High-level Arduino Language

A Compiler for HLMP - High-level Microcontroller Programming

Group SW-4-15
4th Semester, Software

P4 Project

Department of Computer Science
Aalborg University
https://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
High-level Arduino Language

Theme:
Design, definition and implementation of a programming language

Project Period:
February 2022 - May 2022

Project Group:
SW-4-15

Participants:
André Larsen
Jakob Lykke
Kasper Nielsen
Martin Jacobsen
Patrick Reffsøe
Peter Lauridsen

Supervisor:
Martin Zimmermann

Copies: 1

Page Numbers: 95

Date of Completion:
25/05/2022

Abstract:

This report documents the making of a programming language and compiler for Arduino C called High-level Microcontroller Programming. HLMP's purpose was to improve Arduino C's readability and writability when writing large programs. To do this many new features were introduced such as traditional nested functions and a new delay function to improve the function structure of Arduino C and its problematic `delay` function.

Finally, whether HLMP fulfills its goals remains inconclusive as the acceptance tests lack sufficient results to conclude this. However, the results did show a tendency, that HLMP had fulfilled its goals. Ultimately, the goals for the project were fulfilled, as a programming language and compiler were implemented.

Table of Contents

Introduction

In this report, we study the challenges of Arduino and how to improve these. Our motivation for working with Arduino is that we saw several opportunities to improve beforehand. We will investigate the structural and concurrency issues, which will be discussed further in chapter 2.

1.1 Arduino

Arduino is an open-source platform with a focus on easy-to-use hardware and software. The Arduino programming language (Arduino C) is a subset of `C++`. Arduino programs are written in the Arduino IDE[1] and are uploaded to the individual Arduino board. Arduino boards can read inputs and write outputs to different components. The board does this through either a digital or analog pin located on the board itself [1].

At first, the Arduino platform was designed as an easy-to-use tool for fast prototyping, made for students with minimal programming experience. As the popularity of Arduino grew, so did the community of the open-source platform. The Arduino board started changing to meet the wider community's needs with products for IoT and 3D printing, among other things [1].

Due to the growth of popularity of the Arduino board and its accompanying IDE, the target group expanded from students without much programming background to anyone - children, hobbyists, professional programmers, and so on [1].

1.1.1 The Arduino Language

A program written in Arduino C is called a **sketch**. Every sketch contains two standard functions; the *setup* and the *loop* function. The setup function is run once upon boot-up[2], while the loop function runs continuously until it is stopped by

[1] Integrated Development Environment

turning off the Arduino, or if the Arduino crashes [3].

The simple structure of the Arduino language makes it easy for beginners to engage with while still providing much functionality to more seasoned programmers. Unfortunately, Arduino C's simple structure is also a detriment to the level of abstraction available to Arduino C. The lack of abstraction becomes problematic as sketches grow larger. We will discuss this further in sections 2.3 and 4.1.

1.1.2 The Arduino Hardware

There are many different Arduino boards, each with different specifications, meaning that the performance of Arduinos differs from board to board.

All Arduino boards are fitted with physical pins, either digital or analog, which can be configured as an input or an output. Essentially, an Arduino reads inputs and writes outputs through its pins. Thus, the Arduino pins are an important part of the Arduino platform [4].

Arduino boards also have built-in interrupt pins. Interruptions can be used to run code practically instantaneous. The interrupt pins can interrupt any running code to perform a different task before returning to the previously running code. However, this only works for a few specific pins and can only be used for digital inputs, meaning inputs that are either low or high and not specific analog voltages. [5].

The Arduino platform is constantly evolving, with newer and more powerful boards that are using the platform, opening up to more possibilities [6]. With more powerful hardware, the possibility of running more complex tasks on the Arduino platform is becoming more of a reality. With the ever-evolving hardware utilizing the Arduino platform, the Arduino language may need to evolve alongside it.

We now move on to explore the shortcomings of Arduino C.

Challenges with Arduino

In this chapter, we discuss the various challenges found in Arduino C.

As previously mentioned, the Arduino platform is a tool with much potential, but it lacks scalability. It can be difficult to write larger sketches in Arduino C, given its lack of abstraction and simplification. This lack of abstraction can introduce more errors to the code. One example of Arduino's lack of scalability is the `delay` function.

2.1 Controlling Timing in Arduino C

The `delay` function in Arduino C is commonly used to slow down processes. The function halts all processes on the Arduino until the delay is finished[7]. The Arduino cannot perform any other tasks while this function is running. This means that any potential input change during the delay will go unnoticed by the Arduino. This problem can result in the input not being read because the Arduino is busy waiting instead. The problem can be solved using interrupts but as previously mentioned this only works for a few specific pins and digital inputs.

To combat the occurring difficulties from `delay`, many programmers use the `millis` function. This function returns the time since the Arduino was started and can be used to create a delay [8]. This is done by storing a timestamp and then waiting until the `millis` function returns a value that is a specific amount higher than the stored timestamp. It allows the programmer to halt a process for a specified amount of time while still allowing the Arduino to run other functions. This is unlike the `delay` function that stops every process on the Arduino. Unfortunately, the `millis` function is tedious to use and worsens the readability of the Arduino sketch.

Using `millis` is very different from `delay` and requires more code to implement. Code block 2.1 provides an example of a sketch that turns an LED on and off every

second using `millis`. As is shown, it requires a lot of code to perform this simple task. The standard Arduino `delay` is much shorter, as shown in code block 2.2. Therefore, using `millis` is very different from working with `delay`.

Another problem with `millis` is; if more delays are made with `millis`, it will require a new function for every `millis` delay, resulting in a large amount of duplicated code. In code block 2.1 on line 10, a function is called. In this example, the function only returns true, but when a different function should be run while the delay is happening, the function needs to be different for every `millis` delay. This will then require a new `millis` delay for each new function. As is shown, `millis` is unable to substitute a delay without a lot of additional code that worsens readability and writability.

```
1  void loop() {
2    millisDelay(1000);
3    digitalWrite(led,HIGH);
4    millisDelay(1000);
5    digitalWrite(led,LOW);
6  }
7  void millisDelay(int delayTime) {
8      unsigned long startTime = millis();
9      while (millis() < startTime+delayTime) {
10         if (!aFunction()) {
11             return true;
12         }
13     }
14     return false;
15 }
16 bool aFunction() {return true}
```

Code block 2.1: Blinking LED in Arduino language using `millis`

```
1  void loop() {
2    delay(1000);
3    digitalWrite(led,HIGH);
4    delay(1000);
5    digitalWrite(led,LOW);
6  }
```

Code block 2.2: Blinking LED in Arduino language using `delay`

In summary, `millis` can be used to write code that only halts a single process unlike the `delay` function. However, when using `millis` a new `millisDelay` function has to be defined for every `millisDelay`. This will result in repeating code and extra code that worsens the readability and writability of the sketch.

2.2 Arduino I/O Ports

As mentioned in the previous chapter, the I/O[1] ports on an Arduino consist of digital pins and analog pins. Each pin on an Arduino board has a unique identifiable number. The numbers for analog pins have an added *A* to signify that the pin is analog. The pin also needs to be set as an input or output port. This is done through the `pinMode` function. Calling this function in the setup function initializes the pin as input or output before using it in the loop function [9].

It is also common to define a constant int for every pin, as seen on lines 1 to 2 in code block 2.3. This allows the programmer to easily remember what each pin is connected to and allows for the pin number to be changed easily. The name given to the variable through the `const` declaration can now signify whether it is an analog or a digital pin. Two examples of pin initialization and mode specification using `const` and `pinMode` in Arduino C, are seen in code block 2.3.

```
1 const int digitalPin = 2;
2 const int analogPin = A1;
3 void setup() {
4   pinMode(digitalPin, OUTPUT);
5   pinMode(analogPin, INPUT);
6 }
```

Code block 2.3: Initialization and mode specification of Arduino pins using `pinMode`

Not manually setting the `pinMode` will not directly lead to an error because all I/O pins on the Arduino are set as input pins by default [10]. Calling a write function on an input pin will not automatically switch the mode from input to output [11]. This can lead to problems when working with external components connected to the Arduino. These problems can be combatted by remembering to specify the `pinMode`. However, since it is not mandatory to specify the `pinMode`, some users can forget or even omit the specification [12].

2.3 Arduino C's Function Structure

Writing large programs in Arduino can be tedious and worsen writability because of the function structure. Due to the flat structure of the Arduino C language, any function can be called from anywhere. The lack of encapsulation and access limitation makes it difficult to write large programs for Arduino. This flat structure

[1] input and output

also carries through when using multiple files for an Arduino sketch. Having separate scopes for separate files would be beneficial. The individual scopes allow access limitation and reuse of function names. This improves the overall writability of the language and makes it more suitable for larger sketches.

2.4 Read and Write

In order to write a value to a digital or analog pin, one must use `digitalWrite(pin, value)` or `analogWrite(pin, value)` that writes a value to a selected pin. Reading a value from a pin makes use of similar functions named `digitalRead(pin)` and `analogRead(pin)`. We will now go further in-depth with these functions.

2.4.1 digitalRead

`digitalRead` reads a value from a specified digital pin, this value can either be `HIGH` or `LOW`. The `digitalRead` function is relatively simple since it maps inputs to 2 values, whereas the `analogRead` is relatively more complex.

2.4.2 analogRead

The `analogRead` function will map input voltages between 0 and 5 volts to integer values ranging from 0 to 1023 - a total of 1024 values.

2.4.3 digitalWrite

`digitalWrite` gives a digital output to a given pin, configured as `OUTPUT` in pin mode. It takes a pin number and a value which can either be `HIGH` or `LOW`. The value `HIGH` corresponds to 5V for most arduino boards. The value `LOW` corresponds the value 0V. [13]

2.4.4 analogWrite

The `analogWrite` function gives an output to digital pins that supports PWM[2]. The analog value is represented by a value between 0-255, where 255 is equivalent to 5V and 0 corresponds to 0V. When the `analogWrite` is used, it achieves the equivalence of analog signal with digital means, by using the PWM technique. It creates a square wave signal that represents a switch between 5V and 0V repeatedly. To obtain the analog values, you adjust the pulse width[3] of the signal [14]. If repeated fast enough, this on-off pattern will result in the equivalence of an analog signal, as shown in figure 2.1.

[2]Pulse Width Modulation

[3]the duration of the on time

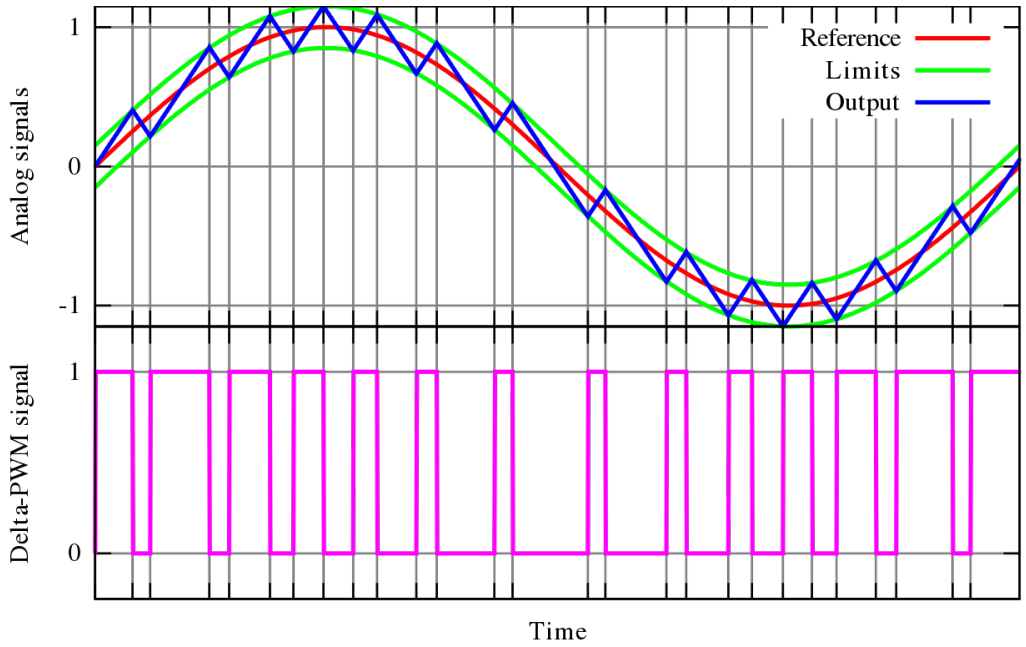

Figure 2.1: PWM signal equivalence to an analog signal. When pulse width is high, so is the analog signal[15].

2.4.5 The problem with analogRead and Write

As previously mentioned, `analogRead` reads integer values between 0-1023 and `analogWrite` is only capable of writing values between 0-255. This difference makes it impossible to read an analog value and write it directly to an analog pin without the need for conversion or mapping [16]. The need for conversion or mapping adds additional clutter to the program, which negatively affects the readability and writeability as shown in code block 2.4.

```
1 int pinOutput = map(analogRead(A0), 0, 1023, 0, 255);
```

Code block 2.4: Reading an analog value and mapping it between 0-255

This chapter has established the various problems found in the Arduino C language. We especially find the challenges regarding timing control and function structure particularly interesting.

Problem Definition

Throughout chapter 2, we established various shortcomings in the readability and writability of the Arduino C language. We see an opportunity to make changes to Arduino C, which should mitigate the previously mentioned shortcomings. As was mentioned, when writing larger sketches, the flat function structure in particular, results in worse readability.

This results in the following problem statement:

How can we design and implement a programming language based on Arduino C with better structure and support for abstraction to provide better read- and writability when writing larger Arduino programs?

The following chapter discusses our proposed solution to our problem statement.

Language Design

This chapter will explore and describe our solutions to the problems discussed in chapter 2. The proposed solutions will be evaluated using the language evaluation criterias from Sebesta [17].

4.1 Nested Scopes

We will now discuss how we will implement nested functions and procedures. Since they are similar in this section we will discuss functions. In section 2.3, we discussed how the function structure of the Arduino language could be tedious and worsen the writability. We wish to counter this by implementing nested functions and procedures. Procedures and functions are nearly identical, both having parameters and a body. The only difference is that procedures do not contain return statements. We have chosen to add this to better accommodate the classical computer science concepts [17]. Also to make it impossible to use a `void` function in an expression like in Arduino C where you will get a `void` error. A procedure cannot be used in an expression then eliminating the problem.

Nested functions are functions or procedures defined in another function or procedure. This introduces encapsulation, which is a form of information hiding. They are useful for dividing procedural or functional tasks into subtasks that are only meaningful locally [18]. Typically, nested functions are used as helping functions or as recursive functions inside another function, as they will be in HLMP.

Let us now give a concrete example of how nested functions are useful. Code block 4.1 shows an example that displays the structure of a program written in our language. The program reads an input from a sensor and has a LED strip connected to it. We have encapsulated all functionality related to a sensor in the `sensorHandler` function and all functionality regarding the LED in the function `ledHandler`. Which is seen on lines 1 to 5 and 7 to 14.

Ultimately, Code block 4.1 exemplifies how nested functions improve readability and writability. For example, the `blink` function (on lines 8 to 11) can either blink fast or slow and shows how we can use nested functions to simplify the logic needed for this functionality. This also shows how real-world objects like a sensor or LED can be encapsulated. This makes it possible to represent their functionality in their own scope thus making them represent the real-world better.

```
1  func Pwm sensorHandler() {
2      proc initializeSensor() {...}
3      func Pwm readSensor() {...}
4      func Bool checkSensor() {...}
5  }
6  proc ledHandler() {
7      proc checkSensor() {}
8      proc blink() {
9          proc blinkFast() {...}
10         proc blinkSlow() {...}
11     }
12     proc fadeUp() {...}
13     proc fadeDown() {...}
14 }
```

Code block 4.1: Example of nested functions in HLMP

Now that we have established the usefulness of our nested functions, let us show how we wish to handle nested function calls. Code block 4.2 shows an example of how nested functions is called in our language. The nested function `checkSensor` which is declared on line 11, is called within the enclosing function's scope on line 7. Functions can only be called in the same scope as they are declared or the function's parent scopes. It is also possible to call a function recursively within itself, as shown on line 18. This example also shows, that it is possible to declare multiple identical identifiers in a program, as long as they are not within the same scope.

```
1  proc setup() {ledHandler();}
2  proc loop() {}
3  func Pwm sensorHandler() {
4      if (notInitialized()) {
5          initializeSensor();
6      }
7      return checkSensor(readSensor());
8      proc initializeSensor() {}
9      func Bool notInitialized() {}
10     func Pwm readSensor() {}
11     func Pwm checkSensor(Pwm input) {}
12 }
13 proc ledHandler() {
```

```
14      proc checkSensor() {
15          if (sensorHandler() == 1) {blink(true);}
16          if (sensorHandler() == 2) {fadeUp();}
17          if (sensorHandler() == 3) {fadeDown();}
18          checkSensor();
19      }
20      proc blink(Bool shouldBlinkFast) {
21          if (shouldBlinkFast) {blinkFast();}
22          else {blinkSlow();}
23          proc blinkFast() {}
24          proc blinkSlow() {}
25      }
26      proc fadeUp() {}
27      proc fadeDown() {}
28  }
```

Code block 4.2: Example of nested functions calls in HLMP

4.2 Data Types

In order to further increase the readability of our language we wish to add new data types. These will help improve readability by encapsulating collections of data in predefined sets. The new types are discussed in the following sections.

4.2.1 Num

We introduce a new data type that encompasses the different numerical data types of Arduino C. The new type is called number and is declared in our programming language as `Num`. Code block 4.3 shows an example of how the declaration of a `Num` is performed in HLMP.

```
1  Num a = 1;       //short
2  Num b = 1.1;     //float
```

Code block 4.3: Example of declaration of numbers

The `Num` data type replaces the *int*, *unsigned int*, *short*, *long*, *unsigned long*, *float*, *unsigned float*, *double* and *unsigned double* data types of Arduino C. It will then be up to the compiler to determine the type of `Num` and ensure the correct data type, based on the value of the declared variable. The type should also be changed based on potential changes in the value of the variable.

4.2.2 Bool

We wish to simplify the current Boolean type of Arduino C to enforce good programming practices. Our Boolean type, annotated as `Bool`, can either be true or false. This means that, unlike Boolean in Arduino C, our `Bool` cannot be represented as numeric values. Boolean expressions will be used as a condition for executing control structures, such as if-else statements. The declaration of a `Bool` data type can be seen in code block 4.4.

```
1 Bool trueVar = true;    // valid
2 Bool falseVar = false; // valid
3 Bool trueVar = 1;       // not valid
4 Bool exprVar = 1 == 1    // evaluates to true
```

Code block 4.4: Example of declaration of Boolean types

4.2.3 Pwm

We also want to a add a new data type called `Pwm`. This type is based on the data type `Byte`, and is used for representing a PWM value. A `Pwm` variable will store an unsigned number, from 0 to 255. The reason we want to add this data type, is because we wish to explain what the data type is used for and not what it is. In this way we encourage programmers to use a type that represents the properties of real-world components connected to the Arudino. Code block 4.5 shows an example of how to declare a `Pwm` variable in our programming language.

```
1 Pwm pwmVar = 128; // byte
```

Code block 4.5: Example of declaration of `Pwm` type

4.2.4 String

We wish to combine the two Arduino C types `Char` and `String` into a single type called `String`. The compiler should then switch between the two types, based on the content within a declaration of a `String` variable. Code block 4.6 shows an example of how to declare a `String` variable in our programming language.

```
1 String charVar = "c";       //Char
2 String stringVar = "str";   //String
```

Code block 4.6: Example of declaration of `String` type

4.2.5 New Data Type For Pins

In section 2.2 we discussed some of the issues found in the Arduino C language, related to the pin type. Due to those issues discussed, we wish to make a new type for the declaration of pins. Code block 4.7 shows how we wish to declare pins in our language. When defining a pin we first type the keyword `Pin` followed by the name of the pin. Then, in curly braces, we specify the pin number and the pin mode separated by a comma. The pin number also defines whether a pin is a digital or an analog pin, either by a `D` or an `A`.

```
1 Pin digitalPin {D5, out}
2 Pin analogPin {A2, in}
```

Code block 4.7: Example of initializing of Pins

With this implementation, we force the pin mode to be specified during pin definition. This eliminates errors, caused by forgetting to specify the pin mode, as was mentioned in section 2.2. This change adds more simplicity and provides our language with an overall better readability and writability.

