0.1 Reading guide

Hi Radu, we are now ready for you to make the final corrections for the following parts:

- Tombstone Diagram - part 2.2
- Parsing - part 2.3
- Design criteria - part 6.1
- Priority table - part 6.2
- The first syntax example - part 7.1
- The second syntax example - part 7.2
- Call-by-methods - part 8.4

Contents

Part I

Introduction

0.2 Introduction

When writing a program for the first time, most programming languages can seem unintuitive and be difficult to understand. This can make it difficult to learn for people without any to little programming experience, especially if they are not assisted by a teacher.

Programming languages differ in various aspects and some are more intuitive for beginners than others. Some languages are complex, which often is due to the fact that they have a lot of features. The complexity of having several different methods to produce the same outcome, this can be confusing and make a language difficult to learn.

A beginner in programming with interest in learning about simple electronics is likely to encounter the programming language of Arduino. Arduino gives programmers the option to program components and use them for a wide variety of purposes. Since Arduino is a small board with just one processor for input and output, the options are relatively few and it is easy to get an overview.

0.3 Motivation

The Arduino language is based on C and C++, which are languages that can be seen as cryptic and unintuitive to newly started programmers. An example of C not being intuitive is when creating an array to hold 3 elements, and the array has to be set to a size that can hold 4 elements. This is needed because there is a special sign to mark the end of an array (zerobit), which takes a slot itself. The programmer can work around it and/or create the functionality, but to the inexperienced programmer it adds complexity. Without prior experience with programming, C and C++ can be difficult languages to learn.

For beginners it could be better if the Arduino language, was simpler and easier to understand. The Arduino language is built upon C and C++, and therefore has similar flaws, at least from a beginners perspective. This project will look into giving beginners an alternative to the Arduino language by creating a new language along with a translator for it, in order to be recognizable as proper code for Arduino. This language will attempt to achieve user-friendliness with beginners in mind. A place to draw inspiration from is Python, which is a programming language praised by many experienced and inexperienced programmers for being intuitive and easy to learn. [**?**] Python is a declarative language which means that the code can be read logically. The programmer will discover that Python looks somewhat like English language. It is easy to understand because the code can be less cryptic compared to other languages. It is this kind of user friendliness that will be sought in this project.

0.4 Compiler

For a programmer to avoid using the Arduino language there are two prerequisites. It requires another language, which the programmer can use to write the desired code. In order to get the code of the new language to work on Arduino, a translator is required. This makes the creation of a new language for Arduino a two step process. Without well-defined translations into the Arduino language, the new language cannot be executed on Arduino. Therefore a translator has to be created alongside the language. A compiler is a program that can take a source language as an input, and generate a target language as the output. The source and target language is defined by the design of the compiler. The compiler will be further explained in chapter 2

0.5 The project

In this project a new programming language and a compiler will be defined and created. The purpose of the new language is to simplify the process of programming for Arduino. In order to make the code from the new language executable on Arduino it has to be translated into the Arduino language. The translation process will be handled by the compiler.

Part II

Arduino

1 Arduino

Arduino was created by the Interaction Design Institute Ivrea (Italy), by Massimo Banzi and David Cuartielles. They were looking for an easy and cheap way for students, who study design, to integrate micro controllers into their projects[Lah09]. Both the board and the programming language was based on the works of Hernando Barragán, one of Massimo Banzi master thesis students [Bar04]

1.1 The hardware components

Arduino is a single-board micro-controller, see figure 1.1. A board consists of open source hardware, which is designed around an 8-bit Atmel AVR micro-controller. Arduino boards varies in sizes. Arduino Uno board for example, has a max width of 2.1" (5,33cm) and a length of 2.7" (6,86cm).

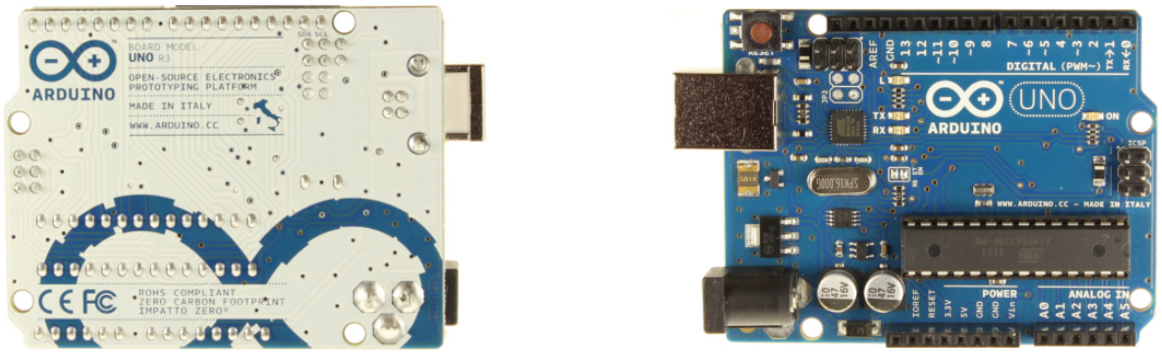

Figure 1.1: Picture of back- and fronside of the Arduino board [**?**]

The board provides some input and output possibilities. However these vary depending on the board, though most have 14 digital I/O and 6 analog inputs. The I/O functions are placed on top of the board, are freely accessible, and consist of 0.1" female headers. Besides the I/O there is also a Power connector, which almost in all cases require 5 volt DC. There is a USB connection on the board, so that processing data to the micro-controller is possible, though it is shown as a virtual com-port on the connected computer. However, on older boards, instead of the USB connection, a RS232 were used for serial communication.

On the board there is a LED diode which is connected to the digital pin 13. When this diode is set to "HIGH" it will be turned on, and if its value is "LOW" it turns off. Besides the LED diode, there is also a reset button. If the button is pressed the micro-controller is reset.

Now what makes Arduino such a good platform for beginners to learn to code, is the fact that it is really easy to hook up the Arduino platform, to a wide array of components, to make it possible for the Arduino to interact with the real world.

LED's	They come in all size, shapes and colors, ranging from small pin size single color LED's, to large multi color led displays.
Sensors	These are some of the main components behind the Arduino success, they give the Arduino the ability to sense its surroundings, for instance the temperature of the room, whether the light is turn on, even advanced sensors like humidity or gas.
Motors	They give the Arduino the ability to manipulate its surrounding, and act on the informations obtained from the sensors. There are two kind of motors, normal motors which can just be turned on/off, they are great for powering wheels or tracks to allow it to move. There are stepper motors with it is possible to control the exact amount of rotation. And finally there is servos, which are precisely controlled motors much like the stepper but much more precise.
Displays	There are a wide range of display available ranging from simple 2 line mono color LCD displays, all the way up to large OLED color and touch displays.
Communication	It is easy to hook up the Arduino with some form of communication allowing it to communicate with other devices either through, RF signals, bluetooth, WiFi, cellular or just plain old Ethernet connection
Shields	Finally the last thing which makes the Arduino great is it modularity, in the form of shields. Shields are boards much like the Arduino it self, but they offer all the capabilities of the components mentioned above, but they do it in a way, which makes it possible for any one with out any electronics experience to use them, you just plug the shield on top of the Arduino board, and all the components need to are included.

1.2 The role of Arduino

In this project Arduino board is used to execute the code, and show a type of output. The output will be in the form of a LCD Display and a LED light. Arduino is also used to get input through buttons and sensors.

Revise when we have the arduino! - Matti: Er denne sektion nødvendig?

Part III

Analysis

2 The compiler

A compiler is a program which translates one language (source language) into another language (target language). When writing a program, the programmer uses a programming language such as Java or C. A computer however only "understands" the language of binary code, the so called machine code. Binary code is in this case the target language reached by translating the source language into the target language, with the use of a compiler. Writing the program in binary code is however not optimal for the programmer as the alphabet of binary consists of 0's and 1's, and can be hard to understand. Writing a program in a high level language such as Java or C and compiling it into machine code is easier for the programmer. The goal of this project is to translate a high-level language to another high-level language - namely, from the source language to the Arduino language. The task of translating a programming language into another can be done using a compiler. The compiler allows the programmer to avoid using the target language. The compiler is therefore responsible for always producing a correct representation of source language in the target language.

