 OrExpression -> AndExpression OrExpressionP
111 OrExpressionP -> or AndExpression OrExpressionP
112                | EPSILON
113
114 AndExpression -> EqExpression AndExpressionP
115 AndExpressionP -> and EqExpression AndExpressionP
116                 | EPSILON
117
118 EqExpression -> RelationalExpression EqExpressionP
119 EqExpressionP -> == RelationalExpression EqExpressionP
120               | != RelationalExpression EqExpressionP
121               | EPSILON
122
123 RelationalExpression -> AddSubExpression RelationalExpressionP
124 RelationalExpressionP -> < AddSubExpression RelationalExpressionP
125                        | > AddSubExpression RelationalExpressionP
126                        | <= AddSubExpression RelationalExpressionP
127                        | >= AddSubExpression RelationalExpressionP
128                        | EPSILON
129
130 AddSubExpression -> MulDivExpression AddSubExpressionP
131 AddSubExpressionP -> + MulDivExpression AddSubExpressionP
132                    | - MulDivExpression AddSubExpressionP
133                    | EPSILON
134
```

```
MulDivExpression -> PowExpression MulDivExpressionP
MulDivExpressionP -> / PowExpression MulDivExpressionP
                   | * PowExpression MulDivExpressionP
                   | % PowExpression MulDivExpressionP
                   | EPSILON

PowExpression -> PrimaryExpression PowExpressionP
PowExpressionP -> ^ PrimaryExpression PowExpressionP
                | EPSILON

PrimaryExpression -> numeral
                   | identifier IdentifierOperation
                   | ( Expression )
                   | ! PrimaryExpression
                   | RegisterType ( Expression ) BitSelector
                   | true
                   | false

IdentifierOperation -> BitSelector
                     | ( ExpressionList )

ExpressionList -> Expression ExpressionListP
                | EPSILON

ExpressionListP -> , Expression ExpressionListP
                 | EPSILON
```

Listing 7.3: Shows the context free grammar for Tang in Backus-Naur Form. Part 3

7.2 Structural Operational Semantics and Type System

After having outlined the syntax for Tang in section 7.1.2, which describes the set of programs that are syntactically valid, we will now look at the semantics and type system of Tang.
To do this, we base this section on the theory for structural operational semantics and type systems described in section 5.4.1 and 5.4.2, respectively.
In connection with this, we will describe the variables used in the semantic rules of Tang and establish an abstract syntax to give an overview of legal constructs in the language. All the rules for the structural operational semantics and type system can be seen in appendix C and D, respectively.

7.2.1 Abstract Syntax

P	$\in$ **Prog**	- Programs
S	$\in$ **Stm**	- Root level statements
s	$\in$ **SimpleStm**	- Scoped level statements
s_{if}	$\in$ **IfStm**	- If Statements
e	$\in$ **Exp**	- Expressions
b	$\in$ **BoolExp**	- Boolean expressions
i	$\in$ **IntExp**	- Integer expressions
x	$\in$ **VNames**	- Variable names
p	$\in$ **PNames**	- Procedure names
n	$\in$ **Num**	- Numerals
T_{Int}	$\in$ **IntTypes**	- Integer types {`int8`, `int16`, `int32`}
T_{Reg}	$\in$ **RegTypes**	- Register types {`register8`, `register16`}
T	$\in$ **Types**	- Types ($T_{Int} \cup T_{Reg} \cup$ {`bool`, `nothing`, `undefined`})
v	$\in$ **Val**	- Values ($\mathbb{Z} \cup \{tt, ff\}$)
r	$\in$ **Ret**	- Return values. **Ret** = (**Types** $\times$ **Val**) $\cup \epsilon$

Table 7.2: Syntactic categories

Table 7.2 shows the syntactical categories that will be used in the semantic specifications for Tang.

$$
\begin{array}{rcl}
P & ::= & S \mid \epsilon \\
S & ::= & S_1 \texttt{ newline } S_2 \mid s \mid T\ p(T_1\ x_1,\ T_2\ x_2,\ \ldots\ T_k\ x_k) \texttt{ indent } s \texttt{ dedent} \mid \texttt{interrupt}(n) \texttt{ indent } s \texttt{ dedent} \\
s & ::= & \texttt{newline} \mid e \mid s_1 \texttt{ newline } s_2 \mid T\ x \mid T\ x\ = e \mid x = e \\
 & \mid & x\{i\} = b \mid T_{Reg}(n)\{i\} = b \mid \texttt{return} \mid \texttt{return } e \mid \texttt{while}(b) \texttt{ indent } s \texttt{ dedent} \\
 & \mid & \texttt{for}(T_{Int}\ x \texttt{ from } i_1 \texttt{ to } i_2) \texttt{ indent } s \texttt{ dedent} \mid s_{if} \\
s_{if} & ::= & \texttt{if}(b) \texttt{ indent } s \texttt{ dedent} \mid \texttt{if}(b) \texttt{ indent } s \texttt{ dedent else } s_{if} \\
 & \mid & \texttt{if}(b) \texttt{ indent } s_1 \texttt{ dedent else indent } s_2 \texttt{ dedent} \\
e & ::= & x \mid i \mid b \mid (e_1) \mid p(e_1,\ e_2,\ \ldots\ e_k) \mid T_{Reg}(n) \\
i & ::= & n \mid e \mid i_1 + i_2 \mid i_1 - i_2 \mid i_1 * i_2 \mid i_1/i_2 \mid i_1 \texttt{ \% } i_2 \mid i_1\hat{}i_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid e \mid e_1 == e_2 \mid e_1 \texttt{!=} e_2 \mid i_1 < i_2 \mid i_1 <= i_2 \mid i_1 >= i_2 \mid i_1 > i_2 \mid \texttt{!}b_1 \\
 & \mid & b_1 \texttt{ and } b_2 \mid b_1 \texttt{ or } b_2 \mid x\{i\} \mid T_{Reg}(n)\{i\}
\end{array}
$$

Table 7.3: Formation rules where `newline`, `indent` and `dedent` are substitutes for their respective tokens along with operators etc.

The formation rules for Tang shown in table 7.3 and the syntactical categories in table 7.2 constitute the abstract grammar of Tang.

7.2.2 Structural Operational Semantics

The following model for representing memory locations and values stored at these locations are based on [53]. We start out by defining what locations are and how to get new available memory locations. Thereafter, we define the environment for variables denoted **EnvV** along with the environment for procedures denoted **EnvP**. A function **Sto** is also defined to map locations to their corresponding type-value pair. The type is stored so that e.g integer overflow can be detected. We use *sto* as a variable for an arbitrary member of **Sto**:

$$\begin{aligned}&\mathbf{Sto} = \mathbf{Loc} \rightharpoonup (\mathbf{Types} \times \mathbf{Val})\\&sto \in \mathbf{Sto}\end{aligned}$$

An update of sto is denoted $sto[l \mapsto (T, v)]$, where the updated sto environment, sto' is:

$$sto'y = \begin{cases} sto\ y & \text{if } y \neq l \\ (T, v) & \text{if } y = l \end{cases}$$

A location points to a specific place in memory and we therefore define the set of locations **Loc** to be the natural numbers where a variable l denotes an arbitrary location:

$$\begin{aligned}&\mathbf{Loc} = \mathbb{N}\\&l \in \mathbf{Loc}\end{aligned}$$

Furthermore a new location is defined by the function below where new returns a new unused location [53, p. 81].

$$\begin{aligned}&new : \mathbf{Loc} \to \mathbf{Loc}\\&new\ l = l + 1\end{aligned}$$

Next we define the set of partial functions **EnvV** that map from a variable name to a location and where env_v is an element in **EnvV**:

$$\begin{aligned}&\mathbf{EnvV} = \mathbf{VNames} \cup \{\text{next}\} \rightharpoonup \mathbf{Loc}\\&env_V \in \mathbf{EnvV}\end{aligned}$$

We denote an update to the variable environment as follows: $env_V[x \mapsto l]$, where the variable name x is mapped to a location l which produces an updated environment env'_V where env'_V is defined as:

$$env'_V\ y = \begin{cases} env_V\ y & \text{if } y \neq x \\ l & \text{if } y = x \end{cases}$$

Finally, we define the set of partial functions **EnvP** where env_p denotes an arbitrary member of **EnvP**:

$$\mathbf{EnvP} = \mathbf{PNames} \rightharpoonup (\mathbf{EnvV} \times \mathbf{SimpleStm} \times (\mathbf{VNames}^k))$$

An update of **EnvP** on the form $env_P[p \mapsto (env_V, s, (x_1, x_2,\ \ldots\ x_k))] = env'_P$ is defined as follows:

$$env'_P\ q = \begin{cases} env_P\ q & \text{if } q \neq p \\ (env_V, s, (x_1, x_2,\ \ldots\ x_k)) & \text{if } q = p \end{cases}$$

The function `default` evaluates to the default value of a type.

$$\begin{aligned}&default : \mathbf{Types} \rightharpoonup \mathbf{Val}\\&default\ T = \begin{cases} 0 & \text{if } T \in \{\texttt{int8}, \texttt{int16}, \texttt{int32}, \texttt{register8}, \texttt{register16}\} \\ ff & \text{if } T = \texttt{bool} \end{cases}\end{aligned}$$

The *set* of values that can be stored in a type is shown in the following table.

Type	Set(Type)
`int8`	$\{-2^{8-1}\ to\ 2^{8-1}-1\}$
`int16`	$\{-2^{16-1}\ to\ 2^{16-1}-1\}$
`int32`	$\{-2^{32-1}\ to\ 2^{32-1}-1\}$
`bool`	$\{tt, ff\}$
`undefined`	$\varnothing$
`register8`	**Loc**
`register16`	**Loc**
`nothing`	$\{NothingVal\}$

Furthermore, we define the following type relation for integers: $int8 \supset int16 \supset int32$.
We also define the function `largestIntegerType` that given a number of types checks if all types are integer types and if they are it returns the biggest integer type that is given to the function.

$$largestIntegerType(T_{Int_1}, T_{Int_2}, \ldots T_{Int_k}) = T_{Int_j} \text{ where } 1 \leq j \leq k \text{ and } T_{Int_i} \supseteq T_{Int_j} \text{ for all } 1 \leq i \leq k$$

Programs

The semantics of **Prog** are given by the transition system $(\Gamma_P, \rightarrow_P, T_P)$ whose configurations are defined by:

$$\Gamma_P = \mathbf{Prog} \cup \mathbf{Ret}$$
$$T_P = \mathbf{Ret}$$

Transitions here have the format $P \rightarrow_P r$ Where execution of a program P results in a return value r. The transition rule $[Statement]$ in table 7.4 states that a non empty program will be executed by scanning the program to fill out a procedure environment and then executing the statement S with respect to an empty store, variable environment and the newly filled procedure environment. The empty variable environment is denoted $env_V^{\varnothing}$.

$$[Statement] \quad \frac{\langle S, env_V^{\varnothing}, env_P^{\varnothing}\rangle \rightarrow_{Scan} (env'_V, env'_P) \quad env'_P \vdash \langle S, sto^{\varnothing}, env_V^{\varnothing}\rangle \rightarrow_S (sto, env_V, r)}{S \rightarrow_P r}$$

$$[empty] \quad \epsilon \rightarrow_P (\texttt{nothing}, nothingval)$$

Table 7.4: The transition rules $\rightarrow_P$

Scan System

In order to call procedures declared after a call statement to the procedure, we use a scan system that fills out the procedure environment before any program code is executed. The **Scan** transition system is defined as such: $(\Gamma_{Scan}, \rightarrow_{Scan}, T_{Scan})$. Its configurations are defined by

$$\Gamma_{Scan} = ((\mathbf{Stm} \cup \mathbf{SimpleStm}) \times \mathbf{EnvP} \times \mathbf{EnvV}) \cup (\mathbf{EnvP} \times \mathbf{EnvV})$$
$$T_{Scan} = \mathbf{EnvP} \times \mathbf{EnvV}$$

The following transitions are for the subset of **Scan** configurations ($\mathbf{Stm} \times \mathbf{EnvP} \times \mathbf{EnvV}) \cup (\mathbf{EnvP} \times \mathbf{EnvV}$) and have the format $\langle S, env_P, env_V\rangle \rightarrow_{Scan} (env'_P, env'_V)$. The rule $[ScanProcedureDeclaration]$ stores a procedure in the procedure environment together with the current variable environment which is filled by the rule $[ScanVariableDeclaration]$.
As such, the procedure will have knowledge of all variables defined prior to the procedure being declared. Since procedures can only be declared in the global scope and variables defined in sub-scopes are not available to the surrounding scope, the scan system does not define transition rules for sub-scopes, as can be seen in rule $[ScanWhileStatement]$.

$$[ScanProcedureDeclaration] \quad \langle T\ p(T_1\ x_1,\ T_2\ x_2,\ \ldots\ T_k\ x_k)\ \texttt{indent}\ s\ \texttt{dedent}, env_P, env_V\rangle \rightarrow_{Scan} (env_P[p \mapsto (env_V, s, (x_1,\ x_2,\ \ldots\ x_k))], env_V)$$

Table 7.5: Subset of the transition rules $\rightarrow_{Scan}$

The following transitions are for the subset of **Scan** configurations ($\mathbf{SimpleStm} \times \mathbf{EnvP} \times \mathbf{EnvV}) \cup (\mathbf{EnvP} \times \mathbf{EnvV}$) and have the format $\langle s, env_P, env_V\rangle \rightarrow_{Scan} (env'_P, env'_V)$.

$$[ScanVariableDeclaration] \quad \langle T\ x, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V[x \mapsto l][next \mapsto new(l)]) \text{ Where } l = env_V(next)$$

$$[ScanVariableDefinition] \quad \frac{\langle T\ x, env_P, env_V\rangle \rightarrow_{Scan} (env'_P, env'_V)}{\langle T\ x = e, env_P, env_V\rangle \rightarrow_{Scan} (env'_P, env'_V)}$$

$$[ScanWhileStatement] \quad \langle \texttt{while}(b)\ \texttt{indent}\ s\ \texttt{dedent}, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V)$$

Table 7.7: Subset of the transition rules $\rightarrow_{Scan}$

Statements

The semantics of **Stm** are given by the transition system $(\Gamma_S, \rightarrow_S, T_S)$ whose configurations are defined by

$$\Gamma_S = (\mathbf{Stm} \times \mathbf{Sto} \times \mathbf{EnvV}) \cup (\mathbf{Sto} \times \mathbf{EnvV} \times \mathbf{Ret})$$
$$T_S = (\mathbf{Sto} \times \mathbf{EnvV} \times \mathbf{Ret})$$

Transitions here have the format $env_P \vdash \langle S, sto, env_V\rangle \rightarrow_S (sto', env'_V, r)$ where the execution of a statement, S, in a store, sto and variable environment env_V evaluates to an updated store and variable environment as well as a return value.

The transitions for compound statements, as defined below, describe the semantics of sequential statement execution and states that: given two statements S_1 and S_2 that are separated by a newline then if execution of S_1 does not produce a return value r, the second statement is executed, otherwise it is not and the return value is propagated. Statements may update the store and define new variables. These updates are reflected in the resulting store and variable environment.

$$[CompoundStatement_1] \quad \frac{env_P \vdash \langle S_1, sto, env_V\rangle \rightarrow_S (sto', env'_V, r)}{env_P \vdash \langle S_1 \texttt{ newline } S_2, sto, env_V\rangle \rightarrow_S (sto', env'_V, r)}$$
If $r \neq \epsilon$

$$[CompoundStatement_2] \quad \frac{\begin{array}{c} env_P \vdash \langle S_1, sto, env_V\rangle \rightarrow_S (sto'', env''_V, r_1) \\ env_P \vdash \langle S_2, sto'', env''_V\rangle \rightarrow_S (sto', env'_V, r_2) \end{array}}{env_P \vdash \langle S_1 \texttt{ newline } S_2, sto, env_V\rangle \rightarrow_S (sto', env'_V, r_2)}$$
If $r_1 = \epsilon$

Table 7.8: The transition rules $\rightarrow_S$

Simple Statements

The semantics of **SimpleStm** are given by the transition system $(\Gamma_s, \rightarrow_s, T_s)$ whose configurations are defined by

$$\Gamma_s = (\mathbf{SimpleStm} \times \mathbf{Sto} \times \mathbf{EnvV}) \cup (\mathbf{Sto} \times \mathbf{EnvV} \times \mathbf{Ret})$$
$$T_s = (\mathbf{Sto} \times \mathbf{EnvV} \times \mathbf{Ret})$$

Transitions here have the format $env_P \vdash \langle s, sto, env_V\rangle \rightarrow_s (sto', env'_V, r)$.
An example of these transitions are the $[RegisterSetBit]$ rule seen in table 7.10. This enables the programmer to set the i'th bit in hardware register x.
The first premise does not get the value stored in register x but instead the address as well as the type T_r which is either `register8` or `register16` depending on what x was declared as. The other premises evaluate the boolean and integer expressions. Two conditions verify that the bit index is within the allowed range for the register type.
A new value v' for the register is calculated, then the value at locality l gets replaced by v' in sto' and the transition is complete.

$$[RegisterSetBit] \quad \frac{\begin{array}{c} env_P, env_V \vdash \langle x, sto\rangle \rightarrow_e (sto''', (T_r, l)) \\ env_P, env_V \vdash \langle i, sto'''\rangle \rightarrow_i (sto'', (T_i, v_i)) \\ env_P, env_V \vdash \langle b, sto''\rangle \rightarrow_b (sto', (T_{b_2}, v_{b_2})) \end{array}}{env_P \vdash \langle x\{i\} = b, sto, env_V\rangle \rightarrow_s (sto'[l \mapsto v'], env_V, \epsilon)}$$
Where $(T_v, v) = sto'\ l$

$$\text{And } v_{b_1} = \begin{cases} tt & \text{if } \left\lfloor \frac{v}{2^{(v_i)}} \right\rfloor mod\ 2 = 1 \\ ff & \text{if } \left\lfloor \frac{v}{2^{(v_i)}} \right\rfloor mod\ 2 = 0 \end{cases}$$

$$\text{And } v' = \begin{cases} v - 2^{(v_i)} & v_{b_1} \wedge \neg v_{b_2} \\ v & v_{b_1} = v_{b_2} \\ v + 2^{(v_i)} & \neg v_{b_1} \wedge v_{b_2} \end{cases}$$
If $v_i \geq 0$

$$\text{And } v_i < \begin{cases} 8 & \text{If } T_r = \texttt{register8} \\ 16 & \text{If } T_r = \texttt{register16} \end{cases}$$

Table 7.10: The transition rules $\rightarrow_s$

Another example of a simple statement transition is the $[VariableDefinition]$ rule.
This rule defines how variables are declared and initialized with a value.
First, the right side of the expression is evaluated to get the value to assign to the variable.
We then evaluate just the type and variable name to get the variable initialised with the default value.
We can then update the sto' returned by the evaluation of e and $T\ x$ and update the store entry, that x references, with the value e evaluated to.
The reason for evaluating e before $T\ x$ is to avoid confusion if you wrote `int8 x = x` and more complex examples of bad coding where the default value is used in a way it was not designed for.

$[VariableDeclaration]$ $\langle T\ x, sto, env_V\rangle \rightarrow_s (sto[l \mapsto (T, default(T))], env_V[x \mapsto l][next \mapsto new(l)], \epsilon)$
Where $l = env_V(next)$

$$[VariableDefinition] \quad \frac{env_P, env_V \vdash \langle e, sto\rangle \rightarrow_e (sto'', (T_e, v)) \quad env_P \vdash \langle T\ x, sto'', env_V\rangle \rightarrow_s (sto', env'_V, \epsilon)}{env_P \vdash \langle T\ x = e, sto, env_V\rangle \rightarrow_s (sto'[l \mapsto (T, v)], env'_V, \epsilon)}$$
Where $env'_V(x) = l$
If $v \in Set(T)$

Table 7.12: The transition rules $\rightarrow_s$

If-statements

The semantics of **IfStm** are given by the transition system $(\Gamma_{s_{if}}, \rightarrow_{s_{if}}, T_{s_{if}})$ whose configurations are defined by:

$$\Gamma_{s_{if}} = (\mathbf{IfStm} \times \mathbf{Sto}) \cup (\mathbf{Sto} \times \mathbf{Ret})$$
$$T_{s_{if}} = (\mathbf{Sto} \times \mathbf{Ret})$$

Transitions here have the format $env_P, env_V \vdash \langle s_{if}, sto\rangle \rightarrow_{s_{if}} (sto', r)$ and as with **SimpleStm**, **IfStm** has the effect of modifying the store and returning a return value r.
Evaluation of an **IfStm** does not result in changes to the current scopes variable environment. This means that any variables defined inside the if statement are not accessible outside of it.
Hereafter follows a subset of the **IfStm** transitions. An $[IfElseStatement]$ evaluates b and if b evaluates to tt then s_1 is evaluated, else if b evaluates to ff then s_2 is evaluated next.

$$[IfElseStatement_1] \quad \frac{\begin{array}{c} env_P, env_V \vdash \langle b, sto\rangle \rightarrow_b (sto'', (T, v)) \\ env_P \vdash \langle s_2, sto'', env_V\rangle \rightarrow_s (sto', env'_V, r) \end{array}}{env_P, env_V \vdash \langle \texttt{if}(b)\ \texttt{indent}\ s_1\ \texttt{dedent else indent}\ s_2\ \texttt{dedent}, sto\rangle \rightarrow_{s_{if}} (sto', r)}$$
If $v = ff$

$$[IfElseStatement_2] \quad \frac{\begin{array}{c} env_P, env_V \vdash \langle b, sto\rangle \rightarrow_b (sto'', (T, v)) \\ env_P \vdash \langle s_1, sto'', env_V\rangle \rightarrow_s (sto', env'_V, r) \end{array}}{env_P, env_V \vdash \langle \texttt{if}(b)\ \texttt{indent}\ s_1\ \texttt{dedent else indent}\ s_2\ \texttt{dedent}, sto\rangle \rightarrow_{s_{if}} (sto', r)}$$
If $v = tt$

Table 7.13: The transition rules $\rightarrow_{s_{if}}$

Expressions

The semantics of **Exp** are given by the transition system $(\Gamma_e, \rightarrow_e, T_e)$ whose configurations are defined by:

$$\Gamma_e = (\mathbf{Exp} \times \mathbf{Sto}) \cup (\mathbf{Sto} \times (\mathbf{Types} \times \mathbf{Val}))$$
$$T_e = (\mathbf{Sto} \times (\mathbf{Types} \times \mathbf{Val}))$$

Transitions here have the format $env_P, env_V \vdash \langle e, sto\rangle \rightarrow_e (sto', (T, v))$.
The semantics of variable evaluation is described by the transition $[VarEvaluation]$. A variable is evaluated by looking up its address and then getting the value of that address via the store.

$$[VarEvaluation] \quad \begin{array}{l} env_P, env_V \vdash \langle x, sto\rangle \rightarrow_e (sto, (T, v)) \\ \text{Where } sto(env_V\ x) = (T, v) \end{array}$$

Table 7.14: The transition rules $\rightarrow_e$

Integer Expressions

The semantics of **IntExp** are given by the transition system $(\Gamma_i, \rightarrow_i, T_i)$ whose configurations are defined by:

$$\Gamma_i = (\mathbf{IntExp} \times \mathbf{Sto}) \cup (\mathbf{Sto} \times (\mathbf{Types} \times \mathbf{Val}))$$
$$T_i = (\mathbf{Sto} \times (\mathbf{Types} \times \mathbf{Val}))$$

Transitions here have the format $env_P, env_V \vdash \langle i, sto\rangle \rightarrow_i (sto', (T, v))$.
In the rules for the transition system for **IntExp** all the big-step semantic rules for arithmetic operations have been defined along with the rule for numerals to integers.
To convert numerals to integers we define the function $\mathcal{N} : Num \rightarrow \mathbb{Z}$, that for every numeral returns the integer representation of that numeral [53].
An example of an arithmetic rule is the following semantic rule for integer division where we evaluate the two operands of the expression to integers and divide v_1 by v_2.

$$[ConstantToInt] \quad env_P, env_V \vdash \langle n, sto\rangle \rightarrow_i (sto, (T, v))$$
Where $\mathcal{N}[\![n]\!] = v$
And $T = \begin{cases} int8 & \text{If } v \in Set(int8) \\ int16 & \text{If } v \in Set(int16) \text{ and } v \notin Set(int8) \\ int32 & \text{If } v \in Set(int32) \text{ and } v \notin Set(int16) \end{cases}$
If $v \in Set(int32)$

$$[IntegerDivision] \quad \frac{env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'', (T_1, v_1)) \quad env_P, env_V \vdash \langle i_2, sto''\rangle \rightarrow_i (sto', (T_2, v_2))}{env_P, env_V \vdash \langle i_1 / i_2, sto\rangle \rightarrow_i (sto', (T_3, v_3))}$$
Where $v_3 = \lfloor v_1 / v_2 \rfloor$
And $T_3 = largestIntegerType(T_1, T_2)$
If $v_2 \neq 0$
And $v_3 \in Set(T_3)$

Table 7.15: The transition rules $\rightarrow_i$

Boolean Expressions

The semantics of **BoolExp** are given by the transition system $(\Gamma_b, \rightarrow_b, T_b)$ whose configurations are defined by:

$$\Gamma_i = (\mathbf{BoolExp} \times \mathbf{Sto}) \cup (\mathbf{Sto} \times (\mathbf{Types} \times \mathbf{Val}))$$
$$T_i = \mathbf{Sto} \times (\mathbf{Types} \times \mathbf{Val})$$

Transitions here have the format $env_P, env_V \vdash \langle b, sto\rangle \rightarrow_b (sto', (T, v))$.
Here follows two examples of transitions in the system.
$[Equals]$ describes that two expressions are equal if they evaluate to the same value. $[RegisterBitToBool]$ describes the semantics of getting a single bit from a register variable.
Firstly, the integer expression i is evaluated to an integer value. Then the value in the register is accessed by following the reference stored in x and afterwards we determine if the bit indicated by v_i is a 1 or a 0.

$$[Equals] \quad \frac{env_P, env_V \vdash \langle e_1, sto\rangle \rightarrow_e (sto'', (T_1, v_1)) \quad env_P, env_V \vdash \langle e_2, sto''\rangle \rightarrow_e (sto', (T_2, v_2))}{env_P, env_V \vdash \langle e_1 \texttt{ == } e_2, sto\rangle \rightarrow_b (sto', (\texttt{bool}, v_3))}$$
Where $v_3 = \begin{cases} tt & \text{if } v_1 = v_2 \\ ff & \text{if } v_1 \neq v_2 \end{cases}$

$$[RegisterBitToBool] \quad \frac{env_P, env_V \vdash \langle i, sto\rangle \rightarrow_i (sto', (T_i, v_i))}{env_P, env_V \vdash \langle x\{i\}, sto\rangle \rightarrow_b (sto', (\texttt{bool}, v))}$$
Where $(T_{Reg}, l) = sto(env_V\ x)$
And $(T_r, v_r) = sto(l)$
And $v = \begin{cases} tt & \text{if } \left\lfloor \frac{v_r}{2^{(v_i)}} \right\rfloor \ mod\ 2 = 1 \\ ff & \text{if } \left\lfloor \frac{v_r}{2^{(v_i)}} \right\rfloor \ mod\ 2 = 0 \end{cases}$

Table 7.16: The transition rules $\rightarrow_b$

7.2.3 Type System

In this section, we define the rules for the type system of Tang. The different rules are grouped together according to their syntactical category as seen in the abstract syntax which can be seen in table 7.3. This section shows some of the type rules for Tang. The type rules are shown in their full extent in Appendix D and the following is an excerpt of these rules.
To map variable and procedure names to their corresponding type we define a partial function, the type environment, E as follows:

$$E : (\mathbf{VNames} \cup \mathbf{PNames}) \rightharpoonup \mathbf{Types}^k$$

Here k is an arbitrate number and is used to denote that an identifier can map to a tuple with an arbitrary number of elements.
An update of the type environment is written as follows $E[x \mapsto (T_1, T_2, \ldots T_k)]$ where the resulting Type environment E' is:

$$E'(y) = \begin{cases} E(y) & \text{if } y \neq x \\ (T_1, T_2, \ldots T_k) & \text{if } y = x \end{cases}$$

When defining new variables we check if the variable-name is already defined in the type environment, before updating the type-environment. We do this by returning `undefined` if the variable is not yet defined in the type environment and making a check in each specific case.

$$E^{\varnothing}(y) = (\texttt{undefined})$$

Programs

Type rules for program statements in Tang has the following type judgement: $E \vdash P : \mathsf{ok}$.
Table 7.17 shows the $[Empty]$ type rule for program statements, which states that our program is well typed if the program is empty.
Furthermore, the $[Program]$ type rule's side condition: $E = E_{Scan}(S, E^{\varnothing}[scope \mapsto \texttt{nothing}])$ will invoke the auxiliary functions in table 7.25, that will recursively go through the source program and put all procedures into the type environment which corresponds to the scope rules described in section 6.5.

$[Program]$	$E \vdash S : \mathsf{ok}$	Where $E = E_{Scan}(S, E^{\varnothing}[scope \mapsto \texttt{nothing}])$
$[Empty]$	$E^{\varnothing}[scope \mapsto \texttt{nothing}] \vdash \varepsilon : \mathsf{ok}$	

Table 7.17: Type rules for Programs

Statements

Type rules for root level statements in Tang has the following type judgement: $E \vdash S : \mathsf{ok}$.
Two type rules for statements can be seen in table 7.18. A $[CompoundStatement]$ is well-typed if

both the first and second statement are well-typed.
Whereas an [*ProcedureDeclaration*] is well-typed if the declaration of each of the formal arguments as variables is well-typed and the function body is well typed.

[*CompoundStatement*]	$\dfrac{E \vdash S_1 : \mathsf{ok} \quad E_1 \vdash S_2 : \mathsf{ok}}{E \vdash S_1 \ \mathtt{newline}\ S_2 : \mathsf{ok}}$	**Where** $E_1 = E(S_1, E)$
[*ProcedureDeclaration*]	$\dfrac{\begin{array}{c} E \vdash T_1\ x_1 : \mathsf{ok} \\ E_1 \vdash T_2\ x_2 : \mathsf{ok} \\ E_2 \vdash T_3\ x_3 : \mathsf{ok} \\ \dots \\ E_{k-1} \vdash T_k\ x_k : \mathsf{ok} \\ E_k[scope \mapsto (T)] \vdash s : \mathsf{ok} \end{array}}{E \vdash T\ p(T_1\ x_1,\ T_2\ x_2,\ \dots\ T_k\ x_k)\ \mathtt{indent}\ s\ \mathtt{dedent} : \mathsf{ok}}$	If $E(p) \neq (\mathtt{undefined})$ Where $E_1 = E(T_1\ x_1, E)$ and $E_2 = E(T_2\ x_2, E_2)$ ... and $E_k = E(T_k\ x_k, E_k)$

Table 7.18: Type rules for Root level statements

Scoped Level Statements

Type rules for scoped level statements in Tang has the following type judgement: $E \vdash s : \mathsf{ok}$.
Table 7.19 shows some of he type rules for the scoped level statements in Tang.
The [$VariableDefinition$] rule states that declaring a variable x of type T and initialising it to the value of expression e, is type correct if e is of the same type, or a type that can implicitly be converted to and if x is not yet defined in the type-environment.
[*Return*] and [$ReturnWithValue$] is well-typed as long as the value returned is of the type that the enclosing scope expects.

[$VariableDefinition$]	$\dfrac{E \vdash e : T_1}{E \vdash T\ x = e : \mathsf{ok}}$	If $E(x) = (\mathtt{undefined})$ if $T \neq \mathtt{nothing}$ if $T = T_1$ or $T \in$ **IntTypes** and $T_1 \in$ **IntTypes** and $T = largestIntegerType(T, T_1)$
[*Return*]	$E \vdash \mathtt{return} : \mathsf{ok}$	If $E(scope) = (\mathtt{nothing})$
[$ReturnWithValue$]	$\dfrac{E \vdash e : T}{E \vdash \mathtt{return}\ e : \mathsf{ok}}$	Where $(T_{scope}) = E(scope)$ If $T_{scope} = (T)$ or $T_{scope} \in$ **IntTypes** and $T \in$ **IntTypes** and $T_{scope} = largestIntegerType(T, T_{scope})$

Table 7.19: Type rules for scoped level statements

If-statements

Type rules for if statements in Tang has the following type judgement: $E \vdash s_{if} : \mathsf{ok}$.
In table 7.20 a type rule for if statements is shown. The $[IfStatement]$ rule states that an if statement is well-typed if b is a boolean expression of type bool and the statement s is well-typed.

$$[IfStatement] \quad \frac{E \vdash b : bool \qquad E \vdash s : \mathsf{ok}}{E \vdash \texttt{if}(b)\ \texttt{indent}\ s\ \texttt{dedent} : \mathsf{ok}}$$

Table 7.20: Type rules for if statements

Expression

Type rules for expressions in Tang has the following type judgement: $E \vdash e : T$.
An excerpt of the type rules for expressions are shown in table 7.21. The rule $[LeftValueEvaluation]$ states that the type of an identifier x is some type T if given the type environment E x's type is T.
The type rule $[ProcedureCallExpression]$ states that a procedure call is type correct if the actual parameters match the type of their associated formal parameters.

$$[LeftValueEvaluation] \qquad E \vdash x : T$$
$$\text{Where } (T) = E(x) \text{ and } T \neq \texttt{undefined}$$

$$[ProcedureCallExpression] \quad \frac{\begin{array}{c} E \vdash e_1 : T_1 \\ E \vdash e_2 : T_2 \\ \dots \\ E \vdash e_k : T_k \end{array}}{E \vdash p(e_1,\ e_2,\ \dots\ e_k) : T_r}$$
$$\text{Where } (T_r,\ T_1,\ T_2,\ \dots\ T_k) = E(p)$$

Table 7.21: Type rules for expressions

Integer Expressions

Type rules for integer expression in Tang has the following type judgement: $E \vdash i : T_{Int}$.
An excerpt of type rules for integer expressions are shown in table 7.22 where the $[Numeral]$ rule states that any numeral n is of type `integer`.
Furthermore all integer-operations consist of 2 integers and evaluate to an integer and are almost identical. The example shown is $[AddOperation]$.

$$[Numeral] \quad E \vdash n : T_{Int}$$

Where $v = \mathcal{N}[\![n]\!]$

And $T_{Int} = \begin{cases} int8 & \text{if } v \in Set(int8) \\ int16 & \text{if } v \in Set(int16) \text{ and } v \notin Set(int8) \\ int32 & \text{if } v \in Set(int32) \text{ and } v \notin Set(int16) \end{cases}$

$$[AddOperation] \quad \frac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash i_1 \ + \ i_2 : T_{Int}}$$

Where $T_{Int} = largestIntegerType(T_{Int_1}, T_{Int_2})$

Table 7.22: Type rules for integer expressions

Boolean Expressions

Type rules for boolean expressions in Tang have the following type judgement: $E \vdash b : bool$.
Examples of type rules for boolean expressions in Tang are shown in table 7.23. The rules $[True]$ and $[False]$ states that the keywords `true` and `false` are both of type `bool`.
Whereas the rule $[EqualBool]$ states that the equality operation to check whether or not two operands b_1 and b_2 are equal, is type correct and of type bool if both operands are of type bool.
$[LessThan]$ compares 2 integers and evaluates to a bool if both integers is type-correct.
$[RegisterGetBit]$ takes a register and an integer and finds the nth bit of register x, and returns it as a bool.

$$[True] \quad E \vdash \texttt{true} : bool$$

$$[False] \quad E \vdash \texttt{false} : bool$$

$$[EqualBool] \quad \frac{E \vdash b_1 : bool \quad E \vdash b_2 : bool}{E \vdash b_1 \ == \ b_2 : bool}$$

$$[LessThan] \quad \frac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash i_1 \ < \ i_2 : bool}$$

$$[RegisterGetBit] \quad \frac{E \vdash x : T_{Reg} \quad E \vdash i : T_{Int_2}}{E \vdash x\{i\} : bool}$$

Table 7.23: Type rules for boolean expressions

Auxilary Functions

Table 7.24 show auxiliary functions for updating the type environment.
An update of the type environment is relevant whenever we have a compound statement in a program.
Consider an example where we first have a declaration of a variable, x followed by a second statement that updates x. The second statement must know whether or not x exists and if so what its type is. The update of the type environment comes in form of the side condition of the $[CompoundStatement]$ rule in table 7.18 which applies one of the auxiliary function of table 7.24.
If we have a compound statement S_1 newline S_2 then if S_1 is a variable declaration of the form $T\ x$

the auxiliary function $[VarAux]$ for updating a variable x with type T is called before evaluating the S_2 statement. The structure of statement S_1 determines which of the auxiliary functions is applied.

$[CompoundAux_1]$	$E(S_1 \texttt{ newline } S_2, E) = E(S_2, E(S_1, E))$
$[CompoundAux_2]$	$E(s_1 \texttt{ newline } s_2, E) = E(s_2, E(s_1, E))$
$[VarAux]$	$E(T\ x, E) = E[x \mapsto (T)]$
$[VarInitAux]$	$E(T\ x = e, E) = E[x \mapsto (T)]$

Table 7.24: Auxiliary functions for updating the type environment

Table 7.25 shows the scan auxiliary functions which is used to fill a type environment with the procedures of a program in a similar manner to that of the auxiliary functions of table 7.24.

$[CompoundAuxScan_1]$	$E_{Scan}(S_1 \texttt{ newline } S_2, E) = E_{Scan}(S_2, E_{Scan}(S_1, E))$
$[CompoundAuxScan_2]$	$E_{Scan}(s_1 \texttt{ newline } s_2, E) = E_{Scan}(s_2, E_{Scan}(s_1, E))$
$[ProcAuxScan]$	$E_{Scan}(T_r\ p(T_1\ x_1,\ T_2\ x_2,\ \ldots\ T_k\ x_k) \texttt{ indent } s \texttt{ dedent}, E) = E[p \mapsto (T_r,\ T_1,\ T_2,\ \ldots\ T_k)]$ If $E(p) = (\texttt{undefined})$

Table 7.25: Auxiliary functions for updating the type environment

7.3 Remaining Language Features

The following will describe the limitations of the syntax and semantic descriptions for Tang.

Missing features Classes, arrays, strings and chars have not been formally described in Tang as these were prioritised as being less important features in the MoSCoW (see section 6.6).

All branches return The type system does not ensure that all branches in a function returns a value.

Interrupts Lastly, the semantics of interrupts are not fully described, as interrupts can interrupt execution after any single hardware instruction, whereas for example the semantics of division might span several hardware instructions. with no simple mechanism for describing execution of some unrelated code at any point.

Informal description of interrupts An interrupt statement: `interrupt`(n) `indent` s `dedent` fills the interrupt vector on location n so that if interrupt n is triggered, the body s will be called, assuming that the global interrupts are enabled and that interrupt n is configured in registers (see Atmega328P datasheet [20]).

Chapter 8

Implementation

Based on the formal specification of Tang in section 7, we are now able to implement a compiler for Tang.
In this section we describe the implementation of a compiler for Tang based on the following phases of the compiler: lexical analysis, parsing, parse-tree to AST, type checking, and code generation.

8.1 Parser Implementation Approaches

Before the parser is implemented, the different approaches for doing this will be examined. Three different approaches will be discussed.
The first approach is to use a tool to generate the parser based on a grammar for the Tang language.
The second approach is to implement the parser manually without using parser generating tools.
The third approach is to develop a parser generator tool ourselves and use this to generate a parser.
Finally, we discuss and decide which approach we will use in this project.

8.1.1 Parser Generating Tools

The tools examined will all support C# as parser implementation language, as this is the implementation language chosen for Tang in section 6.7.3. If a tool requires Visual Studio integration, it is also a requirement that it will work with Visual Studio 2017, as this is the version we are using. Each tool examined in this section will be evaluated in regards to the following points:

- Documentation of the tool, including tutorials, community etc.
- Ease of use including e.g. error messages.

As it is the first time the group has engaged in the making of a compiler, we have chosen to keep the tool's beginner friendliness in mind.

The tokens seen in listing 8.2 are outdated and therefore do not match the tokens explained in table 7.1 exactly. For each tool, the grammar for Tang has been modified to fit the syntax. The different grammars can be seen in appendix L.