4.3 New Read and Write Functions

In section 2.4, we discussed difficulties relating to the interplay between `analogRead` and `analogWrite`. We wish to combat these issues by redesigning the read and write functions of Arduino C.

We have chosen to combine the `digitalWrite` and `analogWrite` into a single function called `.write`. This function should be able to determine whether it is dealing with an analog or a digital pin, based on the given input. This is a quality of life addition, that should improve the writability of the program.

In addition to our new `.write` function we wish to add three new read functions:

- `.ReadPwm` reads from an analog pin in range 0-255
- `.ReadAnalog` reads an analog signal in range 0-1023
- `.ReadDigital` reads the range from LOW-HIGH(0-1) from any pin on the Arduino.

The `.ReadAnalog` and `.ReadDigital` is also present in Arduino C and are necessary for the reading all types of inputs. `.ReadPwm` is a new functions we have added to accommodate the problem discussed in section 2.4.5 with´`analogRead` returning

a value between 0-1023. This allows the programmer to write the value from a `.ReadPwm` function directly to a `Pwm` type or to a `.write` function, without needing to map the value.
These changes to *Read* and *Write* will help improve the overall syntax design and therefore increase the readability and writability of the program. An example of the redesigned read and write functions is shown in the following section on figure 4.1.

4.4 New Delay Function

In section 2.1, we discussed how the `delay` function wastes time on the Arduino by halting all processes. One solution to this problem could be using the `millis` function which is difficult and tedious to implement. Because of this, we wish to improve the syntactical design of `delay` and `millis` by implementing a new procedure named `whileWait`. This new procedure waits for an amount of milliseconds while running another function. In this way, you can still run other processes while the delay is underway.

To better understand how the `whileWait` procedure improves writability, we will give an example. On figure 4.1, we have written a program that blinks an LED until an analog signal reaches a specific value. The left-most code shows a sketch written in Arduino C. The corresponding code written in our language is shown to the right. The code written in our language is shorter and more compact than the Arduino code. The compact nature of our code increases its simplicity, which in turn increases readability and writability. This will be elaborated on in the following section. Thus, the `whileWait` procedure allows us to create a delay similar to `millis`, but without a substantial amount of code.

The `whileWait` procedure will have two parameters. The first parameter is the delay time. The second parameter is a reference to a function. This function will run while the delay is happening. In this way, the program will wait at the caller until the time has run out while still being able to run another process while it is waiting. An example of this is shown on figure 4.1. On line 5 in our code, the `whileWait` procedure takes a delay of 2000 milliseconds and the `shouldBlink` function as its parameters. Then the `shouldBlink` function will be run continuously until the delay is finished. However, this is only a baseline delay. If the duration of fulfilling the task takes longer, it will still run to completion.

```
void loop() {
  while (shouldBlink()) {
    digitalWrite(ledPin1, HIGH);
    digitalWrite(ledPin2, LOW);
    if (whileWaitFunc(2000)) {
      break;
    }
    digitalWrite(ledPin2, HIGH);
    digitalWrite(ledPin1, LOW);
    if (whileWaitFunc(2000)) {
      break;
    }
  }
  digitalWrite(ledPin2, LOW);
  digitalWrite(ledPin1, LOW);
}

int whileWaitFunc(int delayTime) {
  unsigned long startTime = millis();
  timer = millis();
  while (timer < startTime+delayTime) {
    timer = millis();
    if (!shouldBlink()) {
      return 1;
    }
  }
  return 0;
}

int shouldBlink() {
  byte sensorValue = analogRead(analogInPin);
  byte outputValue = map(sensorValue, 0, 1023, 0, 255);
  if(outputValue > 128) {
    return false;
  }
  return true;
}
```

```
proc loop() {
    while (shouldBlink()) {
        ledPin1.Write(HIGH);
        ledPin2.Write(LOW);
        whileWait(2000, shouldBlink);
        ledPin1.Write(LOW);
        ledPin2.Write(HIGH);
        whileWait(2000, shouldBlink);
    }
    ledPin1.Write(LOW);
    ledPin2.Write(LOW);
}

func Bool shouldBlink() {
    Pwm outputValue = sensorPin.ReadPwm();
    if(outputValue > 128) {
        return false;
    }
    return true;
}
```

Figure 4.1: Example of a blinking led that can be interrupted by an analog signal. The code in Arduino C to the left and in our language on the right.

4.5 Language Ambition

Our language ambition is to provide a programming language with high-level features that the relatively small processing power of the Arduino platform can handle. The goal is to provide a programming language aimed at more experienced programmers. We have chosen to name our language **HLMP - High-Level Microcontroller Programming**.

The ambition for HLMP is to provide a high-level language for Arduino C. HLMP will seek to counter the sub-optimal writability and readability of Arduino C by improving structure and abstraction while hiding unnecessary information from the programmer. While Arduino is used by many different groups of users ranging from novice to professional programmers, HLMP will be aimed towards the professional programmer with an understanding of high-level programming.

4.6 Language Evaluation Criteria

Figure 4.2 shows a list of criteria used to evaluate the readability, writability, and reliability of a programming language.

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Figure 4.2: Language evaluation criteria and the characteristics that affect them.[17]

Readability is one of the most important criteria for judging a programming language. It is the ability to read and understand a language, which greatly increases the ease of maintaining the programming language. Meanwhile, **writability** is a measure of how easy a program is to write[17]. HLMP will aim to improve the readability and writability of Arduino C by adding more simplicity[1], new data types, expressivity[2], support for abstraction and better syntax design[3].

With the nested functions, new data types, reworks to Arduino C's read and write functions, and our `whileWait` *procedure, we believe that we are able to improve the Arduino C language significantly. Our additions should increase the readability and writability of Arduino C, which in turn should improve the reliability of our language.*

The following chapters describe the syntax and the semantics of our proposed solution.

[1] A small amount of basic constructs

[2] Convenient ways of specifying computations

[3] Design choices of special words and statements that indicates their purpose, just from their apperearance

Syntax

In this chapter, we will describe the syntax of HLMP through the use of *Context-Free Grammars (CFG)*. Additionally, there will be a definition of what a CFG is. Lastly, the CFG for HLMP is presented as an example of how it can be used, to make a parse tree.

5.1 Context-free Grammars

The components of a CFG is expressed as $\mathtt{G} = (N, \Sigma, P, \mathtt{S})$. The N defines a finite non-terminal alphabet[1], that contains the variables of the given grammar. The Σ defines a finite terminal alphabet[2]. This set is augmented with an \$ to signify an end-of-input. The P defines a finite set of productions which are all the rewriting rules of the grammar. The $\mathtt{S}$ defines the start symbol of the grammar. This is the symbol that initiates all derivations.[19]

A production of a CFG can be viewed as two sides being the left-hand side (LHS) and the right-hand side (RHS). The LHS is a non-terminal which is being led by a rule, these rules are made by the RHS. The RHS can be a combination of tokens[3], lexemes and non-terminals. The lexemes are instances of a token class [19]. An example of this in our language would be `PINNUMBER` : `'D'`$[0-9]+$ | `'A'`$[0-9]+$;. Here the LHS `PINNUMBER` is the token class, and the two RHS are instances of of said token.

There are two conventions used for derivations, being **Leftmost** and **Rightmost** derivation. These conventions ensure that non-terminals are rewritten in a systematic order. Leftmost derivation is the convention that always chooses the leftmost possible non-terminal at each derivation step. Rightmost derivation is the other

[1]Non-Terminal alphabets consist of Non-Terminal Symbols, which are the symbols that take part in the generation of the sentence but are not the component of the sentence.

[2]Terminal alphabet is a set of terminal symbols, which is the set of tokens produced by the scanner.

[3]Tokens are the character that are read from a program.

convention, which without exception expands the rightmost non-terminal[19]. These derivation conventions are both used in different parsing strategies. This will be discussed further in chapter 7.7.

Evaluating a CFG will also reveal the grammars operator precedence. The further down an operator is derived in a CFG, the higher its precedence.

5.2 Derivations Through Our Context-free Grammar

In this section, we will give an example of a derivation through our CFG. Code block 5.1 shows the subset of our CFG needed to complete the derivation in our example. The full CFG for HLMP is shown in section 13.1 in the appendix.

```
1  //Parse Rules
2  program: standardProc content* EOF
3         | content* standardProc content* EOF
4         | content* standardProc EOF;
5
6  content: funcProc | varDecl END
7         | pinLiteral END | comment;
8
9  pinLiteral: PINTYPE id LBRACE PINNUMBER ',' pinmode RBRACE;
10 pinmode: OUT | IN;
11
12 id : ID;
13
14 //Lexer Rules
15 PINTYPE: 'Pin ';
16 ID: [a-zA-Z_] [a-zA-Z_0-9]*;
17 LBRACE: '{';
18 RBRACE: '}';
19 PINNUMBER: 'D' [0-9]+ | 'A' [0-9]+;
20 IN: 'in';
21 OUT: 'out';
```

Code block 5.1: Subset of the CFG of HLMP

Our grammar is an LL grammar meaning that we derive left-to-right. This will be expanded on in chapter 7.7.

Let us consider a pin initialization within the setup procedure. We initialize a pin with the input string "*Pin ledpin1 {D1, out};*". The derivation of this string is shown in code block 5.2. As is seen, we always derive the leftmost rule of the grammar, until we reach an instance of a token (terminal symbol). We show this by coloring

the terminal symbol blue. The rule under derivation is colored red.

```
content
→ pinLiteral END
→ PINTYPE id LBRACE PINNUMBER ',' pinmode RBRACE END
→ Pin id LBRACE PINNUMBER ',' pinmode RBRACE END
→ Pin ledpin1 LBRACE PINNUMBER ',' pinmode RBRACE END
→ Pin ledpin1 { PINNUMBER ',' pinmode RBRACE END
→ Pin ledpin1 { D1 ',' pinmode RBRACE END
→ Pin ledpin1 { D1 , pinmode RBRACE END
→ Pin ledpin1 { D1 , out RBRACE END
→ Pin ledpin1 { D1 , out } END
→ Pin ledpin1 { D1 , out } ; //End of derivation
```

Code block 5.2: Example of a derivation given a specific input string.

This exemplifies how derivations are carried out through our CFG.

Now that we have established the CFG of our language, we move on to define our semantics. The semantics are sets of rules for how a program should operate in our language.

Semantics

This chapter will describe the operational semantics of HLMP. Operational semantics covers the behavior of a programming language.

6.1 Transition Systems

A structural operational semantics defines a transition system. A transition system can be thought of as a directed graph. Its vertices are called configurations and the edges are called transitions. The configurations correspond to a snapshot of the program and its current state. The transitions are steps in the program. Vertices without any transitions leading away from them are called terminal configurations since this corresponds to a terminal state of the current path in the program. Figure 6.1 shows an illustration of a directed graph. Note that it is only possible to picture a transition system this way if there is a set of finite configurations in the transition system [20].

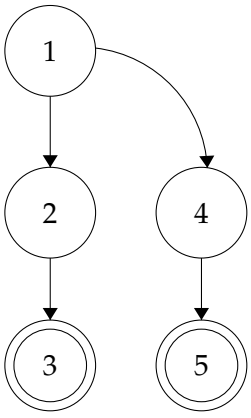

Figure 6.1: Directed graph illustrated a transition system

Thus, a transition system is a triplet $(A, \rightarrow, B)$ where A is the set of all configura-

tions. $\rightarrow$ is the transition relation, which is a subset of the cartesian product $A \times B$. That is, $\rightarrow \subseteq A \times B$. Lastly, $B \subseteq A$ is a set of terminal configurations.[20]

If we take figure 6.1 as an example, we can define the transition system as:

$$\begin{aligned} A &= \{1, 2, 3, 4, 5\} \\ \rightarrow &= \{(1, 2), (1, 4), (2, 3), (4, 5)\} \\ B &= \{3, 5\} \end{aligned}$$

Now that we have defined a transition system, we move on to look at the different types of operational semantics.

6.2 Types of Operational Semantics

There are two kinds of operational semantics, each with its way of describing computations through transitions. A transition is written as $\alpha \rightarrow \alpha'$.

The first kind of semantics is called **big-step semantics**. A single transition in a big-step semantics describes an entire computation. Meaning that the computation starts in configuration α and ends in configuration α'. For this to be the case α' must always be a terminal configuration[20].

The second kind of semantics is **small-step semantics** which describes single steps in a larger computation. Thus, the transition $\alpha \rightarrow \alpha'$ in a small-step semantics describes a single step that is not necessarily the entirety of a computation. Because of this, α' does not need to be a terminal configuration[20].

Both kinds of operational semantics are useful for different tasks. Small-step semantics can describe languages that utilize parallelism or run indefinitely. Big-step semantics cannot describe computations at the same level of detail as small-step semantics. This also means that big-step semantics are often easier to describe and formulate[20].

Big-step semantics are usable when describing terminal transition rules. Small-step semantics are used when we wish for transition rules to run without terminating. Since we are working with the Arduino platform, we need to make use of small-step semantics for some of our transition rules. Given that many Arduino programs run indefinitely, it makes sense to utilize small-step semantics for our non-terminal transition rules. For the remainder of our transition rules, we utilize big-step semantics.

6.3 Abstract Syntax

Before constructing our transition rules, we need to define our abstract syntax. We use our abstract syntax throughout the creation of our operational semantics.

The abstract syntax consists of syntactic categories and accompanying formation rules for each syntactic category. The syntactic categories are the fundamental entities of a programming language [20]. To represent the elements of a syntactic category, we use metavariables. These are seen in table 6.1, where n is the metavariable for *Number* and so on.

Syntactic categories		
$n \in$ **Num**	-	*Number*
$x \in$ **Var**	-	*Variable*
$P_{wm} \in$ **Pwm**	-	*Pwm*
$P_{in} \in$ **Pin**	-	*Pin*
$e \in$ **Exp**	-	*Expessions*
$a \in$ **Aexp**	-	*Arithmetic expressions*
$b \in$ **Bexp**	-	*Boolean expressions*
$S \in$ **Stmt**	-	*Statements*
$D_v \in$ **DecV**	-	*Variable declarations*
$D_{pin} \in$ **DecPin**	-	*Pin declarations*
$Pin_n \in$ **Pinname**	-	*Pin names*
$N_p \in$ **NumP**	-	*Pin numbers*
$M_p \in$ **ModeP**	-	*Pin modes*
$D_p \in$ **DecP**	-	*Procedure declarations*
$P_n \in$ **Pname**	-	*Procedure names*
$D_f \in$ **DecF**	-	*Function declarations*
$F_n \in$ **Fname**	-	*Function names*

Table 6.1: Syntactic categories

Formation rules are context-free production rules, meaning that we do not care about their ambiguity. They describe valid structures of our syntactic categories. We make formation rules for all syntactic categories that are not atomic. Atomic syntactic categories like the ones for *Numbers* and *Variables* do not have any formation rules, since they can only be constructed as is.

We read a formation rule from left to right. On the left side of **::=**, we have the syntactic category is described by the formation rule. The right-hand side describes all the ways we can construct said syntactic category. Our formation rules are as

follows:

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \mid a_1 \% a_2 \mid (a_1)$$

$$b ::= a_1 == a_2 \mid a_1 != a_2 \mid a_1 < a_2 \mid a_1 > a_2 \mid !b_1 \mid b_1 \| b_2 \mid b_1 \&\& b_2 \mid (b_1)$$

$$e ::= a \mid b \mid \texttt{call}\ f_n(x_1 ... x_n) \mid \texttt{call}\ f_n()$$

$$S ::= \texttt{active}\ S\ \texttt{end} \mid x := e \mid S_1; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{call}\ P_n(x_1 ... x_n) \mid \texttt{call}\ P_n() \mid D_{pin} \mid D_f \mid D_p \mid D_v \mid \texttt{skip}$$

$$D_v ::= \textbf{Var}\ x := e;\ D_v \mid \epsilon$$

$$D_f ::= \texttt{func}\ F_n()\ \texttt{is}\ S;\ \texttt{returns}\ e;\ D_f \mid \texttt{func}\ F_n(\ x_1 ... x_n)\ \texttt{is}\ S;\ \texttt{returns}\ e;\ D_f \mid \epsilon$$

$$D_p ::= \texttt{proc}\ P_n()\ \texttt{is}\ S;\ D_p \mid \texttt{proc}\ P_n(\ x_1 ... x_n)\ \texttt{is}\ S;\ D_p \mid \epsilon$$

$$D_{pin} ::= P_{in}\ Pin_n\{N_p, M_p\};\ S \mid \epsilon$$

We now need a way to describe how variables are bound to values in our language. This will be done in the following section.

6.4 Models for Variable Binding

Given that our language contains variables that can be modified, we need to consider how those variables are stored. This is done by using one of two models called ***state model*** *and* ***environment-store model****. These will now be described and discussed.*

State Model

The **state model** considers a state as a partial function[1] $s : \textbf{Var} \rightharpoonup \mathbb{Z}$. This means that we can know the value of a variable based on the state s. The set of states is called **States** and is defined as **States** = **Var** $\rightharpoonup \mathbb{R}$ [20]. In this example, **Var** can map to values contained in $\mathbb{R}$.

In order to update a state in the state model we say that the updated state $s[x \mapsto v]$ is the state s'. This state is given by

$$s'y = \begin{cases} s\,y & \text{if, } y \neq x \\ v & \text{if, } y = x. \end{cases}$$

[1] A partial function means that for every argument a there is at most one value b [20].

$s[x \mapsto v]$ is to be read as s where x is now bound to v [20].

The state model is useful for less complex programming languages. For languages with concepts like references, procedures, and functions, we would use a more sophisticated model that does not view variables as bound to their values directly.

Environment-store Model

The **environment-store model** is a more refined way of describing the state of a program. The difference between the two models is that the environment-store model allows for the binding of variables to actual storage cells (memory), unlike in the state model where a state only tells us the values of variables [20].

A variable is bound to a storage cell and the contents of the said storage cell are the value of the variable. Thus, in the environment-state model, a state both tells us what is stored (value) but also where it is stored (storage cell). This means that we can describe phenomena of programs such as side effects or constructs like references [20].