2.1 The structure of a compiler

Compiling a source language into a target language is not a one step process. The compiler goes through several steps in order to translate a program written in a source language into a program written in the target language. Some steps are necessary, and others are optional.

and its purpose is to create a stream of tokens from the input. Each

- The first task performed by a compiler is scanning. The scanner is part of the lexical analysis. It scans the source language and prepares the code for further treatment. This is done by scanning through the entire code one character at a time. Any unneccessary code, such as the comments and white space, is removed in this process. Every other part of the code is placed in a stream of tokens. These can for example be integers, identifiers and operators. Any user defined elements is furthermore put into a symbol tree, which is accessible throughout the process of compilation. If the user has defined, for example, that whenever the word "POWER" is written in the code of the program, it always refers to the number "9001", this would be put into the symbol tree. The result is that whenever "POWER" is found in the code, it will be replaced with "9001". After the source program has been run through, the output from the scanner is passed on to the parser.

- The second task is parsing. A parser analyses the syntax of the code received from the scanner. The parser checks if the syntax of the code is compatible to the syntax of the target language. If the written code is not proper according to the target language, it is not possible to be translated correctly. As an example, a C parser will return an error if the code states $a + b = c$; rather than $c = a + b$; this is because the syntax of the first expression is illegal in C. It is important to note that at this step it is unknown what the variables contain, which is where the type checker takes over.

- After the syntax has been verified, the third step is the type checking, which deals with the semantics of the code. The type checker verifies that all operations performed in the code are legal regarding to the language. In the previous example, $c = a + b$; is syntactically correct. However what has not been specified is the content of a, b and c. If a and b are strings (text), while c is of the type int (number), the semantic is incorrect and will therefore return an error as the operation does not exist in the defined source language. If no errors occur, the parser will return an abstract syntax tree (AST) which also concludes the analysis of the source code. As seen in figure 2.1 the AST shows the syntactic constructs. For example, an assignment requires a variable and an expression.

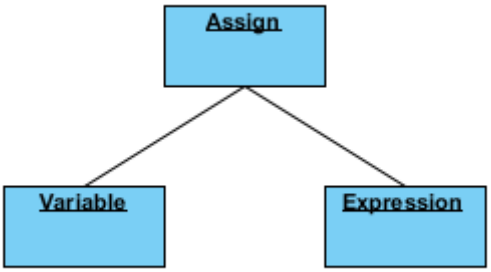

Figure 2.1: AST example

- At this point, the preparation of the source code is complete and the translation into the target language can be performed correctly. This is done by the translator. The translator translates the generated AST, from the parser, to intermediate representation code. IR code is used to describe the meaning of the AST nodes. For example to describe that a while loop actually loops. There is nothing in the AST that implies this. In compilers that do not implement optimization, target code can be generated by a translator without the use of an explicit IR.

  To translate the AST code generation visitors (CodeGenVisitor) are used, which visit the nodes in the AST. There are various different CodeGenVisitors to generate the right low-level language code. The output from the translator is the compiled product of the source code.

Matti: Revise this paragraph about CodeGenVisitor

While this covers the direct compiling process, more optional steps can be used to enhance the compiling process for different purposes.

- An optimizer can be used to analyze, and generate more effective IR code. While a piece of code can be optimal in the source language, the target language might have a more efficient formulation for describing the same action than the direct translation produces. The optimizer locates code which can be optimized and replaces the code with the more efficient model.

- In order to generate target machine code, a code generator can be used to translate the IR code generated by the translator.

Morten: Still a few parts lacking eg code generation

2.2 Tombstone Diagram

When developing a compiler, tombstone diagrams can be used to define the programming languages needed to compile the source code of a program. The tombstone diagram shows the source language, the language of the compiler(s), the language(s) of the different stages of compiling and the target language.

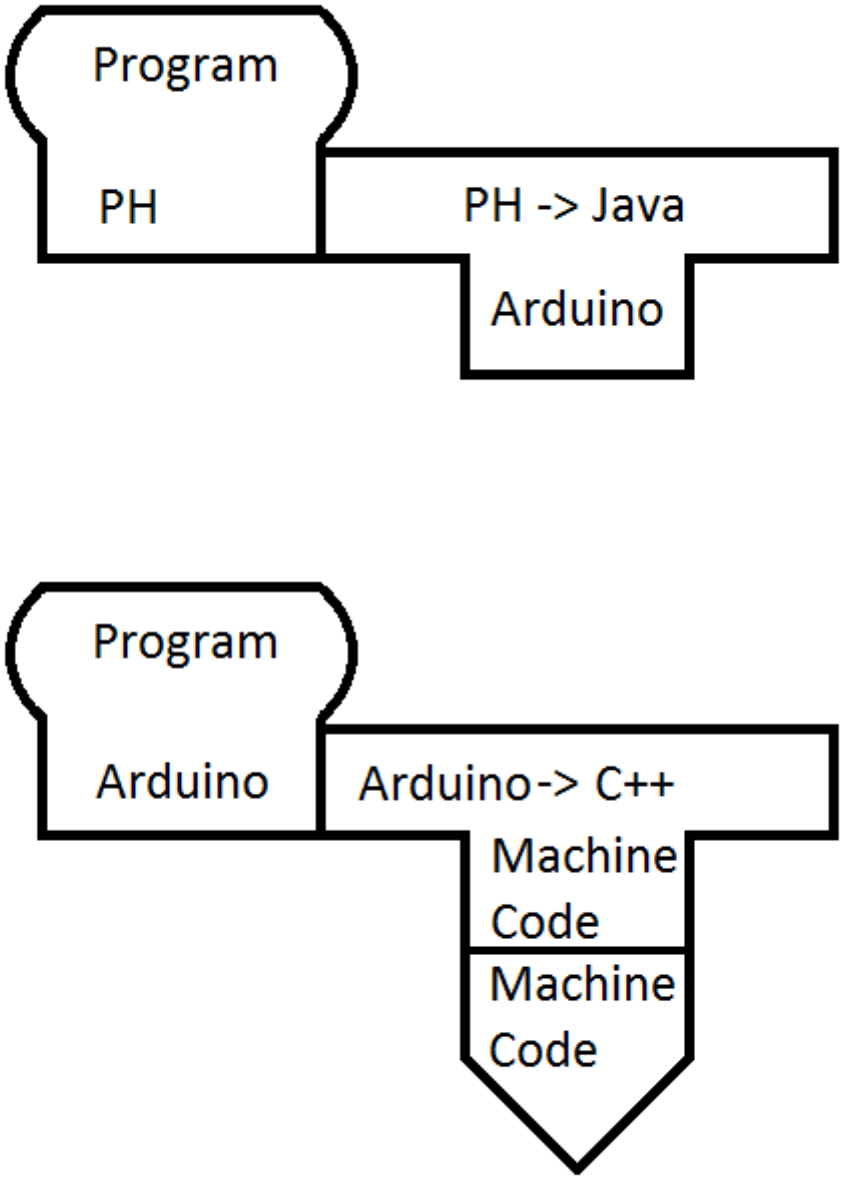

Figure 2.2: Tombstone diagram for the source language to machine code

As seen in figure 2.2 the tombstone diagram for this project is as follows: The program is written in PH . It is then translated by the compiler, written in Java, into Arduino language. Since the Arduino itself only understands machine code, another

Name for source language placeholder

compiler written in C++ is needed to translate from Arduino language into machine code. For this, the standard Arduino IDE is used. Since this project focuses on compiling from PH to Arduino language, the first tombstone diagram in figure 2.2 is the important one in terms of this project.

2.3 Parsing

This section contains a description of the parsing phase of the compiler.
Parsing is the part of the syntax analysis that builds a parse tree and checks the input for syntax errors. The parser handles the token stream generated by the scanner by grouping the tokens into sentences based on the context free grammar. There are two different categories of parsers, top-down and bottom-up parsers.

Top-Down Parsing

This category of parsers is called top-down, because the parser begins with occurrences of the start symbol of the grammar and creates a parse tree of the input, by traveling from its root to its leaves. These parsers read the token stream from left to right, and derives productions using leftmost derivation. Two forms of top-down parsers are described below.

- **Recursive-descent parsers**: This type of parser uses recursive procedures to parse a string. In general each procedure uses one production rule of the grammar.
- **Table-driven LL parsers**: This type of parser uses generic LL(k) parsing engines, and a parse table to keep control of which production rules to use. LL denotes the fact that it scans the input from left to right, and produces a leftmost derivation. Furthermore, it uses *k* symbols of lookahead. Lookahead is explained in 9.5.

Bottom-up Parsing

This category of parsing is the opposite of top-down parsing. Bottom-up parsers are more powerful and efficient than top-down because they handle certain features better, such as recursive productions and common prefixes. It is called bottom-up, because the parser begins with occurrences of the start symbol of the grammer, and creates a parse tree of the input by traveling from its leaves to its root. These parsers read the token stream from left to right, and derives productions using rightmost derivation in reverse. Bottom-up parsers uses the grammar rules to replace the right-hand side of the production with its left-hand side. Top-down parsers does the opposite, by replacing the left-hand side of the production with its right-hand side. The type of bottom-up parser is described below.