```
Program -> Statements
Statements  -> Statement StatementsP
StatementsP -> newline Statement StatementsP
            |  EPSILON
Statement -> simpleType identifier Definition
          |  identifier assign integer
          |  newline
Definition -> assign integer
           |  EPSILON
```

Listing 8.1: Shows the grammar which will be used in the following examples of parsing generator tools.

```
simpleType identifier assign integer newline simpleType identifier
   ↪ assign integer newline identifier assign integer EPSILON
```

Listing 8.2: Shows the input token-stream used to test different parsing generator tools.

The input example used can be seen in listing 8.2.

Coco/R

"*Coco/R is a compiler generator, which takes an attributed grammar of a source language and generates a scanner and a parser for this language.*" [62].
The compiler generator accepts languages with an arbitrary number of lookahead symbols, which means it accepts LL(k) languages [63, p. 1]. It makes use of the recursive descent parsing technique [62].
The compiler description language for Coco/R is called Cocol/R and makes use of EBNF to specify its grammar.
The official documentation of Coco/R can be found as a user manual on their website [63]. The manual is in depth and covers areas such as input language, installation guide, and error messages. No official communities for Coco/R exist. Instead a PowerPoint tutorial is linked to from their website as well as a more in-depth text tutorial [62].

By following the informative tutorial and the user manual, Coco/R is relatively easy to use. It specifies errors and their locations. It also outputs an understandable LL(1) warning, however, we experienced that continuing compiling after the warning resulted in process termination due to StackOverflowException, which also crashed the command prompt on the Windows testing machine.

GOLD

The GOLD parser tool is a versatile tool, used to generate LALR parsers, for a given language. According to [64], the GOLD parser tool supports more programming languages than any other parser tool tested in this section. On the GOLD parser website, a concise and relatively easy to understand guide is listed. On the same website, a tool can be downloaded, called the GOLD parser builder. The tool is used to test a given grammar, designed by the user, and to generate a skeleton program for a parser.

The syntax used for a grammar in the GOLD tool, resembles the standard BNF syntax [65]
The tool is capable of analysis and test of a grammar.
If the grammar is accepted, a LALR table and DFA state table are generated.

As the *GOLD* tool does not create anything other than the barebone structure of the parser program, it involves some manual development from the user, which is a disadvantage compared to other tools analysed.

SableCC

SableCC is a LALR(1) parser [66].
According to [67, p. 21]: *"SableCC represents the result of our research to develop a Java compiler compiler that meets new compiler implementation trends."*.
The research has resulted in the parser achieving results such as support for building an AST of the compiled program and preventing corruption by making each AST node strictly typed [67, p. 22].
The main website for SableCC [68] has links to a documentation page, with links to tutorials and more [69]. It also has links to an inactive community located on the Google forums and to the source code of SableCC on GitHub.

SableCC is fairly straightforward to use, but requires the user to write their own interpreter/-translator class [70, p. 181]. As SableCC can be used with Eclipse IDE, it makes use of the Eclipse console to output error messages for the Java code.

ANTLR4

ANTLR [71] is an abbreviation of *Another Tool for Language Recognition* and is, as the name suggests, a scanner and parser-generator. ANTLR generates code for recognising languages, generating parsing trees, and traversing them by using the visitor or listener pattern [72].
The version used is ANTLR4 and it accepts grammars in the form of a .g4 file containing a EBNF-like grammar.
One can run the ANTLR tool from the command line and specify target language and grammar file and ANTLR will generate code in the target-language, which can then be imported into a project and used.

We succeeded in making it generate a C# scanner and parser, but without proper documentation it is almost impossible to figure out how to use it for C#, and a lot of manual work is needed if the grammar changes, since we need to manually call the tool from the command line and import the generated code into the project by hand.

LLLPG

LLLPG [73] is an LL(k) parser (and scanner) generator, which is created as a C# directed alternative to ANTLR.
Since ANTLR is not directly written for C# and has little documentation on using it with C#.
LLLPG is part of an extension to C# called *LeMP* or *Enhanced C#* [74].
LLLPG has an easy to follow tutorial both for using it inside Visual Studio, but also with the command line tool. LLLPG also has documentation besides the extensive tutorials. To use the command line tool, you create a C# file and before compiling, you run it through the tool by specifying the file in the first argument. If you run the tool without arguments it presents a way of showing how to use the tool, which makes it unnecessary to look up documentation online.

The output from LLLPG is readable and easy to modify, but only contains code to recognise languages (e.g. to check for LL(1)), not parse them. However, this is doable with additional settings or by hand.
LLLPG suffers the same problem as ANTLR. If we change the grammar we need to manually copy the new code into the project and almost rewrite it from the beginning.

8.1.2 Writing a Parser by Hand

A parser can also be written by hand instead of using a tool to automate this process. When writing a parser by hand, most often a top-down recursive descent approach is used, as bottom-up parsers are table-based, which are cumbersome to fill out by hand [1, p. 143] [1, p. 179].
According to [1, p. 143] *"Manual construction of such parsers (recursive-descent parser) is both time consuming and error prone, especially when applied at the scale of a real programming language."*.
Minor changes to the grammar may result in large changes to the code of the parser which means that writing a parser by hand can be an even more time consuming task if the grammar is changed often. Having to rewrite the code often also makes it more prone to error.
However, writing the compiler by hand also has its advantages. It gives the programmer a clearer insight into the code of the parser. It is a more educative experience for the programmers, especially if they have little to no previous experience of compiler programming.
Writing the parser by hand also saves time regarding understanding, installing, and using a parser generating tool. In addition to this, the grammar does not have to be rewritten to fit the grammar of the specific tool.
The implementation of the parser by hand can be made easier by making the grammar LL(1). This is because we are able to parse a string using only a look-ahead of one token.

8.1.3 Writing a Tool for Parser Generation

The third and final approach examines the approach of writing a tool ourselves. Initially, this will be a more time consuming task to implement compared to the two other approaches.
However, writing a tool to generate the parser has certain advantages, such as small changes to the grammar will not require the programmer to make major changes of the parser generator code, thereby shortening the iterative stages of developing a compiler. Writing a parser generator tool also gives us insight in how the parser works and can therefore be considered an educative experience.
Having the parser be generated also makes it less prone to errors once the code for the tool is stable. The disadvantages of writing a parser tool, is that the time it takes to implement the tool, can exceeds the time it would take to write the parser ourselves.

8.1.4 Choice of Parser Implementation Approach

In the previous three sections, three approaches to developing a parser have been discussed. The purpose of this section is to discuss these three approaches and decide upon an approach for this project.
The first approach examined the way of using a preexisting tool to generate the parser from the grammar for Tang. Using a tool requires us to understand it first by e.g. reading documentation and rewriting our grammar to fit the syntax of the input grammar for the tool.
It would not give the group a lot of insight into the the specifics of the parser. Because we consider this knowledge to be an important aspect of the learning goals, we have decided not to use a preexisting tool.
The second approach concerns writing the parser by hand. This approach would, as opposed to using a preexisting tool, give an insight into the ways of the parser. But as explained in section

8.1.2, minor changes to the grammar will result in the code having to be changed to fit the new grammar. Based on the initial learning curve associated with developing a compiler, we assume that the iterative part of the process is going to take up the majority of the time spent. This means that writing a tool by hand, although faster in the initial development, can be tedious and time consuming whenever the grammar of the language changes.
The third and last approach is to develop a parser generating tool ourselves. Although this is a time consuming task at first, it will not require as much time to update the code when the grammar changes. Writing the tool ourselves also has the advantage of educating us regarding parser theory. We choose to make a parser generating tool. The implementation of this tool and the generated compiler for Tang will be explained in the following sections of this chapter.

8.2 Lexical Analysis

The lexer for Tang is configured by a set of rules in the `tang.tokens.json` file (see electronic appendix).
Each rule represents a token from the token specification in section 7.1.1. This includes the name of the token and its regular expression, but also two optional attributes describing whether or not to ignore the token in the output and if the token can span multiple lines. All the regular expressions can be seen in table 7.1. An example of a rule can be seen in listing 8.3, where the regular expression for a block comment is shown. Here we see that a block comment can span multiple lines, by setting `SingleLine` to true, the dot operator now also matches newline [75]. Furthermore, we see that the `Ignore` attribute is true, as comments should not be included in the output of the lexer.

```
1    [...]
2    {
3        "Name": "blockComment",
4        "PatternString": "/\\*.*\\*/",
5        "SingleLine": true,
6        "Ignore": true
7    },
8    [...]
```

Listing 8.3: Listing showing an extract of the tang.tokens.json file.

The responsibility of the `Lexer` class is to group the input characters into tokens corresponding to their regular expression. Furthermore, the lexer will keep track of indentation levels, as indentation is a part of the syntax in Tang, as explained in the paragraph 'Separate and Group Statements' in section 6.4. To keep track of the indentation we use a stack storing the nested indentation levels and create indent or dedent tokens when the indentation level changes. This behaviour is difficult to describe using a regular expression and is thus separate from the rest of the token configuration. The concept and idea behind indent and dedent tokens is inspired by the Python language specification in [76].
Listing 8.4 shows the essential parts of the `Lexer` without the analysis of indentation. As seen in line 5 - 36 the `Lexer` works by iterating until it has analysed all characters in the source file. For each iteration it finds the next token by iterating through the rules in the order of the configuration file (`tang.tokens.json`) until it finds a regular expression that matches. If none of the token's regular expressions match, a `LexicalException` is thrown with information about the unexpected characters, line, and column in the source file. Some tokens' regular expression matches lexemes that are a substring of another token's lexemes e.g. the equal operator `==` can be matched as two

assignment operators = and =. To prevent this the rules in `tang.tokens.json` has been ordered to prioritise e.g. == over =.

```
[...]
public IEnumerable<Token> Analyse(string source, string fileName)
{
    [...]
    while (currentIndex < source.Length)
    {
        token = null;
        [...]
        foreach (LexerRule rule in _rules)
        {
            match = rule.Pattern.Match(source, currentIndex);
            if (match.Success && match.Index == currentIndex)
            {
                currentIndex += match.Value.Length;
                token = new Token
                {
                    Name = rule.Name,
                    Value = match.Value,
                    FileName = fileName,
                    Row = row,
                    Column = column
                };
                UpdateCounters(match, ref row, ref column);
                if (!rule.Ignore)
                {
                    yield return token;
                }
                break;
            }
        }

        if (token == null)
        {
            throw new LexicalException(source.Substring(currentIndex,
                ↪ Math.Min(source.Length - currentIndex, 10)).Split('
                ↪ ')[0] + "...", fileName, row, column);
        }
    }
    [...]
    yield return new Token { Name = "eof", Value = "", Row = row,
        ↪ Column = column };
}
[...]
```

Listing 8.4: Listing showing an extract from Lexer.cs which analyses the source file according to the regular expressions for tokens.

8.3 Parser

Before we describe the parser generator, we will describe the parser generated by it.
Based on the lexer described in section 8.2, we get a stream of tokens based on a source program. In this section, we describe how the parser analyses the stream of tokens to validate the source program according to the grammar of Tang shown in section 7.1.2. The output of the parser is a parse tree also known as a concrete syntax tree (CST) representing the syntactical structure of the source program, based on the derivations used to show that the input source program is in the Tang language. The parser is based on the recursive descent parsing strategy (described in 5.6) as the recursive descent approach is more straightforward to implement since the grammar of Tang is LL(1). Another reason is that when implementing a parser generator (see section 8.3.1) it is easier to read and debug the parser if it is written using the recursive descent approach in contrast to the table-driven approach.

```
1  [. . .]
2  public Compiler.Parsing.Data.OrExpression
       ↪ ParseOrExpression(IEnumerator<Compiler.Parsing.Data.Token>
       ↪ tokens)
3  {
4      Compiler.Parsing.Data.OrExpression node = new
           ↪ Compiler.Parsing.Data.OrExpression(){ Name = "OrExpression"
           ↪ };
5      switch(tokens.Current.Name)
6      {
7          case "numeral":
8          case "identifier":
9          case "(":
10         case "!":
11         case "register8":
12         case "register16":
13         case "true":
14         case "false":
15             node.Add(ParseAndExpression(tokens));
16             node.Add(ParseOrExpressionP(tokens));
17             return node;
18         default:
19             throw new UnexpectedTokenException(tokens.Current);
20     }
21 }
22 [. . .]
```

Listing 8.5: Shows the parse method for parsing an OrExpression.

The recursive descent parser works by having a parse method for each non-terminal in the LL(1) grammar for the Tang language shown in section 7.1.2. Because the grammar of Tang is LL(1) we know which derivation to apply based on a look-a-head of one token. This is implemented using a switch statement as shown in code listing 8.5. To determine which switch cases that correspond to each derivation we use the elements of the predict set for the corresponding derivation. Predict sets are described in section 5.3.4. For each derivation under the corresponding switch cases, there

are calls for each symbol in the derivation.

Besides methods for parsing non-terminals, the parser contains a single method for parsing terminal symbols as shown in code listing 8.6.

```
1  [. . .]
2  public Compiler.Parsing.Data.Token
       ↪ ParseTerminal(IEnumerator<Compiler.Parsing.Data.Token> tokens,
       ↪ string expected)
3  {
4      if(expected == "EPSILON")
5      {
6          return new Compiler.Parsing.Data.Token() { Name = "EPSILON" };
7      }
8      Compiler.Parsing.Data.Token token = tokens.Current;
9      if(token.Name == expected)
10     {
11         tokens.MoveNext();
12         return token;
13     }
14     else
15     {
16         throw new UnexpectedTokenException(token);
17     }
18 }
19 [. . .]
```

Listing 8.6: Shows the parse method for terminals.

The `ParseTerminal` method validates the current token with the expected value and moves to the next token in the token stream unless the expected value is EPSILON i.e. the empty string.
To create the parse tree, each parse method returns a node representing the corresponding non-terminal and inserts the nodes returned by sub calls for each symbol in the selected derivation. To represent nodes in the parse tree, there are classes for each non-terminal in the CFG of Tang (see section 7.1.2) an example is the Statement node in code listing 8.7. Terminals are represented using the `Token` class.

```
1
2  [. . .]
3  public class Statement : Compiler.Parsing.Data.Node
4  {
5      [. . .]
6      public override T
           ↪ Accept<T>(Compiler.Parsing.Visitors.ProgramVisitor<T>
           ↪ visitor)
7      {
8          return visitor.Visit(this);
9      }
10     [. . .]
11 }
12 [. . .]
```

Listing 8.7: Shows the class for Statement.

All classes for non-terminals are implemented using a generic visitor pattern as shown in the `Accept` method in lines 6 - 9. For further elaboration on the visitor method see the theory in section 5.7. The non-terminal classes accept a visitor by calling the visitor's `Visit` method for the corresponding non-terminal.
Implementing a recursive descent parser and classes for the Tang grammar, which contains about 70 non-terminals resulting in 70 parse methods with correct implementation of predict sets and calls for symbols in each derivation, is a tedious and mechanical task. Based on the parser implementation approach in section 8.1 we decided to implement a parser generator to generate the parser and classes for the parse tree based on the grammar for Tang. The code listings 8.5, 8.6, and 8.7 are therefre generated code by the `ParserGenerator` class with the grammar for Tang in 7.1.2 as input. The implementation of the `ParserGenerator` is described in section 8.3.1.

8.3.1 Parser Generator

The `ParserGenerator` generates a recursive descent parser for a language based on its LL(1) grammar. In this section we describe the essential parts of the `ParserGenerator` class with code listings from the `GenerateParserClasses` method. The prototype is as follows.

```
1  public ClassType[] GenerateParserClasses(BNF bnf, string parserName,
       ↪ string dataNamespace, string parserNamespace)
```

To generate the parser we first need information about the predict set of the rules in the grammar. To do this we implement a class `BNFAnalyzer` which analyses the first-, follow-, and predict-sets for the grammar based on the definition of first- and follow-set in [77].

```
1  GrammarInfo grammarInfo = _bnfAnalyzer.Analyze(bnf);
```

To represent the parser we create classes `ClassType`, `MethodType` etc. to represent the overall structure of the generated C# code as shown.

```
1 ClassType parserClass = new ClassType(parserNamespace, "public",
    ↪ parserName, null)
2 [...]
3 parserClass.Methods = new List<MethodType>();
```

The `ParseTerminal` method is first added to the parser (see listing 8.6 for the `ParseTerminal` method).

```
1 MethodType parseTerminalMethod = new MethodType("public",
    ↪ \$"{dataNamespace}.Token", "ParseTerminal")
2 [...]
3 parserClass.Methods.Add(parseTerminalMethod);
```

Then, for each non-terminal in the grammar we add a parse method. The body statements of the parse methods is structured as shown in listing 8.5.

```
1 foreach (var production in bnf)
2 {
3     MethodType parseMethod = new MethodType("public",
        ↪ $"{dataNamespace}.{production.Key}",
        ↪ $"Parse{production.Key}")
4     [...]
5     parserClass.Methods.Add(parseMethod);
6 }
```

Finally, we generate the `UnexpectedTokenException` class used in the parse methods and return it together with the parser class.