Two functions are used to describe a state in this model. Combining these gives us a complete description of the contents of the memory. The functions are as follows:

- The *variable environment* function tells us which variable is bound to which storage location. This corresponds to a symbol table.
- The *store* function describes the contents of the memory by telling us the value found at each storage location.

We consider storage cells as *locations*. The set of locations is called **Loc**. By letting l denote an arbitrary element in **Loc** we say $l \in \mathbf{Loc}$. Furthermore, we always assume that $\mathbf{Loc} = \mathbb{N}$.

By introducing a "`next`" pointer that is bound to the next available location, we can describe that new locations can be allocated to new variables. With the definitions of both **Loc** and the `next` pointer, we can describe the set of variable environments. The set of variable environments is a set of partial functions from variables to storage locations and is denoted as:

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

We also denote a single element in the set of variable environments as $env_V \in \mathbf{EnvV}$.

The set of stores is a set of partial functions from locations to values and is denoted as:

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

Lastly, we say that $sto \in \mathbf{Sto}$.

In order to update a state in the environment-store model, we need to both update the environment and the store. $env_V[x \mapsto l]$ denotes the updated environment env'_V given by:

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

$sto[l \mapsto v]$ denotes the updated store sto' given by:

$$sto' y = \begin{cases} sto\, y & \text{if, } y \neq l \\ v & \text{if, } y = l. \end{cases}$$

Figure 6.2 shows an example of a variable environment and a store. This is a graphical representation of how the *variable environment* function maps variables to storage cells, and the *store* function maps locations to values.

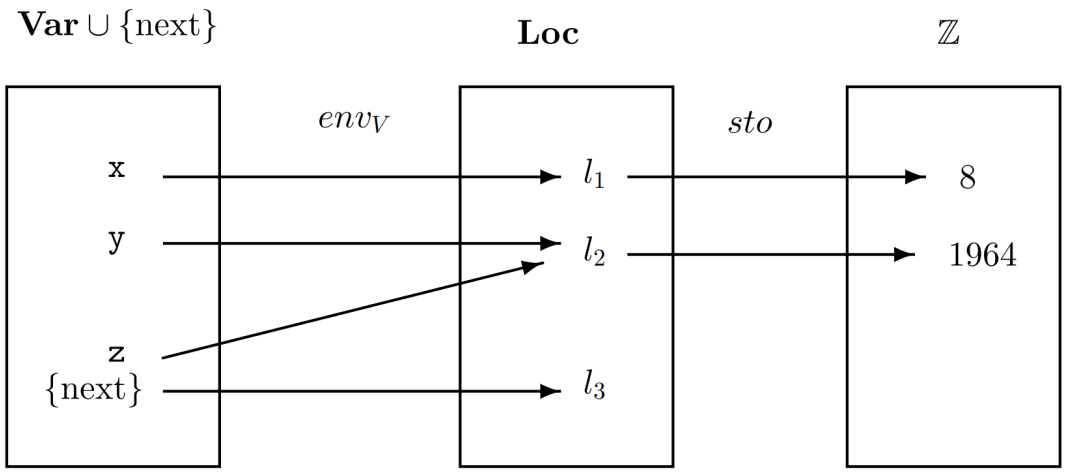

Figure 6.2: A graphical example of variable environment and store [20]

We have chosen to favor the environment-store model in this project. This is due to our use of concepts like functions and procedures.

Our Environments

Let us now consider the various environments that HLMP will utilize. We have the five environments:

- Variable environment: **EnvV**
- Function environment: **EnvF**
- Procedure environment: **EnvP**
- Subroutine environment: **EnvS**
- Runtime-stack environment: **EnvT**

Our set of variable environments look exactly like the one previously introduced in this section: **EnvV** = **Var** $\cup \{next\} \rightharpoonup$ **Loc**.

Our set of function environments is the set of partial functions: **EnvF** = **Fname** $\rightharpoonup$ **Stmt** $\times$ **EnvV** $\times$ **EnvF**. When we use fully static scope rules, the execution of a function call can only use the variable and function bindings that were known when the function was declared. For this reason, the function environments have to remember those bindings. The same holds true for our procedure environments. Therefore, the set of procedure environments is the set of partial functions: **EnvP** = **Pname** $\rightharpoonup$ **Stmt** $\times$ **EnvV** $\times$ **EnvP**.

Our set of subroutine environments is the set of partial functions: **EnvS** = **EnvP** $\cup$ **EnvF**. The subroutine environment operates as a meta environment for the function and procedure environments in our semantics. There are areas of our semantics where we need to describe both the function and procedure environment in a given context. In these contexts, a given semantic will need both environments to describe that transition. Therefore, to reduce redundancy in our semantics, we have chosen to include this subroutine environment.

The set of runtime-stack environments is the set of partial functions: **EnvT** = (**EnvV** $\times$ **EnvS**)*. This environment is needed when describing the active (in current execution) context or part of a program in our small-step semantics. The reason for this is that we wish to keep track of the bindings that are currently in effect in the given execution context (current scope). When we enter a new scope via procedure- or function calls in our program, we push a new pair of (env'_V, env'_S) on top of the runtime stack. The pair of (env'_V, env'_S) closely resembles the concept of activation record - a private block of memory associated with an invocation of a procedure or function. In short, this environment is necessary to keep track of and describe the current bindings and the active part of the program.

Now that we have established both our abstract syntax and a way to bind our variables, we move on to define our transition rules. This is done in the following sections.

6.5 Arithmetic Expressions

This section details our transition rules for our arithmetic expressions, denoted in the big-step semantics. All of our arithmetic expressions are terminal configurations, and as such, can be described through big-step transition rules. We describe the transition rules for our arithmetic expressions first because we use them in the following transition rules.
We use the formation rules for arithmetic expressions constructed in section 6.3 for our transition rules. They were defined as follows:

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \mid a_1 \% a_2 \mid (a_1)$$

Table 6.2 shows our big-step transition rules for arithmetic expressions. Each transition rule consists of multiple parts. The leftmost part shows the name of the rule. The middle part contains the rule itself, which is in turn constructed of multiple parts. The rightmost part shows the rule's side condition.

Let us consider the rule [PLUS$_{BSS}$] as an example. The middle of the rule consists of two main parts. The top part consists of premises and the bottom part is the conclusion of the rule.
Let us also briefly mention the side condition to the [NUM$_{BSS}$] rule. Here we assume the existence of a function $\Re : Num \to \mathbb{R}$ that gives us the value of each number. In this way, we convert the numeral symbol with its actual value.

[NUM$_{BSS}$]	$env_V, sto \vdash n \rightarrow_a v$	if $\Re[n] = v$
[VAR$_{BSS}$]	$env_V, sto \vdash x \rightarrow_a v$	if $env_V\ x = l,\ sto\ l = v$
[PLUS$_{BSS}$]	$\dfrac{env_V, sto \vdash a_1 \rightarrow_a v_1\ env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash\ a_1 + a_2 \rightarrow_a v}$	where $v = v_1 + v_2$
[MINUS$_{BSS}$]	$\dfrac{env_V, sto \vdash a_1 \rightarrow_a v_1\ env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash\ a_1 - a_2 \rightarrow_a v}$	where $v = v_1 - v_2$
[MULTIPLY$_{BSS}$]	$\dfrac{env_V, sto \vdash a_1 \rightarrow_a v_1\ env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash\ a_1 * a_2 \rightarrow_a v}$	where $v = v_1 \times v_2$
[DIVISION$_{BSS}$]	$\dfrac{env_V, sto \vdash a_1 \rightarrow_a v_1\ env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash\ a_1\ /\ a_2 \rightarrow_a v}$	where $v = v_1 \div v_2$
[MODULO$_{BSS}$]	$\dfrac{env_V, sto \vdash a_1 \rightarrow_a v_1\ env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash\ a_1\ \%\ a_2 \rightarrow_a v}$	where $v = v_1\ \mod v_2$

Table 6.2: Big-step transition rules for arithmetic expressions

6.6 Boolean Expressions

Next up are our transition rules for Boolean expressions. Here we use the evaluation of an arithmetic expression as part of the evaluation of our Boolean expression. Namely our [NUM$_{BSS}$] rule is needed in order to map an arithmetic expression a to a value v. In our language, a Boolean expression evaluates a truth value (either tt or ff). Our formation rules for Boolean expressions for HLMP are as follows:

$$b ::= a_1 == a_2 \mid a_1 != a_2 \mid a_1 < a_2 \mid a_1 > a_2 \mid !b_1 \mid b_1 \| b_2 \mid b_1 \&\& b_2 \mid (b_1)$$

Table 6.3 contains our big-step transition rules for Boolean expressions. Each expression evaluates to a truth value. Therefore, we need a way to express both possible outcomes of the evaluation. For this reason, we have both a $TRUE$ and a $FALSE$ transition rule for our Boolean expressions, the only exception being the [PARENTHESES$_{BSS}$] rule.

$[\text{EQUAL}_{BSS}]_{TRUE}$	$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \; env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 = a_2 \rightarrow_b tt}$	if $v_1 = v_2$
$[\text{EQUAL}_{BSS}]_{FALSE}$	$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \; env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 = a_2 \rightarrow_b ff}$	if $v_1 \neq v_2$
$[\text{GREATER}_{BSS}]_{TRUE}$	$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \; env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 > a_2 \rightarrow_b tt}$	if $v_1 > v_2$
$[\text{GREATER}_{BSS}]_{FALSE}$	$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \; env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 > a_2 \rightarrow_b ff}$	if $v_1 \not> v_2$
$[\text{LESS}_{BSS}]_{TRUE}$	$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \; env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 < a_2 \rightarrow_b tt}$	if $v_1 < v_2$
$[\text{LESS}_{BSS}]_{FALSE}$	$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \; env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 < a_2 \rightarrow_b ff}$	if $v_1 \not< v_2$
$[\text{NEGATION}_{BSS}]_{FALSE}$	$\frac{env_V, sto \vdash b \rightarrow_b tt}{env_V, sto \vdash !b \rightarrow_b ff}$	
$[\text{NEGATION}_{BSS}]_{TRUE}$	$\frac{env_V, sto \vdash b \rightarrow_b ff}{env_V, sto \vdash !b \rightarrow_b tt}$	
$[\text{OR}_{BSS}]_{FALSE}$	$\frac{env_V, sto \vdash b_1 \rightarrow_b ff \; env_V, sto \vdash b_2 \rightarrow_b ff}{env_V, sto \vdash b_1 \,\|\| \, b_2 \rightarrow ff}$	
$[\text{OR}_{BSS}]_{TRUE}$	$\frac{env_V, sto \vdash b_i \rightarrow_b tt}{env_V, sto \vdash b_1 \,\|\| \, b_2 \rightarrow_b tt}$	$i \in \{1,2\}$
$[\text{PARENTHESES}_{BSS}]$	$\frac{env_V, sto \vdash b_1 \rightarrow_b v}{env_V, s : to\,(b_1) \rightarrow_b v}$	$v \in \{tt, ff\}$
$[\text{AND}_{BSS}]_{TRUE}$	$\frac{env_V, sto \vdash b_1 \rightarrow_b tt \; env_V, sto \vdash b_2 \rightarrow_b tt}{env_V, sto \vdash b_1 \,\&\&\, b_2 \rightarrow_b tt}$	
$[\text{AND}_{BSS}]_{FALSE}$	$\frac{env_V, sto \vdash b_i \rightarrow_b ff}{env_V, sto \vdash b_1 \,\&\&\, b_2 \rightarrow_b ff}$	$i \in \{1,2\}$

Table 6.3: Big-step transition rules for Boolean expressions

6.7 Statement and Expressions

Next are our transition rules for statements and expressions. Here we need to utilize small-step semantics, since we want them to be able to transition to a non-terminal configuration. Since we are dealing with a run-time stack environment, we need a way to define the active part that is in execution at the given time. This

is called the evaluation context. We use the notation `active` S `end` to mark the evaluation context.
Our formation rules are as follows:

$$S ::= \texttt{active}\ S\ \texttt{end} \mid x := e \mid S_1; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{call}\ P_n() \mid \texttt{call}\ P_n(x_1 ... x_n) \mid D_{pin} \mid D_f \mid D_p \mid D_v \mid \texttt{skip}$$

$$e ::= a \mid b \mid \texttt{call}\ f_n() \mid \texttt{call}\ f_n(x_1 ... x_n)$$

Tables 6.4 and 6.5 show our small-step transition rules for statements and expressions. The declarations D_{pin}, D_f, D_p and D_v will be described later in section 6.8.

[SKIP$_{SSS}$]	$\texttt{skip}, sto, env_T \Rightarrow sto, env_T$
[COMP-1$_{SSS}$]	$\dfrac{\langle S_1, sto, env_T\rangle \Rightarrow \langle S_1', sto', env_T'\rangle}{\langle S_1; S_2, sto, env_T\rangle \Rightarrow \langle S_1'; S_2', sto', env_T'\rangle}$
[COMP-2$_{SSS}$]	$\dfrac{\langle S_1, sto, env_T\rangle \Rightarrow \langle sto', env_T'\rangle}{\langle S_1; S_2, sto, env_T\rangle \Rightarrow \langle S_2', sto', env_T'\rangle}$
[ASSIGN$_{SSS}$]	$x := a, \langle sto, env_T\rangle \Rightarrow (sto[l \to v], env_T)$ where $env_T\ (env_V, env_S) : env_T'$ and $l = env_V\ x$ *and* $env_V, sto \vdash a \to v$
[IF$_{SSS}$] *TRUE*	$\langle \texttt{if}\ b\ \texttt{then}\ S, sto, env_T\rangle \Rightarrow \langle S, sto, env_T\rangle$ if $env_V, sto \vdash b \to tt$ where $env_T = (env_V, env_S) : env_T'$
[IF$_{SSS}$] *FALSE*	$\langle \texttt{if}\ b\ \texttt{then}\ S, sto, env_T\rangle \Rightarrow \langle sto, senv_T\rangle$ if $env_V, sto \vdash b \to ff$ where $env_T = (env_V, env_S) : env_T'$
[IF-ELSE$_{SSS}$] *TRUE*	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ sto, env_T\rangle \Rightarrow \langle S_1, sto, senv_T\rangle$ if $env_V, sto \vdash b \to tt$ where $env_T = (env_V, env_S) : env_T'$
[IF-ELSE$_{SSS}$] *FALSE*	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ sto, env_T\rangle \Rightarrow \langle S_2, sto, senv_T\rangle$ if $env_V, sto \vdash b \to ff$ where $env_T = (env_V, env_S) : env_T'$
[WHILE$_{SSS}$]	$\langle \texttt{while}\ b\ \texttt{do}\ S, sto, env_T\rangle \Rightarrow$ $\langle \texttt{if}\ b\ \texttt{then}\ S\ (S; \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ skip, sto, env_T\rangle$

Table 6.4: Small-step transition rules for statements

[CALL-PROC$_{SSS}$]	$\langle \texttt{call}\ p,\ sto, env_T\rangle \Rightarrow \langle \texttt{active}\ S\ \texttt{end},\ sto, env'_T\ :\ env_T\rangle$ where $env'_S\ p = (env_S, env_V)$ and $env_T = (env_S, env_V) : env'_T$
[CALL-PROC-RE$_{SSS}$]	$\langle \texttt{call}\ p,\ sto, env_T\rangle \Rightarrow$ $\langle \texttt{active}\ S\ \texttt{end},\ sto, env'_T[p \mapsto S, env'_V, env'_S]\ :\ env_T\rangle$ where $env'_S\ p = (env_S, env_V)$ and $env_T = (env_S, env_V) : env'_T$ and $S = (p, env_S, env_V)$
[CALL-PROC$_{SSS}$] *PARAMETER*	$\langle \texttt{call}\ p(x_1 ... x_n),\ sto, env_T\rangle \Rightarrow$ $\langle \texttt{active}\ S\ \texttt{end},\ sto, env'_T\ :\ env_T\rangle$ where $env'_S\ p = (env_S, env_V)$ and $env_T = (env_S, env_V) : env'_T$
[CALL-PROC-RE$_{SSS}$] *PARAMETER*	$\langle \texttt{call}\ p(x_1 ... x_n),\ sto, env_T\rangle \Rightarrow$ $\langle \texttt{active}\ S\ \texttt{end},\ sto, env'_T[p \mapsto S, x_1 ... x_n, env'_V, env'_S]\ :\ env_T\rangle$ where $env'_S\ p = (env_S, env_V)$ and $env_T = (env_S, env_V) : env'_T$ and $S = (p, env_S, env_V)$
[CALL-FUNC$_{SSS}$]	$\langle \texttt{call}\ f\ return\ x,\ sto, env_T\rangle \Rightarrow \langle \texttt{active}\ S\ \texttt{end}\ return\ x,\ sto, env'_T\ :\ env_T\rangle$ where $env'_S\ f = (env_S, env_V)$ and $env_T = (env_S, env_V) : env'_T$
[CALL-FUNC-RE$_{SSS}$]	$\langle \texttt{call}\ f\ return\ x,\ sto, env_T\rangle \Rightarrow$ $\langle \texttt{active}\ S\ \texttt{end}\ return\ x,\ sto, env'_T[f \mapsto S, env'_V, env'_S]\ :\ env_T\rangle$ where $env'_S\ f = (env_S, env_V)$ and $env_T = (env_S, env_V) : env'_T$ and $S = (f, env_S, env_V)$
[CALL-FUNC$_{SSS}$] *PARAMETER*	$\langle \texttt{call}\ f(x_1 ... x_n)\ return\ x,\ sto, env_T\rangle \Rightarrow$ $\langle \texttt{active}\ S\ \texttt{end}\ return\ x,\ sto, env'_T\ :\ env_T\rangle$ where $env'_S\ f = (env_S, env_V)$ and $env_T = (env_S, env_V) : env'_T$
[CALL-FUNC-RE$_{SSS}$] *PARAMETER*	$\langle \texttt{call}\ f(x_1 ... x_n)\ return\ x,\ sto, env_T\rangle \Rightarrow$ $\langle \texttt{active}\ S\ \texttt{end}\ return\ x,\ sto, env'_T[f \mapsto S, x_1 ... x_n, env'_V, env'_S]\ :\ env_T\rangle$ where $env'_S\ f = (env_S, env_V)$ and $env_T = (env_S, env_V) : env'_T$ and $S = (f, env_S, env_V)$

Table 6.5: Small-step transition rules for statements

6.8 Declarations

Finally comes our big-step transition rules for declarations. Table 6.6 shows our transition rules for declarations. We have four declarations which are D_v, D_f, D_p and D_{pin}. Their formation rules are as follows:

$$
\begin{aligned}
D_v &::= \textbf{Var}\ x := e;\ D_v \mid \epsilon \\
D_f &::= \texttt{func}\ F_n()\ \texttt{is}\ S;\ \texttt{returns}\ e;\ D_f \mid \\
&\qquad \texttt{func}\ F_n(\ x_1 ... \ x_n)\ \texttt{is}\ S;\ \texttt{returns}\ e;\ D_f \mid \epsilon \\
D_p &::= \texttt{proc}\ P_n()\ \texttt{is}\ S;\ D_p \mid \texttt{proc}\ P_n(\ x_1 ... \ x_n)\ \texttt{is}\ S;\ D_p \mid \epsilon \\
D_{pin} &::= P_{in}\ Pin_n\{N_p, M_p\};\ S \mid \epsilon
\end{aligned}
$$

As previously mentioned, HLMP contains both functions and procedures, with the only difference being the lack of a return value for procedures. This is also seen in their formation rules. For both procedures and functions, we have made formation and transition rules, with and without formal parameters. Functions and procedures can have either zero, one or multiple parameters. We denote one or multiple parameters in the formation rules by $x_1 ... x_n$. The body of a function or a procedure contains S which makes it possible, to construct nested functions. This is because our declarations are contained within our statements. Now the body of a function or procedure can be part of another function- or procedure declaration.