- **LR(k) parsers**: The LR parser read the token stream from left to right, and traces a rightmost derivation in reverse. As with LL parsers, the LR parser also uses *k* symbols of lookahead.

3 Source language

In this project a new programming language is proposed. The language is designed for the use of beginners in the field of programming and is meant as an alternative for the already existing Arduino programming language (See chapter 1). The goal of the new language, the source language, is to make it simpler and easier to understand compared to the existing Arduino programming language. The syntax should be intuitive in that it can be read naturally, similar to the English language.

The source language is compiled to the Arduino language, rather than machine code. The source language is not a fully functioning programming language with advanced features. However, it is a simple language that can print out text and handle simple mathematics, such as adding, subtracting, multiplying numbers etc. along with handling boolean expressions, built up from equalities and inequalities of numeric expressions.

The features for the source language are listed below:

- Readable, also by users that are not familiar with programming languages.
- Easy to learn, and to be used for basic programming.
- Should be compiled to Arduino.
- The language should be able to handle mathematics and boolean expressions.

Python

The source language that is developed in this project has a syntax that is somewhat similar to Python. Python is a programming language that is open source and supported by operating systems like: Windows, Linux/Unix, OSX. Python is a language with a syntax similar to English. It has been developed following some of the same criteria as our source language. Readable syntax is one of the main features, and it is the reason for creating the source language based on Python.

[pyt13a] There exists a large community of users of Python, which help each others with different problems. There are also many tutorials on the internet and books that teach how to use the language. Python's official website has a tutorial guide.**??**
Python was created back in the 1980s. Its first official release was in 1994. The language was created by Guido van Rossum. When he created Python, the language was also influenced from other existing languages, mainly ABC, which is the core syntax to Python. **??**

3.1 The Arduino language

The Arduino language is based on Wiring and therefore there are a lot of similarities between the two languages, but the Arduino team have added to it, improved the functions, and made it compatible with a wider range of chips.Both the languages are implemented as versions of C/C++, and are using an IDE based on the processing IDE [Bar04][Ard].

In table 3.1, the syntax of the two languages are almost identical, and it is only in the functions parameters, that there are any noticeable difference.
To help facilitate the compatibility with a wide range of AVR chips, the Arduino language makes great use of AVR Libc [AS11], which is an open source C library that supplies the necessary functionality to make it possible to use the Atmel AVR micro controllers.

Wiring

```
1  int ledPin = 8;
2
3  void setup(){
4    pinMode(ledPin, OUTPUT);
5  }
6  void loop(){
7    digitalWrite(ledPin, HIGH);
8    delay(1000);
9    digitalWrite(ledPin, LOW);
10   delay(1000);
11 }
```

Arduino

```
1  int led = 13;
2
3  void setup() {
4    pinMode(led, OUTPUT);
5  }
6  void loop() {
7    digitalWrite(led, HIGH);
8    delay(1000);
9    digitalWrite(led, LOW);
10   delay(1000);
11 }
```

Table 3.1: Code examples in both Wiring and Arduino to make a LED flash

Kunne det ikk være brug-bart at sætte et eksem-pel af C op ved siden af dem også?

3.2 MoSCoW analysis

This overview is based upon the MoSCoW method, which is a technique used in business analysis as well as in software development. It is used to set the importance level of the various criteria needed for the project. **??**. However in this MoSCoW analysis the "Would like" part is left out, as it has no relevance.

A basic classification of the language features is presented below.

- **Must have**: This category shows the most necessary features in the source language. These features are the ones that can not be overlooked in the source language, as it can have a great impact on whether or not the source language will work correctly.
- **Should have**: This category shows the features that should be implemented in the source language, however this category can be left out, without having a huge impact on the functionality of source language.
- **Could have**: This category is features that could be nice to have in the source language, but is only there to give a smother experience while programming in the source language, so leaving these out will have minor impact on how it works.

For this project the following have been chosen:

Must have:

- If-statement
  The feature of choosing one path over another is so common to programming that it would make no sense to leave it out. One must be able to choose what to do based on something important to the program.

- Expressions
  In relation to if-statements it is impossible to leave out the ability to evaluate expressions to true or false.

- Functions, parameters and return values
  The ability to reuse code can reduce the cluttering of code significantly, and help minimize errors, by allowing the use to reuse the same code, in stead of type it multiple times. While it may be possible to leave out functions, it will not make sense from a user-friendly perspective. Making use of parameters and return values removes the complexity of passing by reference.

- Loop - While
  Another very common task is to repeat the same task multiple times. Instead of having to create the code multiple times it is more efficient to simply run the same piece of code.

- Blocks (do/end signs)
  Making it clear for the user where a block begins and ends helps reduce confusion. For example, it might be hard to figure where a while loop ends if it has no "end" keyword or other recognizable sign.

- Print
  The ability to print out the results of a program is an easy way for the programmer to see the outcome.

- Logical operators
  Being able to select and group operators is important to perform decision making in programs. It would be very difficult to leave out of a general purpose language.

- Simple mathematical operators
  Leaving out mathematics removes a lot of functionality and heavily reduces the options of programming.

- Type definitions
  The user has to be capable of identifying what is being used. Without clearly defined types to use it can be very difficult for a beginner to understand how to program.

- Imperative language
  Choosing Imperative language means that the user will have to write code, that describes in exact details what steps has to be done and what is the next step. The Imperative language are supported by most of the mainstream object-oriented programming languages such as Java and C#. **??**

- UTF-8
  UTF-8 is short for **U**CS(Universal Character Set) **T**ransformation **F**ormat-8bit. It can represent every character in the Unicode set and provides compatibility with ASCII. The reason UTF-8 is a must have is because the typeset is very broad. This means the user does not have to worry about whether or not his special characters is acceptable characters, and does as such give the user the ability to code naturally.

Should have:

- Initialization
  There are different ways that initialization can work depending on the language as well as the type of the object.

Could have:

- User input
  While definitely a feature which is relevant for some purposes, the feature is not required in order to have a programming language able to use the features of Arduino.

- Advanced mathematical operators
  Mathematics can be solved without the use of integrated mathematics such as square root and modulus, but having the features is definitely convenient.

- Loops (Other than while)
  The “while-loop” has all the functionality needed, but in some cases there are more intuitive ways to go around a problem. “Foreach” is a good example of this problem.

4 Informal Specification

The purpose of this section is to outline what the programming language, proposed in this project, contains.
The language created in this project is a simplification of the Arduino language. Its purpose is to simplify the process of writing programs for Arduino.

Data types

The language contains the following data types:

int float string boolean

The different data types can hold different ranges or types of values. The values have been specified more below:

- **Int** or integer does not include floating point numbers, and can only hold numbers between -32.768 and 32.767. If however it is unsigned it can range from 0 to 65.535.
- **Float** allows a number within the range $1.175494351e-38$ and $3.402823466e+384$.
- **String** can hold any kind of number and/or character.
- **Boolean** can either be "true" or "false".

Morten: unsigned og signed?

Keywords

The following words are reserved words in the programming language:

if	else	elseif	do	end	function
while	return	true	false	void	int
float	system	string	out	bool	print
loop	setup				

Variables

The variables in the language can only be defined by using among all the upper- and/or lower case English letters. The underscore sign "_ " can also be used in addition to the English letters to make more complex variable-names.

Encapsulation

The encapsulation in the language is defined by **do** and **end**. **do** opens the encapsulation, and **end** closes it.

Loops and conditions

The language contains a **while-loop** with a condition check before the loop is executed. The language also contains an **if then else** condition. Additionally **else if** can be nested inside the **if then else**.

White-space and commentary

The compiler ignores the white-spaces as well as anything to the right-hand side of the comment, where "//" indicates a comment.

5 Design Criteria

This section presents the design criteria for the source language, showing which goals is reached based on the criteria.

5.1 The chosen criteria

Typically one uses a set of criteria to evaluate the usability of a language. The design criteria that were considered to evaluate the source code is listed in table 5. The source language does not fulfil all criteria, since not all of the criteria are relevant for this projects purpose. For instance the "Cost" criteria as well as the "Others" criteria including their sub-criteria are not needed here. However "Readability", "Write-ability" and "Reliability" are relevant for this project and the focus will be on these 3 and their respective sub-criteria.