```
1 classes.Add(parserClass);
2
3 ClassType unexpectedTokenException = new ClassType(parserNamespace,
    ↪ "public", "UnexpectedTokenException", "System.Exception")
4 [...]
5 classes.Add(unexpectedTokenException);
6
7 return classes.ToArray();
```

The `ParserGenerator` class also implements the methods GenerateSyntaxTreeClasses and GenerateVisitorClasses for implementing the data structure and traversal of syntax trees based on a grammar.

8.4 Tree Translation

In this section we describe in general how to translate a syntax tree in one language to another. From the generated parser we are now able to build a concrete syntax tree (CST) based on a list

of input tokens from the Lexer. Because a CST is built based on the LL(1) grammar for Tang in section 7.1.2 it also reflects some of the limitations of using the LL(1) grammar such as large subtrees for simple expressions and inverted associativity. To address these limitations we will translate the CST to an AST following a structure similar to the abstract syntax for Tang in section 7.2. In section 8.4.2 we describe the translation from CST to AST and in section 8.4.4 we describe the translation from AST to C, using these methods.

8.4.1 Translating Syntax Trees in General

Translating a syntax tree in one language (domain tree) to a syntax tree in another language (codomain tree) can be implemented in different ways.
One approach is to implement a translate method inside each class representing a node in the domain tree which then returns the corresponding node in the codomain tree.
Another approach is to use the visitor pattern and implement a visitor that visits and translates each node in the domain tree.
A third approach is to use pattern matching to translate certain patterns in the domain tree to the corresponding structure of the codomain tree.
Choosing between these three approaches revolves around answering the question: *"How should we implement the translation between two languages' syntax trees?"*. We will answer this question later but first we will try to answer the question: *"What defines the translation between two languages' syntax trees?"*. If we can formalise the translation, then we might be able to generate code for the translation between two languages like we generated the parser based on a formal syntax definition.

Defining Syntax Tree Translation

Inspired by structural operational semantics we will in this project define a new language for translation of syntax trees between two languages. To formalise this translation we first describe the possible domain trees and codomain trees. The possible domain trees are defined by the language we are translating from (domain language), which for CFGs can be described as the following tuple (for the description of the elements of the 4-tuple in a CFG see section 5.3.2):

$$L_d = (V_d, \Sigma_d, R_d, S_d)$$

The same applies for the language we are translating to (codomain language).

$$L_c = (V_c, \Sigma_c, R_c, S_c)$$

Let T(L) be set of all possible syntax trees for a language L i.e. trees derived using the language's CFG, we can then define the possible domain trees and codomain trees as the following.

$$T_d = T(L_d)$$

$$T_c = T(L_c)$$

Let S(t) be the set of the tree t and all subtrees of t. Now we define the domain $D(L)$ for a language L as follows

$$D(L) = \bigcup_{t \in T(L)} S(t)$$

The translation can now be defined as a binary relation `->`

$$\texttt{->} \subseteq D(L_d) \times D(L_c)$$

Because some translations depend on more information than a single domain tree and a single codomain tree, we can extend the translation to be a binary relation between a set of n number of domain trees and a set of m number of codomain trees i.e.

$$\texttt{->} \subseteq (D(L_{d1}), D(L_{d2}), \ldots, D(L_{dn})) \times (D(L_{c1}), D(L_{c2}), \ldots, D(L_{cm}))$$

To define this relation, we use inference rules similar to the rules used in structural operational semantics, which consists of a conclusion based on zero or more premises. To denote the rules for the translation we modify the notation used in structural operational semantics to be able to create a string based representation that we can parse and generate code from. This means that we remove the fraction notation and instead write the conclusion of the rule in the top followed by the premises with one level of indentation as shown for the translation rule in listing 8.10.

To understand the syntax of the translation rules we will present a small example with the following domain language.

```
1 Program -> Add
2 Add -> numeral AddP
3 AddP -> + numeral
```

Listing 8.8: Example of a domain language named Program.

And the codomain language for this example is as follows.

```
1 AST -> Add
2 Add -> numeral + numeral
```

Listing 8.9: Example of a codomain language named Program.

Based on these two languages we can now use the translation rules' syntax (see appendix H for the CFG) to define rules for the translation.

```
1 ->:a := Program -> AST
2
3 Program[Add:e1] ->:a AST[e2]
4     e1 ->:a Add:e2
5
6 Add[numeral:e1 AddP[+ numeral:e2]] ->:a Add[e3 + e4]
7     e1 ->:a numeral:e3
8     e2 ->:a numeral:e4
```

Listing 8.10: Example of a translation rules from Program to AST.

In listing 8.10 line 1 we first define the domain and codomain languages for the translation relation `->:a`. In line 3 we define the conclusion for the first rule. This states that a Program node with an Add node as child can be translated to an AST node if the premise in line 4 holds.
The premise states that the Add node of the Program can be translated to an Add node in the codomain language. To make this possible we define a second rule in line 6 - 8 describing the translation of an Add node in the same manner as the first rule. Notice the use of brackets to denote children of a node as well as colon followed by a name to distinguish nodes and relations.

Generating a Syntax Tree Translator

Based on the notation in listing 8.10, we will now implement a `TranslatorGenerator` class that can generate a translator based on the translation rules.
We are now trying to answer the question: *"How should we implement the translation between two languages' syntax trees?"*. The idea behind the implementation is that we parse the rules and based on these rules generate a translator class with methods for each translation.
To parse the rules, we create a LL(1) grammar for the translation language as shown in appendix H and use the `ParserGenerator` class to generate a parser for the rules. From this, we can get a tree structure of the rules that we can generate code based on. For instance if we have the translation relation `->:a` from listing 8.10, we generate a method called `Translatea`. To prevent a translation with many rules to become a large method, we use static overloading to group rules into methods with same top level node types.
The second rule in listing 8.10 defines a translation between two trees, as defined by the arrow `->:a`, and will fall into a method with the prototype `Lc.node Translatea(Ld.Add add)` where `Lc` is the namespace of the classes for the codomain tree and `Ld` is the namespace of the classes for the domain trees. These classes can be generated using the `ParserGenerator` class' `GenerateSyntaxTreeClasses` method based on grammars for the domain and codomain languages in BNF.
Inside each translate method we generate code for all the translation rules with the same top level node type. The code for each rule consists of three parts.
The first part checks that the input domain trees follow the pattern in the rule. For the Add rule in listing 8.10 we need to check that the add parameter contains a numeral as first child and an `AddP` as second child with a `+` as first child and a numeral as second child.
The second part is to check that all the premises hold. We do this by calling the translation method for the premise translation with the left hand side of the premise and then check if the result matches the pattern of the right hand side of the premise.
The third and final part of the code for each rule is to return the tree at the right hand side of the conclusion if all premises hold. The code structure to be generated for the rule in listing 8.10 can be seen in listing 8.11.

```
1  public Lc.Node Translatep(Ld.Add add)
2  {
3      if(add != null && add.Name == "Add" && (add.Count == 2 && add[0]
           ↪ != null && add[0].Name == "numeral" && add[1] != null &&
           ↪ add[1].Name == "AddP" && (add[1].Count == 2 && add[1][0] !=
           ↪ null && add[1][0].name == "+" && add[1][1] != null &&
           ↪ add[1][1].name == "numeral")))
4      {
5          Lc.Node e3 = Translatea(add[0] as Ld.Token);
6          if(e3 != null && e3.Name == "numeral")
7          {
8              Lc.Node e4 = Translatea(add[1][1] as Ld.Token);
9              if(e4 != null && e4.Name == "numeral")
10             {
11                 return new Lc.Add(false) { e3 as Lc.Token, new
                       ↪ Lc.Token() { Name = "+", Value = "+" }, e4 as
                       ↪ Lc.Token };
12             }
13         }
14     }
15     return null;
16 }
```

Listing 8.11: Shows code to be generated for the add node rule in listing 8.10.

As shown in listing 8.11 the rule's use of aliases to refer nodes in the tree is translated to a path of child selectors like the numeral `e2` which is referenced in the generated code as `add[1][1]` i.e. the second child of the second child in the `Add` domain tree. In the following sections we will show how we can use translation rules to implement the parse tree to AST translation and AST to C translation.

8.4.2 Parse Tree to AST Translation

In order to show the translation from a parse tree to an AST we will, as an example, show how the parse tree for the integer expression: `1 + 2 + 3`, is translated into its corresponding AST by using the syntax described in section 8.4.1.

Figure 8.2 shows the AST for the integer expression: `1 + 2 + 3` and figure 8.1 shows its corresponding parse tree. As it can be seen on these two figures the AST is a restructered version of the parse tree where the unnecessary nodes are removed. Furthermore, the AST in figure 8.2 has the right associativity where the numeral nodes for lexemes 1 and 2 are further down the parse tree than the numeral node for lexeme 3 which is not the case for the parse tree in figure 8.1. Note that a dashed line between two nodes indicates that for some ancestor node it has somewhere in its sub-tree a particular descendant node.

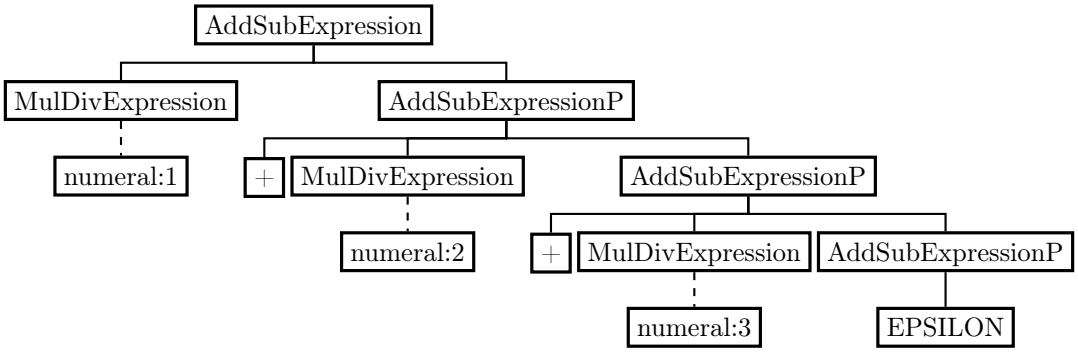


Figure 8.1: Extract of the parse tree for `1 + 2 + 3`.

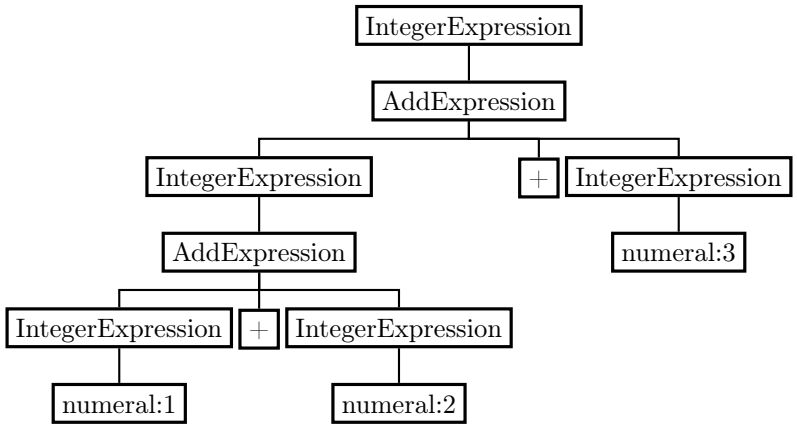


Figure 8.2: AST for: 1 + 2 + 3

In order to translate the parse tree in figure 8.1 to the AST in figure 8.2 a set of translation rules are defined using the syntax described in section 8.4.
The first rule applied to the tree in figure 8.1 is the rule shown below in listing 8.12, which states that an AddSubExpression can be converted into an AST `IntegerExpression` if and only if the premises of the rules hold, which states that both `MulDivExpression`s can be evaluated as `IntegerExpression` ↪ s. As we can see the `MulDivExpression` is first evaluated and then inserted into the tree obtained by evaluating the `AddSubExpressionP` subtree.

```
1 [. . . ]
2 [AddSubExpression[MulDivExpression:expr1 AddSubExpressionP:expr2]
    ↪ SymbolTable:s] -> [intExpr2 <- intExpr1 s]
3     [expr1 s] -> [IntegerExpression:intExpr1 SymbolTable]
4     [expr2 s] -> [IntegerExpression:intExpr2 SymbolTable]
5 [. . . ]
```

Listing 8.12: AddSubExpression translation rule

To evaluate `MulDivExpression` we search for a `MulDivExpression` rule that matches our tree-pattern., which is the rule shown below in listing 8.13. For simplicity figure 8.1 only shows a part of the actual parse tree for the expression `1 + 2 + 3`. The rule states that we obtain the tree

by evaluating the expression, `expr`, which in this case is a single numeral node containing lexeme 1.

```
1 [. . . ]
2 [MulDivExpression[PowExpression:expr MulDivExpressionP[EPSILON]]
    ↪ SymbolTable:s] -> [expr1 s]
3     [expr s] -> [*:expr1 SymbolTable]
4 [. . . ]
```

Listing 8.13: MulDivExpression translation rule

As the node found by evaluating the `MulDivExpression` sub-tree has to be inserted into the tree we obtain by evaluating `AddSubExpressionP` we now try to find a rule to match the tree denoted by topmost `AddSubExpressionP` in figure 8.1. One of the rules for `AddSubExpressionP` states that if the node `AddSubExpressionP` has the following three children: `+`, `MulDivExpression`', and '`AddSubExpressionP`', and if both the `MulDivExpression` and the `AddSubExpressionP` node can evaluate to `IntegerExpressions`, we can obtain an `IntegerExpreesion` tree as denoted in line 2 of listing 8.14 which corresponds to the AST in figure 8.5.

```
1 [. . . ]
2 [AddSubExpressionP[+ MulDivExpression:expr1 AddSubExpressionP:expr2]
    ↪ SymbolTable:s] -> [intExpr2 <-
    ↪ IntegerExpression[AddExpression[%IntegerExpression + intExpr1]]
    ↪ s]
3     [expr1 s] -> [IntegerExpression:intExpr1 SymbolTable]
4     [expr2 s] -> [IntegerExpression:intExpr2 SymbolTable]
5 [. . . ]
```

Listing 8.14: AddSubExpressionP translation rule

In order to obtain this tree we must first evaluate the `MulDivExpression`, which evaluates to a numeral node in the AST-tree more specifically the `numeral:2` node. As denoted by the syntax, `IntegerExpression[AddExpression[%IntegerExpression + intExpr1]] s]`, in listing 8.14 we can now make a `IntegerExpression` tree corresponding to the tree shown below in figure 8.3.

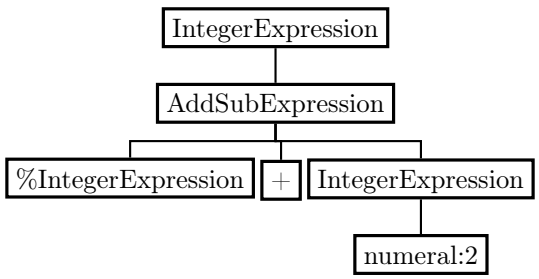


Figure 8.3

However, as denoted by the '<-' arrow in listing 8.14 the tree in figure 8.3 has to be inserted into the tree obtained by evaluating the AddSubExpressionP subtree denoted expr2 in listing 8.14 which is done by pattern matching on the rule in listing 8.15 which gives the tree shown in figure 8.4.

```
1 [. . . ]
2 [AddSubExpressionP[+ MulDivExpression:expr1
    ↪ AddSubExpressionP[EPSILON]] SymbolTable:s] ->
    ↪ [IntegerExpression[AddExpression[%IntegerExpression + intExpr1]]
    ↪ s]
3     [expr1 s] -> [IntegerExpression:intExpr1 SymbolTable]
```

Listing 8.15: AddSubExpression translation rule

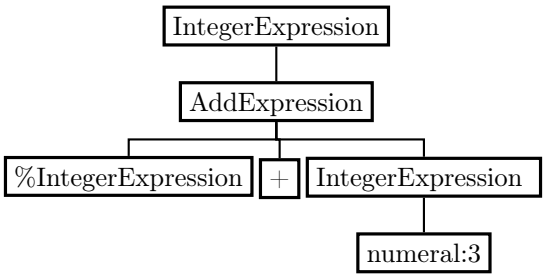


Figure 8.4

After having obtained the tree in figure 8.4, the premises of the rule shown in listing 8.14 are fulfilled and we now insert the tree shown in figure 8.3 into the placeholder denoted by the `%IntegerExpression` node in figure 8.4 which gives the tree in figure 8.5.

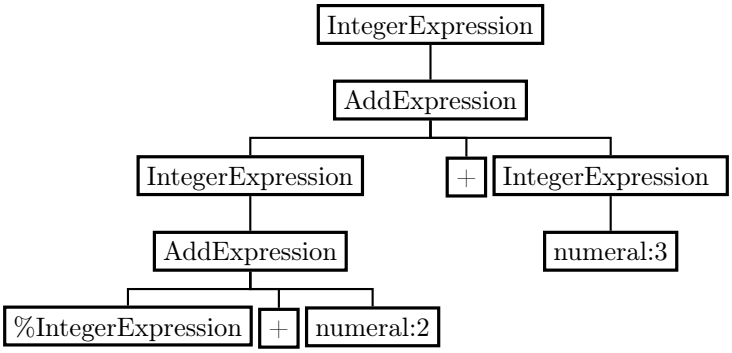


Figure 8.5

The last thing to do in order to obtain the desired AST in figure 8.2, is, in accordance with the rule in listing 8.12, to translate the `numeral` '1' to an IntegerExpression and insert into the placeholder in figure 8.5.
The type and scope checking of Tang is also done in the parse-tree to AST conversion which we describe in section 8.4.3.

8.4.3 Scope and Type Check

In this section we will show how type and scope checks are conducted based on the translation syntax in section 8.4.1.

Figure 5.2 in section 5.5.1 shows how a source program is translated into its corresponding object code. The figure illustrates the parser creating an AST, which is type checked during the construction of a decorated abstract syntax tree.
To do the type checking using the translation syntax, we will for the compiler of Tang do the type and scope checking during the parse-tree to AST conversion which ensures that type and scope errors are found before traversing the AST and generating AVR C code.
During scope and type checking we need a symbol table to store information in. We define a tree-structure for such symbol tables in BNF (see symboltable.bnf in electronic appendix).

Scope check

In this section we will provide examples showing how scope checking is handled in the parse-tree to AST translation.
The rule in listing 8.16 will translate a parse tree into an AST. To do so, we must first scan the input program to find all the functions that have been declared in Tang, as we did in the type system in section 5.4.2.
Line 2 in listing 8.16 shows how this is done by translating a parse-tree and an empty symbol table into a symbol table that contains all function declarations found in the parse-tree. The translation-rule works like the auxiliary scan functions in the type system in table 7.25 and goes through every node in the parse tree to find a function declaration and inserts it into the symbol table.
When all function declarations have been found and no error has occurred, a symbol table denoted `s` is returned which we, together with the parse-tree in listing 8.16, will translate to an AST as explained in section 8.4.2. The reason for inserting the return type, identifier, and parameters of each function into a symbol table is because of the scope-rules of Tang (see section 6.5), which specifies that a function can be called before it is used and we must therefore be able to check if there exists a function in the program that has the same return type, identifier, and parameters for any function call in a Tang program.

```
1 Program:p ->:toAST ast
2     [p SymbolTable[%Declarations[EPSILON]]] ->:scan SymbolTable:s
3     [p s] -> [AST:ast SymbolTable]
```

Listing 8.16:

The type system in section 5.4.2 implies that an identifier cannot be declared in the same scope twice. The translation rule in listing 8.17 shows how to make this scope check by showing an example for boolean declarations.
When a boolean declaration is translated into an AST-tree we check that the identifier has not already been declared by making a look-up in the symbol-table of the current scope. The check can be seen on line 6 in listing 8.17, where we try to evaluate an identifier and a symbol table to a syntax tree, that consists of a declaration node with an `EPSILON` node. This is a look-up in the symbol table where we ask if `s` is undefined. If it is, we know that `s` is not defined in this, or any enclosing scopes and the boolean declaration is therefore type correct and we can update the symbol table where we insert the type `t2` and the name `id2` as well as a placeholder tail declaration, so that later declarations can be added onto the end later.

```
1 [IdentifierDeclaration[BooleanType:t identifier:id
    ↪ Definition[newline]] SymbolTable:s] -> [BooleanDeclaration[t1
    ↪ id1] s <- Declarations[Declaration[Variable[Type[t2] id2]]
    ↪ %Declarations[EPSILON]]]
2     t ->:toAST BooleanType:t1
3     t ->:toSym BooleanType:t2
4     id ->:toAST identifier:id1
5     id ->:toSym identifier:id2
6     [id s] ->:lookup Declaration[EPSILON]
```

Listing 8.17:

Another example of scope rules in Tang is a `while` statement in the sense that any variables declared inside the statement are not accessible outside of it. Listing 8.18 shows the translation rule for while statements. Notice how the symbol table `s` passed to the rule, is the same symbol table as is returned on the right-hand-side on line 1. `s` is passed to the rules that translate `expr` and `stms`, but the resulting symbol tables from these rules are not captured and thus has no effect on the resulting symbol table.

```
1 [WhileStatement[while ( Expression:expr ) indent Statements:stms
    ↪ dedent] SymbolTable:s] -> [WhileStatement[while ( boolExpr )
    ↪ indent stm dedent] s]
2     [expr s] -> [BooleanExpression:boolExpr SymbolTable]
3     [stms s] -> [Statement:stm SymbolTable]
```

Listing 8.18:

Type Check

In this section we will show how type checks are conducted in the translation from a parse-tree to a AST.
Listing 8.19 shows the type check for an integer expression which states that if the two integers denoted `e1` and `e2` can be converted to integer types then the type of the entire expression is the largest of the two types, which corresponds to the [*AddOperation*] type rule we defined for the type system in table 7.22.

```
1 [IntegerExpression[AddExpression[IntegerExpression:e1 +
    ↪ IntegerExpression:e2]] SymbolTable:s] ->:type t3
2     [e1 s] ->:type IntType:t1
3     [e2 s] ->:type IntType:t2
4     [t1 t2] ->:largest IntType:t3
```

Listing 8.19:

Another example of type checking is checking if the actual parameters of a function correspond to the formal parameters. Listing 8.20 shows how this is checked by making a look-up in the symbol table on line 2 that tries to find a function that matches the prototype of the called function. If this premise hold then the formal parameters denoted `p1` and the actual parameters are evaluated with

the transition methods for `params` to see if the actual parameters match the formal parameters. If it does, an expression list is returned and the parse-tree for the `IdentifierStatement` is evaluated to an abstract syntax tree for a function call as can be seen on on line 1 in listing 8.20.

```
1 [IdentifierStatement[identifier:id IdentifierStatementP[(
    ↪ ExpressionList:p )]] SymbolTable:s] -> [Call[id1 ( p2 )] s]
2     id ->:toAST identifier:id1
3     [id s] ->:lookup Declaration[Function[ReturnType identifier
        ↪ Parameters:p1]]
4     [p p1 s] ->:params ExpressionList:p2
```

Listing 8.20:

8.4.4 AST to C Syntax Tree Translation

Now that we know how a parse tree is translated into an AST, we will briefly describe how an AST is translated into a syntax tree for AVR C code, which we can traverse using a `TextPrintVisitor` in order to obtain the target AVR C program. This translation corresponds to the code generation phase of the compiler. We will expand the example in section 8.4.2 to be a fully valid Tang program where the integer expression is assigned to a int8 variable as shown below in listing 8.21.

```
1 int8 a = 1 + 2 + 3
```

Listing 8.21: AddExpression sample

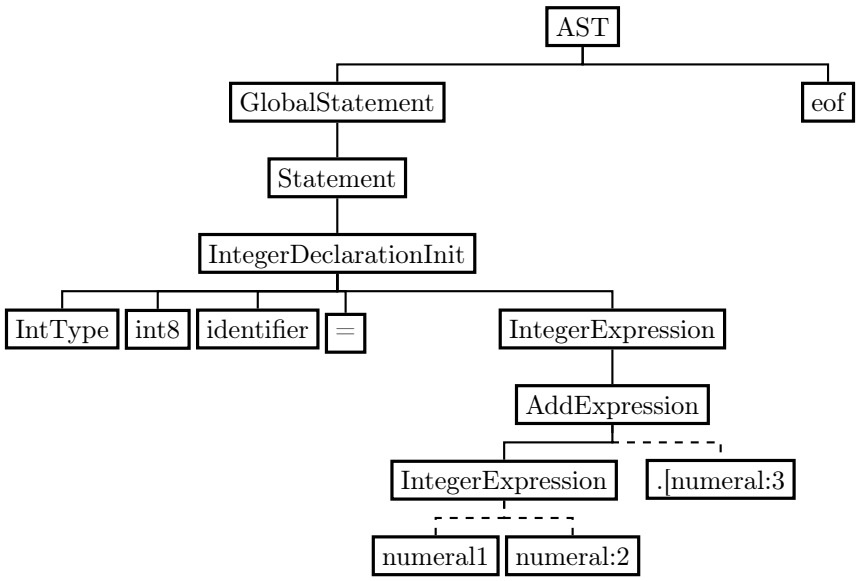


Figure 8.6: AST for listing 8.21

Figure 8.6 shows the parse tree for the statement in listing 8.21 whereas figure 8.7 shows the essential parts of the corresponding syntax tree for AVR C code.

The first thing to note is that an AST is translated into a syntax tree that contains function sub-tree with the main function. This is illustrated by the rule in listing 8.22 for the AST of an empty program which only contains an 'end of file' (eof) node:

```
1 AST[eof] -> C[Declaration[EPSILON] Function[EPSILON]]
```

Listing 8.22: AST to C rule for an AST tree contain the eof node

In comparison to Tang then AVR C does not allow statements in the global scope. This challenge is solved in the code generation phase by inserting translated GlobalStatements into the main function in the target program. To still enable global variables then global declarations is translated into the global scope of the target program. Another difference is that functions must be declared before used in C whereas in Tang functions are allowed to be called before declaration. To implement a translator to do this we define the translation relation to translate a AST to three C syntax trees containing the declarations, functions, and statements. The translation rules can be seen in the file 'ast-c.translator' in electronic appendix.

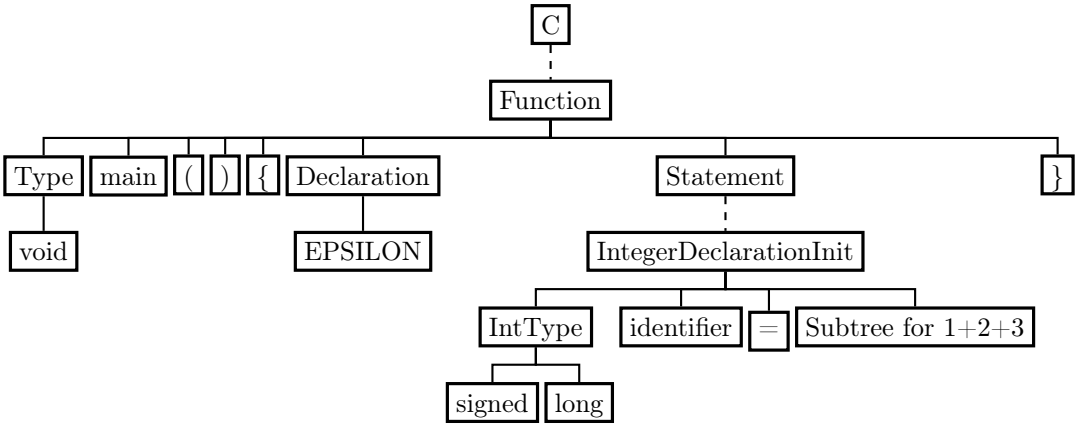


Figure 8.7: Syntax sub tree for the generated C code for the code in listing 8.21

The figure above illustrates some of the key differences between the AST and the syntax tree for the corresponding AVR C code. For instance we see that the AST in figure 8.6 has all its Tang code in a sub-tree of a `GlobalStatement` whereas the syntax tree in figure 8.7 has a subtree for declarations and functions while the functions subtree will include subtrees for each function in the program including the `main` function. When we translate an AST to a syntax tree containing AVR C code we must ensure that constructs like interrupts are declared in the global scope, as AVR C requires, which we do by evaluating the interrupts as global statements and then inserting them as a part of the declaration subtree.

8.5 Not Implemented Features

In the implementation of Tang, we have implemented a subset of the features, described in the language specification (chapter 7). Two of the features that have not been included are signed integer overflow and division by 0. These were not included since we, in the design of Tang, have not specified any mechanism such as exception handling that allows us to deal with run-time errors, as this was thought of too late in the project. What currently happens with integer overflow and

division by 0 is therefore determined by the compiler for AVR C. As division by zero is also undefined in C, any calculation that uses a division by zero results in undefined behaviour [78].

Chapter 9

Test

The purpose of this section is to test Tang in accordance with the test plan defined in section 2.6. This means that we will be conducting four types of tests: unit tests, integration tests, verification tests, and language evaluation tests which are described in section 9.1, 9.2, 9.3, and 9.4, respectively. The unit and integration tests will be conducted using the testing framework xUnit, since xUnit supports data driven testing and supports Windows, Linux, and Mac and thus will run on all group-members' computers without issues [79].
In order to document the tests conducted in this project we will include code snippets that show excerpts of different tests and how we use them to examine if the compiler works as we expect.

9.1 Unit Test

In this section we will describe how we use unit tests to test the lexer, parser, AST, and code generator phases of the compiler. The unit tests are used to identify bugs and prevent unnoticed bugs to occur due to code changes.
To illustrate how each of the phases of the compiler works, we will, besides conducting different tests, walk through the program written in listing 9.1 and show snippets of the output of each of the four phases of the compiler.
We will not do extensive testing for the parse tree, AST, and code generation, since testing these are tedious and prone to errors, as the trees which we we compare them to have to be written manually. Instead, we will focus on conducting integration tests for the program as a whole. We do this in section 9.2 where we test if the result of the compilation is as expected.

Lexer

To test the tokens generated by the lexer (described in section 5.5.1), we first need a test file. The test file can be seen in listing 9.1.