[FUNC-DCL$_{BSS}$] *TRUE*	$\dfrac{env_V \vdash \langle D_f, env_F[F \mapsto (x, env_V, env_F)]\rangle \rightarrow_{D_f} env'_F}{env_F \vdash \langle \texttt{func}\ (x_1, ..., x_n)\ \texttt{returns}\ e\rangle \rightarrow_{D_f} env'_F}$
[FUNC-DCL$_{BSS}$] *FALSE*	$env_V, \vdash \langle env_F, \epsilon\rangle \rightarrow_{D_f} env_F$
[PROC-DCL$_{BSS}$] *TRUE*	$\dfrac{env_V \vdash \langle D_p, env_P[P \mapsto (x, env_V, env_P)]\rangle \rightarrow_{D_p} env'_P}{env_P \vdash \langle \texttt{proc}\ (x_1, ..., x_n)\rangle \rightarrow_{D_p} env'_P}$
[PROC-DCL$_{BSS}$] *FALSE*	$env_V, \vdash \langle env_P, \epsilon\rangle \rightarrow_{D_p} env_P$
[VAR-DCL$_{BSS}$]	$\dfrac{\langle D_V, env''_V, sto[l \mapsto v]\rangle \rightarrow_{D_v} (env'_V, sto')}{\langle \textbf{Var}\ x = e;\ D_V, env_v, sto\rangle \rightarrow_{D_v} (env'_V, sto')}$ where $env_V, sto \vdash e \rightarrow_a v$ and $l = env_V\ next$ and $env''_V = env_V[x \mapsto l][next \mapsto new\ l]$
[VAR-EMPTY$_{BSS}$]	$env_V, \vdash \langle env_V, sto, \epsilon\rangle \rightarrow_{D_v} (env_V, sto)$
[PIN-DCL$_{BSS}$]	$\dfrac{\langle D_{pin}, env''_V, sto[l \mapsto v]\rangle \rightarrow_{D_{pin}} (env'_V, sto')}{\langle P_{in}\ Pin_n\{N_p, M_p\};\ S,\ env_v,\ sto\rangle \rightarrow_{D_{pin}} (env'_V, sto')}$ where $env_V, sto \vdash Pin_n \rightarrow_a v$ and $env_V, sto \vdash N_p \rightarrow_a v$ and $env_V, sto \vdash M_p \rightarrow_a v$ and $l = env_V\ next$ and $env''_V = env_V[x \mapsto l][next \mapsto new\ l]$

Table 6.6: Big-step transition rules for declarations

6.9 Type System

We now move on to define a type system for our programming language. By using a type system, we can avoid various run-time errors. Considering types will allow us to avoid "nonsense" expressions. A simple example would be an addition expression attempting to add a *Number* and a *Boolean* value. Expressions like this would not produce any well-defined output. To avoid this, we need a notion for types. With a type system, we can define a rule for a sum expression that states when we add two operands together, they both must be of the same type. Such a rule lets us discover this error at compile-time instead of run-time.[20]

The classification of data at compile-time will also assist in better memory allo-

cation and better efficiency. This is especially desired when working with microcontrollers, that have limited memory at their disposal. Another reason for introducing types is to improve readability of a program. It is often beneficial for the reader, if the program's variables are labeled according to their type and if its functions are labeled according to the return values.[20]

6.9.1 Our Type System

Our type system is partially defined by our syntactic categories and their formation rules defined in section 6.3. However, to define our type system, we first need to introduce a definition of the syntactic category of *types*. Table 6.7 shows the definition of our syntactic category of types ranged over by T.

Syntactic category for types		
$T \in$ **Types**	-	*types*

Table 6.7: Syntactic category for types

With the added syntactic category for types, we also need to revisit our formation rules and add type information to our variable, pin, function, and procedure declarations. The formation rules for our type system with the added annotation for types is shown by T.

Our new syntactic category for types also needs to have formation rules. To construct our formation rules for types, we need to know our base types which will be ranged over by B. We show our updated formation rules for our syntactic categories with added formation rules for types below.

$$
\begin{aligned}
a &::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \mid a_1 \% a_2 \mid (a_1) \\
b &::= a_1 == a_2 \mid a_1 != a_2 \mid a_1 < a_2 \mid a_1 > a_2 \mid !b_1 \mid b_1 \| b_2 \mid b_1 \&\& b_2 \mid (b_1) \\
e &::= a \mid b \mid \texttt{call}\ f_n(x) \\
S &::= \texttt{active}\ S\ \texttt{end} \mid x := e \mid S_1; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \\
&\quad \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{call}\ P_n(x) \mid D_{pin} \mid D_f \mid D_p \mid D_v \mid \texttt{skip} \\
D_v &::= T\ \textbf{Var}\ x := e;\ D_v \mid \epsilon \\
D_f &::= \texttt{func}\ F_n()\ \texttt{is}\ S;\ \texttt{returns}\ \{e, S\};\ D_f \mid \\
&\quad \texttt{func}\ F_n(T\ x_1 ... T\ x_n)\ \texttt{is}\ S;\ \texttt{returns}\ \{e, S\};\ D_f \mid \epsilon \\
D_p &::= \texttt{proc}\ P_n()\ \texttt{is}\ S;\ D_p \mid \texttt{proc}\ P_n(T\ x_1 ... T\ x_n)\ \texttt{is}\ S;\ D_p \mid \epsilon \\
D_{pin} &::= T\ P_{in}\ T\ Pin_n\{T\ N_p, T\ M_p\};\ S \mid \epsilon \\
B &::= \texttt{Num} \mid \texttt{Bool} \mid \texttt{Pin} \\
T &::= B \mid x : \texttt{B} \rightarrow \texttt{ok}
\end{aligned}
$$

B ranges over our base types being the `Num`, `Bool` and `Pin` type. In our type system, we consider our `Pwm` type as a part of the `Num` type, since they are both numeral values. The `Pin` type contains all types related to our pin declaration. As is seen in our formation rules, our pin declaration contains four individual types, each contained within the `Pin` type. The four types are:

- P_{in} evaluated as type $\texttt{Pin}_t$
- Pin_n evaluated as type `Pinname`
- N_p evaluated as type `NumP`
- M_p evaluated as type `ModeP`

The type of our statements and declarations will be `ok`, meaning that the statement is well-typed.

T ranges over our syntactic category of types. B is, therefore, a subset of T, as is seen in the formation rules. The composite type $x : \texttt{B} \rightarrow \texttt{ok}$ is used for the evaluation of our procedures and functions and states that if we know the type of the formal parameter B, the procedure or function will be well-typed.

6.9.2 Type Environments

We define a type environment that keeps track of our needed types. The type environment is the partial function env_t defined as:

$$env_t : \mathbf{Var} \cup \mathbf{Fname} \cup \mathbf{Pname} \cup \mathbf{Pin} \cup \mathbf{Pinname} \cup \mathbf{NumP} \cup \mathbf{ModeP} \rightharpoonup \mathbf{Types}$$

We annotate an updated state in our type environment env_t' as $env_t[x \mapsto T]$. This update is defined by:

$$env_t'(y) = \begin{cases} env_t(y) & \text{if, } y \neq x \\ T & \text{if, } y = x. \end{cases}$$

We now move on to describe the rules that define how types are assigned to our syntactic categories. This is done for each syntactic category.

6.9.3 Type Rules for Expressions

This subsection details the type rules for both our Boolean and arithmetic expressions. Each type rule contains a type judgement of the form $env_t \vdash e : T$. A simple example of this is our type rule [NUMBER-EXPR] with the type rule $env_t \vdash n$: `number`. This type rule is read as: n has type `number`, given the type bindings of type environment env_t.

For the type rules, we check that if we get two metavariables with the same type we say what type it should be evaluated to. For example with the minus expression we get two metavariables e_1 and e_2 and check that both have the `Num` type, then we say that $e_1 - e_2$ should also be a `Num` type.

[VAR-EXPR]	$\frac{env_t(x) = T}{env_t \vdash v : T}$
[NUMBER-EXPR]	$env_t \vdash n : \texttt{Num}$
[PARENTHESES $-$ $EXPR$]	$\frac{env_t \vdash e_1 = T}{env_t \vdash (e_1) : T}$
[MINUS-EXPR]	$\frac{env_t \vdash e_1 : \texttt{Num}\, env_t \vdash e_2 : \texttt{Num}}{env_t \vdash e_1 - e_2 : \texttt{Num}}$
[PLUS-EXPR]	$\frac{env_t \vdash e_1 : \texttt{Num} \vdash env_t\, e_2 : \texttt{Num}}{env_t \vdash e_1 + e_2 : \texttt{Num}}$
[MULTIPLY-EXPR]	$\frac{env_t \vdash e_1 : \texttt{Num} \vdash env_t\, e_2 : \texttt{Num}}{env_t \vdash e_1 * e_2 : \texttt{Num}}$
[DIVISION-EXPR]	$\frac{env_t \vdash e_1 : \texttt{Num} \vdash env_t\, e_2 : \texttt{Num}}{env_t \vdash e_1 / e_2 : \texttt{Num}}$
[MODULO-EXPR]	$\frac{env_t \vdash e_1 : \texttt{Num} \vdash env_t\, e_2 : \texttt{Num}}{env_t \vdash e_1 \% e_2 : \texttt{Num}}$
[EQUAL-EXPR]	$\frac{env_t \vdash e_1 : T\, env_t \vdash e_2 : T}{env_t \vdash e_1 == e_2 : \texttt{Bool}}$
[AND-EXPR]	$\frac{env_t \vdash e_1 : \texttt{Bool}\, env_t \vdash e_2 : \texttt{Bool}}{env_t \vdash e_1 \,\&\&\, e_2 : \texttt{Bool}}$
[OR-EXPR]	$\frac{env_t \vdash e_1 : \texttt{Bool}\, env_t \vdash e_2 : \texttt{Bool}}{env_t \vdash e_1 \| e_2 : \texttt{Bool}}$
[GREATER-EXPR]	$\frac{env_t \vdash e_1 : \texttt{Num}\, env_t \vdash e_2 : \texttt{Num}}{env_t \vdash e_1 > e_2 : \texttt{Bool}}$
[LESS-EXPR]	$\frac{env_t \vdash e_1 : \texttt{Num}\, env_t \vdash e_2 : \texttt{Num}}{env_t \vdash e_1 < e_2 : \texttt{Bool}}$
[NEGATION-EXPR]	$\frac{env_t \vdash e_1 : \texttt{Bool}}{env_t \vdash !e_1 : \texttt{Bool}}$
[FUNC-CALL-EXPR]	$\frac{env_t \vdash F_n : (x : T \rightarrow \texttt{ok})\ env_t \vdash e : T}{env_t \vdash \texttt{call}\, F_n(e)\ : \texttt{ok}}$

Table 6.8: Type rules for arithmetic- and Boolean expressions

6.9.4 Type Rules for Declarations

Table 6.9 shows our type rules for all declarations in our language.

[EMPTY-DCL]	$env_t : \epsilon \rightarrow \texttt{ok}$
[VAR-DCL]	$\dfrac{env_t[x \rightarrow T] \vdash D_v : \texttt{ok}\ env_t \vdash e : T}{env_t \vdash var\ T\ x := e;\ D_V\ : \texttt{ok}}$
[PIN-DCL]	$\dfrac{env_t \vdash P_{in} : \texttt{Pin}_t,\ Pin_n : \texttt{Pinname},\ N_p : \texttt{NumP},\ M_p : \texttt{ModeP}}{env_t \vdash D_{pin} : \texttt{ok}}$
[PROC-DCL]	$\dfrac{env_t[p \mapsto (x\ :\ t \rightarrow \texttt{ok})] \vdash D_p : \texttt{ok}}{env_t \vdash proc\ p(T\ x)\ is\ S;\ D_p : \texttt{ok}}$
[FUNC-DCL]	$\dfrac{env_t[f \mapsto (x\ :\ t \rightarrow \texttt{ok})] \vdash D_f : \texttt{ok}\ env_t \vdash e : T}{env_t \vdash func\ f(T\ x)\ is\ S\ returns\ e, S;\ D_f\ :\ \texttt{ok}}$

Table 6.9: Type rules for declarations

Let us shortly consider our [PIN-DCL] type rule. This type rule has four premises, combined into $env_t \vdash P_{in} : \texttt{Pin}_t,\ Pin_n : \texttt{Pinname},\ N_p : \texttt{NumP},\ M_p : \texttt{ModeP}$. This premise states that each sub-type of `Pin` is evaluated to their corresponding type. Under this premise, D_p is well typed.

6.9.5 Type Rules for Statements

Table 6.10 shows our type rules for our remaining statements.

[ASSIGN-STMT]	$\dfrac{env_t \vdash x : T \; env_t \vdash e : T}{env_t \vdash x := e : \texttt{ok}}$
[SKIP-STMT]	$env_t \vdash \texttt{skip} : \texttt{ok}$
[COMP-STMT]	$\dfrac{env_t \vdash S_1 : \texttt{ok} \; env_t \vdash S_2 : \texttt{ok}}{env_t \vdash S_1 : S_2 : \texttt{ok}}$
[IF-STMT]	$\dfrac{env_t \vdash b : Bool \; env_t \vdash S : \texttt{ok}}{env_t \vdash \texttt{if}\; b \;\texttt{then}\; S : \texttt{ok}}$
[IF-ELSE-STMT]	$\dfrac{env_t \vdash b : Bool \; env_t \vdash S_1 : \texttt{ok} \; env_t \vdash S_2 : \texttt{ok}}{env_t \vdash \texttt{if}\; b \;\texttt{then}\; S_1 \;\texttt{else}\; S_2 : \texttt{ok}}$
[WHILE-STMT]	$\dfrac{env_t \vdash b : Bool \; env_t \vdash S : \texttt{ok}}{env_t \vdash \texttt{while}\; b \;\texttt{do}\; S : \texttt{ok}}$
[PROC-CALL-STMT]	$\dfrac{env_t \vdash P_n : (x : T \rightarrow \texttt{ok}) \; env_t \vdash e : T}{env_t \vdash \texttt{call}\; P_n(e) \; : \texttt{ok}}$

Table 6.10: Type rules for statements

Now that we have established our syntax and semantics, we move on to describe different ways of translating code from our language to Arduino C.

Compilers & Interpreters

This chapter will discuss the difference between compilers and interpreters.

Before a program can run on a computer, it must be translated into machine code. This is usually done through either a compiler or an interpreter, both with their benefits and drawbacks. Table 7.1 shows the main differences between compilers and interpreters.

Interpreter	Compiler
Translates the program one statement at a time	Translates the entire program into machine code through a single scan
Faster analysis of program, but slower execution	Slower analysis of program, but faster execution time
Very memory effecient, since no Object Code is generated	Less memory effecient, since it generates Object Code

Table 7.1: Main differences between compilers and interpreters

As can be seen, each method presents certain benefits for different use cases. The following sections will discuss both of these methods further. This discussion will result in a decision on the translation method for our language.

7.1 Compilers

A compiler is a program that converts human-readable programming language code into object code[1]. The compiler scans the whole source program to translate it into machine code. Figure 7.1 shows a simple way of viewing the compilation process.

[1] a set of instruction codes that a computer understands at the lowest hardware level[19]

Figure 7.1: A simple view of a compilation process[19]

The process is a bit more complicated than the figure shows. Throughout the compilation phase, the compiler performs multiple steps[17]:

- Syntax analysis
- Contextual analysis
- Code generation

The syntax analysis will produce a syntax tree that contains the grammatical structure of the given source program. The contextual analysis will produce a decorated syntax tree, containing type information of the nodes of the syntax tree. Lastly, the code generation will generate object code that the target machine can understand[19]. The different steps of a compiler will be discussed in chapter 7.5.

7.2 Interpreters

An interpreter is a program that, as its name implies, interprets a given source program without translation. The interpreter uses a virtual machine to simulate a machine that uses high-level language program statements instead of the typical machine code. This has the advantage of making run-time errors easily visible since the source code is not translated. Thus, an error message that shows precisely where the error occurred in the source program can be produced[17]. Figure 7.2 shows a simple diagram of the interpretation process.

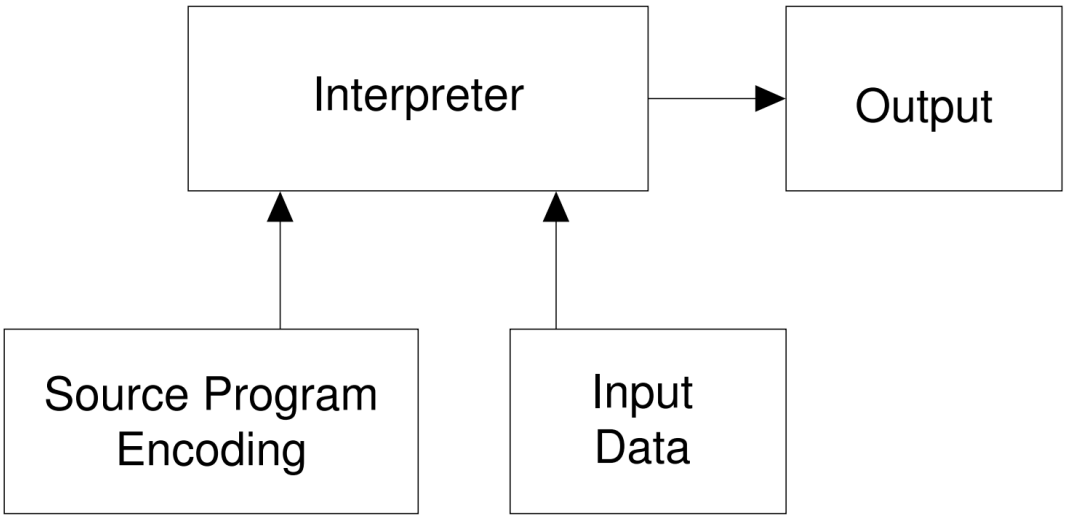

Figure 7.2: A simple view of an interpretation process[19]

However, despite the great error message production, the interpreter has some significant drawbacks. The interpretation method is upwards of 100 times slower in execution than a compilation method. This is mainly because of the decoding of high-level language statements. Even though the number of statements in the high-level code may be fewer than in equivalent machine code, the complexity of those statements means that the interpreter has to make sense of sometimes very complex statements, which takes far longer than a compiler[17].

Interpretation of a program will also require more memory than if the program was compiled.

As discussed, both compilers and interpreters come with their pros and cons. However, there is a third option of language translation. We will now discuss a ***hybrid implementation*** *of the two methods.*

7.3 Hybrid Implementation

A hybrid implementation of compilers and interpreters is a compromise between the two methods. The first part of the method draws inspiration from the compilation method by running the source code through a lexical and syntax analyzer that produces the lexical units and parse tree for the intermediate code generator, thus generating the intermediate code[2]. This intermediate code is then run through an interpreter[17].