Main criteria	*Sub-criteria*
Readability	Overall simplicity Orthogonality Data types and control statements Syntax considerations
Write-ability	Simplicity and orthogonality Support for abstraction Expressivity
Reliability	Type checking Exception handling Aliasing Readability and write-ability
Cost	Training programmers to use the language Writing programs Compiling programs Executing programs Language implementation system Reliability Maintaining programs
Other	Portability Generality Well-definedness

Table 5.1: Table of design criteria

Readability:
The readability criteria is one of the most important criteria for this language, because this language is developed so that it should be easy to read and thereby easy to understand what it is going on.

- Overall simplicity:
  The source language has a manageable set of features, which provides the user

with the basic need to be able to form programs in a quite simple matter. The source language being somewhat limited gives the source language an overall simplicity as there is not much operator overloading possible since the user is more or less closed off from giving the operators another semantics than first given.

- Orthogonality:
  A language that has good orthogonality is a language that has few or a limited amount constructs, where every possible combinations is allowed. Though, too much orthogonality is not always good, as it can cause problems. For instance having too many choices, and which solution would be the best.
  The source language is not quite orthogonal, as every possible combination is not legal though it does not reject all combinations either.

- Data types and control statements:
  In general the source language has adequate data types as well as control statements, however it does not support all the types featured in Arduino, but enough for the most simple tasks. For instance decimal is not supported, which is the preferred data type when making a program that can handle money. In the source language decimal and doubles were chosen not to be included as float provides the source language with plenty opportunities, which can make them obsolete.

- Syntax Consideration:
  The syntax consideration for the source language is mostly to be found within the keywords. For instance "AND" is quite self-describing whereas many programming languages uses "&&" for the same effect. The keywords used in the source language are easy to use as they are close to English language. For instance looking at this example: "expression(1) is true or if expression(2) is true", here the important word is "OR", which is also the keyword for how to distinguish between two boolean expressions.

Write-ability:
The write-ability criteria is quite important as it shows the criteria of how the code should be written.

- Simplicity and Orthogonality:
  It is important for the source language to have a small number of rules required to write code. The code has to be simple to write as well as it being easy to use along with orthogonality.

- Support for abstraction:
  Some languages support abstraction in their programming language. For instance supporting imaginary numbers, The source language does not support abstraction. Having imaginary numbers is without need for the source language as it has no of practical use for this project. The source language does not need this feature to achieve its goals.

- Expressivity:
  The source language is not expressive. It does not support expressions like "$i++$" or predefined functions like "*and then*". These expressions are usually used to

ease coding. For instance, instead of writing "$i++$", one would have write "$i = i+1$". So, by using "$i++$" the programmer would both be saving space and time. Though having focus on expressivity can mean sacrificing some readability.

Reliability:
It is important that the source language is reliable. If the source language keeps producing errors or mistakes, the user is less likely to use the programming language.

- Type checking:
  It is important that the source language is checking for type errors as early as possible. The source language is being checked for these type errors during the code generation.

- Exception handling:
  There is no exception handling in the source language, so if there is an error, the program will not run. To include this would add to the complexity of source language.

- Aliasing:
  It is possible to refer to function-names as well as variable-names in the source language. However, it is not possible to refer to the same slot in memory in two or more distinct ways, which makes the source language weak regarding aliasing.

- Readability and write-ability: When looking at readability and write-ability it is important that the source language supports a natural way of writing and reading code.

It is difficult to be fully objective regarding the design criteria as they can be seen from different angles. Different people will have different opinions on whether or not the source language is fulfilling the criteria.

6 Syntax and Semantics

In this section a description of the syntax and semantics in the programming language is described.

6.1 Syntax

When describing a programming language the syntax of the language is given by the set of rules which defines the words (finite sequences of characters) that can be used for writing a program in that language. The syntax of the language is described using Backus-Naur Form (BNF) which provides the context free grammar of the language. In this report an extended version of BNF, Extended Backus-Naur Form (EBNF), is used. The advantage of using EBNF is to describe the set of rules in a more compact form, and the ability to describe regular expressions in the context free grammar. EBNF does not enhance the descriptive power of BNF, it only increases the readability and the write-ability.

6.2 Syntax definition

Below is the syntax definition written in EBNF which shows the seven syntactic categories followed by the defined rules. As an example of how to read these rules, let a be an arithmetic expression. By looking at the rule for a, this expression could for example be $a_1 + a_2$ meaning it is expanded to be one arithmetic expression consisting in the addition of two other simpler arithmetic expressions. These two expressions could then again be anything within the rule for a, for example the numerals 5 and 7. There are no rules for expanding a numeral, and as such the expression a can be evaluated no further.

```
1  n ∈ Num - Numerals
2  x ∈ Var - Variables
3  a ∈ Aexp - Arithmetic expressions
4  b ∈ Bexp - Boolean expressions
5  S ∈ Stm - Statements
6  Dv ∈ DecV - Variable declarations
7  S_b ∈ Substatement
8
9  a ::= n | x | a_1 + a_2 | a_1 - a_2 | a_1 * a_2 | a_1/a_2 | sqrt(a) | a_1^a_2 |
       a_1 % a_2 | (a)
10
11 b ::= "a_1 equals a_2" | "a_1 > a_2" | "a_1 < a_2" | "a_1 <= a_2" | "a_1 >= a_2"
         | "a_1 notequals a_2" | notb | b_1 and b_2 | b_1 or b_2 | (b)
12
13 S ::= Dv | if b do S end S_b | while b do S end | S_1;S_2
14
15 S_b ::= {elseif b do S end} | else do S end
16
```

```
17 Dv ::= x = a; Dv
```

Code 6.1: Syntax formation rules

Numerals

In our programming language there are three numeral systems called integers, doubles, and floats. These types are needed to represent integers (for example -17, -239586, 0, 237, 9001). and decimal numbers (such as -12.3456, -30.009, 90.01 and 990.1259). Float and double are both used for decimal numbers with double being more precise than float at the cost of taking up more space. The numeral systems are however finite as the numerals can not exceed the size allocated to each float, integer and double. As such the limitations are:

- Integers: From -2^{n-1} to $2^{n-1}-1$
- Floats: From $3.4E-38$ to $3.4E+38$
- Doubles: From $1.7E-308$ to $1.7E+308$

Variables

A variable in the language ranges over text, expressions, boolean expression, numerals etc.

Variable declarations

The following describes the specified rule for declaration a variable. The rule states that a variable declaration is composed by variable name on the lefthand side of the assignment character "=", and then the value on the righthand side is thereby assigned to the left side. The ";" character closes the variable declaration.

Arithmetic expressions

The source language features arithmetic expressions. The transition rules for the expressions can be read in table **??**. The rules listed in listing 6.1 states that well-formed arithmetic expressions in the source language can consist of:

- x - Variable
- n - Numeral
- Addition "$a_1 + a_2$" - Addition between two numbers.
- Subtraction "$a_1 - a_2$" - Subtraction between two numbers.
- Multiplication "$a_1 * a_2$" - Multiplication between two numbers.
- Division "a_1/a_2" - Division between two numbers.
- Square root "sqrt(a)" - Finds the square root of a number.
- Power "$a_1 \hat{} a_2$" - Allows a number to be powered by another number.

- Modulus "a_1 % a_2" - Allows modulus to be used, which gives the opportunity to find the remainder between two numbers.
- Parenthesis "(a)" - Specifies that an expression can be surrounded by parenthesis.

Boolean expressions

The source language supports boolean expressions, which evaluate to *true* or *false*. The boolean transition rules are listed in table **??**.

- Equals to (a_1 equals a_2) - *equals* allows the programmer to equate two expressions, which will evaluates to *true* if their evaluations match; if not it evaluates to *false*.
- Greater than ($a_1 > a_2$) - "*>*" checks if one value is greater than another value. If a_1 is greater than a_2 it evaluates to *true*, else it evaluates to *false*.
- Less than ($a_1 < a_2$) - "*<*" checks if the value of the first expression is lesser than the value of the second one. If a_1 is less than a_2 it evaluates to *true*, else *false*.
- Greater than or equal to ($a_1 >= a_2$) - "*>=*" checks if one value is greater then or equals another value. If a_1 is greater then or equals a_2 it evaluates to *true*, else *false*.
- Less than or equal to ($a_1 <= a_2$) - "*<=*" checks if one value is less then or equals another value. If a_1 is less then or equals a_2 it evaluates to *true*, else *false*.
- Different from (a_1 notequals a_2) - "*notequals*" checks if two values are different. If a_1 is different from a_2 it evaluates *true*, else *false*.
- Negation (not$b$) - "*not*" checks if a value is boolean value is false. If b is false it evaluates *true*, else false.
- Conjunction (b_1 and b_2) - "*and*" check if two boolean values are both *true*. If both b_1 and b_2 are *true* it evaluates *true*, else *false*.
- Disjunction (b_1 or b_2) - "*or*" checks if one of the boolean values are *true*. If either b_1 or b_2 are *true* it evaluates *true*, else *false*.
- Parenthesis ((b)) - Allows the boolean expressions to be insulated in parenthesis. The parenthesis will then evaluate to *true* or *false*.