```
1 int8 a = 1 + 2 + 3
```

Listing 9.1: The AddExpression sample from section 8.4.

Here we can see that int8 is an int8 type according to the regular expressions in tang.tokens.json. An extract from tang.tokens.json showing the regular expression for this integer type is shown in listing 9.2. The reason for the last part of the pattern string: $(?![a-zA-Z\_0-9])$ is to help the

lexer know that a statement such as *int8varName* is a variable name and not a int8 type.

```
    [...]
    "Name": "int8",
    "PatternString": "int8(?![a-zA-Z_0-9])"
    [...]
```

Listing 9.2: Extract from tang.tokens.json showing the regular expression for int8.

Knowing in advance which tokens exist in our test file allows us to assert that they are correct. This is shown in listing 9.3.

```
[...]
// Initialise Lexer
Lexer l = new Lexer(AppContext.BaseDirectory +
    ↪ "\\TestFiles\\Tokens.cfg.json");

// Read from test file
IEnumerable<Token> tokens =
    ↪ l.Analyse(File.ReadAllText(AppContext.BaseDirectory +
    ↪ "\\TestFiles\\AddExpression.tang"));
[...]
Assert.AreEqual(tokens.Count(), 10);
[...]
Assert.AreEqual(tokens.ElementAt(2).Name, "=");
Assert.AreEqual(tokens.ElementAt(8).Name, "newline");
[...]
```

Listing 9.3: Extract from LexerTests.cs showing how we check if our results are correct.

In some cases the lexer might see input such as `===`. In a case like this, the expected behaviour is to first match an equality token (`==`) followed by an assignment token (`=`). The same applies for e.g. the string (`2test`), where we first want to match a numeral followed by an identifier. Tests covering this are shown in listing 9.4.

```
[...]
IEnumerable<Token> tokens = l.Analyse("=== 2test");

Assert.AreEqual(tokens.ElementAt(0).Name, "==");
Assert.AreEqual(tokens.ElementAt(1).Name, "=");
Assert.AreEqual(tokens.ElementAt(2).Name, "numeral");
Assert.AreEqual(tokens.ElementAt(3).Name, "identifier");
[...]
```

Listing 9.4: Test method to test that tokens are split correctly by the lexer.

For error messages we want to specify exactly where the error occurs. Therefore, we choose to add

row and column numbers as attributes to tokens. This needs to be tested as well. This is done by asserting that the row and column attributes are what we know them to be. An extract from a test for tokens attributes is seen in 9.5.

```
1 [. . . ]
2 // '=' should be at row 0 and column 8 since there are 8 symbols until
     ↪ '=' is hit
3 Assert.AreEqual(tokens.ElementAt(2).Row, 0);
4 Assert.AreEqual(tokens.ElementAt(2).Column, 8);
5 [. . . ]
```

Listing 9.5: Test extract showing how we test token attributes.

Parser

The parser takes a stream of tokens as an input, and generates a tree of token objects. To test that the parser has successfully created a valid tree, a simple stream of tokens is created from the *AddExpression* program seen in listing 9.1.

To test the parser, tokens are created by using the Lexer, as seen in listing 9.6. These tokens are manually inspected, and inserted as a string. Hereafter, the `ParseProgram` method from the parser is called with the tokens as input, which produces a parse tree. To ensure this parse tree is generated correctly, a parse tree is also made by hand and the equivalence of these two is asserted by calling a helper method, called `TreeAsserter`, that traverses two trees and checks if they are equal.

```
1  //Tokens for program AssExpression. These can are derived from the
       ↪ unit test for the lexer.
2  string[] testTokens = { "int8", "identifier", "=", "numeral", "+",
       ↪ "numeral", "+", "numeral", "newline", "eof"};
3
4  var tokenEnumerator = testTokens.Select(t => new Parsing.Data.Token()
       ↪ { Name = t }).GetEnumerator();
5
6  tokenEnumerator.MoveNext();
7
8  ProgramParser pp = new ProgramParser();
9
10 var parseTree = pp.ParseProgram(tokenEnumerator);
11
12 // Build parse tree manually for the AddExpression.tang program, kfg
       ↪ edit is used to visualise how it will look
13 var parseTreeManual = new Parsing.Data.Program(true)
14 [...]
15 TreeAsserter(parseTree, parseTreeManual);
```

Listing 9.6: The initialising of the token list.

An extract from the manual creation of the parse tree can be seen in listing 9.7.

```
1  var parseTreeManual = new Parsing.Data.Program(true)
2  {
3      new Parsing.Data.GlobalStatements(true)
4      {
5          [...]
6      },
7      new Parsing.Data.Token(){ Name = "eof" }
8  };
```

Listing 9.7: A part of the reference tree, showing the nesting of the nodes.

In listing 9.8, the `TreeAsserter` helper method can be seen. If any discrepancy between the two trees is found, the parser test will fail. The method compares the name of each token as to test for both errors regarding individual nodes, and the structure overall.

```
1  public void TreeAsserter(Parsing.Data.Node node, Parsing.Data.Node
       ↪ nodeTest)
2          {
3              Assert.AreEqual(node.Name, nodeTest.Name);
4
5              if(node is Parsing.Data.Token){}
6              else
7              {
8                  Parsing.Data.Node[] nodeChildren =
                       ↪ node.Nodes<Parsing.Data.Node>();
9                  Parsing.Data.Node[] nodeChildrenTest =
                       ↪ nodeTest.Nodes<Parsing.Data.Node>();
10                 for (int i = 0;i < nodeChildren.Length; i++)
11                 {
12                     TreeAsserter(nodeChildren[i], nodeChildrenTest[i]);
13                 }
14             }
15         }
```

Listing 9.8: The recursive method used to compare and assert each node in the trees.

AST

As abstract syntax trees grow large even for minor programs, it is difficult to assert equality between a generated AST and an AST written manually. However, two minor unit tests for this part of the compiler have been written. A small program written in Tang is put through the initial phases of the compiler until the conversion to AST and a corresponding AST is created manually. Hereafter, a helping method is called (see listing 9.9), which walks each tree and asserts the equality of each corresponding node. The input program written in Tang can be seen in listing 9.1.

```
1  public void TreeAsserter(AST.Data.Node node, AST.Data.Node nodeTest)
2  {
3      Assert.AreEqual(node.Name, nodeTest.Name);
4
5      if (node is AST.Data.Token) { }
6      else
7      {
8          AST.Data.Node[] nodeChildren = node.Nodes<AST.Data.Node>();
9          AST.Data.Node[] nodeChildrenTest =
               ↪ nodeTest.Nodes<AST.Data.Node>();
10         for (int i = 0; i < nodeChildren.Length; i++)
11         {
12             TreeAsserter(nodeChildren[i], nodeChildrenTest[i]);
13         }
14     }
15 }
```

Listing 9.9: The helper method used to assert tree equality by traversing each tree and comparing nodes.

To write an AST manually, the tokens produced from the Tang program are identified. The BNF of the AST is then used to write the tree. Although this is a time consuming process, it is doable and is important to ensure the correctness of the program to AST translator. The AST equivalent to the Tang program in listing 9.1 can be seen in appendix K.

Code Generator

The code generator runs by calling the Translate method in the `ASTToCTranslator` class. There are several `Translate` methods depending on which type of translation is needed. Which method is run is based on the parameters in the method call.
The code generator takes as input an AST and produces a tree with AVR C code as its leaves.

To test the code generator, a tree is first created by going through the compiler's phases with an input program. This example will, like the other unit tests, use the `AddExpression` program, which can be seen in listing 9.1. A tree is then written by hand by using the logic of the C BNF determined beforehand and the equality of the two trees is determined by using the `TreeAsserter` method which is also described in listing 9.8.
A code extract of how the AVR C tree is generated by hand can be seen in listing 9.10.

```
var cExpected = new C.Data.C(true)
{
    new C.Data.Declaration(true)
    {
        [. . .]
    },
    new C.Data.Function(true)
    {
        [. . .]
    }
};
```

Listing 9.10: Listing showing the root of the AVR C tree along with the first two children.

9.2 Integration Test

In this section we will be conducting the integration test of Tang as described in the test plan in section 2.6. In our integration tests we will use data-driven testing [80] since we want to put some code through the compiler and compare the outputted C code to some previously compiled C code, which has been checked for errors manually. This means that if we have to write a new test for each file, we would have a lot of tests where the only difference is the two files they read.
xUnit enables us to call tests with parameters defined elsewhere by some code, which means that we can make any number of different tests by making one and using the input-files as parameters. The areas of the compiler tested in our integration test includes:

- Types
- Operators
- Declaration and assignment, both in one line and separately
- Control statements, including: for-loops and if-statements
- Direct bit operations
- Throw exception when necessary, e.g. when a reserved keyword is used
- Functions
- Scope
- Type conversion

```
[Theory]
[MemberData("FilesInTang", MemberType = typeof(CompilerTestsData))]
// Method to test that all programs compiles correctly in Tang folder
public void ProgramsCompileCorrectly(string file)
{
    TangCompiler tc = new TangCompiler();
    string expected =
        File.ReadAllText(file + ".c").Replace("\r","");
    string actual =
        tc.Compile(file, "tang.tokens.json").Replace("\r", "");
    Assert.Equal(expected, actual);
}
```

Listing 9.11: Extract from CompilerTests.cs showing how we ensure that the programs we designed to compile correctly, compiles correctly

Listing 9.11 shows the test method where we test all the programs we wrote specifically to pass the compiler in a certain way to test the different parts of the compiler. This method works by defining some `MemberData` to be passed to this method at run-time. This `MemberData` consists of a class that we wrote, which returns all files in "Testfiles/tang". xUnit then calls the method with each file as parameter and the functions check that the files compile correctly. When the programs are compiled, the method checks the output C code against some C-files also located in a folder named tang. These files have been compiled before and inspected by members of the group to ensure that the files are compiled correctly. This means that if there are any differences between the compiled code and the .c file, the test will fail, since the test cannot determine if the change is essential and will change the behaviour of the program.

```
[Theory]
[MemberData("FilesFailingInTang", MemberType =
    ↪ typeof(CompilerTestsData))]
// Method to test that all programs does not compile correctly in Fail
    ↪ folder
public void ProgramsDoesntCompileCorrectly(string file)
{
    TangCompiler tc = new TangCompiler();
    Assert.ThrowsAny<Exception>(() => tc.Compile(file,
        ↪ "tang.tokens.json"));
}
```

Listing 9.12: Extract from CompilerTests.cs testing if the files designed to fail, actually fail.

Listing 9.12 also uses `MemberData` as listing 9.11. This test gets all files in "Testfiles/fail", which are all the files that are designed specifically to fail for different reasons, like using reserved keywords or passing the wrong type of arguments. If a compilation fails, the `TangCompiler` will throw an exception, which the `Assert` catches. If no exception is thrown, the test fails.

```
public static IEnumerable<object[]> FilesInTang
{
    get {
        List<object[]> data = new List<object[]>();
        var files = Directory.GetFiles("Testfiles/tang/");
        foreach(string file in files) {
            if(file.EndsWith(".tang"))
                data.Add(new object[]{file});
        }
        return data;
    }
}
```

Listing 9.13: Extract from MemberData class, generating the parameters for listing 9.11

Listing 9.13 is the method that generates the data for listing 9.11. The method returns a list of all files in the directory "Testfiles/tang" and returns it. A similar method is created for listing 9.12.

9.3 Verification Test

In this section, we will conduct the verification tests of Tang which are explained in section 2.6. In order to do so, we will show code excerpts of how we test some of the functionality of Tang. All the sample files can be found in the electronic appendix in the 'samples' directory.

Register test

To test registers in Tang we write a sample program in Tang.

```
register8(36){5} = true
register8(37){5} = false
int32 c
c = 0
while(true)
    if(c == 300000)
        register8(37){5} = !register8(37){5}
        c = 0
    c = c + 1
```

Listing 9.14: A register test written in Tang.

First two registers are initialised at the addresses 36 and 37. The first register is the Data Direction Register for `PORTB`, the DDRB register, which we know from the datasheet for the ATmega328P (see [20]) and the other register is the `PORTB` register itself.
In the first register we set the fifth bit to true to indicate that we want data to be going OUT on pin `PB5` (Port B 5), which is connected to the digital pin 13 and the onboard LED on the Arduino Nano [20]. We then set the fifth bit in `PORTB` to false to indicate that we want that pin to be low.

We then enter our while-loop which counts to 300.000 and flips the value of the fifth bit in the `PORTB` register. This causes the pin to be connected to 5V and the LED connected to it, to light up and after another 300.000 iterations, turn off again. The reason for the counter counting to 300.000 is because it takes about 0.5 seconds and slows down things enough for us to actually be able to see that the LED is turning on and off.
A set of standard libraries are implemented for Tang. To test the functionality of the standard libraries, a program is written, which utilises it.

```
1  import Nano
2
3  int16 dutyCycle = 40
4  int32 counter = 0
5  while(true)
6      counter = counter + 1
7      dutyCycle = counter%511
8      if(dutyCycle >= 256)
9          dutyCycle = 255 - dutyCycle%255
10     SetFastPWMPin6(dutyCycle)
11     delay(5)
```

Listing 9.15: An example, which uses the standard library Nano Tang.

As seen in listing 9.15, the program imports the file *Nano*, and uses the $setFastPWMPin6$ function, which cycles through the PWM of a pin, alternating between an LED being on and off in this example.

Both programs produced the expected result, where the first made the LED blink, and the second one made it oscillate between on and off [81].

9.4 Language Evaluation Test

In this section, the information gathered in section 3.2, will be evaluated alongside Tang. To properly conduct this evaluation Tang has to be exposed to the same analysis as the previously mentioned languages in section 3.2.
To evaluate the languages we use the test plan in section 2.6.

The Blink Program in Tang

To evaluate the Tang language, the program *blink* will be evaluated, and compared to the same program, written in other languages.

```
register8  ddrb = register8(36)
register8  port = register8(37)
ddrb {5} = true
port {5} = true
int32  counter = 0
while(true)
    if(counter == 100000)
        counter = 0
        port{5} = !port{5}
    else
        counter = counter + 1
```

Listing 9.16: The blink program written in Tang.