[2]middle-level language code

The use of the intermediate code makes this hybrid translation method faster than the pure interpretation method. This is because interpreting the intermediate language is far easier than interpreting a high-level language. The source code is only decoded once, as opposed to pure interpretation, where the source code has to be decoded every time it has to be interpreted[17].

7.4 HLMP's Translation Process

HLMP is an extension of Arduino C, which is a compiled language. This choice of translation makes sense when considering the small amount of storage available on an Arduino. The low processing power of Arduino's makes compilation a more reasonable option of translation, since compiled code is more efficient and has faster execution time. Thus it does make more sense to compile rather than interpret.

*We have chosen to favor **compilation** rather than interpretation for our language.*

7.5 The Phases of a Compiler

We will now describe the different phases of a compiler.

7.5.1 Syntax Analysis

The syntax analysis phase of a compiler consists of two parts being the scanner and the parser.

The Scanner

The scanner condenses the source program into a stream of tokens and eliminates unnecessary information, where each token represents an instance of some terminal symbol. The scanner is responsible for reading the given input as a continuous string of characters called lexemes. While reading, the scanner groups lexemes into tokens like identifiers or reserved words. These tokens are then used by the parser. The scanner can also begin the production of symbol tables. Here, the scanner would enter information regarding the tokens found during the scan of the source program into the symbol table [19].

The Parser

The parser is responsible for reading the given stream of tokens and grouping them based on the rules given by the CFG. This is done to check whether the syntax of the source program is correct. The parser will therefore produce a syntax error

message if any errors are found [19].

The parser recognizes the syntactic structure of the source program. This will usually result in the construction of a syntax tree. This tree will then be used during the compiler's contextual analysis [19].

There are different approaches to parsing, both in the way a parse is performed and whether a parse is performed once or several times. Both of these concepts will be discussed in the following chapters.

7.5.2 Contextual Analysis

During the contextual analysis phase, the previously generated syntax tree is traversed. This is done to perform semantic analysis. The compiler will use a **type checker** to check the static semantics of each node in the given syntax tree. This check verifies that each node is legal within the context of the semantics. It also checks that types are correctly annotated according to the type system of the language and that used identifiers are declared beforehand. Should a semantic error be discovered, the type checker will produce an error message. If the type checker verifies the semantic correctness of a node, that node will be decorated by adding type information to it. Furthermore, as the semantic analysis discovers the actual meaning of the nodes in the syntax tree, operators, and values may be replaced with their actual meaning instead of being a symbol [19].

Another task performed during contextual analysis would be **scope resolution**. Here, the scopes of variables and identifiers in a program are determined. The scope rules of a language regulate the visibility of identifiers. If an identifier has been used where it is not visible, the scope resolution will catch the error [19].
While contextual analysis checks the static semantics of a program, it is unable to check dynamic semantics also known as run-time semantics. As the name implies, these are the semantics that occurs during the run-time of a program. Thus, only static semantics can be evaluated during compilation.

Upon completion of the syntax and contextual analysis, the compiler is ready for code generation.

7.5.3 Code Generation

Code generation is often the final phase of a compiler, although some compilers have post-code generation optimization. The generated code is object code of some lower-level language that resides below the source language in the language stack. The lower-level language is required to at minimum carry the meaning of the

source language and should be efficient in terms of processor usage and memory management. To accomplish this, the code generator should have an understanding of the target machine's run-time environment and instruction set [19]. This allows the code generator to intelligently decide on the order in which instructions are executed. The optimal order of operations will result in better performance.

Another way that the compiler can achieve better target code performance is by performing code optimization.

7.5.4 Code Optimization

Code optimization is an optional compilation step, either performed after the code generation phase or after the completion of the contextual analysis. This depends on whether the code optimization step is machine-dependent or machine-independent.

Machine-dependent optimization is done after the target code is generated and when the code is transformed according to the target machine architecture. This form of optimization can involve CPU register allocation and may have absolute memory references rather than relative ones. This is done to take advantage of the memory architecture and hierarchy of the target machine. The machine-dependent optimization can also allocate memory registers to the most frequently used variables in the program to speed up data retrieval and reduce run-time[19].

An optimization available to both the machine-dependent and independent optimization is the peephole optimization. This is the recognition of program patterns that could be rewritten to produce faster code[19]. For example by replacing an if statement whose condition will always be true with the statement in the true branch.

Machine-independent optimization takes intermediate code and attempts to make it more efficient by transforming sections of code. This can be done in several ways, **dead code elimination** being one of them. Dead code is any code that is never executed and thus can be removed without affecting the rest of the program. Another optimization is to identify **common subexpressions** that can be eliminated. An expression that occurs often in a program and is evaluated each time could be eliminated to reduce the waste of resources and time. The common expression can thus be replaced with an already evaluated variable containing that common value[19].

The code optimization phase is optional, and thus programs can be compiled and run without it. Code optimization comes with some significant benefits, as has

been discussed here. However, compilation time will increase with the addition of the code optimization step.

7.5.5 Error Recovery

A compiler typically has some error recovery. Error recovery lets the compiler continue parsing a source program, even if an error is found. There are four types of errors that the compiler can encounter. The four types of errors are lexical, syntactical, semantical, and logical errors. For each type of error, there are four common strategies of error recovery.

Error productions are used for common errors known by the designers of the compiler. Error productions recognize common errors by adding productions to a CFG that recognizes and sometimes corrects errors [21].
Global correction looks at a given source program in its entirety and attempts to figure out what the program is supposed to do to then produce the closest match without errors [22].
Panic mode is a simple form of error recovery that skips input tokens until it reaches the end of the erroneous statement. Then it continues to read the given source program [19].
Statement mode attempts to ensure that the remaining input code can be parsed. This process can involve modifying the parse stack and remaining input. However, this can lead to further errors if the designers of the compiler are not careful when defining the corrections.[19]

Now that we have covered the typical phases of a compiler, we move on to consider concrete decisions for our compiler. The following chapter will discuss one-pass and multi-pass compilers.

7.6 Compiler Types

A **pass** in terms of a compiler is a way to describe the compiler's traversal through a program. Upon completing a pass, the intermediate program is created (possibly represented as an abstract syntax tree (AST)). In the case of multi-pass compilers, the intermediate program is used in the subsequent pass.

The following sections will discuss one-pass and multi-pass compilers.

7.6.1 One-pass Compilers

A one-pass compiler only passes through the source code once, meaning that no code optimization is performed. This makes the one-pass compilers compilation

process faster than multi-pass compilers. The lack of code optimization being done means that the compilation process is relatively simple and makes the process faster[23].

Fewer compiler passes mean that fewer potential errors are caught. The lack of code optimization and the small number of compiler passes are the major drawbacks of one-pass compilers.

7.6.2 Multi-pass Compilers

A multi-pass compiler passes through the source code multiple times. Each pass is executed by using the intermediate language given by the previous pass. This further optimizes the final code.

Since the final output of a multi-pass compiler has been passed several times and checked for errors, the resulting machine code is well optimized and as error-free as possible. The quality of the resulting code is, therefore, higher than that of a one-pass compiler[23].

The drawback of a multi-pass compiler is the compilation time. Since it passes several times, the compilation process is far slower than a one-pass compiler[23].

We have chosen to utilize a multi-pass compiler in this project. The details and benefits of this choice will become apparent in chapter 8.

There are two main strategies when performing a pass. These will be discussed in the following chapter.

7.7 Parsing Strategies

The task of a parser is to check the syntactical correctness[3] of a program based on the syntax of the programming language. If the parser finds any syntactical errors, it will produce a diagnostic message. Afterward, the parser will return to a normal state. By returning to a normal state, the parser can continue with the analysis of the program, resulting in the parser being able to find as many errors as possible in a single analysis run. The second task of a parser is to create parse trees. Traversing these trees generates the basis for translating a given program. A parser takes an input consisting of a stream of tokens of a program and produces

[3]The notion that certain words, word forms, and syntactic structures meet the standards and conventions

an output of a sequence of grammar rules [17].

Generally, there are two main categories of parsing algorithms. Top-down and bottom-up. The names of these two categories refer to how they build their parse trees.

7.7.1 Top-down Parsing

A top-down parsing strategy constructs its parse tree starting from the root of the parse tree down towards the leaves. The tree is traced in the same order it was built, starting from the root, visiting each node before the node's branches are traced. The branches of a given node are traced left-to-right, corresponding to a leftmost derivation [17].

The top-down parser begins at the start symbol of a given grammar. It then tries to expand leaf nodes from left to right, constantly matching against the given input string. If a match fails, the parser will continue to try alternatives until the input string is matched entirely [19]. This method is known as backtracking and is found in recursive-descent parsers.

There also exists top-down parsers without backtracking. With this strategy, the parser uses the information available in the part of the input string it is trying to match to figure out how to further the parse tree [19]. This type of top-down parsing is known as non-recursive descent parsing.

Top-down parsers belong in the LL category of parsing strategies. The first L states that the parser scans the given input left-to-right. The second L states that a leftmost derivation is generated by the parser [17].

In theory, top-down parsers are not as powerful as bottom-up parsers. However, because of their simplicity, performance, and excellent error diagnostics, top-down parsers have been quite attractive [19]. We now move on to discuss bottom-up parsers.

7.7.2 Bottom-up Parsing

A bottom-up parsing strategy constructs its parse tree from the leaves upwards towards the root and uses the rightmost derivation method. Thus, the parse tree is built from the bottom in a right-to-left manner [17]. Given an input string, a bottom-up parser will attempt to work backward toward the tree's root. This is done by figuring out which right-hand-side (RHS) rule of the grammar must be

reduced to its left-hand-side (LHS) rule.

Bottom-up parsers are also called shift-reduce algorithms, which refer to the two main actions that a bottom-up parser performs [17]. The shift action places the next input token on top of the parser's stack. The reduce action takes an RHS rule and reduces it to its corresponding LHS rule. These rules are applied continuously until the parse tree is fully constructed.

Most bottom-up parsers are classified as LR parsers. The L refers to the fact that the parser is left recursive. The R states that the parser generates a rightmost derivation in reverse. There are three advantages to LR parsers. Firstly, they are buildable for all programming languages. Also, they can detect syntax errors as soon as possible in a left-to-right scan. Lastly, the class of LR grammars is a proper superset of the class that is parsable by LL parsers, which means that many left recursive grammars are LR, but none are LL.[17]

The only disadvantage of LR parsing is that it is difficult for the parsing table, for a given grammar for a complete programming language, to be produced by hand. However, this is not a serious disadvantage, as there are programs available that take input as grammar and produce the parsing table.[17]

There are three commonly used algorithms used for constructing LR parsers. The first method is called SLR (Simple LR). This algorithm is the fastest LR parsing strategy, but it only works on small sets of grammars. However, it is simple to construct, given its small amount of states. The second algorithm is called LALR (look-ahead LR). This algorithm works on medium-sized grammars and has the same amount of states as an SLR strategy. The final algorithm is the LR algorithm. It works on full-sized LR(1) grammars, but is the slowest of the three algorithms, given its large amount of states [17].

There are several tools available for parsing a program. In the following chapter, we will discuss popular parsing tools. Our choice of parsing strategy will be discussed in section 7.8.3.

7.8 Compiler Tools

In this chapter, we explain and discuss different widely used compiler tools to make a decision on which tool should be used in our project.

7.8.1 Java Cup

Java Cup is a parser generator that produces a parser written in Java. The input grammar for this parser generator has to be of type LALR(1)[4]. The output of Java Cup includes a Java source file named parser.java, which defines a class named parser and a method named parse. Java Cup also generates another Java source file named sym.java, containing a class named sym, which declares one public final static int for each terminal declared in the Java Cup specification. Java Cup does not have a built-in lexical analyzer generator. However, a tool named JFlex is often used to provide this lexical analyzer.[24]

7.8.2 JavaCC

JavaCC is a popular parser generator written in Java. The parser generator reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. The generated parser is a top-down recursive descent parser. Top-down parsers have multiple advantages compared to bottom-up parsers. Easier debugging, having the ability to parse any non-terminal in the grammar and being able to pass attributes both up and down the parse tree while parsing.
In most cases, JavaCC generates an LL(1) parser. However, when some grammar is not LL(1), JavaCC generates an LL(k) parser instead to offer the capability of syntactic and semantic lookahead to resolve shift-shift ambiguities locally at these points. The LL(1) will always be prioritized when possible, as it has better performance than LL(k). Additionally, JavaCC also provides tree building with a tool named JJTree included with JavaCC.[25]

7.8.3 ANTLR

ANTLR[5] is a parser generator tool written in Java. ANTLR is used for generating parsers and scanners for several target languages. It accepts any CFG[6] that is not indirect or hidden left-recursive. Indirect left-recursive rules calls themselves through another rule.[7] A rule is hidden left-recursive if an empty production exposes left recursion.[8] ANTLR generates a recursive descent parser, which uses an ALL(*)[9] parsing strategy. While generating the parser, ANTLR generates a scanner, which creates the necessary amount of look-ahead tokens. ANTLR can also be used for building and traversing parse trees.[26]

[4]Look-Ahead, Left-to-right, Rightmost derivation in reverse
[5]ANother Tool for Language Recognition
[6]Context free grammar
[7]For example; $A \rightarrow B, B \rightarrow A$.
[8]For example; $A \rightarrow BA, B \rightarrow \varepsilon$.
[9]Adaptive Left-to-right Leftmost derivation (*)

Ultimately, we have chosen ANTLR as our parse generator since ANTLR is a complete solution in terms of having a built-in lexical analyzer generator, a parse generator, and an option to build a parse tree. JavaCC also provided this. However, it proved to be more tedious to learn and use when compared to ANTLR. ANTLR was the tool that was easiest for us to become familiar with. We want to dedicate most of our time to implement the compiler. Therefore, ANTLR is a great option as it works right out of the box. Lastly, ANTLR has an efficient and powerful parsing strategy, which we will now have a closer look at.

ALL(*) Parsing

As mentioned before, ANTLR uses an ALL(*) parsing algorithm.

The generated ALL(*) parser is usable for any CFG without indirect or hidden left-recursion. ALL(*) combines the simplicity, efficiency, and predictability of conventional top-down LL(k) parsers with the power of a GLR[10]-like mechanism to make parsing decisions. The simplicity of the top-down parser comes from the way the parser operates. The parser constructs a parse tree, starting from the root node at the top and then moving step-by-step down to the leaf nodes as grammar tokens are read [26]. The power of a GLR parser lies in the way it operates. A GLR parser uses the rightmost form to create right-associative trees and work from the leaves to the root. This might be less intuitive than working from the root towards the leaves. However, it is a much more powerful way to parse, and unlike top-to-bottom parsing, it can handle left- and right recursion.

Shifting grammar analysis to parse-time caching analysis results in a lookahead DFA is a critical innovation for efficiency. This is another advantage of the ALL(*) parsing strategy. Therefore, no static grammar analysis is needed. In this way, the undecidability of static LL(*) grammar analysis is avoided, and correct parsers can be generated for any non-left-recursive CFG [26].

With our chosen compiler tool, we now move on to our implementation process. During this process, we also cover the phases of our compiler.

[10]Generalized Left-to-right Rightmost derivation

Implementation

This chapter will discuss a subset of the implementation details for the HLMP compiler. We have omitted the syntax analysis phase of our compiler from this chapter since most of this phase is implemented by the ANTLR compiler tool.

The general structure of our compiler is discussed first, followed by our handling of symbols and scopes. This is an interesting part of our compiler because of our nested functions and procedures which are a core feature of our compiler. Next, we discuss our type checking since this subject is a central part of the compiler. We also discuss exception handling and we connect it with ANTLR. Lastly, we cover our code generation. Here our translation process from HLMP to Arduino C produces some interesting cases because of our nested functions and scopes.

Throughout this chapter, we show code examples where we replace any irrelevant code with three dots to reduce the clutter of our examples. Also, when discussing both functions and procedures in one instance, they will be referred to as subroutines.

8.1 General Structure

Our compiler is written in Java and consists of three phases, syntax analysis, contextual analysis, and code generation, as illustrated in figure 8.1. As previously mentioned, we have chosen to use ANTLR for our syntax analysis. ANTLR can throw syntax errors when given a grammar and input code. In this way, we only have to handle the errors ANTLR throws in the syntax analysis phase. ANTLR also provides us with a parse tree, that we utilize in the contextual analysis phase. Here we first create a symbol table for each scope, where the symbols[1] are then inserted. This allows us to check for undefined variables and declarations of already declared variables. Next, we utilize our symbol table and parse tree to perform type checking. Finally, when we have checked the code for errors, we reach the

[1]Everything that can be found in a scope i.e variable declarations, statements, etc.

code generation phase, where we create the C code for the Arduino.

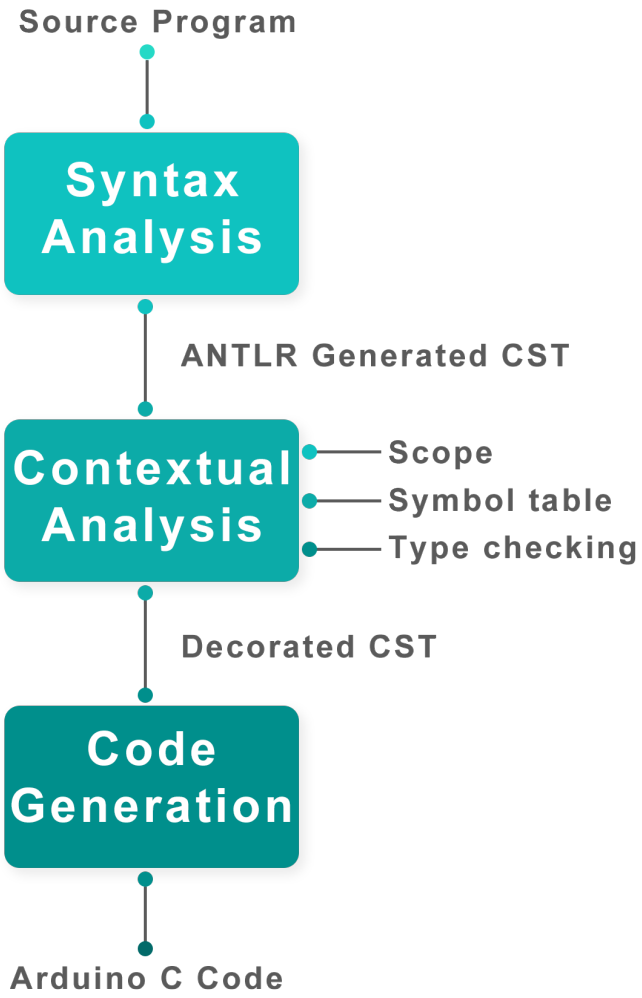

Figure 8.1: The three phases in our compiler; syntax analysis, contextual analysis, and code generation.

We will now explain the different phases of the compiler in more detail.