Statements

S ::= Dv | if b do S end S_b | while b do S end | S_1;S_2

- Variable declaration - A statement can consist of a variable declaration, section 6.2.
- If-statement - A conditional expression involving one boolean expression and two statements, or a statement and a substatement. If the boolean expression evaluates to true, the first statement is called. If the boolean condition evaluates to false, then the program executes the second statement or the substatement.

- While loop - It involves a boolean expression and the while loop will run if the boolean expression evaluates to true. As long as the boolean expression is true, the statements inside the loop will keep repeatedly execute over and over. When the boolean expression evaluates to false, the loop is exited.
- Sequential Composition - To make programs run statements sequentially in the source language.

Substatements

S_b ::= elseif b do S end | else do S end

- Elseif - If an if-statement is executed the source language allows to follow up using an elseif statement. If and if-statement is not present, an elseif is not allowed to be used. Elseif follow the same rules as if-statements, and just like if-statements it has a boolean expression, which if true the statement is run. If the boolean expression is evaluated to false, the program checks for additional elseif statements, if none, then carries on to the else-statement. Elseif can run zero to many times.
- Else - If the boolean expression in an if-statement evaluates to false, an else statement can be reached which will execute the other statement. An if-statement is required though to be able to use the else-statement.

6.3 Semantics

The semantics of a language is the description of what happens when a program is executed. In table **??**, **??** and **??** are the descriptions of the operations in the source language.

In table **??** this arrow $\rightarrow_a$ is used. This arrow represents the transition system. For instance, taking in the first line in "PLUS", it represents the transition from state a_1 to v_1 as well as the transition from a_2 to v_2. The label on the arrow indicates that the transition system only allows values that all are arithmetic expressions.

Name	*Rule*	*Notes*
$[PLUS_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 + a_2 \rightarrow_a v}$	where $v = v_1 + v_2$
$[MINUS_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 - a_2 \rightarrow_a v}$	where $v = v_1 - v_2$
$[MULT_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 * a_2 \rightarrow_a v}$	where $v = v_1 * v_2$
$[DIV_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 / a_2 \rightarrow_a v}$	where $v = \frac{v_1}{v_2}$ and $v_2 \neq 0$
$[MOD_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 \ \% \ a_2 \rightarrow_a v}$	where $v = v_1 mod v_2$
$[POW_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 \ \hat{} \ a_2 \rightarrow_a v}$	where $v = v_1 \hat{} v_2$
$[SQRT_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1}{s \vdash sqrt(a_1) \rightarrow_a v_1}$	where $v = \sqrt{v_1}$ and $v_1 >= 0$
$[PARENT_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1}{s \vdash (a_1) \rightarrow_a v_1}$	
$[NUM_{BSS}]$	$s \vdash n \rightarrow_a v \ if \ \mathcal{N}[\![n]\!] = v$	
$[VAR_{BSS}]$	$s \vdash x \rightarrow_a v \ if \ sx = v$	

Table 6.1: The semantics of arithmetic operations

First is table **??**, which describes the arithmetic expressions - the more common ones are addition, subtraction, multiplication and division (PLUS(+), MINUS(-), MULT(*), DIV(/)). Their rules all look near identical: A statement, "s", with two expressions, v_1 and v_2, which both are arithmetic as seen by $\rightarrow_a$ can be added, subtracted, multiplied or divided with each other. This results in v, which is arithmetic. However, when taking a look upon DIV's side-note, take notice that it states that v_2 cannot be divided by zero, as it is not possible to divide any number by zero, so this side-note making sure that it is not possible to do so.
Furthermore, v is the product of the operator used on v_1 and v_2. Both modulus and power of (MOD(%), POW(^) are identical to these operations in procedure.
SQRT, also known as square-root($\sqrt{}$) is a simple rule to allow any positive arithmetic

expression, v_1 to be squared, where v obviously is $\sqrt{v_1}$. Also here, like on division, is an extra side-note, which indicates that the number that is to be square-rooted is either a zero or a positive number, as the source language is not going to allow imaginary numbers.

The **PARENT**, or parentheses(()), rule states that any arithmetic expression can be parenthesized, allowing chained arithmetic calculations to be performed in the correct order.
VAR describes, that in a state s, a value v can be assigned to a variable x, allowing variable declaration.
NUM, numerals also known as absolute value, allows within a statement S to get the numerical value of a variable v. A numeric value is a the distance a number has from zero, for instance $|2|$ and $|-2|$ are both 2 units from 0.

Name	*Rule*	*Notes*
$[NOT1_{BSS}]$	$\frac{s \vdash b_1 \rightarrow_b tt}{s \vdash not\ b_1 \rightarrow_b ff}$	
$[NOT2_{BSS}]$	$\frac{s \vdash b_1 \rightarrow_b ff}{s \vdash not\ b_1 \rightarrow_b tt}$	
$[EQUAL1_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1\ equals\ a_2 \rightarrow_b tt}$	if $v_1 = v_2$
$[EQUAL2_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1\ equals\ a_2 \rightarrow_b ff}$	if $v_1 \neq v_2$
$[NOTEQUAL1_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1\ notequals\ a_2 \rightarrow_b tt}$	if $v_1 \neq v_2$
$[NOTEQUAL2_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1\ notequals\ a_2 \rightarrow_b ff}$	if $v_1 = v_2$
$[GREATER1_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 < a_2 \rightarrow_b tt}$	if $v_1 < v_2$
$[GREATER2_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 < a_2 \rightarrow_b ff}$	if $v_1 \not< v_2$
$[LESSER1_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 > a_2 \rightarrow_b tt}$	if $v_1 > v_2$
$[LESSER2_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 > a_2 \rightarrow_b ff}$	if $v_1 \not> v_2$
$[LEQ1_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 >= a_2 \rightarrow_b tt}$	if $v_1 \leq v_2$
$[LEQ2_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 >= a_2 \rightarrow_b ff}$	if $v_1 \not\leq v_2$
$[GEQ1_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 <= a_2 \rightarrow_b tt}$	if $v_1 \geq v_2$
$[GEQ2_{BSS}]$	$\frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 <= a_2 \rightarrow_b ff}$	if $v_1 \not\geq v_2$
$[PARENT_{BSS}]$	$\frac{s \vdash b_1 \rightarrow_b v_1}{s \vdash (b_1) \rightarrow_b v_1}$	
$[AND1_{BSS}]$	$\frac{s \vdash b_1 \rightarrow_b tt \quad s \vdash b_2 \rightarrow_b tt}{s \vdash b_1\ and\ b_2 \rightarrow_b tt}$	
$[AND2_{BSS}]$	$\frac{s \vdash b_i \rightarrow_b ff}{s \vdash b_1\ and\ b_2 \rightarrow_b ff}$	where $i \in \{1, 2\}$
$[OR1_{BSS}]$	$\frac{s \vdash b_1 \rightarrow_b tt \quad s \vdash b_2 \rightarrow_b tt}{s \vdash b_1\ or\ b_2 \rightarrow_b tt}$	
$[OR2_{BSS}]$	$\frac{s \vdash b_i \rightarrow_b ff}{s \vdash b_1\ or\ b_2 \rightarrow_b ff}$	

Table 6.2: The semantics of boolean operations

Table **??** holds the rules of boolean operations. Once again a lot of operations look

near identical, namely EQUAL1(=), EQUAL2(≠), NOTEQUAL1(≠), NOTEQUAL2(=), GREATER1(<), GREATER2(≮), LESSER1(>), LESSER2(≯), LEQ1(≤), LEQ2(≰), GEQ1(≥) and GEQ2(≱). The common factor is that these are all comparison expressions, respectively checks for equality, equals, not equals, greater than, less than, less or equal to and greater or equal to. Having a v_1 and v_2, which are both arithmetic as seen by $\rightarrow_a$, a boolean evaluation is used to conclude whether comparison is true (tt) or false (ff).
The NOT1(¬), NOT2(¬), AND1(∧), AND2(∧), OR1(∨) and OR2(∨) expressions are similar as well. Arithmetic expressions are tested with some criteria and evaluated to true or false. In the case of NOT1 where the evaluation is true, the evaluation of NOT1 is obviously false as it is not "not", with NOT2 being the opposite case. AND1, AND2, OR1 and OR2 follow same principles. In AND1 and AND2, two expressions need to be true if the entire expression is to be true, while OR1 and OR2 just needs one of two expressions.
As with the PARENT rule in table **??** any boolean expression can be parenthesized as well as seen in this PARENT rule.