```
import Nano
PinModeD(6, OUTPUT)
while(true)
    delay(500)
    DigitalFlip(6)
```

Listing 9.17: The blink program written in Tang, using the standard library written for it

The blink example in listing 9.16 makes an on-board LED blink using only core functionality in Tang. The specifics of how it functions, are described in 9.3, with the register test, which functions in the same manner.
A library for Tang could be included in the blink example, shown in listing 9.17, significantly shortening and simplifying the program. The reason for not doing this is, that it would not make for a fair comparison as not all the other languages that we evaluate have a library that allows the programmer to e.g. set the output of a pin, which would cause the readability and writability of the programs that uses these libraries to be higher than those that do not.

Readability

For the evaluation of the Arduino language we have decided that the Arduino.h file, which can be found in [82] is part of the Arduino language as the user of the Arduino language does not have to include this file themselves. However, it is automatically included during processing as can be seen in listing 3.2. Because of the Arduino.h file, the Arduino language has a lot of meaningful statements, such as the `delay(1000)` statement, seen in listing 3.1, the readability of the Arduino language is relatively high in comparison to languages like the AVR C language that do not have similar constructs as a part of the language in accordance with the theory for readability in section 5.1.1.

However, since we see the Arduino.h file as a part of the language this also adds to the total number of constructs of the language, thereby decreasing the overall simplicity of the language (see section 5.1.1).
We have weighted the readability, added by the expressivity of the constructs, higher than the

readability, decreased by the increased number of constructs which make the Arduino language more readable, than e.g. the AVR C language that does not have these expressive constructs.

As for the AVR assembler the different constructs of the language used in the blink example in listing 3.4 are not meaningful e.g. the constructs used for setting a specific pin as an output pin. Furthermore, the orthogonality of the AVR Assembler language is low because some of the assembly instructions require specific registers given as parameters like the `cbi` (clear bit) constructs used in the blink example for AVR Assembler in listing 3.4.
Based on the theory in section 5.1.1 this makes AVR Assembler less readable than e.g. AVR C that is more expressive and has higher orthogonality. As for the preprocessed blink example for AVR C in listing 3.10 we see that the code required to turn on a LED requires a complex statement that includes the binary *or* and $left$ shifting which in contrast to the blink example in Tang in listing 9.16, only requires the programmer to access a bit in a register and set it to true. Given the target group, Tang is therefore evaluated as being more readable than AVR C.
As for the readability between the Arduino language and Tang, the Arduino language has more meaningful constructs like `Digital.Write` and `Delay` even though Tang has standard libraries in form of Nano.tang, which includes these functions and provides an easy interface to pins, interrupts, PWM, and the ADC, much like the Arduino language, we choose not to count this as part of the language for reasons explained later in this section. Listing I in appendix I shows a blink example written in Java. In terms of readability, Java is, for the blink example, as readable as AVR C as it uses many of the same constructs like left-shifting, the only difference is that since the Java example imports the Haiku library the `DDRB` and `PORB` registers are provided.
The table below summarises the readability of the five languages in relation to the comparison of the readability of the languages conducted above.

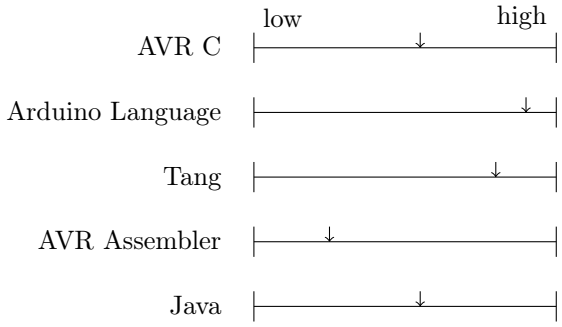


Table 9.1: Readability

Writability

According to the theory in section 5.1.1, the writability of a language depends on the simplicity, expressivity, and orthogonality of a programming language, but also depends on readability. The writability of the Arduino language is in contrast to the blink example written in AVR C in listing 3.10 not that writable as the expressivity of the statements used to change bits for pins, and setting a delay is lower than for Arduino as the AVR C code for e.g. setting a bit is much more complex and requires more language constructs as can be seen by the two blink examples in listing 3.1 and 3.10. In relation to AVR Assembler, we reasoned previously that the language has relatively low orthogonality which decreases the languages writability. As for the expressivity of the language we see in listing 3.4 that the constructs for setting a delay in AVR Assembler is not very expressive which also decreases its writability. If we also look at the readability of AVR Assembler, the readability

was previously determined to be lower than that of AVR C which along with the low expressivity and orthogonality of AVR Assembler makes the language less writable for the blink example than AVR C. As for Tang the blink example in listing 9.16 shows that for the blink example the different statements for setting delay and e.g. clearing a bit is more expressive than the similar statements for AVR C. However, since these same statements are more readable and expressive in the Arduino language than in Tang, the Arduino language is for the blink example more writable than Tang.

The Java example in listing I is not as expressive as Tang, but is still relatively expressive due to the use of compound assignments. However, due to the similarities with AVR C, for the blink example, Java is as writable as AVR C.

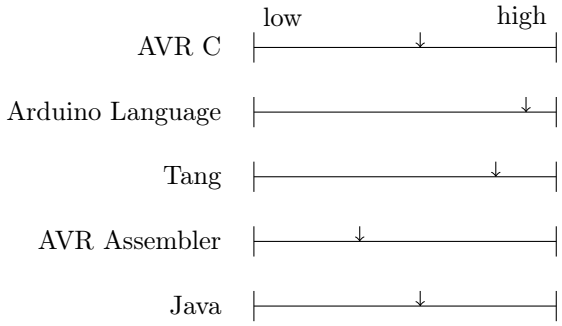


Table 9.2: Writability

Reliability

A program is reliable if it performs according to its specifications [41, p. 37]. In order to evaluate different languages, we have to look at the language specifications for each of the five languages.
In the blink example for AVR C, we conduct signed integer addition which according to [78] can lead to signed integer overflow. The C standards specifies signed integer overflow as having undefined behaviour which makes the program less reliable than programs that have defined behaviour for overflow like Java [83] [41, p. 37]. This is also the case with the Arduino language and Tang, which also decreases their reliability [41, p. 37]. AVR C uses some type-checking at compile-time [84], but not at run-time which increases its reliability over AVR Assembler [26] which has no type-checking [41, p. 37]. This means that AVR Assembler has lower reliability than AVR C, because in AVR Assembler it is the programmer's responsibility that he performs the correct operations on the correct types[41, p. 37]. It can be argued that this would make AVR Assembler have higher reliability, but since we see reliability as the compiler making sure that the programmer is not misunderstood, we came to the conclusion that AVR Assembler has lower reliability[41, p. 37].
AVR Assembler also has low readability and writability, which also makes it has lower reliability[41, p. 37]. This puts AVR Assembler below AVR C.
Arduino is based on C++, where classes and exception-handling are possible, which increases readability and thus reliability [41, p. 37]. C++ is an expansion of C and thus includes pointers which create aliasing and decreases reliability [41, p. 37]. Arduino abstracts over pins and other hardware through standard libraries, making it more readable and writable and thus reliability, and therefore puts it above AVR C [41, p. 37].
Tang is based on AVR C, but does not include C-pointers directly. Instead it includes the `register` type, which also works at specific memory-addresses like C's pointers, but have more limitations and cannot be cast to other types and thus only decreases reliability a bit [41, p. 37]. Tang in turn performs more type-checking and reduces the number of constructs and different types, which

makes for a more reliable language than Arduino [41, p. 37].
Java, as the only language, performs automatic type-checking at run-time [85], which increases the reliability [41, p. 37], but Java also includes many constructs and is pure object oriented and have lower readability as concluded, which decreases its readability and thus reliability [41, p. 37]. Java also includes exception-handling, which also increases its reliability [41, p. 37]. Furthermore Java does not include pointers which also increases it reliability [41, p.37]. All things considered, Java is more reliable than Arduino language and Tang, according to this test.

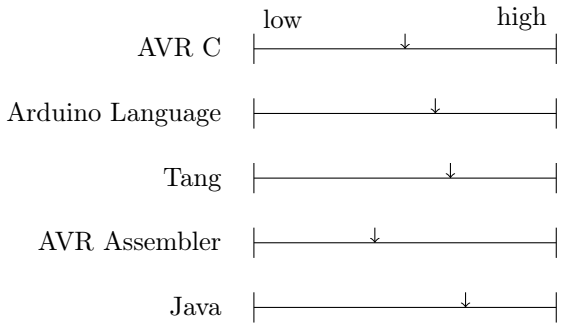


Table 9.3: Reliability

Cost

Cost of a programming language is concerned with simplicity, orthogonality, writability, performance of compiler, language implementation system, reliability, and maintainability 5.1.1. To test the performance of the compiler we will check the size of the compiled program and the compilation time of each of the tested languages. To test the execution speed of the different languages, each blink example is compiled and executed on the Arduino, and then the speed with which the led blinks is used to discuss the execution speed of each program. All programs counts with the same value, as to accurately compare the execution speed. As for the maintainability of the programs written in the different languages we say that the maintainability of the different languages in relation to the blink example is equal to their readability as these two are closely related (see section 5.1.1) and since comparing other things like features, paradigms etc. would not make for a fair comparison as not all these features are comparable and/or relevant for embedded programming. Simplicity and orthogonality has to do with cost of training programmers to use a language 5.1.1, but since these criteria has to do with readability and writability they will not be further elaborated on in this section.
Table 9.4 shows the size of the object code generated by the compiler in each language. The AVR Assembler produces the object code that takes up the least amount of memory, using only 164 bytes, whereas the higher level languages like Arduino and Java uses 942 bytes and 5324 bytes respectively. The reason that the size of the object code for the blink example in Arduino language is larger than that for AVR C is that Arduino includes Arduino.h, which adds to the total program size (see table 3.2). Tang takes up more memory than AVR C but less than the Arduino language due to the fact that Tang does not include a lot of constructs like the Arduino language, but does not optimise the code, so functions like pow is included in Tang whether used or not. Java takes up more than 5000 bytes because it includes a JVM [86] and Java bytecode.

Language	Size of object code
AVR Assembler	164 bytes
Java	5324 bytes
AVR C	282 bytes
Tang	758 bytes
Arduino Language	942 bytes

Table 9.4: Size of compiled object code for the blink examples (see electronic appendix for hex files).

To further measure the cost of compilation, listing 9.5 shows the compilation-time required to compile the blink program in each of the below languages. The compilation time for compiling the AVR C and the Arduino language is similar, whereas the compilation time of the AVR Assembler language is slightly faster. This result is as expected since the AVR Assembler instructions has a direct mapping to machine code instructions [26]. Tang has the second highest compilation time of 1.262 seconds which is due to the compilation time of Tang not being highly prioritised during development. The compilation time of Java is approximately 3.9 seconds due to the fact that Java is a large and complicated language with many constructs in comparison to the four other languages.

Language	Compilation-time
AVR Assembler	0.022 seconds
Java	3.894 seconds
AVR C	0.036 seconds
Tang	1.262 seconds
Arduino Language	0.35 seconds

Table 9.5: Time used for compilation (not upload) for the blink examples (See electronic appendix for code).

Table 9.6 shows the interval it takes for a LED to blink in the compared languages. We use this to measure the execution speed in relation to cost of executing the object code generated by each compiler. AVR Assembler is by far the most cost-efficient of the languages in terms of execution time which is expected as the AVR Assembler instructions have a one-to-one correspondence with the generated machine-code instructions, and the number 3.2 million comes from the AVR Assembler, where it takes approx. 1 second to count to 3.2 million. In Tang the interval of each blink is approx. 12 seconds which is slower than AVR Assembler, Arduino language and AVR C, but faster than Java where the interval is over four minutes, partly due to the JVM needing to compile every statement before running it. The reason for the difference of the execution speed between AVR C, Arduino language and Tang is that the Tang code is not exposed to any optimisation by the Tang compiler and contains more checks to ensure reliability and is not called with the optimise flag by the gcc compiler as this will sometimes produce unexpected results where the originally intent of a program has been modified by the optimiser, which can be almost impossible to debug if you are not aware. Note that this test is just a bench mark for measuring execution speed of for-loops and only tests a small subset of the different languages.

Language	Blink interval in seconds
AVR Assembler	1.00 seconds
Java	4 minutes 51.12 seconds
AVR C	3.64 seconds
Tang	12.4 seconds
Arduino Language	7.12 seconds

Table 9.6: Run-time for counting to 3.2 million 1 time in each language. A video can be found in [87].

As for the cost for the implementation time of the different languages these are similar for most of them, as all the languages works on the family of AVR microcontrollers.
The table 9.6 gives an overview of the estimated cost of using each language in relation to the theory on cost of using a language in section 5.1.1 and is based on simplicity, writability, compilation time, and execution time. It is important to note, that a low cost is considered advantageous for a language, in contrast to the other criteria, where it is the high score.

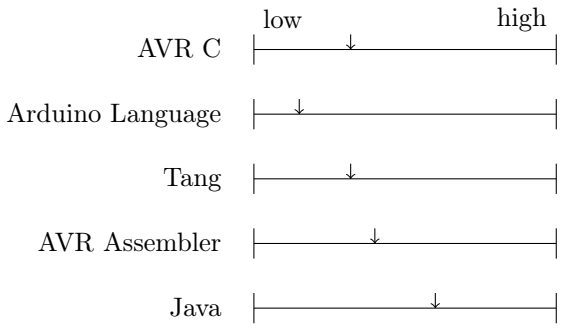


Table 9.7: Cost

User Test

In order to get insight into Tang from another perspective than the members of the group, we asked two students from a neighbouring computer science group to solve the tasks in appendix J. The test method is based on the test plan in section 2.6.
As appendix J and J.1 shows, one of these programmers corresponds to the intended target group of Tang whereas the other programmer has some experience in embedded programming. Appendix J shows the tasks that the two programmers were asked to solve. In the first task the programmers were asked to read a Tang program and answer two questions related it. The first programmer answered both of these programs wrong as he understood the for-loop: “for int i from 1 to 6” as iterating through the values (1,2,3,4,5) not including the numeral 6 (see listing J.1.1 in appendix J).
The other programmer correctly solved this assignment and thus also the for-loop (see listing J.1.2 in appendix J.1. The programmer seen in J.1.2 was the test person with embedded programming experience, while J.1.1 did not have any experience.
Even though these test do not provide absolute conclusive results, it seems like the for-loop has a somewhat ambiguous meaning. In relation to assignment 2 both programmers were able to understand the essence of working with registers and able to set and clear bits for the necessary hardware pins. However, the first programmer incorrectly sat the pin-mode of the LED pin to ’INPUT’

(listing J.1.1) . And as no follow-up questions were asked it is not clear whether or not this was intentionally or unintentionally but highlights a possible concern of the language.
In both tests, the programmers were able to use the functions they have written in assignment 2.c and 2.d (see appendix J) to write a blink program in Tang. However, only the second test person's program was able to compile as the first programmer used curly brackets instead of indentation. This programmer later stated, that he did it this way due to him not knowing that the programming used indentation instead of curly brackets.

Even though the test in itself does not indicate whether or not Tang has the intended readability and writability, it provides insight into some of the challenges of the language. During the tests we discovered that testing Tang was difficult since programming hardware can be challenging without more than a brief introduction of how to to this.

Chapter 10

Discussion

10.1 General Concerns

In this section, we will summarise and discuss some of the general concerns regarding the implementation of Tang.
In section 6.3, we selected different features for Tang from the imperative and object-oriented paradigm. Though we did not consider all possible features from these paradigms and as such we may have missed some. The language may have been better designed if we had included members of the target-group in the process.
The syntax for Tang was designed based on the prioritised language evaluation criteria and influenced by data from a scientific paper for the Quorum language (see section 6.4). However, not all features of Tang were examined in the Quorum paper and the design of these features were instead based on our own assessment of how intuitive these features were for beginners of hardware programming.
The evaluation of Tang and other Arduino languages, was based on a description and prioritisation of the following four language criteria from [41]: *writability, readability, reliability, and cost.*
Using these criteria to evaluate the different languages in section 9.4 was found to be difficult, as the evaluation criteria do not state directly how to measure each of the criteria, but instead describes the characteristics of each criteria. These characteristics can be somewhat contradicting, and in some cases it is difficult to determine if a specific feature enhances a specific evaluation criteria of the language. For instance [41] says that a characteristic of a writable language is a language that is expressive and readable. However, [41] then states that expressive features happen at the expense of readability of a language, which makes it hard to determine if a language feature that is expressive, but not readable, increases or decreases the writability of a language.
In most cases, the characteristics of the four criteria can indicate whether a feature enhances a specific language criteria. We found it difficult to use it as a basis for comparing different languages, because it is hard to compare e.g. the readability of two languages when readability is not directly measurable, whereas it is easier to determine if adding a feature makes a language more readable based on the characteristics of readability from [41].
In relation to the measurability of the criteria, we experienced that if we have two languages that have two distinct characteristics for enhancing some language evaluation criteria, it is hard to determine which of these characteristics are most important for the specific evaluation criteria as it is not defined by [41]. An example of this, which we experienced in 9.4, is that the Arduino language used more expressive features for the blink example than AVR C did, but AVR C was simpler in terms of the total constructs used, which both are characteristics of writable languages, but do not directly state which language are most writable.
The concerns of the current approach of evaluating languages in this project is therefore that there

is no deterministic way of evaluating a language. Another concern is that the evaluation conducted on the six different languages: AVR C, AVR Assembly, the Arduino language, Java, JavaScript, and Tang only looks at a subset of each of these languages. This means that other issues might have been discovered if other subsets of these languages were evaluated. However, the reason for evaluating the languages in relation to only a subset of the languages is that if we were to evaluate the entire languages it would be difficult and time-consuming as the tested languages are from different paradigms and contain a lot of different constructs that are not comparable.
Another reason for testing only a subset of the languages is that the subset that is tested is related to hardware programming and has to do with e.g. changing bits in registers which is central to hardware programming, and thus also this project.
It is also a troublesome task to analyse if the blink programs written in the languages stated above are equally making use of the features of the specific languages. This may mean that if the blink program written in e.g. Java was not optimally written, then it could cause Java to be unfairly slow relative to the other languages.

10.2 Choice of Software Development Method

In chapter 2, we chose to use the iterative waterfall method as the software development method for this project.
The reason we chose this, was that we did not have any experience with the design and implementation of a programming language. The iterative waterfall method would allow us to understand each phase of designing and creating a language before moving on to the next phase while also allowing us to iterate through and modify earlier phases. This approach was suitable as it allowed us to understand the problems with current Arduino languages before designing the features we wanted in the finished version of Tang. This helped the design of the language as we knew which kind of language we wanted to design, and which problems we wanted to solve.

After having formally described some of the features of Tang, we discovered that it was a time consuming task to formalise a language and because we did not know the time taken to formalise and implement the entire language, we decided to do this in iterations.
Because we had already decided which features were to be included in Tang, we were able to make a MoSCoW analysis where we prioritised the language features of Tang into four categories based on their importance. We therefore moved to a more iterative approach to formalise and describe a subset of the language where we, in each iteration, created the features that correspond to the four categories of the MoSCoW analysis starting with the most important features (see section 6.6).
Even though the chosen development method was modified towards a more iterative approach to accommodate the time constraints of the project, the iterative waterfall method allowed us to get the full overview of the language we wanted to design, while also allowing us to iterate and modify different phases of the design and implementation of the language. We viewed this as a necessity in our project, as we discovered that compiler and language design is an iterative process, where a lot of changes and modifications occur as new experiences and knowledge is obtained.

10.3 Language Evaluation Criteria

In this section the findings of the language evaluation in section 9.4 will be discussed.
In accordance to the evaluation criteria established for each of the languages, we see that the subset of Arduino language we tested is the most readable and writable of the different language subsets. One reason for this, is that the Arduino language adds a layer of abstraction where it, based on the

selected Arduino board in the Arduino IDE, imports a library that contains functions for setting a delay, or setting a hardware pin to LOW/HIGH according to the print on the board.
The problem with doing so is that the Arduino language is basically C++ code with predefined functions which means that if the user wants to write some of these functions, in order to get a better understanding of the hardware, they have to write code according to the C++ specification's meaning that they have to use complex operations like bit-shifting.
In Tang, we wanted to avoid this and therefore did not include functions for setting a delay etc. as part of the pre-processing of Tang. The reason for this choice is that we in Tang wanted to make a programming language where programmers that had not previously programmed hardware would get to work with hardware registers and pins the way they are represented on the micro-controller board without having to learn multiple bitwise operators.

10.4 Implementation considerations

Based on the choice of parser implementation approach in section 8.1.4, we implemented a parser generator. We extended this implementation approach to the development of a compiler generator to generate the parser, type checking, and syntax tree translation for the Tang compiler. In this section we will discuss the advantages and disadvantages of our implementation approach. The choice of making a LL(1) grammar enabled us to implement a parser generator. Generating a parser based on this grammar was a manageable task after the calculation of predict sets were implemented. The disadvantage of using a LL(1) grammar was that the structure of the CST reflected the left factorisations done in the grammar. For instance, this resulted in a non-preferable structure of expressions (see section 8.4.2).
Some limitations could be solved by using a less restrictive grammar class like LL(k), that requires the parser to be able to apply rules based on a look-a-head of multiple tokens. However, this adds complexity to the parser and thereby also the parser generator. The approach of creating a parser generator in this project was therefore possible due to the simplicity of the recursive descent parsing strategy and the choice of LL(1) grammar. For projects with higher requirements for the parser it might be preferable to use an existing parser generator in order to save time. To translate a CST to an AST for Tang we invented a new language to denote rules of syntax tree translation. This translation language can be categorised as a logical programming language as it defines relations between syntax trees based on inference rules and axioms (see section 5.2.3).
The use of inference rules makes it similar to the use of rules in the structural operational semantics and type systems. Besides translating a CST to an AST, the translation language can be used for type checking and code generation as well (see section 8.4.2 and 8.4.4). A disadvantage of using the translation language for most of the compiler was that we needed to implement a translator generator to convert the translation language to the C# implementation language of the Tang compiler. For this, we discovered an advantage of creating a parser generator as we could use this to generate the parser for the translation language. This means that parts of the generator are generated.
A disadvantage of our implementation approach is that the implementation of the compiler generator has been a time consuming process. On the other hand the advantage is that, besides fulfilling the primary implementation goal i.e. implementing a compiler based on the formalisation of Tang, we have introduced a new language for syntax tree translation and implemented a compiler generator that can be used to generate a compiler's components in future projects.

Chapter 11

Conclusion

In this project, we have designed, formalised, implemented, and tested a new programming language called Tang. Tang is an imperative programming language for the Arduino that has been developed to answer the problem statement:
"How can a programming language for Arduino be developed for programmers with no previous experience in embedded programming?".

To answer the problem statement, the design and implementation of Tang have been developed to answer the four research questions related to the problem statement (see section 4).

As an answer to the question *How can the programmer be able to control hardware components on the Arduino?* we added, in the design of Tang, features for controlling hardware components like register types, a bit index operator, and an interrupt construct (see section 6.3). Furthermore, we added support for bit-length on register and integer types which allows programmers to make better use of the low memory capabilities of the Arduino board (see section 3.1.1).
Additional operators, types, and control-statements were also added to the design of Tang to enable control-flow, arithmetic operations, and to represent binary values in the language. These additional features can be used along with e.g. registers, to control hardware components. The blink example in listing 9.16 shows how a subset of these features can be used to control hardware components.

As an answer to the question *How can the programming language support abstractions of hardware components?* we implemented a register and an interrupt construct which are abstractions of hardware registers and hardware interrupts.
Furthermore, we added functions and classes to the design of Tang which allows the programmer to abstract over hardware components by e.g. creating classes to represent hardware components which, could be a pin class containing functions for reading and writing from a specific hardware pin to Tang.
The implementation of Tang developed in this project includes support for registers, interrupts, and functions but not classes since these were prioritised low in the MoSCoW analysis (see section 6.6).

The research question *How can syntax and semantics be defined for the programming language?* is answered b formally specifying the tokens of Tang using regular expressions in 7.1.1, describing the syntactic rules of Tang using a CFG in section 7.1.2, describing the run-time semantics of Tang using structural operational semantics in section 5.4, and formally describing the legal types using a type system in section 7.2.3.
The constraints of the formal descriptions of the syntax and semantics of Tang are described in

section 7.3. Some of these constraints concern the operational semantics not giving a formal description of the run-time semantics of interrupts, which are instead informally described.

To test the language, we devised a test plan in section 2.6, which includes four types of tests: unit, integration, verification, and evaluation tests.
These tests were described in chapter 9 and the unit tests assess the individual components of the compiler to check that they function as intended.
The integration tests check if these components work together by compiling a bunch of programs written specifically for this purpose and the run-time tests validate that different programs written in Tang execute properly on an Arduino Nano.
In the language evaluation test, we compared Tang to other languages for embedded programming in order to understand the differences between these languages. The language evaluation test shows that the Arduino language is the most writable and readable of all the languages compared, due to having a lot of predefined functions that are imported upon pre-processing. Furthermore, it shows that the reliability of Tang to some degree is constrained by the reliability of C (see section 9.4).

To evaluate Tang from another perspective than those of this project group, we tested the language on two students from a computer science group at Aalborg University. The test highlights an ambiguity in the syntax-design of for-loops, as well as a possible confusion of whether or not specific hardware pin's mode are set to output or input. In later iterations of Tang the language should be tested on a larger set of the target group. Furthermore, the following missing features should be formally described, implemented, and tested: strings, floats, chars, arrays, and classes.

As discussed in chapter 10, we developed a tool to generate most of the compiler. Since this was not the purpose of the project, it was important for us, that its development did not distract from the design and implementation of Tang. We did not create a tool because we thought it was a more time efficient way to implement a compiler, compared using existing tools, but it provided us with insight into the development of a compiler while also giving us some of the benefits of a tool, like more maintainable code.
In conclusion, we have designed, implemented, and tested a programming language targeted at programmers with little to no experience with developing for embedded systems. The language enables programmers to interact with the underlying hardware components on the Arduino board through abstractions over some low-level systems like registers and interrupts. The language also allows programmers to create their own higher-level abstractions through the use of functions. The syntax of the language is formalised as an LL(1) grammar in BNF format and the semantics is formally described using structural operational semantics and a type system.
The status of the implementation of Tang is that we have implemented the features that have been formalised in the language specification of Tang including file import and comments.

Bibliography

[1] Charles Fischer. *Crafting a Compiler*. Harlow: Pearson Education Limited, 2016. ISBN: 0-13-606705-0.

[2] Lone Leth Thomsen. *A new programming language for Arduino*. Danish. https://www.moodle.aau.dk/pluginfile.php/888450/mod_page/content/16/SW4_ideas_lone.pdf.

[3] AAU. *Studieordning for Bacheloruddannelsen i Software*. Danish. http://www.sict.aau.dk/digitalAssets/266/266284_software-bachelor-f17-03_07_2016.pdf.

[4] Winston W. Royce. *MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS*. English. http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf. 2017.

[5] Barry Boehm. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston: Addison Wesley, 2009. ISBN: 978-0321186126.

[6] Nitrit Deepak. *Flexible Self-Managing Pipe-line Framework Reducing Development Risk to Improve Software Quality*. https://www.researchgate.net/publication/279195731_Flexible_Self-Managing_Pipe-line_Framework_Reducing_Development_Risk_to_Improve_Software_Quality. 2015.

[7] Ian Sommerville. *Software engineering*. Harlow, England New York: Addison-Wesley, 2007. ISBN: 9780321313799.

[8] Jacob Nielsen. *Iterative User Interface Design*. English. https://www.nngroup.com/articles/iterative-design/. 2017.

[9] Arduino. *Blink*. English. https://www.arduino.cc/en/tutorial/blink.

[10] Svetomir Kurtev, Tommy Aagaard Christensen, and Bent Thomsen. *Discount Method for Programming Language Evaluation*. English. http://dl.acm.org/citation.cfm?doid=3001878.3001879. 2016.

[11] Arduino. *What is Arduino?* English. https://www.arduino.cc/en/Guide/Introduction. 2017.

[12] Processingl. *Processing Environment*. English. https://processing.org/reference/environment/. 2017.

[13] Arduino. *Arduino Products*. English. https://www.arduino.cc/en/Main/Products. 2017.

[14] Arduino. *ARDUINO UNO REV3*. English. https://store.arduino.cc/arduino-uno-rev3. 2017.

[15] Arduino. *ARDUINO NANO*. English. https://store.arduino.cc/arduino-nano. 2017.

[16] czb6721960 (Ebay user). *Nano V3.0 Mini USB ATmega328 5V 16M Micro-controller Board CH340G Arduino Cable*. English. http://www.ebay.com/itm/Nano-V3-0-Mini-USB-ATmega328-5V-16M-Micro-controller-Board-CH340G-Arduino-Cable-/162002876661?hash=item25b81fb8f5:g:RYYAAOSw5cNYfJ0j. 2017.

[17] microchip DIRECT. *ATMEGA328P*. English. http://www.microchipdirect.com/ProductDetails.aspx?Category=ATmega328P&&keywords=ATMEGA328P-AN. 2017.

[18] Frédéric Gaillard. *Microprocessor (MPU) or Microcontroller (MCU)? What factors should you consider when selecting the right processing device for your next design*. English. http://www.atmel.com/Images/MCU_vs_MPU_Article.pdf. 2017.

[19] Paul Zandbergen. *Motherboard definition and function*. English. http://study.com/academy/lesson/what-is-a-motherboard-definition-function-diagram.html. 2017.

[20] Atmel Corporation. *ATmega328/P*. http://www.atmel.com/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf. 2016.

[21] AVR-Tutorials. *AVR 8 bit microcontrollers Interrupts*. English. http://www.avr-tutorials.com/interrupts/about-avr-8-bit-microcontrollers-interrupts. 2016.

[22] Arduino. *Arduino Frequently Asked Questions*. English. https://www.arduino.cc/en/Main/FAQ. 2017.

[23] Arduino. *Examples from Libraries*. English. https://www.arduino.cc/en/Tutorial/LibraryExamples. 2017.

[24] Arduino. *"Hello World!"* English. https://www.arduino.cc/en/Tutorial/HelloWorld. 2017.

[25] Richard Blum. *Professional assembly language*. Indianapolis, IN: Wiley, 2005. ISBN: 9780764579011.

[26] Atmel Corporation. *AVR Instruction Set Manual*. http://www.atmel.com/images/Atmel-0856-AVR-Instruction-Set-Manual.pdf. 2016.

[27] Atmel Corporation. *ATmega48A/PA/88A/PA/168A/PA/328/P*. http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf. 2015.

[28] Arduino. *ATmega168/328-Arduino Pin Mapping*. English. https://www.arduino.cc/en/Hacking/PinMapping168. 2017.

[29] Arduino. *Arduino Uno & Arduino Genuino*. English. https://www.arduino.cc/en/Main/ArduinoBoardUno. 2017.

[30] Inc. Free Software Foundation. *avr-gcc(1) - Linux man page*. English. https://linux.die.net/man/1/avr-gcc. 2010.

[31] Inc. Free Software Foundation. *avr-objcopy(1) - Linux man page*. English. https://linux.die.net/man/1/avr-objcopy. 2009.

[32] Brian S. Dean. *avrdude(1) - Linux man page*. English. https://linux.die.net/man/1/avrdude. 2017.

[33] nongnu. *avr-libc and assembler programs*. English. http://www.nongnu.org/avr-libc/user-manual/assembler.html. 2017.

[34] Tutorials Point. *C - Preprocessors*. English. https://www.tutorialspoint.com/cprogramming/c_preprocessors.htm. 2017.

[35] ISO/IEC. *ISO/IEC 9899:201x*. English. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf. 2007.

[36] Till Harbaum. *The NanoVM - Java for the AVR*. http://www.harbaum.org/till/nanovm/index.shtml. 2017.

[37] Genom Bob. *HaikuVM: a small JAVA VM for microcontrollers*. http://haiku-vm.sourceforge.net/. 2017.

[38] Stephan Korsholm. "Java for Cost Effective Embedded Real-Time Software". In: (2012).

[39] BitRot. *Using Arduino, Firmata and Processing Together*. English. https://adestefawp.wordpress.com/learning/using-arduino-firmata-and-processing-together/. 2017.

[40] Johnny Five. *Johnny Five*. English. http://johnny-five.io/. 2017.

[41] Robert Sebesta. *Concepts of programming languages*. Harlow: Pearson Education Limited, 2016. ISBN: 978-0-13-394302-3.

[42] Richard Cammack. *Oxford dictionary of biochemistry and molecular biology*. Oxford New York: Oxford University Press, 2006. ISBN: 9780198529170.

[43] Kurt Nørmark. *Overview of the four main programming paradigms*. English. http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html. 2017.

[44] Kurt Nørmark. *Programming Paradigms*. English. http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigms.html. 2017.

[45] Maurizio Gabbrielli. *Programming languages : principles and paradigms*. London New York: Springer, 2010. ISBN: 978-1-84882-913-8.

[46] Milena Vujošević-Janičić and Dušan Tošić. *THE ROLE OF PROGRAMMING PARADIGMS IN THE FIRST PROGRAMMING COURSES*. English. http://elib.mi.sanu.ac.rs/files/journals/tm/21/tm1122.pdf. 2008.

[47] Gary T. Leavens. *Major Programming Paradigms*. English. http://www.cs.ucf.edu/~leavens/ComS541Fall97/hw-pages/paradigms/major.html. 2017.

[48] Michael Sipser. *Introduction to the theory of computation*. Andover: Cengage Learning, 2013. ISBN: 978-1-133-18781-3.

[49] Dave Child. *Regular Expression Cheat Sheet by DaveChild*. English. https://www.cheatography.com/davechild/cheat-sheets/regular-expressions/. 2017.

[50] Bent Thomsen. *Languages and Compilers Lecture 5 slides*. English. https://www.moodle.aau.dk/pluginfile.php/923513/mod_folder/content/0/SPOF17-5.pdf. 2017.

[51] Maggie Johnson. *LALR Parsing*. English. https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/140%20LALR%20Parsing.pdf. 2017.

[52] Maggie Johnson. *Miscellaneous Parsing*. English. https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/150%20Miscellaneous%20Parsing.pdf. 2017.

[53] Hans Hüttel. *Transitions and Trees : an Introduction to Structural Operational Semantics*. Cambridge: Cambridge University Press, 2010. ISBN: 978-0-521-14709-5.

[54] David Watt. *Programming Language Processors in Java : COMPILERS AND INTERPRETERS*. Harlow, England New York: Prentice Hall, 2000. ISBN: 0130257869.

[55] Bent Thomsen. *Languages and Compilers, Lecture 1*. English. https://www.moodle.aau.dk/pluginfile.php/923509/mod_folder/content/0/SPOF17-1.pdf?forcedownload=1. Accessed: 09/03/2017. 2017.

[56] Erich Gamma et al. *Design Patterns - Elements of reusable Object-Oriented Software*. English. 1994.

[57] SUSANNA SIEBERT ANDREAS STEFIK. "An Empirical Investigation into Programming Language Syntax". In: (2013). URL: http://dl.acm.org/citation.cfm?id=2534973.

[58] Microsoft. *C Identifiers*. English. https://msdn.microsoft.com/en-us/library/e7f8y25b.aspx. 2017.

[59] Cafe au Lait Java News and Resources. *Identifiers in Java*. English. http://www.cafeaulait.org/course/week2/08.html. 2017.

[60] Microsoft. *C# Operators*. English. https://msdn.microsoft.com/en-us/library/6a71f45d.aspx. 2017.

[61] David Benyon. *Designing interactive systems : a comprehensive guide to HCI, UX and interaction design*. Harlow, England: Pearson, 2014. ISBN: 978-1-4479-2011-3.

[62] Hanspeter Mössenböck et. al. *The Compiler Generator Coco/R*. English. http://www.ssw.uni-linz.ac.at/Coco/. 2014.

[63] Hanspeter Mössenböck et. al. *The Compiler Generator Coco/R - User Manual*. English. http://www.ssw.uni-linz.ac.at/Coco/Doc/UserManual.pdf. 2010.

[64] Devin Cook. *Why Use GOLD?* http://www.goldparser.org/about/why-use-gold.htm. 2017.

[65] Devin Cook. *GOLDparser*. http://www.goldparser.org/. 2017.

[66] Martin Fowler. *HelloSablecc*. https://www.martinfowler.com/bliki/HelloSablecc.html. 2007.

[67] Étienne Gagnon. *SABLECC, AN OBJECT-ORIENTED COMPILER FRAMEWORK*. English. http://sablecc.sourceforge.net/thesis/thesis.html. 1998.

[68] SableCC. *SableCC*. English. http://www.sablecc.org/. 2017.

[69] SableCC. *SableCC Documentation*. English. http://www.sablecc.org/documentation. 2017.

[70] Seth D. Bergmann. *Compiler Design: Theory, Tools, and Examples*. English. http://elvis.rowan.edu/~bergmann/books/java/CompilerDesignBook.pdf. 2016.

[71] Terence Parr. *ANTLR*. English. http://www.antlr.org/. 2014.

[72] Safari Books Online. *ANTLR Patterns*. English. https://www.safaribooksonline.com/library/view/the-definitive-antlr/9781941222621/f_0020.html. 2017.

[73] qwertie256. *Enhanced C Sharp, LLLPG*. English. http://ecsharp.net/lllpg/. 2017.

[74] qwertie256. *Enhanced C Sharp, LeMP*. English. http://ecsharp.net/lemp/. 2017.

[75] Jan Goyvaerts. *Specifying Modes Inside The Regular Expression*. English. http://www.regular-expressions.info/modifiers.html. 2017.

[76] Python. *Lexical Analysis - Indentation*. English. https://docs.python.org/2/reference/lexical_analysis.html#indentation. 2017.

[77] James Brunskill. *First and Follow Sets*. English. https://www.jambe.co.nz/UNI/FirstAndFollowSets.html. 2017.

[78] gnu. *The GNU C Reference Manual*. English. https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html. 2017.

[79] Xunit. *Xunit test framework*. English. https://xunit.github.io/.

[80] Microsoft. *Data Driven Test*. English. https://docs.microsoft.com/da-dk/visualstudio/test/how-to-create-a-data-driven-unit-test.

[81] Mathias Pihl. *Blink and PWM Example*. English. youtu.be/jyWI7dkbzIk.

[82] Arduino. *Arduino.h*. English. https://github.com/arduino/Arduino/blob/master/hardware/arduino/avr/cores/arduino/Arduino.h. 2017.

[83] Wikipedia. *Integer overflow*. English. https://en.wikipedia.org/wiki/Integer_overflow. 2017.

[84] GCC Wiki. *AVR-GCC Type Layout*. https://gcc.gnu.org/wiki/avr-gcc. 2017.

[85] Arduino. *Java Type-checking*. English. http://www.programcreek.com/2011/12/an-example-of-java-static-type-checking/. 2017.

[86] Margaret Rouse. *Java Virtual Machine*. English. http://www.theserverside.com/definition/Java-virtual-machine-JVM. 2017.

[87] Mathias Pihl. *Video of different languages blinking on the Arduino*. English. https://youtu.be/xVhtEskgUK8. 2017.

Appendices

Appendix A

Disassemble of Binaries for Blink Example in C

```
00000000 <__vectors>:
   0:   0c 94 34 00      jmp     0x68    ; 0x68 <__ctors_end>
   4:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
   8:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
   c:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  10:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  14:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  18:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  1c:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  20:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  24:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  28:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  2c:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  30:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  34:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  38:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  3c:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  40:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  44:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  48:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  4c:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  50:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  54:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  58:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  5c:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  60:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>
  64:   0c 94 3e 00      jmp     0x7c    ; 0x7c <__bad_interrupt>

00000068 <__ctors_end>:
  68:   11 24           eor     r1, r1
  6a:   1f be           out     0x3f, r1        ; 63
  6c:   cf ef           ldi     r28, 0xFF       ; 255
  6e:   d8 e0           ldi     r29, 0x08       ; 8
```

```
  70:   de bf           out     0x3e, r29       ; 62
  72:   cd bf           out     0x3d, r28       ; 61
  74:   0e 94 40 00     call    0x80    ; 0x80 <main>
  78:   0c 94 56 00     jmp     0xac    ; 0xac <_exit>

0000007c <__bad_interrupt>:
  7c:   0c 94 00 00     jmp     0       ; 0x0 <__vectors>

00000080 <main>:
  80:   25 9a           sbi     0x04, 5 ; 4
  82:   2d 9a           sbi     0x05, 5 ; 5
  84:   2f ef           ldi     r18, 0xFF       ; 255
  86:   83 ed           ldi     r24, 0xD3       ; 211
  88:   90 e3           ldi     r25, 0x30       ; 48
  8a:   21 50           subi    r18, 0x01       ; 1
  8c:   80 40           sbci    r24, 0x00       ; 0
  8e:   90 40           sbci    r25, 0x00       ; 0
  90:   e1 f7           brne    .-8             ; 0x8a <main+0xa>
  92:   00 c0           rjmp    .+0             ; 0x94 <main+0x14>
  94:   00 00           nop
  96:   2d 98           cbi     0x05, 5 ; 5
  98:   2f ef           ldi     r18, 0xFF       ; 255
  9a:   83 ed           ldi     r24, 0xD3       ; 211
  9c:   90 e3           ldi     r25, 0x30       ; 48
  9e:   21 50           subi    r18, 0x01       ; 1
  a0:   80 40           sbci    r24, 0x00       ; 0
  a2:   90 40           sbci    r25, 0x00       ; 0
  a4:   e1 f7           brne    .-8             ; 0x9e <main+0x1e>
  a6:   00 c0           rjmp    .+0             ; 0xa8 <main+0x28>
  a8:   00 00           nop
  aa:   eb cf           rjmp    .-42            ; 0x82 <main+0x2>