8.2 Symbol Table and Scope

8.2.1 Symbol Table Listener

When creating scopes and inserting symbols into them, we need to traverse the parse tree. ANTLR provides two ways of traversing the parse tree, `Listener` and `Visitor`. They both take advantage of the visitor design pattern. Essentially, in the

`Visitor`, each time a node is visited, it returns a generic type and combines it with the other returned generic types in a bottom-up fashion using the parse tree. For the `Listener` it only visits the note without returning anything. For symbol table and scope generation, we use the `Listener` because we do not want to change the parse tree as we traverse it. Therefore, we find it more efficient to utilize the listener. We have created the class `SymbolTblListener` that inherits from the ANTLR generated `Listener`. By using a parse tree walker, ANTLR will call the methods `Enter` and `Exit` for each node in the parse tree. By overriding these methods from ANTLR, we can create our own implementation of the `Enter` and `Exit` methods.

Code block 8.1 shows an example of the `Enter` and `Exit` methods for `FuncDefinition`. On line 3, we create a new symbol to represent the function. We have created classes to represent different symbols in our language. Here, we use the symbol for function definition and give the symbol an identifier(id) through its constructor. The id is later used as a key for finding the symbol in a dictionary data structure. Afterward, on lines 4 and 5, we set the return type of the function and add the symbol to the symbol table. On line 6, we call the method `enterScope` with the id and parse tree. When we exit the `FuncDefinition`, we call the method `exitScope` in our symbol table on line 11. In order to explain `enterScope` and `exitScope`, we first have to understand how scopes are represented in our compiler.

```
1  @Override
2  public void enterFuncDefinition(HlmpParser.FuncDefinitionContext ctx)
       {
3      FuncDefSymbol symbol = new FuncDefSymbol(ctx.funcHead().id().
           getText(), makeUniqueId());
4      symbol.setType(ctx.funcHead().type());
5      symbolTbl.addSymbol(symbol);
6      symbolTbl.enterScope(ctx.funcHead().id().getText(), ctx);
7  }
8
9  @Override
10 public void exitFuncDefinition(HlmpParser.FuncDefinitionContext ctx) {
11     symbolTbl.exitScope();
12 }
```

Code block 8.1: Example of `Enter` and `Exit` for `FuncDefinition`.

8.2.2 Scopes and Symbol Table

In our compiler, a scope is represented as an object of the class `Scope`. By doing this, we can keep track of an endless amount of scopes, which enables us to easily represent nested subroutines. Instead of using a list structure or any similar

structure, all scopes can be accessed through their parent scope and subscopes. In code block 8.2, a snippet of the class `Scope` is shown. On lines 3 and 4, we see the `parent` and `subScopes` field variables that are used to access all scopes. The `Scope` class also has a dictionary to keep track of all symbols in the scope. This can be seen on line 2. All scopes also have an id, which is used to ensure that a scope is not created twice. This can be seen on line 5. The id is a unique number, while for subroutines and variables, it is equal to the name of the subroutine or variable.

```
1 public class Scope {
2     Map<String, Symbol> symbolDictionary = new HashMap<String, Symbol
          >();
3     private List<Scope> subScopes = new ArrayList<>();
4     public Scope parent;
5     public String id;
6     ...
7 }
```

Code block 8.2: Snippet of the `Scope` class.

Now that we know the representation of scopes, we can explain the `enterScope` method. The method is seen in code block 8.3, and it is defined in the class `SymbolTbl`. The purpose of this method is to create a new scope and add it to the parse tree. On line 3, we create the new scope. Afterwards, on lines 4 and 5, we ensure that the new scope is added as a subscope and then update the `currentScope`. Lastly, on line 12, we add the scope to the parse tree, as it will be used later in the type checker.

```
1 @Override
2 public void enterScope(String id, ParseTree tree) {
3     Scope scope = new Scope(currentScope, id);
4     currentScope.addSubScope(scope);
5     currentScope = scope;
6     scopesProperty.put(tree, currentScope);
7 }
```

Code block 8.3: The `enterScope` method.

While we are in a scope, we add the symbols we encounter until we exit the scope. Code block 8.4 shows the method for adding a symbol to the current scope. This method is in the `SymbolTbl` class and is called in the `SymbolTblListener` whenever we enter a node that should be added to the symbol table. Before we add the symbol to the scope's dictionary of symbols, we need to check if the symbol already exists. We do this on line 2 in code block 8.4. If it does not exist, we add the symbol on line 3, otherwise, we throw an exception on line 6. We use the `isSymbol`

method on line 2 to achieve this. We will now explain the `isSymbol` method in further detail.

```
1 public void addSymbol(Symbol symbol) {
2     if (!isSymbol(symbol.getId(), currentScope)) {
3         currentScope.addThisSymbol(symbol);
4     }
5     else {
6         throw new AlreadyDeclared(symbol.getId());
7     }
8 }
```

Code block 8.4: The `addSymbol` method.

The `isSymbol` method checks if an id already exists in the symbol table. This method is shown in code block 8.5. The method is called with two parameters being the id we wish to search for and the scope we wish to search in. In code block 8.4, we call the `isSymbol` method with `currentScope` as a parameter. In this way, we check if the id already exists in this scope as seen on line 2 in code block 8.5. Given that all symbols are stored in different scopes, we cannot simply check one scope. Therefore, we also need to check all the parent scopes. We do this on lines 6 and 7, where we first check if the scope has a parent and then recursively call `isSymbol` until there are no more parent scopes. In this way, we check all parent scopes and if the id was found, we return true, otherwise false. The same approach is used for getting a symbol in our scopes dictionary.

```
1  private boolean isSymbol(String id, Scope scope) {
2      if (scope.containsId(id)) {
3          return true;
4      }
5      else {
6          if (scope.parent != null) {
7              return isSymbol(id, scope.parent);
8          }
9          else {
10             return false;
11         }
12     }
13 }
```

Code block 8.5: The `isSymbol` method.

We have now created all scopes and inserted all symbols into them. However, we still need to check if an id is used, but not declared. We have chosen to do this in a separate listener called `DeclarationCheckListener`. In this way, we can go

through the parse tree twice. The first time we go through the parse tree, all the ids are being added to the symbol table. The second time we go through the parse tree, we check for use of undeclared ids. In this way, we do not encounter issues like in C where prototypes for functions can be required. We have chosen to do this because it improves writability when writing larger programs by not having to write prototypes. In the `DeclarationCheckListener` we have `enter` and `exit` methods for all symbols that have an id. Here, we simply check if the id exists in the scope's symbol dictionary with the method `isSymbol` as described earlier.

8.3 Type Checking

Type checking ensures that the syntactic and semantic conventions are followed while also checking the type rules. We have chosen to perform static type checking during compile-time since we are compiling to an Arduino, where run-time resources are limited. We utilize ANTLR's generated visitor for our type checking, where we override the visit methods as needed. We also utilize the ANTLR-generated parse tree to retrieve the relevant context needed for the given type check.

When we call the `visit` method on a node, that node is visited until all return paths are explored. The visitor then combines all return results into one type. This is useful for visiting structures such as expressions composed of other expressions. Instead of receiving a list of those subexpression components, the visitor produces one type that is a combined type of all the expressions. Once the visitor has completed its traversal of the node, we return to the original method, where the visit method was called, and continue onward. In this way, we can get the type of complex expressions by simply calling `visit` on them. The visitor pattern also gives complete control of the traversal order through the parse tree. This will be useful for us when we need to handle a case like expressions, where we need to visit multiple nodes. Therefore, we have chosen to apply ANTLR's visitor pattern instead of a listener. We will now show examples of visitor methods from our type checking.

8.3.1 Variable Declaration

Code block 8.6 shows our type checker for variable declarations. On line 3, we get the type of the variable declaration. Then on line 4, we check if it is of the type `Pwm`. If that is the case, we convert the type to `Num`. We do this because both the types are numbers, and therefore, we treat them as the same type. Because we do not have runtime type checking, we do not check if values used with `Pwm` type are integers between 0-255. This is acceptable because when the `byte` type that is used

for `Pwm` overflows, it just resets from 0. The programmer can take advantage of this to create more compact code. If a float value is inputted to a `byte`, the value will be rounded down, which we see as a sufficient solution. Afterward, on line 6, we check if the type corresponds to the expression of the assignment. If this is not the case, we throw a `TypeException`. Otherwise, we return the type.

```
1  @Override
2  public Integer visitVarDeclarationAssign(HlmpParser.
       VarDeclarationAssignContext ctx) {
3      Integer type = ctx.type().start.getType();
4      if (type == HlmpLexer.PWMTYPE)
5          type = HlmpLexer.NUMTYPE;
6      if (visit(ctx.expr()) == type) {
7          return type;
8      }
9      else {
10         throw new TypeException("In variable declaration assignment -
               \""+ ctx.expr().getText() + "\" is not of type " + ctx.type
               ().getText());
11     }
12 }
```

Code block 8.6: Type checking for variable declarations

8.3.2 Arithmetic Expressions

In code block 8.7, we have an example of type checking in our language. Here we type check our arithmetic expressions. On lines 2 and 3, we visit our operand nodes through the visitor pattern. Then on line 4, we check whether both operands are of type `Num`. This is done by using the lexer. If both operands are type `Num`, we return `NUMTYPE`, otherwise, we throw a `TypeException`.

```
1  public Integer visitExprBinaryFloat(HlmpParser.ExprBinaryNumberContext
       ctx) {
2      Integer left = visit(ctx.left);
3      Integer right = visit(ctx.right);
4      if (left == HlmpLexer.NUMTYPE && right == HlmpLexer.NUMTYPE) {
5          return HlmpLexer.NUMTYPE;
6      }
7      else
8          throw new TypeException("Expected the type: Num");
9  }
```

Code block 8.7: Type checking for arithmetic expressions

8.3.3 Function Definition

In code block 8.8, we show our type checker for function definitions. On line 3, we get the return type of the function and on line 4, we convert the type to `Num` if it is of type `Pwm`. Next, we get the current scope with the current context on line 5. Then, on line 6 we check if the function type corresponds to the body context. If that is the case, we update the current scope on line 7 to the scope stored `scopesProperty` as showed in code block 8.3. If their types do not match, we throw a `TypeException`.

```
1  @Override
2  public Integer visitFuncDefinition(HlmpParser.FuncDefinitionContext
       ctx) {
3      Integer funcType = ctx.funcHead().type().start.getType();
4      funcType = ifPwmConvertToNum(funcType);
5      symbolTbl.updateCurrentScope(ctx);
6      if (funcType == visit(ctx.body())) {
7          symbolTbl.updateCurrentScope(ctx);
8          return defaultResult();
9      }
10     else {
11         throw new TypeException("Expected return type \"" + ctx.
               funcHead().type().getText() + "\" for \"" + ctx.funcHead().
               id().getText() + "\"");
12     }
13 }
```

Code block 8.8: Type checking for function definition

8.3.4 Function Call

Code block 8.9 shows our type checking of a subroutine call. Firstly, we check if the subroutine is a whileWait. If that is the case, we expect two parameters and a comma and return `BOOLTYPE`. Afterward, on line 8, we check if there exist any parameters. If any parameters exist, we visit them. Next, we get the id of the subroutine on line 11. Now, on line 12, we must check if the subroutine is declared, such that we can make the subroutine call. If this is not the case, we throw a `NotDeclared` exception. Lastly, on lines 15 to 22, we return `null` if the return type of the subroutine is `null`. Otherwise, we return the return type of the subroutine call.

```
1  @Override
2  public Integer visitFunctionCall(HlmpParser.FunctionCallContext ctx) {
3      if (ctx.id().getText().equals("whileWait")){
```

```
4           if (ctx.args().children.size() == 3) { // size should be 3
                because ',' is included
5               return HlmpLexer.BOOLTYPE;
6           }
7       }
8       if (ctx.args().children != null) {
9           visit(ctx.args());
10      }
11      FuncDefSymbol symbol = (FuncDefSymbol) symbolTbl.getSymbol(ctx.id
            ().getText());
12      if (symbol == null) {
13          throw new NotDeclared(ctx.id().getText());
14      }
15      else if (symbol.getType() == null) {
16          return null;
17      }
18      else {
19          Integer type = symbol.getType().start.getType();
20          type = ifPwmConvertToNum(type);
21          return type;
22      }
23  }
```

Code block 8.9: Type checking for subroutine call

8.3.5 Actual Parameters

Code block 8.10 shows the type checker for the actual parameters for a subroutine. In this method, we check whether the types of our actual parameters match the types of the formal parameters. On line 3, we create an `List` for our actual parameters. Then on lines 4 to 6, we insert our parameters into the newly created `List`. On line 7, we get the id of the subroutine. In order to do this, we need a node in the same hierarchy that contains the id. Therefore, we need to visit the parent node, in order to traverse to the correct child node.

From lines 8 to 11, we get the symbol for the subroutine that is called. If it does not exists in the symbol table, it will return `null` and throw an exception on line 10. If the id of the subroutine call does exist in the symbol table, we create a new list with our formal parameters on line 12. Then, we perform a check on lines 13 to 15, where we make sure that the amount of actual and formal parameters is the same. From lines 16 to 22, we iterate through the list of formal parameters and check whether their types correspond to the types of the actual parameters. If that is not the case, we throw a `TypeException`. Otherwise, we return the types of the actual parameters.

```java
@Override
public Integer visitArguments(HlmpParser.ArgumentsContext ctx) {
    List<Integer> parametersType = new ArrayList<>();
    for (HlmpParser.ExprContext i : ctx.expr()) {
        parametersType.add(visit(i));
    }
    String id = ctx.getParent().children.get(0).getText();
    FuncDefSymbol symbol = (FuncDefSymbol) symbolTbl.getSymbol(id);
    if (symbol == null) {
        throw new NotDeclared("A function or procedure match could not
             be found for call");
    }
    List<TypeSymbol> parameters = symbol.getParameters();
    if (parametersType.size() != parameters.size()) {
        throw new TypeException("The amount of formal parameters
            supplied does not match the actual parameters");
    }
    for (int i = 0; i < parameters.size(); i++) {
        int type = parameters.get(i).getType().start.getType();
        type = ifPwmConvertToNum(type);
        if (type != parametersType.get(i)) {
            throw new TypeException("Expected parameter type: \"" +
                parameters.get(i).getType().getText()+"\"");
        }
    }
    return parametersType.get(0);
}
```

Code block 8.10: Type checking for subroutine arguments

8.4 Code Generation

We have chosen to utilize ANTLR's visitor pattern during code generation. This lets us override certain visit methods, as we wish. Firstly, we override `defaultResult`, which returns null as default. We want the default value to be an empty string instead. Thus, any visit method not overwritten, will now return an empty string through `defaultResult`. We also override the method `aggregateResult`, which has the strings `aggregate` and `nextResult` as parameters. `aggregate` is the previous aggregate value. The default implementation of the method `aggregateResult` returns `nextResult`, which is the result of the immediately proceeding call to visit a child node. However, we want it to return the `aggregate` value in addition to `nextResult`. This allows us to concatenate the two strings into one string instead of overwriting each string. The overwritten visit methods are shown in code block 8.11.

```
@Override
protected String defaultResult() {
    return "";
}

@Override
protected String aggregateResult(String aggregate, String nextResult){
    return aggregate+nextResult;
}
```

Code block 8.11: Code generation for type

8.4.1 Program

Not all code can be written as it is visited by the `ArduinoGenVisitor`. Therefore, we use field variables to store content that will be used later. These field variables are seen in code block 8.12. Here we have strings on lines 1 to 3 that store the content that should be added later. In code block 8.13 that shows the `visitProgram` method, we can see an example of this. On line 10, we append the `globalContent` field variable after we have visited all `Content`. In this way, we can append code in the Global scope no matter which visit method we are visiting.

```
1 private String setupContent = "";
2 private String globalContent = "";
3 private String topContent = "";
4 List<String> refVarsAddress = new ArrayList<>();
5 List<String> whileWaitsAdded = new ArrayList<>();
```

Code block 8.12: The field variables for `ArduinoGenVisitor`

We use a `StringBuilder` to assemble all the output code in the `visitProgram` method. The `StringBuilder` is seen on line 3 in code block 8.13. First, we append the loop function to the `StringBuilder` on line 4. Then we visit all `ContentContext` nodes and append them on lines 5 to 7. After this, everything except the setup procedure has been visited. The setup procedure code is only appended after everything else has been visited. Because when visiting a `PinLiteral` it stores code in the `setupContent` field variable as we will discuss later. Because of this, we append the setup procedure on line 9 after everything else has been visited. Then `globalContent` is appended and the `topContent` is inserted at the start of the string, to ensure that the Arduino compiler does not give a `not declared` error.

```
1 @Override
```

```
2  public String visitProgram(ProgramContext ctx) {
3      StringBuilder sb = new StringBuilder();
4      sb.append(visit(ctx.standardProc().loopDef()));
5      for (ContentContext c : ctx.content()) {
6          sb.append(visit(c));
7      }
8      resetRefVarsAddress();
9      sb.append(visit(ctx.standardProc().setupDef()));
10     sb.append(globalContent);
11     sb.insert(0, topContent+"\n");
12     return sb.toString();
13 }
```

Code block 8.13: Overriding the `visitProgram` method

8.4.2 Setup Definition

In code block 8.14, we visit the `SetupDefinitionContext`. On line 3, we initialize our `result` string with "void", as the setup procedure does not have a return type. Then, on line 6, we append the field variable `setupContent`, which we elaborated on in the previous section. Next, we visit the body for our setup definition and append it to `result`. Lastly, we return the string.

```
1  @Override
2  public String visitSetupDefinition(SetupDefinitionContext ctx) {
3      String result = "void ";
4      result += "setup";
5      result += "() {";
6      result += setupContent;
7      result += visit(ctx.procBody());
8      result += "}\n";
9      return result;
10 }
```

Code block 8.14: Code generation for setup procedure definition

8.4.3 Pin Literal Definition

When visiting a pin literal definition, we first need to determine which type it is. In code block 8.15, on lines 3 to 7, we switch on the type of the pin literal. Depending on which type it is, we pick the corresponding string to provide. As previously mentioned in HLMP, you will need to specify whether a pin is analog or digital with either an 'A' or a 'D'. But in Arduino C, you either write 'A' for analog or just the literal without the 'D' for digital. On lines 8-11, we make a new string and

add an 'A' if it is an analog pin number. On line 12 we add the pin number to the string except for the first character. In this way, we get the desired pin number for Arduino. On line 13, we add a pin mode specification to the `SetupContent`. Here we can set the pin mode before the other code is run. Initializing a pin literal is only possible in the global scope of our language. Therefore, we add the pin literal definition to `TopContent`, as shown on line 14. Lastly, we return an empty string because the pin definition is already added to `TopContent`.

```
1  @Override
2  public String visitPinLiteralDef(PinLiteralDefContext ctx) {
3      String pinmode = switch (ctx.pinmode().start.getType()) {
4          case HlmpLexer.OUT → "OUTPUT";
5          case HlmpLexer.IN → "INPUT";
6          default → defaultResult();
7      };
8      String pinNum = "";
9      if (ctx.PINNUMBER().getText().charAt(0) == 'A') {
10         pinNum += 'A';
11     }
12     pinNum += ctx.PINNUMBER().getText().substring(1);
13     addSetupContent("pinMode(" + pinNum + "," + pinmode + ");");
14     addTopContent("int " + ctx.id().getText() + " = " + pinNum + ";");
15     return defaultResult();
16 }
```

Code block 8.15: Code generation for pin literal definition

8.4.4 Type

When we generate code for our type `Num`, we will simply generate it as a floating-point number. In chapter 4, we stated that we wished for our type `Num` to be multiple types determined dynamically. However, as it stands now when generating code to Arduino C, it converts to a float. Ultimately, this came down to prioritizing other features of our compiler. Chapter 10 will expand further on our reasoning for this decision.