Name	*Rule*	*Notes*
$[ASS_{BSS}]$	$\langle x := a, s\rangle \rightarrow s[x \mapsto v]$	where $s \vdash a \rightarrow_a v$
$[SKIP_{BSS}]$	$\langle skip, s\rangle \rightarrow s$	
$[COMP_{BSS}]$	$\frac{\langle S_1,s\rangle \rightarrow s'' \quad \langle S_1,s''\rangle \rightarrow s'}{\langle S_1;S_2,s\rangle \rightarrow s'}$	
$[IFTRUE_{BSS}]$	$\frac{\langle S_1,s\rangle \rightarrow s'}{\langle if\ b\ do\ S_1\ else\ S_2,s\rangle \rightarrow s'}$	if $s \vdash b \rightarrow_b tt$
$[IFFALSE_{BSS}]$	$\frac{\langle S_2,s\rangle \rightarrow s'}{\langle if\ b\ do\ S_1\ else\ S_2,s\rangle \rightarrow s'}$	if $s \vdash b \rightarrow_b ff$
$[WHILETRUE_{BSS}]$	$\frac{\langle S,s\rangle \rightarrow s''\langle while\ b\ do\ S,s''\rangle \rightarrow s'}{\langle while\ b\ do\ S,s\rangle \rightarrow s'}$	if $s \vdash b \rightarrow_b tt$
$[WHILEFALSE_{BSS}]$	$\langle while\ b\ do\ S, s\rangle \rightarrow s\ if\ s \vdash b \rightarrow_b ff$	

Table 6.3: The semantics of statements

The table of statement semantics, **??** holds rules affecting statements.
ASS, assignments, describes that with a given expression and state (a, s), a variable x can be assigned a value v. The SKIP rule states that within a state s, a skip can occur, leading to a state s. COMP, composition, allows compositional statements meaning that a series of statements can occur. IFTRUE and IFFALSE allows choosing an appropriate sub-statement based on boolean expression within a statement. WHILETRUE allows the statement to call itself again, making it recursive for as long as the expression is true. Whereas WHILEFALSE will do nothing for as long as the expression is false.
All of these rules, arithmetic, boolean and statements makes the base of the source language.

7 Scope rules

This section describese scope rules and the environment-store model for the language. It is important to know how the values of variables are being stored as well as knowing how the language handles encapsulation.

7.1 The environment-store model

It is important to know how the environment-store model works, because the model describes how the content of variables is stored and what location a variable is bound to. Furthermore it also describes what content is stored on a given location. Figure 7.1, illustrates the environment-store model. The model shows three boxes. From left to right the boxes represent the environment, location and store. The environment is the variables. A variable is bound to a location, illustrated by the arrow, env_v, connecting the **environment** and **location**. env_v is a function that retrieves the location of a variable. The content of a variable can be stored on a location, which is illustrated by the *sto* arrow, connecting **location** and **store**. *sto* is a function that retrieves the content stored on a location.

For example, the model shows that the variable x is bound to location 25, which has the value 5 stored. Both the variables y and z are bound to 26. This means they refer to the same location, which also of course has the same value stored. If for example, the value 13 in store is changed to 15, then the value for both variables y and z changes.

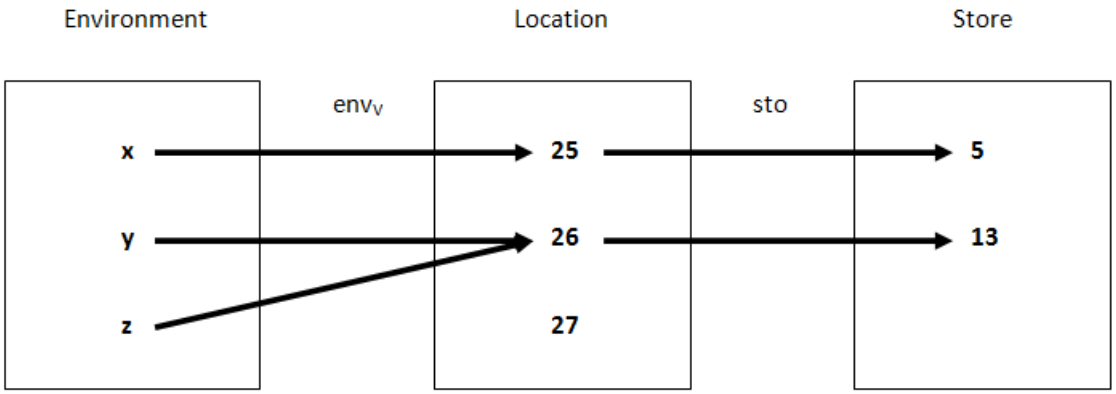

Figure 7.1: Environment-store model

7.2 The scope rules

Matti: Er der en som kan forklare denne? Eller en som ved hvad den betyder

$$[CALL-STAT-STAT_{BSS}] \quad \frac{env'_v\ [next \rightarrow l], env'_p \vdash \langle S, sto\rangle \rightarrow sto'}{env_v,\ env_p \vdash \langle call\ p, sto\rangle \rightarrow sto'}$$

where $env_p p = (S, env'v, env'p)$
and $l = env_v next$

7.3 Priority Table

In this section a priority table of the syntax for the language is described. The priority table contains different sections, the first is level hierarchy. Level hierachy describes the operators precedence. The operators with the highest precedence are evaluated first, and the operators with the lowest precedence are evaluated last. The level 1 operators have the highest precedence, and the level 4 operators have the lowest.

The table also shows which symbol is used for the different operators, and a short description of the operator. Furthermore it also describes the associativity of the operator. Associativity describes which direction an operator is evaluated in, either left to right or right to left.

Level	*Symbol*	*Description*	*Associativity*
1	not	Logical negation	Left to right
	*	Multiplication	Left to right
	/	Division	Left to right
	%	Modulus	Left to right
	sqrt()	Square root	
	^	Exponentiation	
2	+	Addition	Left to right
	-	Subtraction	Left to right
3	<	Less than	Left to right
	>	Greater than	Left to right
	<=	Less than or equal to	Left to right
	>=	Greater than or equal to	Left to right
	equals	Equal to	Left to right
	notequals	Not equal to	Left to right
	or	Logical or	Left to right
	and	Logical and	Left to right
4	=	Assignment	Right to left

Table 7.1: Table of operator priority

As seen in table **??**, on level 1 of the language is negation, multiplication, division, modulus, square root, and exponentiation. This precedence has been chosen because it matches that of mathematics. Level 2 contains addition and subtraction, which are placed above the boolean expressions in precedence, in order to have arithmetic and boolean expressions in the same statement but separate them in precendence. The last level is assignment.

Matti: This chapter is missing a section about how encapsulation is handled in our

8 Syntax examples

8.1 The first syntax example

This section contains the first code example that is written in the new language for Arduino. The code written in the Arduino language can be seen in Appendix **??**. If compared, it shows the difference between the new language, and the original one. This first code example shows how the "Hello, world!" program is written in the new language. "Hello, world!" is a simple test program that prints a string, which contains the text "Hello, world!".

Code 8.1: Hello World code example based on the source language

```
1 // include the library code:
2 #include <LiquidCrystal.h>
3
4 // initialize the library with the numbers of the interface
    pins
5 LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
6
7 void setup() begin
8   pinMode(9, OUTPUT);
9   analogWrite(9, 20);
10  // set up the LCD's number of columns and rows:
11  lcd.begin(8, 2);
12  // Print a message to the LCD.
13  lcd.print("hello, world!");
14  delay(2000);
15 end
16
17 void loop() begin
18  // set the cursor to column 0, line 0
19  // (note: line 1 is the second row, since counting begins
      with 0):
20  lcd.setCursor(0, 0);
21  // print the number of seconds since reset:
22  lcd.print(millis()/1000);
23 end
```

8.2 The second syntax example

This section contains the second example of a program written in the new language. This program is written based on a program from the original Arduino language. The original code can be found in Appendix **??**. The following test program performs a simple task, it counts from 1 to 100, by incrementing 1 at a time. Meanwhile it prints

a different output depending on the current value of the counter.