000000ac <_exit>:
  ac:   f8 94           cli

000000ae <__stop_program>:
  ae:   ff cf           rjmp    .-2             ; 0xae <__stop_program>
```

Listing A.1: Shows the disassemble of the compiled binaries from the blink program in listing 3.4. The blink example program was compiled with avrgcc using flags mmcuātmega328p and Os.

Appendix B

Java LED Class for Arduino

```
1  package processing.examples._01_Basics;
2
3  import static processing.hardware.arduino.cores.arduino.Arduino.*;
4
5  public class Led {
6      int ledPin;
7      int blinkInterval;
8      BlinkThread blinkThread;
9      boolean blinking = false;
10
11     public Led(int pin) {
12         ledPin = pin;
13         blinkInterval = 1000;
14         pinMode(ledPin, OUTPUT);
15         blinkThread = new BlinkThread();
16     }
17
18     class BlinkThread extends Thread {
19         boolean running = false;
20         public void run() {
21             running = true;
22             while(running) {
23                 digitalWrite(ledPin, HIGH);
24                 delay(blinkInterval);
25                 digitalWrite(ledPin, LOW);
26                 delay(blinkInterval);
27             }
28         }
29         public void kill() {
30             if(running) {
31                 running = false;
32                 stop();
33             }
34         }
35     }
```