Code block 8.16 shows an example where we generate code when visiting `Type`. The switch statement shown from line 3 to 8 switches on the type. Depending on which type the context is, the method returns the corresponding string. For example, if the type of `TypeContext` is a `NUMTYPE`, the `visitType` method returns a string which contains `float`.

```
1  @Override
```

```
2  public String visitType(TypeContext ctx) {
3      return switch (ctx.start.getType()) {
4          case HlmpLexer.NUMTYPE → "float ";
5          case HlmpLexer.BOOLTYPE → "bool ";
6          case HlmpLexer.PWMTYPE → "byte ";
7          default → defaultResult();
8      };
9  }
```

Code block 8.16: Code generation for type

8.4.5 Function Definition

Before we look at the code generation for function definition, we need to walk through the code generation for a function head as it is a part of the function definition. Making the `funcProcHead` is done using the helping method `makeFuncProcHead`. This helping method was made because we need to construct heads for both functions and procedures. `makeFuncProcHead` helps us avoid duplicating code. This method will be expanded on later in section 8.4.6.

Code block 8.17 shows the code generation for a function definition. Here, we will build a string containing the entirety of a function definition by visiting its different parts. On line 8, we initialize a string named `result`, where we store the function head of the `FuncDefinitionContext` by visiting `ctx.funcHead()`. Afterward, on line 9, we visit the context for the body and append it to `result`. On line 10, we append a string with a curly bracket to close the function and a new line to improve readability. On line 11, we call `resetRefVarsAddress` to reset the list of parameters that need to be dereferenced, which is used for the parameters of the nested functions. This matter will be elaborated on in subsection 8.4.6. Lastly, we return `result` containing the entire string of a function definition.

```
1  @Override
2  public String visitFuncHead(FuncHeadContext ctx) {
3      return makeFuncProcHead(ctx.type(), symbolTbl.idProperty.get(ctx),
           ctx.parameter(), scope);
4  }
5
6  @Override
7  public String visitFuncDefinition(FuncDefinitionContext ctx) {
8      String result = visit(ctx.funcHead());
9      result += visit(ctx.body());
10     result += "}\n";
11     resetRefVarsAddress();
12     addGlobalContent(result);
```

```
13      return result;
14  }
```

Code block 8.17: Code generation for function definition

8.4.6 Nested Subroutines

Because we have nested subroutines in our language and Arduino C does not, our code generation becomes significantly more complicated. When we convert nested subroutines to a flat block structure[2], we need to move each subroutine to the global scope. When doing this, we have to consider the parameters and local variables. An inner subroutine can access the outer subroutine's variables and parameters. Therefore, we need to add a reference to them, when we move the subroutines to the flat block structure. We also have to transfer the variables and parameters as pointers, because if a value in an inner scope is modified, it also needs to be modified in the outer scope. We have shown this in code blocks 8.18 and 8.19 where the variable on line 2 in code block 8.18 is added as a pointer on line 2 in code block 8.19. This is also done for the parameters. We will now show how this is done in the code generation of our compiler.

```
1  proc func1(Num param1) {
2      Num var1;
3      proc func2(Num param2) {}
4  }
```

Code block 8.18: Example of a nested procedure before translated into Arduino C

```
1  void func1(Num param1) { }
2  void func2(Num param2, Num *param1, Num *var1) {}
```

Code block 8.19: Generated code for a nested procedure in Arduino C

Code block 8.20 shows the method that makes the head of a subroutine. The head consists of the identifier, parameters, and optionally the type. This is handled on lines 2 to 7 in code block 8.20. Afterward, the parameters need to be added. We first add the parameters that should be pointers from variables and parameters from other scopes. On line 8 we call the method `getTypeSymbolsFromScope` in our `symbolTbl` class. This method will return all the `TypeSymbols` from the given scope. The scope is given as a parameter and is the current scope of the subroutine it is in.

[2]When subroutines can only be defined in the global scope, and only two scopes levels exist the Local and the Global scope

On line 9, we then append a comma-separated list of the symbols, which is made using the `addCommaSeperated` method. Afterward, we add a comma-separated list of parameters, which was originally written in the input code for the subroutine. Next, we will explain the `addCommaSeperated` method.

```
1  private String makeFuncProcHead(TypeContext typeCtx, String idCtx,
       List<ParameterContext> parameters, Scope scope) {
2      String result = "void ";
3      if (typeCtx != null) {
4          result = visit(typeCtx);
5      }
6      result += idCtx;
7      result += "(";
8      List<TypeSymbol> symbols = symbolTbl.getTypeSymbolsFromScope(scope
           .parent);
9      result += addCommaSeparated(symbols, makeParamPointer, parameters.
           size() > 0);
10     result += addCommaSeparated(parameters, makeParseTree, false);
11     result += ") {";
12     return result;
13 }
```

Code block 8.20: Code generation for the head of a subroutine definition

The `addCommaSeperated` method, shown in code block 8.21, takes a list and adds the content in a comma-separated sequence. This appears frequently in our code, therefore, we have chosen to make a generic version. The first parameter of `addCommaSeperated` is a generic list. Next, a method is taken as a parameter. This is done through the functional interface named `CommaSeparatedFunc`, which allows us to pass a lambda expression as a parameter to a method. Lastly, a parameter for appending a comma in the end. In code block 8.21, we first check on line 2, if the size of the list is zero. If that is the case, we return `defaultResult`. Otherwise, we call the method parameter with the first element of the list as an actual parameter. Next, we append the rest of the elements, separated by a comma on lines 7 to 10. Lastly, we return the string `result`.

```
1  private <T> String addCommaSeparated(List<T> list, CommaSeparatedFunc
       func, boolean endWithComma) {
2      if (list.size() == 0) {
3          return defaultResult();
4      }
5      String result = "";
6      result += func.invoke(list.get(0));
7      for (int i = 1; i < list.size(); i++) {
8          result += ", ";
9          result += func.invoke(list.get(i));
```

```
10      }
11      if (endWithComma) {
12          result += ", ";
13      }
14      return result;
15  }
```

Code block 8.21: `addCommaSeparated` in code generation

When the `addCommaSeperated` method is used, as we do in code block 8.20 on line 9, we pass a method as a parameter. In this case, we pass the `makeParamPointer` method, which is shown in code block 8.22. The method is a lambda expression, which has the parameter `t`. In this case, `t` is a `TypeSymbol`. Therefore, on line 2, `t` is typecasted as a `TypeSymbol`. Next, on line 3, we add the symbol's identifier to the field variable `refVarsAddress`. This will later be used for dereferencing identifiers in expressions. This will be elaborated on later. Lastly, on line 4 we visit the symbol's type and append its id with a pointer operator in between, which is returned as a string.

```
1  private CommaSeparatedFunc makeParamPointer = (t) → {
2      TypeSymbol symbol = (TypeSymbol) t;
3      refVarsAddress.add(symbol.getId());
4      return visit(symbol.getType()) + "*" + symbol.getId();
5  };
```

Code block 8.22: `addWithPointer` method in code generation

Now that we have added all the formal parameters, we also need to add the address operator to every pointer actual parameter in the subroutine call. When a subroutine call is made, the actual parameters are created by the `visitArguments` method. This is shown in code block 8.23. Firstly, on line 4, we get the id of the subroutine that is called assigned to `calledId`. This is the subroutine's identifier of the arguments we are visiting. Next, on line 5, we get the scope related to this id. This is the subroutine's declaration that is called. On line 6, we get the scope that the subroutine call exists in, which is the current scope assigned to `scope`. However, it is important that we know whether it is an inner or outer scope.

When we call an inner scope, we wish to transfer all the `TypeSymbols` from the current scope and its parent's scope to the called subroutine. In this way, we maintain the scope rules of HLMP. When an outer subroutine is called, we wish to get the `TypeSymbols` of the scope that is called. However, we do not want to include variable declarations, therefore, we utilize the parent's scope for outer calls. Because of these reasons, we must differentiate between the outer and inner scope.

On line 7 in code block 8.23, we call the method `isInnerScope` where we pass the parameters `calledId` and `scope`, in order to determine if the subroutine call is to an outer or an inner scope. This method returns true, if the `calledId` exists in an inner scope. If the call is in an outer scope, the scope is set to the called scope's parent on line 8. This will then be used to get the symbols in that scope. Before that, we check on line 10 if `calledScope` is equal to the current scope, which means a recursive call will be executed. If this is the case, we can simply append a comma-separated list of the `refVarsAddress` using the `addCommaSeperated` method. Otherwise, we get the `TypeSymbols` from the scope.

Now that we know which scope to use, we can add the `TypeSymbols` from that scope. As seen on line 14, we use the `getTypeSymbolsFromScope` method to get all `TypeSymbols`. Then, on lines 16 to 18, we add these symbols to a new list of type string, such that it is easier to utilize on line 19, where the result of the `addCommaSeperated` method is appended to `result`. Lastly, on line 21 we append the remaining parameters from the original subroutine declaration and return `result`. By doing all of this, we add the actual parameters for a subroutine call with the address operators where necessary.

```
1  @Override
2  public String visitArguments(ArgumentsContext ctx) {
3      String result = "";
4      String calledId = ctx.parent.getChild(0).getText();
5      Scope calledScope = symbolTbl.getScope(calledId);
6      Scope scope = symbolTbl.scopesProperty.get(ctx.parent);
7      if (!symbolTbl.isInnerScope(calledId, scope) == null) {
8          scope = calledScope.parent;
9      }
10     if (calledScope.equals(scope)) {
11         result += addCommaSeparated(refVarsAddress, makeActualParam,
               ctx.expr().size() > 0);
12     }
13     else {
14         List<TypeSymbol> symbols = symbolTbl.getTypeSymbolsFromScope(
               scope);
15         List<String> strSymbols = new ArrayList<>();
16         for (TypeSymbol ts: symbols) {
17             strSymbols.add(ts.getId());
18         }
19         result += addCommaSeparated(strSymbols, makeActualParam, ctx.
               expr().size() > 0);
20     }
21     result += addCommaSeparated(ctx.expr(), makeVisit, false);
22     return result;
23 }
```

Code block 8.23: `visitArguments` method in code generation

Finally, whenever a referenced parameter is utilized in an expression, we must dereference it. As mentioned earlier, we added the field variable `refVarsAddress`, which keeps track of the variables that need to be dereferenced. Whenever a variable is used in an expression, we can simply check if the id of that variable exists in the `refVarsAddress`. We do this with the method `shouldDeRef` shown in code block 8.24, where we check on line 3 if the id is contained in `deRefVars`. If that is the case, we can return the dereference operator.

```
1 @Override
2 private String shouldDeRef(String id) {
3     if (refVarsAddress.contains(id)) {
4         return "*";
5     }
6     return defaultResult();
7 }
```

Code block 8.24: `ShouldDeRef` method

Code block 8.25 shows an example of where `shouldDeRef` is used. The method shown is `visitExprOperand`. This method is called whenever an expression operand is visited. This is used when an expression contains an identifier. Here we call the method `shouldDeRef`, with the id as a parameter, before the id is written. In this way, if a referenced parameter is used in an expression, the deference operator is added in front of the id.

```
1  @Override
2 public String visitExprOperand(ExprOperandContext ctx) {
3     return shouldDeRef(ctx.getText()) + super.visitExprOperand(ctx);
4 }
```

Code block 8.25: `visitExprOperand` method in code generation

8.4.7 New Delay/WhileWait

For the code generation for `whileWait`, we need to implement a procedure that can create a delay using `millis`. This procedure needs to be called from where the `whileWait` procedure is. On line 13 in code block 8.26, we create the code that replaces the `whileWait` procedure call. Here we call the new procedure that creates a delay with `millis`. The new procedure's name is a unique identifier, followed

by `whileWait`. In this way, we can have multiple of these procedures for different `whileWaits`. We call the new procedure with a parameter for the amount of time it has to wait. When creating the procedure that creates a delay with `millis`, we first make a string. The string can be seen on line 5 in code block 8.26, where we write a specific Arduino C function that will call the given procedure until the specified time has run out. The given function is called on line 8, where it will return a Boolean value. If the Boolean value is true, the while loop will continue, else the function will return, and the delay will be interrupted.

```
1  @Override
2  public String visitWhileWait(WhileWaitContext ctx) {
3      String id = ctx.id().getText();
4      if (!whileWaitsAdded.contains(id)) {
5          String str = "bool "+symbolTbl.idProperty.get(ctx)+"WhileWait(
               int delayTime) {" +
6           "unsigned long startTime = millis();" +
7           "while (millis() < startTime+delayTime) {" +
8           "if (!"+symbolTbl.getSymbol(id).getUniqueId()+"()) {" +
9           "return true;}}return false;}";
10          addGlobalContent(str);
11          whileWaitsAdded.add(id);
12     }
13     return symbolTbl.idProperty.get(ctx)+"WhileWait(" + visit(ctx.expr
           ()) + ");";
14 }
```

Code block 8.26: `visitValueId` method

8.5 Error Reports

We achieve customizable exception handling by inheriting from the `CancellationException` class. This allows us to create multiple exception classes such as `AlreadyDeclared`, `NotDeclared`, `SyntaxException` and `typeException`. The reason we inherit specifically from the `CancellationException` class, is because ANTLR uses exactly this exception for its visitor. Since we utilize this, we need an exception type compatible with it. In code block 8.27, we override the base visitor method `visitStmtAssign` on line 1 and make our own implementation. Because our exception classes inherited the `CancellationException`, we are able to throw our `TypeException` on line 5. We do not use error recovery, as we have prioritized other features of our compiler. The compiler simply halts and issues a message in the terminal for the programmer.

```
1  @Override
```

```
2 public Integer visitStmtAssign(HlmpParser.StmtAssignContext ctx) {
3     ...
4     else {
5         throw new TypeException("Assignment -\"" + ctx.id().getText()
              + "\" is not of expected type:" +symbol.getType().getText()
              );
6     }
7 }
```

Code block 8.27: Exception handling

Tests

This chapter discusses the testing phase of our project. Our testing consists of three parts, unit tests, integration tests, and lastly user acceptance tests. Each will be explained and discussed in their respective sections.

We use the testing framework **JUnit** for our unit and integration testing. JUnit is a testing framework designed for the Java programming language[27]. The testing framework enables us to easily write unit and integration tests for our compiler instead of writing our tests by hand.

9.1 Unit Testing

We perform unit tests to validate if individual units of our compiler run as expected[28]. Our tests follow the arrange-act-assert structure. In the **arrange** step, we arrange our inputs and targets. During the **act** step, we act on the target behavior. Lastly, we **assert** the expected outcome.

Code block 9.1 contains our first unit test for the `enterScope` method. Here we test if the method creates a new scope. In our arrange step, on lines 4 and 5, we create a new instance of our `SymbolTbl` and a new id that does not exist within the new instance of the `SymbolTbl`. Then, on line 8 in our act step, we call the `enterScope` method in our `SymbolTbl` with our newly created id. This should then create a new `SymbolTbl` object with the same id as the one created on line 5. Lastly, in our assert step seen on line 11, we assert if a new id has been created in the `SymbolTbl` that matches the id from line 5.

```
1     @Test
2     public void enterScope_newId_shouldCreateNewScope() {
3         // Arrange
4         SymbolTbl symbolTbl = new SymbolTbl();
5         String id = "func";
6
```

```
7           // Act
8           symbolTbl.enterScope(id, null);
9
10          // Assert
11          Assertions.assertEquals(id, symbolTbl.currentScope.id);
12      }
```

Code block 9.1: Example of `Enter` and `Exit` for `Scope`.

In code block 9.2 we test the method `getSubScopes`. In our arrange step, on lines 4 to 7, we create three new scopes and a `subScopesList`. Then, on lines 10 and 11 in our act step, we add the two subscopes created on lines 5 and 6, to the scope created on line 4. The subscopes are then added to the `subScopesList` on lines 13 and 14. Now, the `scope` and `subScopesList` should both contain the subscopes previously defined. Lastly, in our assert step, we assert if the `getSubScopes` method corretly gets the added subscopes.

```
1       @Test
2       public void getSubScopes_shouldGet() {
3           // Arrange
4           Scope scope = new Scope(null, null);
5           Scope subScope = new Scope(scope, "subScope");
6           Scope subScope2 = new Scope(scope, "subScope2");
7           List<Scope> subScopesList = new ArrayList<>();
8
9           // Act
10          scope.addSubScope(subScope);
11          scope.addSubScope(subScope2);
12
13          subScopesList.add(subScope);
14          subScopesList.add(subScope2);
15
16          // Assert
17          Assertions.assertEquals(scope.getSubScopes(), subScopesList);
18      }
```

Code block 9.2: Unit test that test if method **getSubsScopes** can return the correct scopes.

9.2 Integration Testing

We perform integration testing to test the interactions between the modules of our compiler and ensure that the modules operate as expected when combined[28].

In code block 9.3, we have the first test for our type checker. This test verifies the expected behavior of our `BinaryFloatComparison`. The annotation `@ParaterimizedTest`

on line 4 is a part of the JUnit framework that allows us to run the same test multiple times with different data sets. The `@ValueSource` on line 5 defines the data sets used for our test. The test is performed from line 7 to line 11. If the `BinaryFloatComparison` functions as expected, the test passes and does not throw any exception.

```
1  public class TypeCheckingTests {
2      private static String setUpLoop = " proc setup() {} proc loop() {}
           ";
3
4      @ParameterizedTest
5      @ValueSource( strings = {"proc test() { Num var1 = 2; Num var2 =
           3; Num var3; var3 = var1 + var2; }"
6                               ," proc test() { Num var1 = 2; Num var2 =
                                    3; Num var3; var3 = var1 + var2; }"})
7      public void BinaryFloatComparison_ShouldPass(String testCode) {
8          Assertions.assertDoesNotThrow(() → {
9              ParseTree tree = DoSyntax(setUpLoop + testCode);
10             DoContextual(tree);
11         });
12     }
```

Code block 9.3: An integration test that test for binary comparison between floats

Code block 9.4 shows the second test of our `BinaryFloatComparison`. The test is very similar to the previous test. However, this time we assert whether an exception is thrown when the `BinaryFloatComparison` is given data sets of mismatched data.

```
1      @ParameterizedTest
2      @ValueSource( strings = {"proc test() { Num var1 = 2; Bool var2 =
           false; Num var3; var3 = var1 + var2; }",
3                               "proc test() { Num var1 = 2; Bool var2 =
                                    3; Num var3; var3 = var1 - var2; }"})
4      public void BinaryFloatComparison_ShouldFail(String testCode) {
5          Assertions.assertThrows(TypeException.class, () → {
6              compile(setUpLoop + testCode);
7          });
8      }
```

Code block 9.4: An integration test that test for binary comparison between floats

9.3 User Acceptance Testing

User acceptance tests (UAT) are the final phase of testing executed after unit tests and integration tests. UATs are performed by the end-user or client to verify that

the program fulfills the goal. The purpose of UAT is to validate end-to-end business flow. It does not focus on cosmetic errors, spelling errors, or system testing.[28]

To perform UAT, we have created three test cases that consist of small programming tasks in the HLMP language. However, due to time constraints, we find it hard to properly test nested functions, as it requires a quite large program to excel. Therefore, in these tests, we have focused on the readability and writability on a smaller scale with the idea of nested functions. This will also be discussed later, in section 10.6. The three test cases are presented below in table 9.1.