Count	print
Dividebel with 3	Foo
Dividebel with 5	Bar
Dividebel with 3 and 5	FooBar
Anything else	the count

If the count is at 9, it will print Foo, and if the count is at 15, it will print FooBar. Despite it being a simple program, it makes use of a lot of different operators and expressions.

Code 8.2: LCD code example based on the source language

```
// include the library code:
#include <LiquidCrystal.h>

// initialize the library with the numbers of the interface
    pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
int count;

void setup() begin
  pinMode(9, OUTPUT);
  analogWrite(9, 12);
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  // Print a message to the LCD.
  lcd.print("Foo Bar");
  count = 1;
  Serial.begin(9600);

  delay(2000);
end

void loop() begin
  lcd.clear();
  delay(200);
  //If count divided by 3 and 5 equals 0 write Foo Bar
  if(count % 3 equals 0 && count % 5 equals 0)then
      lcd.print("Foo Bar");
      Serial.println("Foo Bar");
  end
  //If count divided by 3 equals 0 write Foo
  elseif(count % 3 equals 0)then
    lcd.print("Foo");
    Serial.println("Foo");
  end
    //If count divided by 5 equals 0 write Bar
```

```
35   elseif(count % 5 equals 0)then
36     lcd.print("Bar");
37     Serial.println("Bar");
38   end
39   else then
40   //All other times write the number
41     lcd.print(count);
42     Serial.println(count);
43   end
44 end
45
46   count = count + 1;
47   if(count > 100)then
48     count = 2;
49   end
50   // delay at the end of the full loop:
51   delay(1000);
52
53
54 end
```

Part IV

Implementation

9 JavaCC - Java Compiler Compiler

JavaCC is a parser generator that generates a fully functioning parser. JavaCC takes a file containing EBNF grammar and then makes a program that can recognize matches between the input code and the grammar, this part of the parser is called the scanner. JavaCC also creates other things like e.g. an AST (Abstract Syntax Tree, see section 9.2) that decides the derivation of the code. [Jav]

JavaCC generates top-down parsers (LL(k) parsers) which means that it replaces every nonterminal until the string is created. LL(k) grammars produce a left-to-right symbol scan which means that it produces a leftmost derivation. The LL(k) grammars use a maximum of *k* symbols of lookahead(This is why it is called a LL(k) grammar).For more information on lookahead see section 9.5. [**?**]

The process of a parser program:

- Input a set of token definitions, grammar and actions
- Outputs a Java program which performs lexical analysis
  - Finding tokens.
  - Parses the tokens according to the grammar.
  - Executes actions.

9.1 Grammar example

Code table 9.1 is containing an example of how a grammar for an "if"-statement can be constructed. The grammar in the example states the following:

- Skip all whitespace and newlines.
- An "if statement" consists of a condition (C), a statement (S) and optionally an "else statement" which contains a Statement (S)·
- A condition (C) consists of the string "TBD" - not anything else.
- A statement (S) also consists of the string "TBD" or a new "if"-statement

```
1
2 PARSER_BEGIN(Example)
3
4 public class Example {
5
6   public static void main(String args[]) throws
       ParseException {
7     Example parser = new Example(System.in);
8     parser.IfStm();
9   }
10
11 }
```

```
12
13 PARSER_END(Example)
14
15 SKIP :
16 {
17   " "
18 | "\t"
19 | "\n"
20 | "\r"
21 }
22
23 void IfStm() :
24 {}
25 {
26  "if" C() S() [ "else" S() ]
27 }
28
29 void C() :
30 {}
31 {
32   "TBD"
33 }
34
35 void S() :
36 {}
37 {
38   "TBD"
39 |
40   IfStm()
41 }
```

Code 9.1: One of JavaCC's standart examples on how to make a grammar that accepts "if"-statements.

Using JavaCC a parser is generated that is used in the final compiler to determine wether the source language is written correctly or not. The code generation part of the compiler (more info in section) uses the stream of tokens sent from the parser to specify how to translate the source language into the target language.

ref here

9.2 Abstract Syntax Tree

As described in section 2.1, the parser generates an abstract syntax tree (AST). An AST is a representation of the syntax in a programming language in the structure of a tree. Each node of the tree representates a token in the given sentence or code block. Since the tree is abstract not every single syntactic construct is shown in the tree. As javaCC is used in the creation of this project, the AST is generated automatically based on the grammar of the source language.

AST for Function and Variable Declaration

Code 9.2: Function with variable declaration

```
1  void function Procedure() do
2    int h = 10 + 11;
3    string hello = "World!";
4  end
```

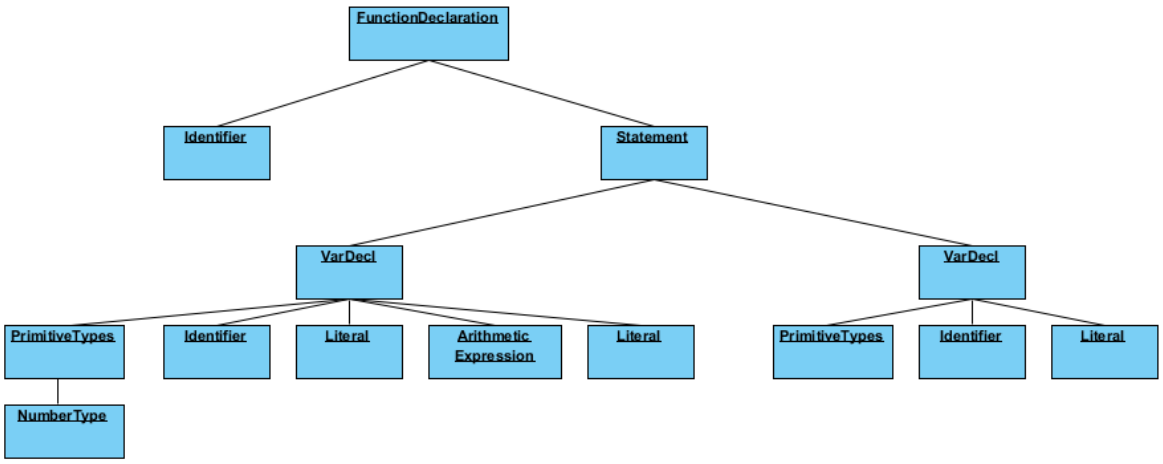

Figure 9.1: AST for listing 9.2

In listing **??** is seen a function with two different variable declarations in the source language. Figure 9.1 shows the AST generated by javaCC using the beforementioned code example. It is not concrete to the specific token. As seen in the AST there are different opportunities for declaring a variable in the source language.

AST for While Loops and If Statements

Code 9.3: While loop with if-statement

```
1  while(b <= i) do
2    if(a EQUALS s) do
3      string ok = "works!";
4    end
5  end
```

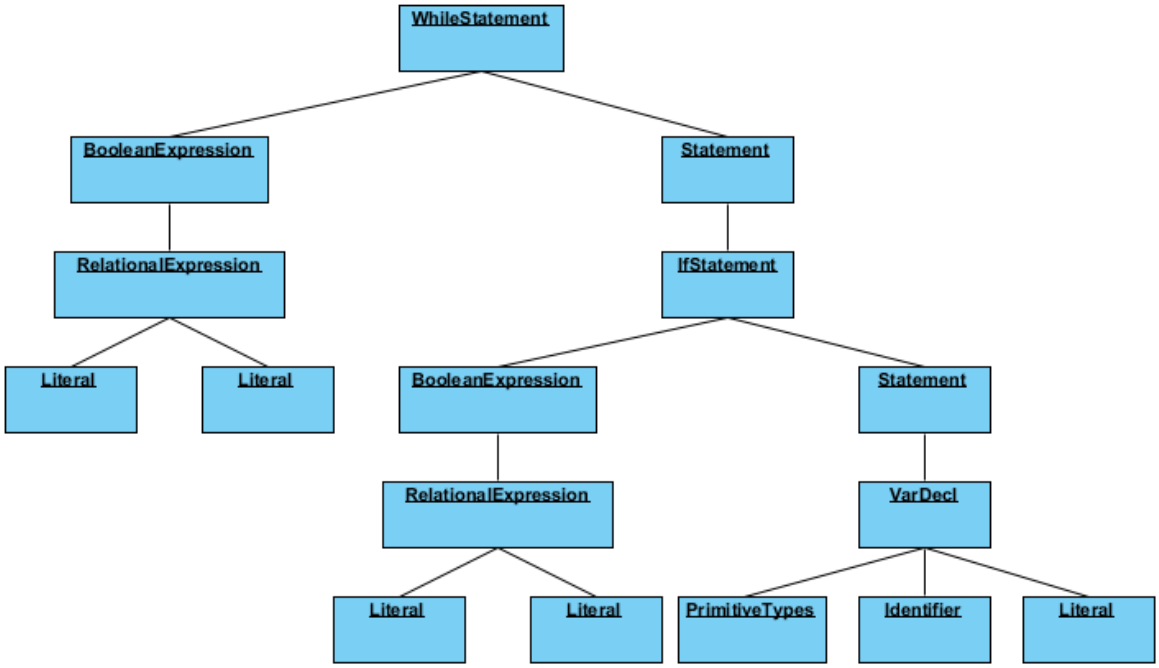

Figure 9.2: AST for listing 9.3

In listing 9.3 is seen a while loop with an if-statement in the source language. Figure 9.2 shows the AST generated from the code example.