```
36
37      public void blink(int interval) {
38          blinkThread.kill();
39          blinkInterval = interval;
40          blinkThread = new BlinkThread();
41          blinkThread.start();
42      }
43
44      public void on() {
45          blinkThread.kill();
46          digitalWrite(ledPin, HIGH);
47      }
48
49      public void off() {
50          blinkThread.kill();
51          digitalWrite(ledPin, LOW);
52      }
53
54      public boolean isOn() {
55          return digitalRead(ledPin) == HIGH;
56      }
57  }
```

Listing B.1: Shows Led class implemented in Java using HaikuVM.

Appendix C

Structural Operational Semantics

Abstract syntax

P	$\in$ **Prog**	- Programs
S	$\in$ **Stm**	- Root level statements
s	$\in$ **SimpleStm**	- Scoped level statements
s_{if}	$\in$ **IfStm**	- If Statements
e	$\in$ **Exp**	- Expressions
b	$\in$ **BoolExp**	- Boolean expressions
i	$\in$ **IntExp**	- Integer expressions
x	$\in$ **VNames**	- Variable names
p	$\in$ **PNames**	- Procedure names
n	$\in$ **Num**	- Numerals
T_{Int}	$\in$ **IntTypes**	- Integer types {`int8`, `int16`, `int32`}
T_{Reg}	$\in$ **RegTypes**	- Register types {`register8`, `register16`}
T	$\in$ **Types**	- Types ($T_{Int} \cup T_{Reg} \cup$ {`bool`, `nothing`, `undefined`})
v	$\in$ **Val**	- Values ($\mathbb{Z} \cup \{tt, ff\}$)
r	$\in$ **Ret**	- Return values. **Ret** = (**Types** $\times$ **Val**) $\cup \epsilon$

Table C.1: Syntactic categories

$$
\begin{array}{rcl}
P & ::= & S \mid \epsilon \\
S & ::= & S_1 \texttt{ newline } S_2 \mid s \mid T\ p(T_1\ x_1,\ T_2\ x_2,\ \ldots\ T_k\ x_k) \texttt{ indent } s \texttt{ dedent} \mid \texttt{interrupt}(n) \texttt{ indent } s \texttt{ dedent} \\
s & ::= & \texttt{newline} \mid e \mid s_1 \texttt{ newline } s_2 \mid T\ x \mid T\ x\ = e \mid x = e \\
 & \mid & x\{i\} = b \mid T_{Reg}(n)\{i\} = b \mid \texttt{return} \mid \texttt{return } e \mid \texttt{while}(b) \texttt{ indent } s \texttt{ dedent} \\
 & \mid & \texttt{for}(T_{Int}\ x \texttt{ from } i_1 \texttt{ to } i_2) \texttt{ indent } s \texttt{ dedent} \mid s_{if} \\
s_{if} & ::= & \texttt{if}(b) \texttt{ indent } s \texttt{ dedent} \mid \texttt{if}(b) \texttt{ indent } s \texttt{ dedent else } s_{if} \\
 & \mid & \texttt{if}(b) \texttt{ indent } s_1 \texttt{ dedent else indent } s_2 \texttt{ dedent} \\
e & ::= & x \mid i \mid b \mid (e_1) \mid p(e_1,\ e_2,\ \ldots\ e_k) \mid T_{Reg}(n) \\
i & ::= & n \mid e \mid i_1 + i_2 \mid i_1 - i_2 \mid i_1 * i_2 \mid i_1/i_2 \mid i_1 \texttt{ \% } i_2 \mid i_1\texttt{\^{}}i_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid e \mid e_1 == e_2 \mid e_1\texttt{!=}\ e_2 \mid i_1 < i_2 \mid i_1 <= i_2 \mid i_1 >= i_2 \mid i_1 > i_2 \mid \texttt{!}b_1 \\
 & \mid & b_1 \texttt{ and } b_2 \mid b_1 \texttt{ or } b_2 \mid x\{i\} \mid T_{Reg}(n)\{i\}
\end{array}
$$

Table C.2: Formation rules where `newline`, `indent` and `dedent` are substitutes for their respective tokens

Environment-store model definitions

$$\begin{array}{l}
\mathbf{Loc} = \mathbb{N} \\
l \in \mathbf{Loc}
\end{array}$$

$$\begin{array}{l}
\mathbf{Sto} = \mathbf{Loc} \rightharpoonup (\mathbf{Types} \times \mathbf{Val}) \\
sto \in \mathbf{Sto} \\
sto[l \mapsto (T, v)] = sto' \\
sto'y = \begin{cases} sto\ y & \text{if } y \neq l \\ (T, v) & \text{if } y = l \end{cases}
\end{array}$$

$$\begin{array}{l}
\mathbf{EnvV} = \mathbf{VNames} \cup \{\text{next}\} \rightharpoonup \mathbf{Loc} \\
env_V \in \mathbf{EnvV} \\
env_V[x \mapsto l] = env'_V \\
env'_V\ y = \begin{cases} env_V\ y & \text{if } y \neq x \\ l & \text{if } y = x \end{cases}
\end{array}$$

$$\begin{array}{l}
\mathbf{EnvP} = \mathbf{PNames} \rightharpoonup (\mathbf{EnvV} \times \mathbf{SimpleStm} \times (\mathbf{VNames}^k)) \\
env_P \in \mathbf{EnvP} \\
env_P[p \mapsto (env_V, s, (x_1, x_2,\ \ldots\ x_k))] = env'_P \\
env'_P\ q = \begin{cases} env_P\ q & \text{if } q \neq p \\ (env_V, s, (x_1, x_2,\ \ldots\ x_k)) & \text{if } q = p \end{cases}
\end{array}$$

The function *new* evaluates to a new location following the given location.

$$\begin{array}{l}
new : \mathbf{Loc} \to \mathbf{Loc} \\
new\ l = l + 1
\end{array}$$

The function *default* evaluates to the default value of a type.

$$\begin{array}{l}
default : \mathbf{Types} \rightharpoonup \mathbf{Val} \\
default\ T = \begin{cases} 0 & \text{if } T \in \{\texttt{int8}, \texttt{int16}, \texttt{int32}, \texttt{register8}, \texttt{register16}\} \\ \mathit{ff} & \text{if } T = \texttt{bool} \end{cases}
\end{array}$$

The function *bitlen* evaluates to the number of bits a variable of the given type can store.

$$\begin{array}{l}
bitlen : \mathbf{Types} \to \mathbf{Val} \\
bitlen\ T = \begin{cases} 0 & \text{if } T = \texttt{nothing} \\ 1 & \text{if } T = \texttt{bool} \\ 8 & \text{if } T \in \{\texttt{int8}, \texttt{register8}\} \\ 16 & \text{if } T \in \{\texttt{int16}, \texttt{register16}\} \\ 32 & \text{if } T = \texttt{int32} \end{cases}
\end{array}$$

Type	Set(Type)
`int8`	$\{-2^{8-1}\ to\ 2^{8s-1}-1\}$
`int16`	$\{-2^{16-1}\ to\ 2^{16-1}-1\}$
`int32`	$\{-2^{32-1}\ to\ 2^{32-1}-1\}$
`bool`	$\{tt, ff\}$
`undefined`	$\varnothing$
`register8`	**Loc**
`register16`	**Loc**
`nothing`	$\{NothingVal\}$

Furthermore we define the following type relation for integers: $int8 \supset int16 \supset int32$. We also define the function largestIntegerType that given a number of types return the biggest subset of the super-type that is given to the function

$$largestIntegerType(T_{Int_1}, T_{Int_2}, \ldots T_{Int_k}) = T_{Int_j} \text{ where } 1 \leq j \leq k \text{ and } T_{Int_i} \supseteq T_{Int_j} \text{ for all } 1 \leq i \leq k$$

The empty variable environment is denoted $env_V^{\varnothing}$.
The semantics of **Prog** are given by the transition system $(\Gamma_P, \rightarrow_P, T_P)$ whose configurations are defined by

$$\Gamma_P = \mathbf{Prog} \cup \mathbf{Ret}$$
$$T_P = \mathbf{Ret}$$

Transitions here have the format $P \rightarrow_P r$

$$[Statement] \quad \frac{\langle S, env_V^{\varnothing}, env_P^{\varnothing}\rangle \rightarrow_{Scan} (env'_V, env'_P) \quad env'_P \vdash \langle S, sto^{\varnothing}, env_V^{\varnothing}\rangle \rightarrow_S (sto, env_V, r)}{S \rightarrow_P r}$$

$$[empty] \qquad \epsilon \rightarrow_P (\texttt{nothing}, nothingval)$$

Table C.3: The transition rules $\rightarrow_P$

In order to fill the procedure environment before executing any statements we define the **Scan** transition system as follows: $(\Gamma_{Scan}, \rightarrow_{Scan}, T_{Scan})$. Its configurations are defined by

$$\Gamma_{Scan} = ((\mathbf{Stm} \cup \mathbf{SimpleStm}) \times \mathbf{EnvP} \times \mathbf{EnvV}) \cup (\mathbf{EnvP} \times \mathbf{EnvV})$$
$$T_{Scan} = \mathbf{EnvP} \times \mathbf{EnvV}$$

The following transitions are for the subset of **Scan** configurations $(\mathbf{Stm} \times \mathbf{EnvP} \times \mathbf{EnvV}) \cup (\mathbf{EnvP} \times \mathbf{EnvV})$ and have the format $\langle S, env_P, env_V\rangle \rightarrow_{Scan} (env'_P, env'_V)$

$[ScanCompoundStatement]$	$\dfrac{\langle S_1, env_P, env_V\rangle \rightarrow_{Scan} (env''_P, env''_V) \quad \langle S_2, env''_P, env''_V\rangle \rightarrow_{Scan} (env'_P, env'_V)}{\langle S_1 \texttt{ newline } S_2, env_P, env_V\rangle \rightarrow_{Scan} (env'_P, env'_V)}$
$[ScanProcedureDeclaration]$	$\langle T\ p(T_1\ x_1,\ T_2\ x_2,\ \ldots\ T_k\ x_k) \texttt{ indent } s \texttt{ dedent}, env_P, env_V\rangle \rightarrow_{Scan} (env_P[p \mapsto (env_V, s, (x_1,\ x_2,\ \ldots\ x_k))], env_V)$
$[ScanInterrupt]$	$\langle \texttt{interrupt}(n) \texttt{ indent } s \texttt{ dedent}, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V)$

Table C.4: Subset of the transition rules $\rightarrow_{Scan}$

The following transitions are for the subset of **Scan** configurations ($\mathbf{SimpleStm} \times \mathbf{EnvP} \times \mathbf{EnvV}$)$\cup$ ($\mathbf{EnvP} \times \mathbf{EnvV}$) and have the format $\langle s, env_P, env_V\rangle \rightarrow_{Scan} (env'_P, env'_V)$

$[ScanNewline]$	$\langle \texttt{newline}, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V)$
$[ScanExpression]$	$\langle e, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V)$
$[ScanCompoundSimpleStatement]$	$\dfrac{\langle s_1, env_P, env_V\rangle \rightarrow_{Scan} (env''_P, env''_V) \quad \langle s_2, env''_P, env''_V\rangle \rightarrow_{Scan} (env'_P, env'_V)}{\langle s_1 \texttt{ newline } s_2, env_P, env_V\rangle \rightarrow_{Scan} (env'_P, env'_V)}$
$[ScanVariableDeclaration]$	$\langle T\ x, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V[x \mapsto l][next \mapsto new(l)])$ Where $l = env_V(next)$
$[ScanVariableDefinition]$	$\dfrac{\langle T\ x, env_P, env_V\rangle \rightarrow_{Scan} (env'_P, env'_V)}{\langle T\ x = e, env_P, env_V\rangle \rightarrow_{Scan} (env'_P, env'_V)}$
$[ScanAssignment]$	$\langle x = e, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V)$
$[ScanRegisterSetBit]$	$\langle x\{i\} = b, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V)$
$[ScanRegisterLiteralSetBit]$	$\langle T_{Reg}(i_1)\{i_2\} = b, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V)$
$[ScanReturn]$	$\langle \texttt{return}, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V)$
$[ScanReturnWithValue]$	$\langle \texttt{return } e, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V)$
$[ScanWhileStatement]$	$\langle \texttt{while}(b) \texttt{ indent } s \texttt{ dedent}, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V)$
$[ScanForStatement]$	$\langle \texttt{for(int } x \texttt{ from } i_1 \texttt{ to } i_2) \texttt{ indent } s \texttt{ dedent}, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V)$
$[ScanIfStatement]$	$\langle s_{if}, env_P, env_V\rangle \rightarrow_{Scan} (env_P, env_V)$

Table C.6: Subset of the transition rules $\rightarrow_{Scan}$

The semantics of **Statements** are given by the transition system $(\Gamma_S, \rightarrow_S, T_S)$ whose configurations are defined by

$$\Gamma_S = (\mathbf{Stm} \times \mathbf{Sto} \times \mathbf{EnvV}) \cup (\mathbf{Sto} \times \mathbf{EnvV} \times \mathbf{Ret})$$
$$T_S = (\mathbf{Sto} \times \mathbf{EnvV} \times \mathbf{Ret})$$

Transitions here have the format $env_P \vdash \langle S, sto, env_V \rangle \rightarrow_S (sto', env'_V, r)$

$[CompoundStatement_1]$ $$\frac{env_P \vdash \langle S_1, sto, env_V \rangle \rightarrow_S (sto', env'_V, r)}{env_P \vdash \langle S_1 \texttt{ newline } S_2, sto, env_V \rangle \rightarrow_S (sto', env'_V, r)}$$ If $r \neq \epsilon$

$[CompoundStatement_2]$ $$\frac{env_P \vdash \langle S_1, sto, env_V \rangle \rightarrow_S (sto'', env''_V, r_1) \quad env_P \vdash \langle S_2, sto'', env''_V \rangle \rightarrow_S (sto', env'_V, r_2)}{env_P \vdash \langle S_1 \texttt{ newline } S_2, sto, env_V \rangle \rightarrow_S (sto', env'_V, r_2)}$$ If $r_1 = \epsilon$

$[SimpleStatement]$ $$\frac{env_P \vdash \langle s, sto, env_V \rangle \rightarrow_s (sto', env'_V, r)}{env_P \vdash \langle s, sto, env_V \rangle \rightarrow_S (sto', env'_V, r)}$$

$[ProcedureDeclaration]$ $env_P \vdash \langle T\ p(T_1\ x_1,\ T_2\ x_2,\ \ldots\ T_k\ x_k)\ \texttt{indent}(n)\ \texttt{indent}\ s\ \texttt{dedent}, sto, env_V \rangle \rightarrow_S (sto, env_V, \epsilon)$

$[Interrupt]$ See section 7.3

Table C.7: The transition rules $\rightarrow_S$

The semantics of **SimpleStm** are given by the transition system $(\Gamma_s, \rightarrow_s, T_s)$ whose configurations are defined by

$$\Gamma_s = (\mathbf{SimpleStm} \times \mathbf{Sto} \times \mathbf{EnvV}) \cup (\mathbf{Sto} \times \mathbf{EnvV} \times \mathbf{Ret})$$
$$T_s = (\mathbf{Sto} \times \mathbf{EnvV} \times \mathbf{Ret})$$

Transitions here have the format $env_P \vdash \langle s, sto, env_V \rangle \rightarrow_s (sto', env'_V, r)$

$[Newline]$ $env_P \vdash \langle \texttt{newline}, sto, env_V \rangle \rightarrow_s (sto, env_V, \epsilon)$

$[Expression]$ $$\frac{env_P, env_V \vdash \langle e, sto \rangle \rightarrow_e (sto', v)}{env_P \vdash \langle e, sto, env_V \rangle \rightarrow_s (sto', env_V, \epsilon)}$$

$[CompoundSimpleStatement_1]$ $$\frac{env_P \vdash \langle s_1, sto, env_V \rangle \rightarrow_s (sto', env'_V, r)}{env_P \vdash \langle s_1 \texttt{ newline } s_2, sto, env_V \rangle \rightarrow_s (sto', env'_V, r)}$$ If $r_1 \neq \epsilon$

$[CompoundSimpleStatement_2]$

$$\frac{env_P \vdash \langle s_1, sto, env_V\rangle \rightarrow_s (sto'', env_V'', r_1) \quad env_P \vdash \langle s_2, sto'', env_V''\rangle \rightarrow_s (sto', env_V', r_2)}{env_P \vdash \langle s_1 \texttt{ newline } s_2, sto, env_V\rangle \rightarrow_s (sto', env_V', r_2)}$$

If $r_1 = \epsilon$

$[VariableDeclaration]$

$$\langle T\ x, sto, env_V\rangle \rightarrow_s (sto[l \mapsto (T, default(T))], env_V[x \mapsto l][next \mapsto new(l)], \epsilon)$$

Where $l = env_V(next)$

$[VariableDefinition]$

$$\frac{env_P, env_V \vdash \langle e, sto\rangle \rightarrow_e (sto'', (T_e, v)) \quad env_P \vdash \langle T\ x, sto'', env_V\rangle \rightarrow_s (sto', env_V', \epsilon)}{env_P \vdash \langle T\ x = e, sto, env_V\rangle \rightarrow_s (sto'[l \mapsto (T, v)], env_V', \epsilon)}$$

Where $env_V'(x) = l$

If $v \in Set(T)$

$[Assignment]$

$$\frac{env_P, env_V \vdash \langle e, sto\rangle \rightarrow_e (sto', (T_e, v))}{env_P \vdash \langle x = e, sto, env_V\rangle \rightarrow_s (sto'[l \mapsto (T_x, v)], env_V, \epsilon)}$$

Where $l = env_V(x)$

And $(T_x, v_x) = sto'(l)$

If $v \in Set(T_x)$

$[RegisterSetBit]$

$$\frac{env_P, env_V \vdash \langle x, sto\rangle \rightarrow_e (sto''', (T_r, l)) \quad env_P, env_V \vdash \langle i, sto'''\rangle \rightarrow_i (sto'', (T_i, v_i)) \quad env_P, env_V \vdash \langle b, sto''\rangle \rightarrow_b (sto', (T_{b_2}, v_{b_2}))}{env_P \vdash \langle x\{i\} = b, sto, env_V\rangle \rightarrow_s (sto'[l \mapsto v'], env_V, \epsilon)}$$

Where $(T_v, v) = sto'\ l$

And $v_{b_1} = \begin{cases} tt & \text{if } \left\lfloor \frac{v}{2^{(v_i)}} \right\rfloor mod\ 2 = 1 \\ ff & \text{if } \left\lfloor \frac{v}{2^{(v_i)}} \right\rfloor mod\ 2 = 0 \end{cases}$

And $v' = \begin{cases} v - 2^{(v_i)} & v_{b_1} \wedge \neg v_{b_2} \\ v & v_{b_1} = v_{b_2} \\ v + 2^{(v_i)} & \neg v_{b_1} \wedge v_{b_2} \end{cases}$

If $v_i \geq 0$

And $v_i < \begin{cases} 8 & \text{If } T_r = \texttt{register8} \\ 16 & \text{If } T_r = \texttt{register16} \end{cases}$

$[RegisterLiteralSetBit]$

$$\frac{env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'''', (T_{i1}, v_{i1})) \quad env_P, env_V \vdash \langle i_2, sto''''\rangle \rightarrow_i (sto''', (T_{i2}, v_{i2})) \quad env_P, env_V \vdash \langle T_{Reg}(n_{i1})\{n_{i2}\}, sto'''\rangle \rightarrow_b (sto'', (T_r, v_r)) \quad env_P, env_V \vdash \langle b, sto''\rangle \rightarrow_b (sto', (T_b, v_b))}{env_P \vdash \langle T_{Reg}(i_1)\{i_2\} = b, sto, env_V\rangle \rightarrow_s (sto'[v_{i1} \mapsto (T, v')], env_V, \epsilon)}$$

If $v_{i1} \geq 0$

And $v_{i2} \geq 0$

$$\text{And } v_{i2} < \begin{cases} 8 & \text{If } T_{Reg} = \texttt{register8} \\ 16 & \text{If } T_{Reg} = \texttt{register16} \end{cases}$$
Where $n_{i1} = \mathcal{N}^{-1}[\![v_{i1}]\!]$
And $n_{i2} = \mathcal{N}^{-1}[\![v_{i2}]\!]$
And $(T, v) = sto'\ v_{i1}$
$$\text{And } v' = \begin{cases} v - 2^{(v_{i2})} & v_r \wedge \neg v_b \\ v & v_r = v_b \\ v + 2^{(v_{i2})} & \neg v_r \wedge v_b \end{cases}$$

[*Return*]
$$env_P \vdash \langle \texttt{return}, sto, env_V \rangle \rightarrow_s (sto, env_V, (\texttt{nothing}, NothingVal))$$

[*ReturnWithValue*]
$$\frac{env_P, env_V \vdash \langle e, sto \rangle \rightarrow_e (sto', (T, v))}{env_P \vdash \langle \texttt{return}\ e, sto, env_V \rangle \rightarrow_s (sto', env_V, (T, v))}$$

[$WhileStatement_1$]
$$\frac{env_P, env_V \vdash \langle b, sto \rangle \rightarrow_b (sto', (T, v))}{env_P \vdash \langle \texttt{while}(b)\ \texttt{indent}\ s\ \texttt{dedent}, sto, env_V \rangle \rightarrow_s (sto', env_V, \epsilon)}$$
If $v = ff$

[$WhileStatement_2$]
$$\frac{\begin{array}{c} env_P, env_V \vdash \langle b, sto \rangle \rightarrow_b (sto'', (T, v)) \\ env_P \vdash \langle s, sto'', env_V \rangle \rightarrow_s (sto', env'_V, r) \end{array}}{env_P \vdash \langle \texttt{while}(b)\ \texttt{indent}\ s\ \texttt{dedent}, sto, env_V \rangle \rightarrow_s (sto', env_V, r)}$$
If $v = tt$ and $r \neq \epsilon$

[$WhileStatement_3$]
$$\frac{\begin{array}{c} env_P, env_V \vdash \langle b, sto \rangle \rightarrow_b (sto''', (T, v)) \\ env_P \vdash \langle s, sto''', env_V \rangle \rightarrow_s (sto'', env'_V, r) \\ env_P \vdash \langle \texttt{while}(b)\ \texttt{indent}\ s\ \texttt{dedent}, (sto'', env'_V) \rangle \rightarrow_s (sto', env'_V, r) \end{array}}{env_P \vdash \langle \texttt{while}(b)\ \texttt{indent}\ s\ \texttt{dedent}, sto, env_V \rangle \rightarrow_s (sto', env_V, r)}$$
If $v = tt$ and $r = \epsilon$

[$ForStatement_1$]
$$\frac{\begin{array}{c} env_P, env_V \vdash \langle i_1, sto \rangle \rightarrow_i (sto'''', (T_{i1}, v_{i1})) \\ env_P, env_V \vdash \langle i_2, sto'''' \rangle \rightarrow_i (sto''', (T_{i2}, v_{i2})) \\ env_P \vdash \langle T_{Int}\ x = n_{i1}, sto''', env_V \rangle \rightarrow_s (sto''', env''_V, r_1) \\ env_P \vdash \langle s, sto''', env''_V \rangle \rightarrow_s (sto'', env'_V, r_2) \\ env_P \vdash \langle \texttt{for}(T_{Int}\ x\ \texttt{from}\ n_{i1} + n_c\ \texttt{to}\ n_{i2})\ \texttt{indent}\ s\ \texttt{dedent}, sto'', env_V \rangle \rightarrow_s (sto', env'''_V, r_3) \end{array}}{env_P \vdash \langle \texttt{for}(T_{Int}\ x\ \texttt{from}\ i_1\ \texttt{to}\ i_2)\ \texttt{indent}\ s\ \texttt{dedent}, sto, env_V \rangle \rightarrow_s (sto', env_V, r_3)}$$
Where $n_{i1} = \mathcal{N}^{-1}[\![v_{i1}]\!]$
And $n_{i2} = \mathcal{N}^{-1}[\![v_{i2}]\!]$
And $n_c = \mathcal{N}^{-1}[\![(v_{i2} - v_{i1})/|v_{i2} - v_{i1}|]\!]$
If $v_{i1} \neq v_{i2}$
And $r_2 = \epsilon$
And $T_{Int} = largestIntegerType(T_{Int}, T_{i2})$

[$ForStatement_2$]
$$\frac{\begin{array}{c} env_P, env_V \vdash \langle i_1, sto \rangle \rightarrow_i (sto'', (T_{i1}, v_{i1})) \\ env_P, env_V \vdash \langle i_2, sto'' \rangle \rightarrow_i (sto', (T_{i2}, v_{i2})) \end{array}}{env_P \vdash \langle \texttt{for}(T_{Int}\ x\ \texttt{from}\ i_1\ \texttt{to}\ i_2)\ \texttt{indent}\ s\ \texttt{dedent}, sto, env_V \rangle \rightarrow_s (sto', env_V, \epsilon)}$$
If $v_{i1} = v_{i2}$

$[ForStatement_3]$

$$\frac{\begin{array}{c} env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'''', (T_{i1}, v_{i1})) \\ env_P, env_V \vdash \langle i_2, sto''''\rangle \rightarrow_i (sto''', (T_{i2}, v_{i2})) \\ env_P \vdash \langle T_{Int}\ x = n_{i1}, sto''', env_V\rangle \rightarrow_s (sto'', env''_V, r_1) \\ env_P \vdash \langle s, sto'', env''_V\rangle \rightarrow_s (sto', env'_V, r_2) \end{array}}{env_P \vdash \langle \texttt{for}(T_{Int}\ x\ \texttt{from}\ i_1\ \texttt{to}\ i_2)\ \texttt{indent}\ s\ \texttt{dedent}, sto, env_V\rangle \rightarrow_s (sto', env_V, r_2)}$$

Where $n_{i1} = \mathcal{N}^{-1}[\![v_{i1}]\!]$

If $v_{i1} \neq v_{i2}$

And $r_2 \ \neq \ \epsilon$

And $T_{Int} = largestIntegerType(T_{Int}, T_{i2})$

$[IfStatement]$

$$\frac{env_P, env_V \vdash \langle s_{if}, sto\rangle \rightarrow_{s_{if}} (sto', r)}{env_P \vdash \langle s_{if}, sto, env_V\rangle \rightarrow_s (sto', env_V, r)}$$

Table C.9: The transition rules $\rightarrow_s$

The semantics of **IfStm** are given by the transition system $(\Gamma_{s_{if}}, \rightarrow_{s_{if}}, T_{s_{if}})$ whose configurations are defined by

$$\Gamma_{s_{if}} = (\mathbf{IfStm} \times \mathbf{Sto}) \cup (\mathbf{Sto} \times \mathbf{Ret})$$
$$T_{s_{if}} = (\mathbf{Sto} \times \mathbf{Ret})$$

Transitions here have the format $env_P, env_V \vdash \langle s_{if}, sto\rangle \rightarrow_{s_{if}} (sto', r)$

$$[IfStatement_1] \quad \frac{env_P, env_V \vdash \langle b, sto\rangle \rightarrow_b (sto', (T, v))}{env_P, env_V \vdash \langle \texttt{if}(b)\ \texttt{indent}\ s\ \texttt{dedent}, sto\rangle \rightarrow_{s_{if}} (sto', \epsilon)} \quad \text{If } v = \mathit{ff}$$

$$[IfStatement_2] \quad \frac{\begin{array}{c} env_P, env_V \vdash \langle b, sto\rangle \rightarrow_b (sto'', (T, v)) \\ env_P \vdash \langle s, sto'', env_V\rangle \rightarrow_s (sto', env'_V, r) \end{array}}{env_P, env_V \vdash \langle \texttt{if}(b)\ \texttt{indent}\ s\ \texttt{dedent}, sto\rangle \rightarrow_{s_{if}} (sto', r)} \quad \text{If } v = \mathit{tt}$$

$$[IfElseIf_1] \quad \frac{\begin{array}{c} env_P, env_V \vdash \langle b, sto\rangle \rightarrow_b (sto'', (T, v)) \\ env_P, env_V \vdash \langle s_{if}, sto''\rangle \rightarrow_{s_{if}} (sto', r) \end{array}}{env_P, env_V \vdash \langle \texttt{if}(b)\ \texttt{indent}\ s\ \texttt{dedent}\ \texttt{else}\ s_{if}, sto\rangle \rightarrow_{s_{if}} (sto', r)} \quad \text{If } v = \mathit{ff}$$

$$[IfElseIf_2] \quad \frac{\begin{array}{c} env_P, env_V \vdash \langle b, sto\rangle \rightarrow_b (sto'', (T, v)) \\ env_P \vdash \langle s, env_V, sto''\rangle \rightarrow_s (sto', env'_V, r) \end{array}}{env_P, env_V \vdash \langle \texttt{if}(b)\ \texttt{indent}\ s\ \texttt{dedent}\ \texttt{else}\ s_{if}, sto\rangle \rightarrow_{s_{if}} (sto', r)} \quad \text{If } v = \mathit{tt}$$

$$[IfElseStatement_1] \quad \frac{\begin{array}{c} env_P, env_V \vdash \langle b, sto\rangle \rightarrow_b (sto'', (T, v)) \\ env_P \vdash \langle s_2, sto'', env_V\rangle \rightarrow_s (sto', env'_V, r) \end{array}}{env_P, env_V \vdash \langle \texttt{if}(b)\ \texttt{indent}\ s_1\ \texttt{dedent}\ \texttt{else}\ \texttt{indent}\ s_2\ \texttt{dedent}, sto\rangle \rightarrow_{s_{if}} (sto', r)} \quad \text{If } v = \mathit{ff}$$

$$[IfElseStatement_2] \quad \frac{\begin{array}{c} env_P, env_V \vdash \langle b, sto\rangle \rightarrow_b (sto'', (T, v)) \\ env_P \vdash \langle s_1, sto'', env_V\rangle \rightarrow_s (sto', env'_V, r) \end{array}}{env_P, env_V \vdash \langle \texttt{if}(b)\ \texttt{indent}\ s_1\ \texttt{dedent}\ \texttt{else}\ \texttt{indent}\ s_2\ \texttt{dedent}, sto\rangle \rightarrow_{s_{if}} (sto', r)} \quad \text{If } v = \mathit{tt}$$

Table C.10: The transition rules $\rightarrow_{s_{if}}$

The semantics of **Exp** are given by the transition system $(\Gamma_e, \rightarrow_e, T_e)$ whose configurations are defined by

$$\Gamma_e = (\mathbf{Exp} \times \mathbf{Sto}) \cup (\mathbf{Sto} \times (\mathbf{Types} \times \mathbf{Val}))$$
$$T_e = (\mathbf{Sto} \times (\mathbf{Types} \times \mathbf{Val}))$$

Transitions here have the format $env_P, env_V \vdash \langle e, sto\rangle \rightarrow_e (sto', (T, v))$

$[VarEvaluation]$
$$env_P, env_V \vdash \langle x, sto\rangle \to_e (sto, (T, v))$$
Where $sto(env_V\ x) = (T, v)$

$[IntegerExpression]$
$$\frac{env_P, env_V \vdash \langle i, sto\rangle \to_i (sto', (T, v))}{env_P, env_V \vdash \langle i, sto\rangle \to_e (sto', (T, v))}$$

$[BooleanExpression]$
$$\frac{env_P, env_V \vdash \langle b, sto\rangle \to_b (sto', (T, v))}{env_P, env_V \vdash \langle b, sto\rangle \to_e (sto', (T, v))}$$

$[ParenthesisExpression]$
$$\frac{env_P, env_V \vdash \langle e_1, sto\rangle \to_e (sto', (T, v))}{env_P, env_V \vdash \langle (e_1), sto\rangle \to_e (sto', (T, v))}$$

$[ProcedureCall]$
$$\frac{\begin{array}{c} env_P, env_V \vdash \langle e_1, sto\rangle \to_e (sto^{(1)}, (T_1, v_1)) \\ env_P, env_V \vdash \langle e_2, sto^{(1)}\rangle \to_e (sto^{(2)}, (T_2, v_2)) \\ \dots \\ env_P, env_V \vdash \langle e_k, sto^{(k-1)}\rangle \to_e (sto^{(k)}, (T_k, v_k)) \\ env_P \vdash \langle s, sto^{(k)}[l_1 \mapsto v_1][l_2 \mapsto v_2][\dots][l_k \mapsto v_k], \\ env'_V[x_1 \mapsto l_1][x_2 \mapsto l_2][\dots][x_k \mapsto l_k][next \mapsto new(l_k)]\rangle \to_s (sto', env''_V, (T_r, r)) \end{array}}{env_P, env_V \vdash \langle p(e_1,\ e_2,\ \dots\ e_k), sto\rangle \to_e (sto', (T_r, r))}$$
Where $env_P\ p = (env'_V, s, (x_1,\ x_2,\ \dots\ x_k))$
And $l_1 = env'_V(next)$
And $l_2 = new(env'_V(next))$
And $l_3 = new(new(env'_V(next)))$
$\dots$
And $l_k = new^{k-1}(env'_V(next))$

Table C.11: The transition rules $\to_e$

The semantics of **IntExp** are given by the transition system $(\Gamma_i, \to_i, T_i)$ whose configurations are defined by
$$\Gamma_i = (\mathbf{IntExp} \times \mathbf{Sto}) \cup (\mathbf{Sto} \times (\mathbf{Types} \times \mathbf{Val}))$$
$$T_i = (\mathbf{Sto} \times (\mathbf{Types} \times \mathbf{Val}))$$

Transitions here have the format $env_P, env_V \vdash \langle i, sto\rangle \to_i (sto', (T, v))$

$[ConstantToInt]$
$$env_P, env_V \vdash \langle n, sto\rangle \to_i (sto, (T, v))$$
Where $\mathcal{N}[\![n]\!] = v$
And $T = \begin{cases} int8 & \text{If } v \in Set(int8) \\ int16 & \text{If } v \in Set(int16) \text{ and } v \notin Set(int8) \\ int32 & \text{If } v \in Set(int32) \text{ and } v \notin Set(int16) \end{cases}$
If $v \in Set(int32)$

$[ExpressionToInt]$
$$\frac{env_P, env_V \vdash \langle e, sto\rangle \to_e (sto', (T, v))}{env_P, env_V \vdash \langle e, sto\rangle \to_i (sto', (T, v))}$$

$$[IntegerPlus] \quad \frac{env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'', (T_1, v_1)) \quad env_P, env_V \vdash \langle i_2, sto''\rangle \rightarrow_i (sto', (T_2, v_2))}{env_P, env_V \vdash \langle i_1 + i_2, sto\rangle \rightarrow_i (sto', (T_3, v_3))}$$
Where $v_3 = v_1 + v_2$
And $T_3 = largestIntegerType(T_1, T_2)$
If $v_3 \in Set(T_3)$

$$[IntegerMinus] \quad \frac{env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'', (T_1, v_1)) \quad env_P, env_V \vdash \langle i_2, sto''\rangle \rightarrow_i (sto', (T_2, v_2))}{env_P, env_V \vdash \langle i_1 - i_2, sto\rangle \rightarrow_i (sto', (T_3, v_3))}$$
Where $v_3 = v_1 - v_2$
And $T_3 = largestIntegerType(T_1, T_2)$
If $v_3 \in Set(T_3)$

$$[IntegerMultiplication] \quad \frac{env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'', (T_1, v_1)) \quad env_P, env_V \vdash \langle i_2, sto''\rangle \rightarrow_i (sto', (T_2, v_2))}{env_P, env_V \vdash \langle i_1 * i_2, sto\rangle \rightarrow_i (sto', (T_3, v_3))}$$
Where $v_3 = v_1 \cdot v_2$
And $T_3 = largestIntegerType(T_1, T_2)$
If $v_3 \in Set(T_3)$

$$[IntegerDivision] \quad \frac{env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'', (T_1, v_1)) \quad env_P, env_V \vdash \langle i_2, sto''\rangle \rightarrow_i (sto', (T_2, v_2))}{env_P, env_V \vdash \langle i_1 / i_2, sto\rangle \rightarrow_i (sto', (T_3, v_3))}$$
Where $v_3 = \lfloor v_1 / v_2 \rfloor$
And $T_3 = largestIntegerType(T_1, T_2)$
If $v_2 \neq 0$
And $v_3 \in Set(T_3)$

$$[IntegerModulus] \quad \frac{env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'', (T_1, v_1)) \quad env_P, env_V \vdash \langle i_2, sto''\rangle \rightarrow_i (sto', (T_2, v_2))}{env_P, env_V \vdash \langle i_1 \ \texttt{\%}\ i_2, sto\rangle \rightarrow_i (sto', (T_3, v_3))}$$
Where $v_3 = v_1 \ mod \ v_2$
And $T_3 = largestIntegerType(T_1, T_2)$
If $v_3 \in Set(T_3)$

$$[IntegerPower] \quad \frac{env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'', (T_1, v_1)) \quad env_P, env_V \vdash \langle i_2, sto''\rangle \rightarrow_i (sto', (T_2, v_2))}{env_P, env_V \vdash \langle i_1 \ \hat{}\ i_2, sto\rangle \rightarrow_i (sto', (T_3, v_3))}$$
Where $v_3 = v_1^{(v_2)}$
And $T_3 = largestIntegerType(T_1, T_2)$
If $v_3 \in Set(T_3)$
And $v_2 \geq 0$

Table C.13: The transition rules $\rightarrow_i$

The semantics of **BoolExp** are test hest given by the transition system $(\Gamma_b, \rightarrow_b, T_b)$ whose configurations are defined by

$$\Gamma_i = (\mathbf{BoolExp} \times \mathbf{Sto}) \cup (\mathbf{Sto} \times (\mathbf{Types} \times \mathbf{Val}))$$
$$T_i = \mathbf{Sto} \times (\mathbf{Types} \times \mathbf{Val})$$

Transitions here have the format $env_P, env_V \vdash \langle b, sto\rangle \rightarrow_b (sto', (T, v))$

$[TrueConstant]$ $\quad env_P, env_V \vdash \langle \texttt{true}, sto\rangle \rightarrow_b (sto, (\texttt{bool}, tt))$

$[FalseConstant]$ $\quad env_P, env_V \vdash \langle \texttt{false}, sto\rangle \rightarrow_b (sto, (\texttt{bool}, ff))$

$[ExpressionToBoolean]$
$$\frac{env_P, env_V \vdash \langle e, sto\rangle \rightarrow_e (sto', (T, v))}{env_P, env_V \vdash \langle e, sto\rangle \rightarrow_b (sto', (T, v))}$$

$[Equals]$
$$\frac{\begin{array}{c} env_P, env_V \vdash \langle e_1, sto\rangle \rightarrow_e (sto'', (T_1, v_1)) \\ env_P, env_V \vdash \langle e_2, sto''\rangle \rightarrow_e (sto', (T_2, v_2)) \end{array}}{env_P, env_V \vdash \langle e_1 \texttt{ == } e_2, sto\rangle \rightarrow_b (sto', (\texttt{bool}, v_3))}$$
$$\text{Where } v_3 = \begin{cases} tt & \text{if } v_1 = v_2 \\ ff & \text{if } v_1 \neq v_2 \end{cases}$$

$[NotEquals]$
$$\frac{\begin{array}{c} env_P, env_V \vdash \langle e_1, sto\rangle \rightarrow_e (sto'', (T_1, v_1)) \\ env_P, env_V \vdash \langle e_2, sto''\rangle \rightarrow_e (sto', (T_2, v_2)) \end{array}}{env_P, env_V \vdash \langle e_1 \texttt{ != } e_2, sto\rangle \rightarrow_b (sto', (\texttt{bool}, v_3))}$$
$$\text{Where } v_3 = \begin{cases} tt & \text{if } v_1 \neq v_2 \\ ff & \text{if } v_1 = v_2 \end{cases}$$

$[IntegerLessThan]$
$$\frac{\begin{array}{c} env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'', (T_1, v_1)) \\ env_P, env_V \vdash \langle i_2, sto''\rangle \rightarrow_i (sto', (T_2, v_2)) \end{array}}{env_P, env_V \vdash \langle i_1 < i_2, sto\rangle \rightarrow_b (sto', (\texttt{bool}, v_3))}$$
$$\text{Where } v_3 = \begin{cases} tt & \text{if } v_1 < v_2 \\ ff & \text{if } v_1 \geq v_2 \end{cases}$$

$[IntegerLessThanOrEquals]$
$$\frac{\begin{array}{c} env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'', (T_1, v_1)) \\ env_P, env_V \vdash \langle i_2, sto''\rangle \rightarrow_i (sto', (T_2, v_2)) \end{array}}{env_P, env_V \vdash \langle i_1 \leq i_2, sto\rangle \rightarrow_b (sto', (\texttt{bool}, v_3))}$$
$$\text{Where } v_3 = \begin{cases} tt & \text{if } v_1 \leq v_2 \\ ff & \text{if } v_1 > v_2 \end{cases}$$

$$[IntegerGreaterThanOrEquals] \quad \frac{env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'', (T_1, v_1)) \quad env_P, env_V \vdash \langle i_2, sto''\rangle \rightarrow_i (sto', (T_2, v_2))}{env_P, env_V \vdash \langle i_1 \geq i_2, sto\rangle \rightarrow_b (sto', (\texttt{bool}, v_3))}$$

Where $v_3 = \begin{cases} tt & \text{if } v_1 \geq v_2 \\ ff & \text{if } v_1 < v_2 \end{cases}$

$$[IntegerGreaterThanTrue] \quad \frac{env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'', (T_1, v_1)) \quad env_P, env_V \vdash \langle i_2, sto''\rangle \rightarrow_i (sto', (T_2, v_2))}{env_P, env_V \vdash \langle i_1 > i_2, sto\rangle \rightarrow_b (sto', (\texttt{bool}, v_3))}$$

Where $v_3 = \begin{cases} tt & \text{if } v_1 > v_2 \\ ff & \text{if } v_1 \leq v_2 \end{cases}$

$$[NotOperator] \quad \frac{env_P, env_V \vdash \langle b_1, sto\rangle \rightarrow_b (sto', (T_1, v_1))}{env_P, env_V \vdash \langle \texttt{!}\, b_1, sto\rangle \rightarrow_b (sto', (\texttt{bool}, v_2))}$$

Where $v_2 = \neg v_1$

$$[BooleanAnd] \quad \frac{env_P, env_V \vdash \langle b_1, sto\rangle \rightarrow_b (sto'', (T_1, v_1)) \quad env_P, env_V \vdash \langle b_2, sto''\rangle \rightarrow_b (sto', (T_2, v_2))}{env_P, env_V \vdash \langle b_1 \texttt{ and } b_2, sto\rangle \rightarrow_b (sto', (\texttt{bool}, v_3))}$$

Where $v_3 = v_1 \wedge v_2$

$$[BooleanOr] \quad \frac{env_P, env_V \vdash \langle b_1, sto\rangle \rightarrow_b (sto'', (T_1, v_1)) \quad env_P, env_V \vdash \langle b_2, sto''\rangle \rightarrow_b (sto', (T_2, v_2))}{env_P, env_V \vdash \langle b_1 \texttt{ or } b_2, sto\rangle \rightarrow_b (sto', (\texttt{bool}, v_3))}$$

Where $v_3 = v_1 \vee v_2$

$$[RegisterBitToBool] \quad \frac{env_P, env_V \vdash \langle i, sto\rangle \rightarrow_i (sto', (T_i, v_i))}{env_P, env_V \vdash \langle x\{i\}, sto\rangle \rightarrow_b (sto', (\texttt{bool}, v))}$$

Where $(T_{Reg}, l) = sto(env_V\ x)$

And $(T_r, v_r) = sto(l)$

And $v = \begin{cases} tt & \text{if } \left\lfloor \frac{v_r}{2^{(v_i)}} \right\rfloor \ mod\ 2 = 1 \\ \\ ff & \text{if } \left\lfloor \frac{v_r}{2^{(v_i)}} \right\rfloor \ mod\ 2 = 0 \end{cases}$

$$[RegisterLiteralBitToBool] \quad \frac{env_P, env_V \vdash \langle i_1, sto\rangle \rightarrow_i (sto'', (T_{i1}, v_{i1})) \quad env_P, env_V \vdash \langle i_2, sto''\rangle \rightarrow_i (sto', (T_{i2}, v_{i2}))}{env_P, env_V \vdash \langle T_{Reg}(i_1)\{i_2\}, sto\rangle \rightarrow_b (sto', (\texttt{bool}, v))}$$

If $v_{i1} \geq 0$

If $v_{i2} \geq 0$ And $v_{i2} < bitlen(T_{Reg})$

Where $(T_r, v_r) = sto(v_{i1})$

$$\text{And } v = \begin{cases} \mathit{tt} & \text{if } \left\lfloor \frac{v_r}{2^{(v_{i2})}} \right\rfloor \ mod\ 2 = 1 \\ \\ \mathit{ff} & \text{if } \left\lfloor \frac{v_r}{2^{(v_{i2})}} \right\rfloor \ mod\ 2 = 0 \end{cases}$$

Table C.15: The transition system $\rightarrow_b$

Appendix D

Type System

To map variable and procedure names to their corresponding type we define a partial function, the type environment, E as follows:

$$E : (\mathbf{VNames} \cup \mathbf{PNames}) \rightharpoonup \mathbf{Types}^k$$

An update of the type environment is written as follows $E[x \mapsto (T_1, T_2, \ldots T_k)]$ where the new Type environment E' is:

$$E'(y) = \begin{cases} E(y) & \text{if } y \neq x \\ (T_1, T_2, \ldots T_k) & \text{if } y = x \end{cases}$$

When defining new variables we check if the variable-name is already defined in the type environment, before updating the type-environment. We do this by returning `undefined` if the variable is not yet defined in the type environment and making a check in each specific case.

$$E^{\varnothing}(y) = (\texttt{undefined})$$

Programs

Type rules for program statements in Tang has the following type judgement: $E \vdash P : ok$
Table D.1 shows the type rules for program statements.

$[Program]$	$E \vdash S : \texttt{ok}$	Where $E = E_{Scan}(S, E^{\varnothing}[scope \mapsto \texttt{nothing}])$
$[Empty]$	$E^{\varnothing}[scope \mapsto \texttt{nothing}] \vdash \varepsilon : \texttt{ok}$	

Table D.1: Type rules for Programs

Statements

Type rules for root level statements in Tang has the following type judgement: $E \vdash S : \texttt{ok}$
The type rules for statements can be seen in table D.2.

$[CompoundStatement]$	$\dfrac{E \vdash S_1 : \mathsf{ok} \quad E_1 \vdash S_2 : \mathsf{ok}}{E \vdash S_1 \ \mathtt{newline}\ S_2 : \mathsf{ok}}$	**Where** $E_1 = E(S_1, E)$
$[ProcedureDeclaration]$	$\dfrac{\begin{array}{c} E \vdash T_1\ x_1 : \mathsf{ok} \\ E_1 \vdash T_2\ x_2 : \mathsf{ok} \\ E_2 \vdash T_3\ x_3 : \mathsf{ok} \\ \dots \\ E_{k-1} \vdash T_k\ x_k : \mathsf{ok} \\ E_k[scope \mapsto (T)] \vdash s : \mathsf{ok} \end{array}}{E \vdash T\ p(T_1\ x_1,\ T_2\ x_2,\ \dots\ T_k\ x_k)\ \mathtt{indent}\ s\ \mathtt{dedent} : \mathsf{ok}}$	If $E(p) \neq (\mathsf{undefined})$ Where $E_1 = E(T_1\ x_1, E)$ and $E_2 = E(T_2\ x_2, E_2)$ ... and $E_k = E(T_k\ x_k, E_k)$
$[InteruptStatement]$	$\dfrac{E \vdash n : T_{Int} \quad E[scope \mapsto (\mathsf{nothing})] \vdash s : \mathsf{ok}}{E \vdash \mathtt{interrupt}(n)\ \mathtt{indent}\ s\ \mathtt{dedent} : \mathsf{ok}}$	

Table D.2: Type rules for Root level statements

Scoped level statements

Type rules for scoped level statements in Tang has the following type judgement: $E \vdash s : \mathsf{ok}$
Table D shows the type rules for the scoped level statements in Tang. The auxiliary function $[VarAux]$ in table D.9 states that x is bound to the type of T.

$[Newline]$	$E \vdash \mathtt{newline} : \mathsf{ok}$
$[CompoundSimpleStatement]$	$\dfrac{E \vdash s_1 : \mathsf{ok} \quad E_1 \vdash s_2 : \mathsf{ok}}{E \vdash s_1 \ \mathtt{newline}\ s_2 : \mathsf{ok}}$ Where $E_1 = E(s_1, E)$
$[VariableDeclaration]$	$E \vdash T\ x : \mathsf{ok}$ If $E(x) = (\mathsf{undefined})$ and $T \neq \mathtt{nothing}$
$[VariableDefinition]$	$\dfrac{E \vdash e : T_1}{E \vdash T\ x = e : \mathsf{ok}}$ If $E(x) = (\mathsf{undefined})$ if $T \neq \mathtt{nothing}$ if $T = T_1$ or $T \in$ **IntTypes** and $T_1 \in$ **IntTypes** and $T = largestIntegerType(T, T_1)$
$[Assignment]$	$\dfrac{E \vdash x : T \quad E \vdash e : T_1}{E \vdash x = e : \mathsf{ok}}$ If $T = T_1$ or $T \in$ **IntTypes** and $T_1 \in$ **IntTypes** and $T = largestIntegerType(T, T_1)$
$[RegisterBitAssigment]$	$\dfrac{E \vdash x : T_{Reg} \quad E \vdash i : T_{Int} \quad E \vdash b : bool}{E \vdash x\{i\} = b : \mathsf{ok}}$

$[RegisterLiteralAssignment]$	$\frac{E \vdash i_1 : T_{Int} \quad E \vdash i_2 : T_{Int} \quad E \vdash b : bool}{E \vdash T_{Reg}(i_1)\{i_2\} = b : \mathsf{ok}}$
$[Return]$	$E \vdash \texttt{return} : \mathsf{ok}$ If $E(scope) = (\texttt{nothing})$
$[ReturnWithValue]$	$\frac{E \vdash e : T}{E \vdash \texttt{return}\ e : \mathsf{ok}}$ Where $(T_{scope}) = E(scope)$ If $T_{scope} = (T)$ or $T_{scope} \in \textbf{IntTypes}$ and $T \in \textbf{IntTypes}$ and $T_{scope} = largestIntegerType(T, T_{scope})$
$[WhileStatement]$	$\frac{E \vdash b : bool \quad E \vdash\ s : \mathsf{ok}}{E \vdash \texttt{while}(b)\ \texttt{indent}\ s\ \texttt{dedent} : \mathsf{ok}}$
$[ForStatement]$	$\frac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2} \quad E[x \mapsto (T_{Int_3})] \vdash\ s : \mathsf{ok}}{E \vdash \texttt{for}(T_{Int}\ x\ \texttt{from}\ i_1\ \texttt{to}\ i_2)\ \texttt{indent}\ s\ \texttt{dedent} : \mathsf{ok}}$ Where $T_{Int_3} = largestIntegerType(T_{Int_1}, T_{Int_2})$ If $E(x) = (\texttt{undefined})$

Table D.4: Type rules for scoped level statements

IfStatements

Type rules for if statements in Tang has the following type judgement: $E \vdash s_{if} : ok$
In table D.5 the type rules for if statements are shown.

$[IfStatement]$	$\frac{E \vdash b : bool \quad E \vdash s : \mathsf{ok}}{E \vdash \texttt{if}(b)\ \texttt{indent}\ s\ \texttt{dedent} : \mathsf{ok}}$
$[IfElseIf]$	$\frac{E \vdash b : bool \quad E \vdash s : \mathsf{ok} \quad E \vdash s_{if} : \mathsf{ok}}{E \vdash \texttt{if}(b)\ \texttt{indent}\ s\ \texttt{dedent}\ \texttt{else}\ s_{if} : \mathsf{ok}}$
$[IfElseStatement]$	$\frac{E \vdash b : bool \quad E \vdash s_1 : \mathsf{ok} \quad E \vdash s_2 : \mathsf{ok}}{E \vdash \texttt{if}(b)\ \texttt{indent}\ s_1\ \texttt{dedent}\ \texttt{else}\ \texttt{indent}\ s_2\ \texttt{dedent} : \mathsf{ok}}$

Table D.5: Type rules for if statements

Expression

Type rules for expressions in Tang has the following type judgement: $E \vdash e : T$
The type rules for expressions are shown in table D.6.

$[LeftValueEvaluation]$	$E \vdash x : T$ Where $(T) = E(x)$ and $T \neq$ `undefined`
$[ParenthesisExpression]$	$\dfrac{E \vdash e : T}{E \vdash (e) : T}$
$[ProcedureCallExpression]$	$\dfrac{\begin{array}{c} E \vdash e_1 : T_1 \\ E \vdash e_2 : T_2 \\ \cdots \\ E \vdash e_k : T_k \end{array}}{E \vdash p(e_1,\ e_2,\ \ldots\ e_k) : T_r}$ Where $(T_r,\ T_1,\ T_2,\ \ldots\ T_k) = E(p)$
$[RegisterLiteralExpression]$	$\dfrac{E \vdash n : T_{Int}}{E \vdash T_{Reg}(n) : T_{Reg}}$

Table D.6: Type rules for expressions

Integer expressions

Type rules for integer expression in Tang has the following type judgement: $E \vdash i : T_{Int}$
The different type rules for integer expressions are shown in table D.7.

$[Numeral]$ $\quad E \vdash n : T_{Int}$

Where $v = \mathcal{N}[\![n]\!]$

And $T_{Int} = \begin{cases} int8 & \text{if } v \in Set(int8) \\ int16 & \text{if } v \in Set(int16) \text{ and } v \notin Set(int8) \\ int32 & \text{if } v \in Set(int32) \text{ and } v \notin Set(int16) \end{cases}$

$[AddOperation]$ $\quad \dfrac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash i_1 \ + \ i_2 : T_{Int}}$

Where $T_{Int} = largestIntegerType(T_{Int_1}, T_{Int_2})$

$[SubtractOperation]$ $\quad \dfrac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash i_1 \ - \ i_2 : T_{Int}}$

Where $T_{Int} = largestIntegerType(T_{Int_1}, T_{Int_2})$

$[MultiplicityOperation]$ $\quad \dfrac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash i_1 \ * \ i_2 : T_{Int}}$

Where $T_{Int} = largestIntegerType(T_{Int_1}, T_{Int_2})$

$[DivisionOperation]$ $\quad \dfrac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash i_1 \ / \ i_2 : T_{Int}}$

Where $T_{Int} = largestIntegerType(T_{Int_1}, T_{Int_2})$

$[ModuloOperation]$ $\quad \dfrac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash i_1 \ \texttt{\%} \ i_2 : T_{Int}}$

Where $T_{Int} = largestIntegerType(T_{Int_1}, T_{Int_2})$

$[PowerOperation]$ $\quad \dfrac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash i_1 \hat{} \ i_2 : T_{Int}}$

Where $T_{Int} = largestIntegerType(T_{Int_1}, T_{Int_2})$

Table D.7: Type rules for integer expressions

Boolean expressions

Type rules for boolean expressions in Tang has the following type judgement: $E \vdash b : bool$
The type rules for boolean expressions are shown in table D.8.

$[True]$	$E \vdash \texttt{true} : bool$
$[False]$	$E \vdash \texttt{false} : bool$
$[EqualInt]$	$\dfrac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash i_1 \ == \ i_2 : bool}$
$[EqualBool]$	$\dfrac{E \vdash b_1 : bool \quad E \vdash b_2 : bool}{E \vdash b_1 \ == \ b_2 : bool}$
$[LessThan]$	$\dfrac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash i_1 \ < \ i_2 : bool}$
$[GreaterThan]$	$\dfrac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash i_1 \ > \ i_2 : bool}$
$[LessThanEqual]$	$\dfrac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash i_1 \ <= \ i_2 : bool}$
$[GreaterThanEqual]$	$\dfrac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash i_1 \ >= \ i_2 : bool}$
$[Not]$	$\dfrac{E \vdash b : bool}{E \vdash \texttt{!}b : bool}$
$[LogicalAnd]$	$\dfrac{E \vdash b_1 : bool \quad E \vdash b_2 : bool}{E \vdash b_1 \ \texttt{and} \ b_2 : bool}$
$[LogicalOr]$	$\dfrac{E \vdash b_1 : bool \quad E \vdash b_2 : bool}{E \vdash b_1 \ \texttt{or} \ b_2 : bool}$
$[RegisterGetBit]$	$\dfrac{E \vdash x : T_{Reg} \quad E \vdash i : T_{Int_2}}{E \vdash x\{i\} : bool}$
$[RegisterLiteralGetBit]$	$\dfrac{E \vdash i_1 : T_{Int_1} \quad E \vdash i_2 : T_{Int_2}}{E \vdash T_{Reg}(i_1)\{i_2\} : bool}$

Table D.8: Type rules for boolean expressions

Auxilary functions

$[CompoundAux_1]$	$E(S_1 \ \texttt{newline} \ S_2, E) = E(S_2, E(S_1, E))$
$[CompoundAux_2]$	$E(s_1 \ \texttt{newline} \ s_2, E) = E(s_2, E(s_1, E))$
$[VarAux]$	$E(T \ x, E) = E[x \mapsto (T)]$
$[VarInitAux]$	$E(T \ x = e, E) = E[x \mapsto (T)]$

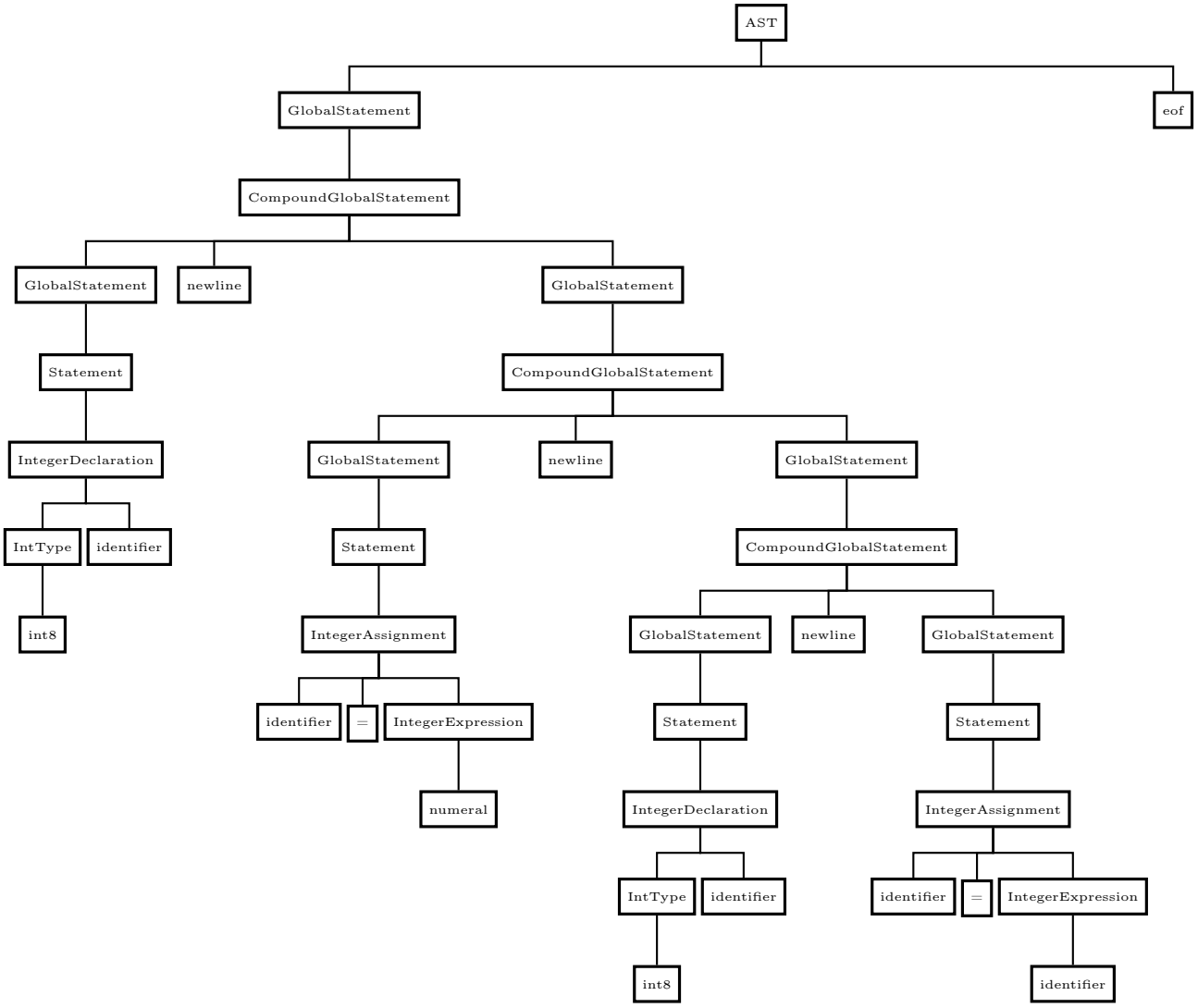
Table D.9: Auxiliary functions for updating the type environment

$[CompoundAuxScan_1]$	$E_{Scan}(S_1 \texttt{ newline } S_2, E) = E_{Scan}(S_2, E_{Scan}(S_1, E))$
$[CompoundAuxScan_2]$	$E_{Scan}(s_1 \texttt{ newline } s_2, E) = E_{Scan}(s_2, E_{Scan}(s_1, E))$
$[ProcAuxScan]$	$E_{Scan}(T_r\ p(T_1\ x_1,\ T_2\ x_2,\ \ldots\ T_k\ x_k) \texttt{ indent } s \texttt{ dedent}, E) = E[p \mapsto (T_r,\ T_1,\ T_2,\ \ldots\ T_k)]$ If $E(p) = (\texttt{undefined})$

Table D.10: Auxiliary functions for updating the type environment

Appendix E

AST of Alias.tang



Appendix F

Language Documentation

The purpose of this section is to give a brief documentation of the usage of Tang.

Overview of Tang

The main purpose of Tang, is to bridge the gap for programmers, wanting to learn embedded programming. As of now, getting into programming embedded systems, i.e. Arduino, can be tedious, and involves learning a large number of new constructs. What Tang tries to accomplish, is to identify these constructs, and simplify them. An example of this, is working with bitwise operators. Other structural changes are made, such as removing brackets as a form of structuring the scope of program. Instead, indentation serves this purpose, and enforces a cleaner, more readable code in general.

Tang tutorial

In this tutorial, it is assumed, that the target embedded system used is an Arduino Nano.
To begin learning Tang, one needs to understand, how a program is executed. Tang works in a top-down fashion, which means that any chip running a Tang program starts at the top of the source code, and moves downwards until the end of the file. This means that there is no need to define a main function to run a program. A result of this is that short programs are simple to write, as not much structure is needed to make a functional program.
As Tang is designed to be used with embedded programming, there usually is no console to print text. Therefore, the *Blink* example will be used instead of the famous $Hello\ World$ example. The *Blink* example, is a simple program, designed to make the on-board diode on an embedded board blink, where in this tutorial, it is the diode on an Arduino board.
An important aspect of working with embedded systems, is knowing how to manipulate the different ports and pins on the board. To access ports on the board, they first have to be initialised as a variable. In Tang, this is done by assigning a specific port to the *register* type. The number of a given port, varies from board to board and it is therefore important to look at the data sheet for the specific board used. In this example, the name of the port we want to use, is $PORTB$, where the LED we want to use is located on pin 5. On the data sheet, seen here [20], the reference number given is hexadecimal $0x24$, which is 36 in decimal. In our blink example, the instantiating of the variables looks like this:

```
1    register8 ddrb = register8(36)
2    register8 port = register8(37)
```

Listing F.1: Assigning a register type in Tang

The reason for using two registers, namely *ddrb* and *port*, is that, *ddrb* serves the purpose of defining our port as an output port, whereas *port* is used for the actual manipulation of the pins on that specific port. Now that we have our register types, we need to define which pin on the port we want to work with. As mentioned earlier, the pin we want to use is pin 5, and we begin by setting it to an output pin. This is done by setting *ddrb*5 to true, meaning that the fifth bit of the ddrb register is flipped to true. Next we want the LED turned on at the beginning of the program. This is done by setting *port*5 to true as well. To keep track of how long the LED has been on or off, an integer variable has to be declared. In Tang, integers are defined with the keyword *int*, followed by the bit length of the integer, which can be either 8, 16, or 32. In this example, a bit length of 8 is sufficient, and the integer can be defined as unsigned, since we have no need for negative numbers. That means the keyword we want to use here is *int*8. The reason we need an integer, is to keep track of how the time since last blink. An example of both the pin and integer declarations can be seen below:

```
1    ddrb{5} = true
2    port{5} = true
3    int8 counter = 0
```

Listing F.2: Pin and integer declarations in Tang

Now that we have declared the necessary variables, its time to define the loop, which switches the LED on and off. Here it is optimal to use a *while* loop, which is done by writing the keyword *while*, followed by a boolean expression in parenthesis. To make the loop run forever, use the boolean expression *true*. The code you want the loop to run, needs to be indented, in order for the program to identify it. In the loop, we want to increment the integer, until it reaches a certain point, where we want to switch the state of the LED, and reset the counter. This can be done by a simple *if* statement. The entire loop is shown below:

```
1    while(true)
2        if(counter == 100000)
3            counter = 0
4            port{5} = !port{5}
5        else
6            counter = counter + 1
```

Listing F.3: An example of a loop, and a conditional statement in Tang

As seen in the example, the counter keeps running the else statement, where the integer is increased, until it reaches 100000. This number is not chosen for any other reason, than that it corresponds to the LED switching on and off roughly every second. When the counter reaches 100000, the counter is set to 0, and *port*5 flips value. Since *port*5 is a bit, therefore a boolean value, it is switched between its on and off state. As the counter is now 0, the loop start over by increasing it by one every cycle, until it reaches 100000 again. The full *Blink* example is shown below:

```
    register8 ddrb = register8(36)
    register8 port = register8(37)
    ddrb{5} = true
    port{5} = true
    int8 counter = 0
    while(true)
        if(counter == 100000)
            counter = 0
            port{5} = !port{5}
        else
            counter = counter + 1
```

Listing F.4: The entire blink example in Tang

Cheat Sheet

A cheat sheet is a way to give a short an concise representation of a language. A cheat sheet is therefore created for the language Tang, showing the types, operators, control structures, and precedence of the language. To give context for the cheat sheet, an example is shown using the cheat sheet in practice.

Table F.1: Tang Operators, their associativity and precedence.

Precedence	Operator (Associativity)
1	() (Left to right)
	! (Right to left)
2	$\wedge$ (Left to right)
3	/ (Left to right)
	% (Left to right)
4	+ (Left to right)
	- (Left to right)
5	> (Left to right)
	< (Left to right
	>= (Left to right)
	<= (Left to right)
6	= (Right to left)
	!= (Left to right)
7	and (Left to right)
8	or (Left to right)

Table F.2: Tang Types and control statements.

Type	Value		Flow Control
int8	$-2^7\texttt{to}2^7-1$		for(Statements)
int16	$-2^15\texttt{to}2^15-1$		while(boolean)
int32	$-2^31\texttt{to}2^31-1$		if(boolean)
bool	{true, false }		else if(boolean)
register8	$0\texttt{to}2^16-1$		else
register16	$0\texttt{to}2^16-1$		functionName(parameters)
nothing			

Code example showing constructs in Tang

An example in Tang, using the constructs from the the cheat sheet F.2, F.1.

```
// Declaring variables
int8 i
i = 5
int16 a = 1025
bool b
b = false
bool b = true


// Using registers

register8 DDRB = register(36)
DDRB{3} = false
register16 r16 = register(12)

// This is a function, with the return type int8 and two formal
   ↪ parameters of type int8
int8 foo(int8 i, int8 i2)
    int8 r = i / i2
    return r

// calling foo

// using conditional statements
int8 i1 = 10
int8 i2 = 5
int8 r
[...]
```

Listing F.5: An example of Tang

```
[...]
if(i2 != 0)
    r = foo(i1, i2)
else
    r = 0

// using while loop
r = 0
while(r < 20)
    r = r + 1

// using logical operators
bool b = true or false

// using recursion

int32 factorial(int32 a)
    if (a == 0)
        return 1
    return a * factorial(a-1)
```

Listing F.6: An example of Tang

Appendix G

Parser Generator Snippets

This appendix contains snippets used to explain the implementation of the parser generator in section 8.3.1. The entire source code can be seen in the electronic appendix as explained in the preamble.

```
[...]
MethodType parseTerminalMethod = new MethodType("public",
    ↪ \$"{dataNamespace}.Token", "ParseTerminal")
{
    Parameters = new List<ParameterType>()
    {
        new ParameterType(\$"IEnumerator<{dataNamespace}.Token>",
            ↪ "tokens"),
        new ParameterType("string", "expected")
    },
    Body = new List<string>()
    {
        "if(expected == \"EPSILON\")",
        "{",
        \$"    return new {dataNamespace}.Token() {{ Name =
            ↪ \"EPSILON\" }};",
        "}",
        \$"{dataNamespace}.Token token = tokens.Current;",
        "if(token.Name == expected)",
        "{",
        "    tokens.MoveNext();",
        "    return token;",
        "}",
        "else",
        "{",
        "    throw new Exception();",
        "}"
    }
};

[...]
```

Listing G.1: Shows the code for generating the `ParseTerminal` method

```
[...]
foreach (var production in bnf)
{
    MethodType parseMethod = new MethodType("public",
        ↪ \$"{dataNamespace}.{production.Key}",
        ↪ \$"Parse{production.Key}")
    {
        Parameters = new List<ParameterType>()
        {
            new ParameterType(\$"IEnumerator<{dataNamespace}.Token>",
                ↪ "tokens")
        }
    };

```

```
12      List<string> methodStatements = new List<string>();
13
14      methodStatements.Add(\$"{dataNamespace}.{production.Key} node =
            ↪ new {dataNamespace}.{production.Key}(){{ Name =
            ↪ \"{production.Key}\" }};");
15
16      methodStatements.Add("switch(tokens.Current.Name)");
17      methodStatements.Add("{");
18
19      for(int expansionIndex = 0; expansionIndex <
            ↪ production.Value.Count; expansionIndex++)
20      {
21          foreach (var predictSymbol in
                ↪ grammerInfo.PredictsSets[(production.Key,
                ↪ expansionIndex)])
22          {
23              methodStatements.Add(\$"    case \"{predictSymbol}\":");
24          }
25
26          foreach (var expansionSymbol in
                ↪ bnf[production.Key][expansionIndex])
27          {
28              if(IsTerminal(expansionSymbol, bnf))
29              {
30                  methodStatements.Add(\$"node.Add(ParseTerminal(tokens,
                        ↪ \"{expansionSymbol}\"));");
31              }
32              else
33              {
34                  methodStatements.Add(\$"node.Add(Parse{expansionSymbol}(tokens));");
35              }
36          }
37
38          methodStatements.Add(\$"return node;");
39
40      }
41
42      methodStatements.Add(\$"default:");
43      methodStatements.Add(\$"throw new Exception();");
44
45      methodStatements.Add("}");
46
47      parseMethod.Body = methodStatements;
48
49      parserClass.Methods.Add(parseMethod);
50  }
51  [. . .]
```

Listing G.2: Shows the code for generating methods for parsing the non terminals in the BNF for Tang

Appendix H

Translator Syntax

```
Translator -> Systems Rules eof

Systems -> System newline Systems
         | EPSILON

System -> goto Alias := Domain goto Domain
        | <=> Alias := Domain <=> Domain
        | </> Alias := Domain </> Domain

Domain -> ListDomain
        | TreeDomain

ListDomain -> [ Domains ]

Domains -> TreeDomain Domains
         | EPSILON

TreeDomain -> Symbol

Symbol -> symbol
        | escapedSymbol

Rules -> Rule RulesP
RulesP -> newline Rule RulesP
        | EPSILON

Rule -> Conclusion Premises
      | EPSILON

Conclusion -> Pattern NewlineGoto Alias Structure

NewlineGoto -> newline goto
             | goto

Premises -> indent Premis PremisesP dedent
```

```
          | EPSILON
PremisesP -> newline Premis PremisesP
           | EPSILON

Premis -> Structure StructureOperation
        | EPSILON

StructureOperation -> Goto
                    | Equal
                    | NotEqual

Goto -> goto Alias Pattern

Equal -> <=> Alias Structure

NotEqual -> </> Alias Structure

Pattern -> ListPattern
         | TreePattern

ListPattern -> [ Patterns ]

TreePattern -> Name Alias ChildrenPattern

Name -> Symbol

Alias -> : symbol
       | EPSILON

ChildrenPattern -> [ Patterns ]
                 | EPSILON

Patterns -> TreePattern Patterns
          | EPSILON

Structure -> ListStructure
           | TreeStructure

ListStructure -> [ Structures ]

TreeStructure -> Placeholder Name ChildrenStructure Insertion

Placeholder -> %
         | EPSILON

Insertion -> insert TreeStructure
           | EPSILON

ChildrenStructure -> [ Structures ]
                   | EPSILON

```

```
87 Structures -> TreeStructure Structures
88              | EPSILON
```

Listing H.1: Shows the LL(1) grammar for the translator syntax in Backus Naur Form.

Appendix I

Blink Java

```
1  package arduino.tutorial;
2
3  import static haiku.avr.lib.arduino.WProgram.*;
4  import static haiku.avr.AVRConstants.*;
5
6  public class blink {
7
8      public static void _wait() {
9          for(long i = 0; i < 3200000; i++) {}
10     }
11
12     public static void main(String[] args) {
13         DDRB |= 1<<5;
14         while(true) {
15             PORTB |= 1<<5;              // Turn LED on
16             _wait();                     // waits for a second
17             PORTB &= ~(1<<5);           // Turn LED of
18             _wait();                     // waits for a second
19         }
20     }
21 }
```

Appendix J

Programming Tasks in Tang

The Tang Programming Language

Task 1

```
1   int32 s = 0
2   bool b
3   b = false
4
5   for(int8 i from 1 to 6)
6       s = f(s, i)
7
8   int32 f(int16 v1, int16 v2)
9       return v1 + v2
10
11  if(s > 20)
12      b = true
13  else
14      b = false
```

Figure 1: Shows a program in The Tang Programming Language.

a) What is the value of b after executing the program in figure 1?

- true
- false

b) What is the value of s?

Answer: _____

Task 2

Figure 2 shows an Arduino Nano. An Arduino Nano is a microcontroller which can be programmed using

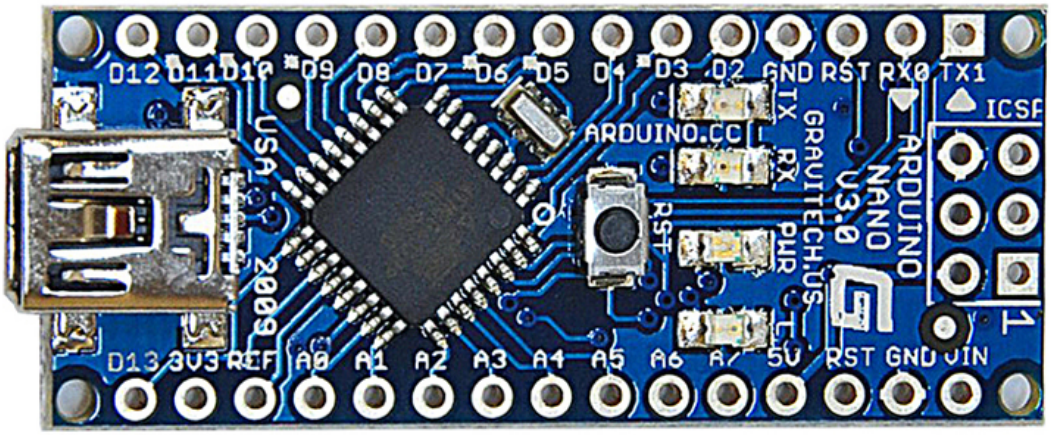


Figure 2: Shows an Arduino Nano

a) Mark which of the following devices you have programmed before:

[ ] Arduino Uno

[ ] Arduino Nano

[ ] Other microcontrollers or embedded devices

[ ] PC/Mac

As seen in figure 2 the Arduino Nano has what is called pins along the sides. Each pin has a name like D2, D3, D4 etc.

Pins are used to connect the Arduino with other devices like LEDs and sensors. A pin can either be used as input or output.

A special pin on the Arduino Nano is the pin D13 which is connected to the LED on the Arduino Nano.

To make this Led turn on and off you need make D13 output 5 or 0 volts. To do this you need to configure hardware registers which are parts of the Arduino Nano's memory that control the behavior of pins like D13.

The register at address 37 does also contain 8 bits where the bit at index 5 determines if pin D13's output value is 5 or 0 volts. The output voltage is 5 volts when the bit value is set to 1.

In The Tang Programming language, you can change the value of a hardware register using the register8 datatype.

An example of a register variable declaration of a register at address 40 is shown below in figure 3.

```
register8 PORTC = register8(40)
```

Figure 3: Shows how to declare a register8 variable

To change the bit values of the register then you can use the bit index operator as shown in figure 4.

```
r{3} = false
```

Figure 4: Shows how to change the bit value at index 3 to false.

b) Write a program in The Tang Programming Language that sets pin D13 as output and the output value to 5 volts. This should turn on the Led.

c) Write a function called ledOn in The Tang Programming Language that turns the Led on.

d) Write a function called ledOff in The Tang Programming Language that turns the Led off.

e) Make program that turns the Led on and off continuously with an interval of 1 second. Assume there exist a function called delay that stops execution in a duration based on a parameter in milliseconds.

J.1 Task Answers

This appendix shows the answers given by test person A and B to the tasks in appendix J.

J.1.1 Testperson A

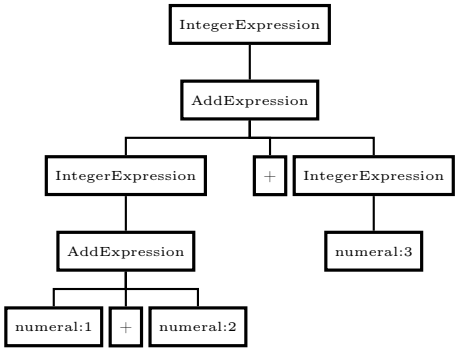
```
task 1:
a) false
b) 0 + 1 = 1 + 2 = 3 + 3 = 6 + 4 = 10 + 5 = 15
task 2:
a) PC/Mac
b)
    register8 PORT36 = register8(36)
    register8 PORT37 = register8(37)
    PORT36{5} = true
    PORT37{5} = true
c)
    void LedOn() {
        register8 PORT36 = register8(36)
        register8 PORT37 = register8(37)
        PORT36{5} = true
        PORT37{5} = true
    }
d)
    void LedOff() {
        register8 PORT37 = register8(37)
        PORT37{5} = false
    }
e)
    while(true) {
        LedOn()
        delay(1000)
        LedOff()
        delay(1000)
    }
```

J.1.2 Testperson B

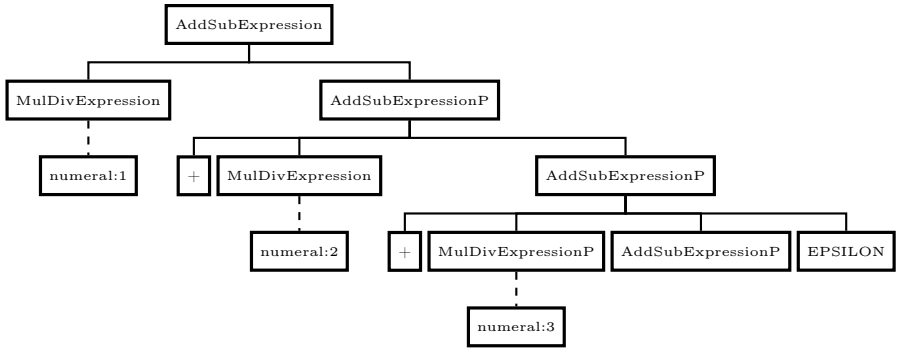
```
task 1)
    a: true
    b: 21
task 2)
    a: uno, nano, pc
b:
register8 r36 = register8(36)
register8 r37 = register8(37)
r36{5} = true
r37{5} = true
nothing ledOn()
    r36{5} = true
    r37{5} = true
nothing ledOff()
    r36{5} = true
    r37{5} = false
while (true)
    ledOn()
    delay(1000)
    ledOff()
    delay(1000)
```

Appendix K

AddExpression Parse-tree



(a) ParseTree for example



(b) AST for AddExpression example used in multiple unit tests.

Appendix L

Input Grammars For Parser Generator Tools

Listed in this appendix are the input grammars for the tools examined and documented in section 8.1. Listing L.1 shows a subset of the grammar of Tang, which is used as input grammar for the tools analysed. The grammar is not up-to-date with the actual grammar for Tang, but this is insignificant as the purpose is to show how grammar looks in the syntax of different tools.

Input Grammar

```
1 Program -> Statements
2 Statements -> Statement StatementsP
3 StatementsP -> newline Statement StatementsP
4             | EPSILON
5 Statement -> simpleType identifier Definition
6             | identifier assign integer
7             | newline
8 Definition -> assign integer
9             | EPSILON
```

Listing L.1: The input grammar used for the examples below in BNF.

Coco/R

```
1 Program = Statements.
2 Statements = Statement StatementsP.
3 StatementsP = "newline" Statement StatementsP
4             | "EPSILON".
5 Statement = "simpleType" "identifier" Definition
6             | "identifier" "assign" "integer"
7             | "newline".
8 Definition = "assign" "integer"
9             | "EPSILON".
```

Listing L.2: Grammar for the Coco/R tool.

GOLD

```
"Start Symbol" = <Program>

<Program>       ::= <Statements>
<Statements>    ::= <Statement> <StatementsP>
<StatementsP>   ::= newline <Statement> <StatementsP>
                  | EPSILON
<Statement>     ::= simpleType identifier <Definition>
                  | identifier assign integer
                  | newline
<Definition> ::= assign integer
                  | EPSILON
```

Listing L.3: Grammar for the GOLD tool.

SableCC

```
Productions
 program = {statements};
 statements = {statement} {statementsp};
 statementsp = newline {statement} {statementsp}
            | epsilon;
 statement = simpletype identifier {definition}
            | identifier assign integer
            | newline;
 definition = assign integer
            | epsilon;
```

Listing L.4: Grammar for the SableCC tool.

ANTLR4

```
grammar NameOfFile;
program : 'begin' statements 'end';
statements : statement statementsP;
statementsP : newline statement statementsP
            | ; //epsilon
statement : simpleType identifier definition
            | identifier assign integer
            | newline;
Definition : assign integer
            | ; //epsilon
```

Listing L.5: Grammar for the ANTLR4 tool.

LLLPG

```
LLLPG (lexer) {
    public rule Program @{Statements};
    public rule Statements @{Statement StatementsP};
    public rule StatementsP @{newline Statement StatementsP
                | EPSILON};
    public rule Statement @{simpleType identifier Definition
                |identifier assign integer
                |newline};
    public rule Definition @{assign integer
                |EPSILON};
};
```

Listing L.6: Grammar for the LLLPG tool.

Appendix M

Command-line Arguments for Compiling Bare Minimum Blink Programs

The numbers seen in table 3.2 describes the size of a blink program written in the languages listed in addition to the size of a bare minimum (empty) program.
The blink program written in AVR Assembler and the blink program written in C are compiled with the command *-mmcu=atmega328p option and -Os flag*, where -Os refers to optimising the code in regards to size of the program.
The blink program written in the Arduino language is compiled by using the Arduino IDE version 1.8.2.
The Java blink program was compiled by haiku with –Config ArduinoIDEUpload option [37].