ID	Keyword	Description
1	Understand code	Look at the code and explain what is going on.
2	Write code	Write a program that switches a LED on and off every second.
3	Write advanced code	With assistance, write a program that turns on a LED and turns it off every two seconds, if a potentiometer exceeds a certain value, the LED must turn off instantly with the use of the whileWait function.

Table 9.1: The titles and description of the test cases

9.3.1 Results

The test cases have been performed by one test person who is experienced in Arduino programming. We had hoped to have multiple test persons, but due to time constraints, this was not feasible for us.

We first introduced the test person to the features of HLMP, which were needed to perform the tests. During the first test case, the test person had to read and understand a program written in HLMP. This program takes input from a sensor and provides output in the form of an LED that fades or blinks. Here, the focus is on nested subroutines, recursive calls, and `whileWait`. As soon as the test person opened the program, he noticed the nested functions.

Afterward, they stumbled upon `whileWait`, after which a conversation about the function ensued. Here, the `millis` function was also brought up. The test person ended up concluding that the reduced amount of code required to write our `whileWait` procedure compared to `millis`, was desirable.

Finally, during the first test case, the test person expressed their likeness to the idea

of the type `Num`. They added, that it improves the simplicity when working with types.

In the second test case, the test person had to write a program that switches an LED on and off every second. This task was easily completed by the test person, with very few errors. Our compiler informed the them of the error, and the they were able to quickly locate the error. Once the error was fixed, the code was compiled and run with no errors.

For the last test case, the test person was given the skeleton of a program, where they had to fill out the remaining code to make the program work. The test person easily managed to solve the assignment. Once the assignment was completed, we entered a discussion where we brought up the Arduino C code, generated by their written code. The test person was surprised by the extra amount of code needed to write an equivalent program in Arduino C.

The results of our UAT will be discussed in section 10.6.

Discussion

In this chapter, we discuss the fulfillment of the evaluation criteria, and if HLMP meets the expectations from chapter 4.5, where we introduced criterias such as readbility and writability and how to meet these criterias. Furthermore, we will discuss the challenges of the project and what consequences they have.

10.1 Data Types

In section 4.2, we specified five new data types: `Num`, `String`, `Pin`, `Pwm`, and `Bool`, which we wanted to add. These would add simplicity and then increase readability and writability.

We ended up not implementing the `String` type. When considering the usefulness of the `String` type, we saw it as the least used type in Arduino C. Strings are sometimes used to display text on a external screen but can also be used to write to the serial monitor. Our language does not support the serial monitor because it is only used for debugging and setup. Therefore, we did not prioritize this. When connecting a display to a Arduino, a library is almost always used, which our language does not support. While it is possible to use a display without a library it is very difficult, so it is not commonly done. Therefore, without a serial monitor and libraries, `String` is not useful. For these reasons, we chose to prioritize `Num`, `Bool`, `Pwm` and `Pin` types, which are all used more frequently.

The `Num` data type has been implemented, resulting in the reduction of numeric data types from ten to one. This change adds more simplicity to HLMP, thus improving the overall read- and writability. Our `Num` currently only encompasses floats. As was stated in section 4.2.1, we initially wished for the `Num` type to be a dynamic type that was able to switch its internal type based on the given value. The decision of which type to choose, would be up to the compiler. As the project progressed, we realized that we did not have enough time to implement this dynamic version of `Num`. Therefore, we chose to represent `Num` as a floating-point

number. While this outcome is not ideal, we believe that we still managed to capture the essence of a high-level numeric type. The lack of adaptiveness of `Num` does result in worse memory management, which is less than ideal when working with Arduinos that have limited storage. However, it is easier to handle for the programmer when using our language. Chapter 12 will discuss this dynamic `Num` type further.

We have added the new data type `Pin` for specifying a pin. This data type adds more simplicity to the configuration of specific pins. Now, the pin initialization is performed in a single line of code, instead of defining the pin number and the pin mode separately. This was also shown in chapter 4.

We have implemented the `Bool` data type, which improves the overall syntax design and readability. We have removed the ability to represent Boolean values as numeric values, in order to create a better distinction between the types in HLMP. Now boolean values are represented only by `true` and `false`. This change increases the simplicity of HLMP, which in turn provides better readability, writability, and reliability.

Lastly, we have added the new data type `Pwm`. This type was introduced and discussed in section 4.2. `Pwm` was implemented in HLMP to better mimic the real-world usage of the components connected to the Arduino. Furthermore, the addition of `Pwm` lets us avoid unnecessary conversions, when working with analog- and PWM signals.

10.2 WhileWait

In section 4.4, we discussed a new `delay` function to our language. As was described, the new function intended to provide better read- and writability of the original `delay` function. Delays are common in Arduino sketches, and therefore we found it important to provide a better alternative. The implementation of this new function was covered in section 8.4.7. We believe that we have achieved the goal of providing a better alternative to the `delay` function by allowing for tasks to be performed while the delay is happening, without having to write extensive amounts of code like is the case with the `millis` function.

10.3 Functions and Procedures

We have introduced procedures and functions in HLMP to improve the writability of larger scaled programs for Arduinos. Having both functions and procedures is intended to give the programmer a clear distinction between working with or

without return values. We expect using procedures to be easy for any seasoned programmer, considering that the concept of procedures is familiar to most computer scientists. We also saw the ease of use of procedures during our usage tests, where our test person quickly adapted to procedures.

10.4 Nested Scopes

The addition of nested scopes provides a better structure to Arduino programs, enables access control, and abstraction. This is very helpful when working with bigger Arduino projects. We determined that nested scopes would fulfill the missing encapsulation capabilities the way Arduino C does not provide.

10.5 Syntax and Semantics

We revisited our syntax several times as our language design changed. These revisits resulted in the syntax development phase taking much longer than we anticipated. This added development time also extended to the development of our semantics. These revisits of our syntax and semantics gave us a deeper understanding of both topics, which we believe resulted in a better end product for our syntax and semantics.

Working with semantics has given us a deeper understanding of how structural operational semantics work. In this regard, we have especially gained a better grasp of the environment concept and small-step semantics. Our semantics also provided a better understanding and foundation for the implementation of our language. We are satisfied with the result of our syntax and semantics development, even if the time spent was more than expected.

10.6 Testing Phase

This section discusses the testing phase and our experiences while performing it. We will consider the code coverage of our unit tests and integration tests and finally, we will conclude with the outcome of the UAT.

During our unit- and integration testing phase, we achieved a code coverage of 85.11%. This number only represents the code we have written ourselves and excludes any ANTLR-generated code. Ultimately, 75 was the total amount of unit tests and integration tests. Some segments in the compiler are either trivial or similar, so we did not find it necessary to cover them.

The results of our UAT shows a tendency towards a positive reception of HLMP. However, given that we only performed one UAT, it must remain inconclusive whether the real-world read- and writability is a significant improvement over the Arduino C language. Ideally, multiple test persons would be preferred to verify this. Additionally, the tests were in a non-to semi-structured fashion. Ideally, we wanted to be better prepared when performing these tests. Ultimately, time constraints were the issue.

Conclusion

In this project, we set out to combat the poor scalability of Arduino C. Chapter 2 discussed the challenges with the language. Among the challenges found in Arduino C, were the flat function structure and the problematic `delay` function. We especially saw issues regarding Arduino C's function structure. This resulted in our problem definition, which is:

How can we design and implement a programming language based on Arduino C with better structure and support for abstraction to provide better read- and writability when writing larger Arduino programs?

Chapter 4 introduced our proposed solution. At the end of the chapter, in section 4.6, we described how our proposed solution would combat the shortcomings of Arduino C. Here we argued that we would be able to achieve better readability and writability compared to the Arduino C language, in large part due to our introduction of nested functions and procedures, as well as our `whileWait` function. As was discussed in chapter 10, we have successfully implemented our proposed solution. We believe that this has resulted in improved read- and writability, as was also displayed through the examples given in chapters 2 and 4.

We also introduced a rework to the traditional data types found in Arduino C to provide more simplicity and higher-level programming. Section 4.2 discussed the ideas of our reworked data types, and section 10.1 discussed the outcome of our implementation of those data types. Here we also concluded that we achieved our goal of providing higher-level data types.

During our testing phase, we investigated whether HLMP had fulfilled its purpose of improving Arduino's readability and writability through user acceptance testing. The results from the test indicates a significant improvement. However, as we only had one test person, it must remain inconclusive that HLMP is an actual improvement compared to Arduino C. It does show a tendency that it is an im-

provement, which could be proven with further testing.

With the implementation of our proposed solution, we conclude that we have achieved a satisfying result and reached our goal for this semester's project.

Future Work

This chapter discusses functionalities we would have liked to implement if we had more time. The functionalities discussed here had a lower priority than those we implemented.

12.1 Data Types

As discussed earlier, we wished to make `Num` a dynamic type that would change its datatype based on the given value. We believe that this addition would enable better memory management. However, to check for a potential change in the datatype of `Num`, we would have to run a check function each time the value of a variable changes. If given more time, we would have explored this new `Num` type further.

12.2 Code Optimization

We briefly mentioned code optimization in chapter 7.5. Here we explained both the pros and cons of optimization. In this project, we have not made any code optimization. While this lack of an optimization step reduces compile time, we feel that the benefits would outweigh the drawback of the increased compile time. Code optimization would increase the efficiency of a running program. Since an Arduino typically only compiles once and then runs indefinitely, the increased compile-time would not be a significant disadvantage.

12.3 Error Recovery

In section 8.1, we introduced the different phases of a compiler. Where we also described different ways of achieving error recovery. We currently do not have any error recovery in our compiler, which means it simply stops the compilation process once an error is found. If given more time, we would like to add error recovery to the compiler. Error recovery lets the compiler catch more potential errors, which

results in more errors being reported to the user. This would reduce the amount of compilation attempts, before all errors are caught. We imagine using panic mode as our choice of error recovery. This is due to the simple nature of panic mode, and in extension the safety it provides.

Error productions might also make sense to incorporate. With error productions, we are able to ensure the needed elements of a program is inserted by the compiler. This can come in the form of type annotations or the insertion of the mandatory `setup` and `loop` procedures.

Statement mode attempts to modify the remainder of the input code, which can result in infinite loops if the attempt to correct the input goes poorly. Therefore, if we were to implement error recovery using statement mode, we would have to ensure that the corrections made, would not lead to further errors.

Bibliography

1. *About arduino* https://www.arduino.cc/en/Guide/Introduction. Last visited 21/3/2022.
2. Arduino. *setup()* https://www.arduino.cc/reference/en/language/structure/sketch/setup/. Last visited 28/2/2022.
3. Arduino. *loop()* https://www.arduino.cc/reference/en/language/structure/sketch/loop/. Last visited 28/2/2022.
4. *Digital Pins* https://www.arduino.cc/en/Tutorial/Foundations/DigitalPins. Last visited 08/4/2022.
5. Arduino. *attachInterrupt()* https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/. Last visited 28/2/2022.
6. *Arduino Hardware* https://www.arduino.cc/en/hardware. Last visited 08/4/2022.
7. *delay()* https://www.arduino.cc/en/reference/delay. Last visited 23/3/2022.
8. *millis()* https://www.arduino.cc/en/Reference/Millis. Last visited 23/3/2022.
9. *PinMode()* https://www.arduino.cc/en/Reference/PinMode. Last visited 24/3/2022.
10. *Digital Pins* https://www.arduino.cc/en/Tutorial/Foundations/DigitalPins. Last visited 24/3/2022.
11. Arduino. *Digital Pins* https://docs.arduino.cc/learn/microcontrollers/digital-pins. Last visited 18/5/2022.
12. WolMat. *Pinmode needed??* https://forum.arduino.cc/t/pinmode-needed/495648. Last visited 20/5/2022.
13. *Digital Write* https://www.arduino.cc/reference/en/language/functions/digital-io/digitalwrite/. Last visited 21/3/2022.
14. *Analog Write* https://www.arduino.cc/reference/en/language/functions/analog-io/analogwrite/. Last visited 21/3/2022.
15. *Pulse-width modulation* https://en.wikipedia.org/wiki/Pulse-width_modulation. Last visited 09/5/2022.

16. *Arduino get started analogWrite()* https://arduinogetstarted.com/reference/analogwrite. Last visited 08/4/2022.

17. Sebesta, R. W. *Concepts of Programming Languages* ISBN: 9781292100555 (Pearson Education Limited, 2016).

18. *Nested function* https://en.wikipedia.org/wiki/Nested_function. Last visited 16/3/2022.

19. Charles N. Fischer Ron K. Cytron, R. J. L. J. *Crafting a Compiler* ISBN: 9780136067054 (Pearson Education Limited, 2010).

20. Hüttel, H. *Transitions and Trees* ISBN: 9780521197465 (Cambridge University Press, 2010).

21. C. N. Fischer, J. M. *On the role of error productions in syntactic error correction* https://www.sciencedirect.com/science/article/pii/0096055180900065. Last visited 23/5/2022.

22. Tutorialspoint. *Compiler Design - Error Recovery* https://www.tutorialspoint.com/compiler_design/compiler_design_error_recovery.htm. Last visited 23/5/2022.

23. *Compiler Design: Introduction* https://medium.com/an-introduction-to-compiler-design/compiler-design-introduction-38aea30cd8c8. Last visited 06/4/2022.

24. *Java Cup* https://pages.cs.wisc.edu/~fischer/cs536.s06/course.hold/html/NOTES/4a.JAVA-CUP.html. Last visited 24/3/2022.

25. *JavaCC* https://javacc.github.io/javacc/. Last visited 24/3/2022.

26. *Adaptive LL(*) Parsing: The Power of Dynamic Analysis* https://www.antlr.org/papers/allstar-techreport.pdf. Last visited 23/3/2022.

27. team, J. *JUnit 5* https://junit.org/junit5/. Last visited 19/5/2022.

28. Stephens, R. *Beginning Software Engineering* ISBN: 8126555378 (John Wiley & Sons Inc., 2015).

Appendix

13.1 Grammar

```
grammar Hlmp;
//Parse Rules

program: standardProc content* EOF
       | content* standardProc content* EOF
       | content* standardProc EOF;

standardProc: setupDef loopDef
            | loopDef setupDef content;

content: funcProc
       | varDecl END
       | pinLiteral END
       | comment;


funcProc: funcHead LBRACE body RBRACE
        | procHead LBRACE procBody RBRACE;

funcHead: FUNC type id LPAREN (parameter (',' parameter)*)? RPAREN;
procHead: PROC id LPAREN (parameter (',' parameter)*)?  RPAREN;

parameter: type id;

type: NUMTYPE | BOOLTYPE | PWMTYPE;

body: (funcProc body)?
    | stmt body
    | comment body
    | (returnExpr END)+ body;

procBody: (funcProc procBody)?
        | stmt procBody
        | comment procBody;

```

```
stmt: varDecl END
    | id ASSIGN expr END
    | whileWaitCall END
    | waitCall END
    | funcCall END
    | writeFunc END
    | readFunc END
    | ifStmt elseStmt?
    | whileExpr;

pinLiteral: PINTYPE id LBRACE PINNUMBER ',' pinmode RBRACE;

varDecl: type id
       | type id ASSIGN expr;

expr: operand
    | readFunc
    | LPAREN expr RPAREN
    | op=NEG expr
    | left=expr op=(DIVIDE|MULT) right=expr
    | left=expr op=(PLUS|MINUS|MODULU) right=expr
    | left=expr op=(LESSTHAN|GREATERTHAN) right=expr
    | left=expr op=(EQUAL|NOTEQUAL) right=expr
    | left=expr op=LOGAND right=expr
    | left=expr op=LOGOR right=expr;

operand: id
       | sfloat
       | bool
       | funcCall;

readFunc: id READPWM LPAREN RPAREN
        | id READA LPAREN RPAREN
        | id READD LPAREN RPAREN;

returnExpr: RETURN expr;

whileWaitCall: WHILEWAIT LPAREN expr ',' id RPAREN;

waitCall: WAIT LPAREN expr RPAREN;

funcCall: id LPAREN args RPAREN;
args: (expr (',' expr)*)?;

writeFunc: id WRITE LPAREN val RPAREN;

val: TRUE
   | FALSE
   | sfloat
   | id
   | TOGGLE;
```

```
87
88  ifStmt: IF LPAREN expr RPAREN LBRACE body RBRACE;
89
90  elseStmt: (ELSE LBRACE body RBRACE)?;
91
92  whileExpr: WHILE LPAREN expr RPAREN LBRACE body RBRACE;
93
94  setupDef: PROC SETUP LPAREN RPAREN LBRACE procBody RBRACE;
95  loopDef: PROC LOOP LPAREN RPAREN LBRACE procBody RBRAC;
96
97  comment: COMMENT
98         | LINECOMMENT;
99
100 pinmode: OUT | IN;
101 bool: TRUE | FALSE;
102 sfloat: INT | SFLOAT | (NEGATIVE)?INT;
103 id : ID;
104
105 //Lexer Rules
106 INT: [0-9]+;
107
108 NUMTYPE: 'Num ';
109 BOOLTYPE: 'Bool ';
110 PWMTYPE: 'Pwm ';
111 PINTYPE: 'Pin ';
112
113 NEG: '!';
114 PLUS: '+';
115 MINUS: '-';
116 DIVIDE: '/';
117 MULT: '*';
118 MODULU: '%';
119 LESSTHAN: '<';
120 GREATERTHAN: '>';
121 EQUAL: '==';
122 NOTEQUAL: '!=';
123 LOGAND: '&&';
124 LOGOR: '||';
125
126 WHILEWAIT: 'whileWait';
127 WAIT: 'wait';
128 SETUP: 'setup';
129 LOOP: 'loop';
130 FUNC: 'func ';
131 PROC: 'proc ';
132
133 SFLOAT: (NEGATIVE)? FLOAT;
134 FLOAT: [0-9]+'.'[0-9]+;
135
136 WRITE: '.Write';
137 READPWM: '.ReadPwm';
```

```
138 READA: '.ReadAnalog';
139 READD: '.ReadDigital';
140
141 LPAREN: '(';
142 RPAREN: ')';
143 LBRACE: '{';
144 RBRACE: '}';
145 END: ';';
146 ASSIGN: '=';
147 NEGATIVE: '~';
148
149 TOGGLE: 'toggle';
150 TRUE: 'true';
151 FALSE: 'false';
152 IN: 'in';
153 OUT: 'out';
154
155 IF: 'if';
156 ELSE: 'else';
157 RETURN: 'return';
158 WHILE: 'while';
159 PINNUMBER: 'D' [0-9]+ | 'A' [0-9]+;
160
161 COMMENT: '/*' .*?  '*/' -> skip;
162 LINECOMMENT: '//' ~( '\r' | '\n' )* -> skip;
163
164 WS: (' '|'\t'|NEWLINE)+ -> skip;
165 NEWLINE: ('\r\n'|'\n'|'\r');
166
167 ID: [a-zA-Z_] [a-zA-Z_0-9]*;
```

Code block 13.1: Grammar of HLMP