9.3 Implementation of the Scanner

When creating the lexical analyzer part of the parser generation, context free grammar is used in the form of EBNF (see Section ??). In this section a few of the EBNF implementations will be described.

```
1 void ArithmeticExpression() :
2 {}
3 {
4   (<PLUS> | <MINUS> | <MULTIPLY> | <DIVIDE> | <MOD> | <SQRT>
        | <POW>)
5
6 }
```

One of the features for the source language is basic arithmetic expressions. These different expressions have been implemented as seen in Listing 9.3. Since the lexical analyzer reports error whenever there is a wrong use of syntax.

```
1 void IfStatement() :
2 {}
3 {
4     <IF> <LPAREN> BooleanExpression() <RPAREN> <DO>
          Statement() <END> [<ELSEIF> <LPAREN> Expression() <
          RPAREN> <DO> Statement() <END>] [<ELSE> <DO> Statement
          () <END>]
5 }
```

Another feature is the use of if statements, but with the opportunity to use only the if statement, follow up with the amount of elseif statements the programmer desires, or end with an else statement. This has been implemented as seen in listing 9.3, most importantly to note is the ability to use elseif zero or many times with the use of a regular expression. To make sure the order of the if statements are created in the right order, having an if statement before an elseif or else is required. To make sure there is no dangling else problems, every if, elseif, or else statement requires a "do" "end" block to avoid ambiguity.

9.4 Call-by methods

There are three different methods for parsing parameter to functions, these are, call-by-value, call-by-reference and call-by-name. It is possible to implement more then one of these methods.

Call-by-reference

With call-by-reference, $func(a)$ the parameter a is the address of the parameter. So consider the following example;

Code 9.4: call-by-reference example

```
1 y = 0;
2
3 func(y){
4   y + 1;
5 }
```

Since it is the location of y that is passed, the value of y after the function call will be incremented by 1.

Call-by-value

With call-by-value, ass opposed to call-by-reference, the value is copied to a local variable in the function, rather then using the original parameter. If using the example above code 9.4, the value of y after the function call will still be 0. Since it is only the value that is copied, it is also possible to make calls, using arithmetic expressions, if $y + 1$ is passed instead of y, then the result of the expression 1 will be passed.

Call-by-name

Call-by-name is less obvious than call-by-value and call-by-reference. Basically this method allows the passing of expressions without evaluating them. Using the previous example with call-by-value, passing $y + 1$ would be evaluated first and then passed as 1. Call-by-name is different in that it passes the expression itself. The expression will not be evaluated until it is accessed in the function. Furthermore it will be evaluated on every call. As such the pass to the function holds $y + 1$, and only when accessed

will the expression be evaluated to 1. Unique to call-by-name, this evaluation happens every time it is accessed. This allows y to be evaluated correctly every time, even if it should somehow be assigned to a new value.

Implementation

To keep it simple and easy for the user, only the call-by-value method will be implemented in PH, this will allow for passing of parameters to functions, while keeping it simple, by not including features like pointers, to help confuse the user with more then one way of accomplishing a certain task.

Name for source language placeholder

9.5 Lookahead

A parser can in some cases require a feature called lookahead. This feature allows for choosing the right path in a grammar where it is not possible to determine the correct outcome immediately. Such points leading to different paths can for example be represented by brackets in the grammar which are optional parts. A part of a grammar could be "*a*""*b*"["*c*"], an acceptable string can be a "*ab*" or "*abc*". In this case the parser will first look for "*a*", then for "*b*". Then it has to decide whether it is done reading (The string is *ab*) or if it should continue, making the string *abc*. Assuming that the string *abc* is matched, what if the caller of this grammar function now requires a "*c*"? The parser will then have made a mistake in matching *abc* too early and must back-trace all the way to the choice point to take the other option. Back-tracing is however inefficient and undesirable which is where lookahead comes in. Instead of blindly choosing one path over another lookahead allows to look ahead in the grammar and make qualified decisions. JavaCC uses this feature and does not back-trace . By default lookahead(1) is used which is also what we stick to in our grammar. While lookahead is more efficient than backtracking it can still be very inefficient. As such we have made our grammar efficient in regards to lookahead and actually the lookahead(1) is only used in our if-else grammar. In the if-else grammar, the if part will be read first. After this the parser has to decide between whether the next part is an elseif, an else or another statement.

Morten: Source needed

JONAS!! GØR VI DET??!?!?!?

Morten: Need verification/clarification

10 Test

Part V

Discussion

11 Evaluation of the Product

11.1 Improvement

11.2 Conclusion

Part VI

Appendix

11.3 Original Syntax Example 1

This is the first example of the original way to write code to Arduino. This example is a Hello World.

Code 11.1: This is the first original Arduino example [?]

```
1 // include the library code:
2 #include <LiquidCrystal.h>
3
4 // initialize the library with the numbers of the interface
    pins
5 LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
6
7 void setup() {
8   pinMode(9, OUTPUT);
9   analogWrite(9, 20);
10  // set up the LCD's number of columns and rows:
11  lcd.begin(8, 2);
12  // Print a message to the LCD.
13  lcd.print("hello, world!");
14  delay(2000);
15 }
16
17 void loop() {
18  // set the cursor to column 0, line 0
19  // (note: line 1 is the second row, since counting begins
      with 0):
20  lcd.setCursor(0, 0);
21  // print the number of seconds since reset:
22  lcd.print(millis()/1000);
23 }
```

11.4 Original Syntax Example 2

This is the second example of the original way to write code to Arduino. This is about the LCD example.

Code 11.2: This is the second original Arduino example [?]

```
1   #include <LiquidCrystal.h>
2
3 // initialize the library with the numbers of the interface
    pins
4 LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
5 int count;
6
7 void setup() {
```

```
8   pinMode(9, OUTPUT);
9   analogWrite(9, 12);
10  // set up the LCD's number of columns and rows:
11  lcd.begin(16, 2);
12  // Print a message to the LCD.
13  lcd.print("Foo Bar");
14  count = 1;
15  Serial.begin(9600);
16
17  delay(2000);
18 }
19
20 void loop() {
21  lcd.clear();
22  delay(200);
23  //If count divided by 3 and 5 equals 0 write Foo Bar
24  if(count % 3 == 0 && count % 5 == 0){
25    lcd.print("Foo Bar");
26    Serial.println("Foo Bar");
27  }
28  else{
29    //If count divided by 3 equals 0 write Foo
30    if(count % 3 == 0){
31      lcd.print("Foo");
32      Serial.println("Foo");
33    }
34    else{
35      //If count divided by 5 equals 0 write Bar
36      if(count % 5 == 0){
37        lcd.print("Bar");
38        Serial.println("Bar");
39      }
40      else{
41        //All other times write the number
42        lcd.print(count);
43        Serial.println(count);
44      }
45    }
46  }
47
48  count++;
49  if(count > 100){
50    count = 2;
51  }
52  // delay at the end of the full loop:
53  delay(1000);
54
55
56 }
```

Todo list

Bibliography

[Ard] http://arduino.cc/en/main/software.

[AS11] Dmitry Xmelkov Joerg Wunsch Marek Michalkiewicz Ruddick Lawrence Ruud Anatoly Sokolov, Eric Weddington. Avr c runtime library, Febuary 2011.

[Bar04] Hernando Barragán. Wiring: Prototyping physical interaction design. Master's thesis, Interaction Design Institute Ivrea, 2004. http://wiring.org.co/download/Wiring_thesis.pdf.

[com12] Anatomy of a compiler. http://www.cs.man.ac.uk/ pjj/farrell/comp3.html, 2012.

[Jav] http://javacc.java.net/.

[Lah09] Justin Lahart. Taking an open-source approach to hardware. The Wall Street Journal, November 2009.

[pyt13a] Official python site - about page. http://python.org/about, 2013.

[pyt13b] Official python tutorial. http://docs.python.org/2/tutorial/, 2013.