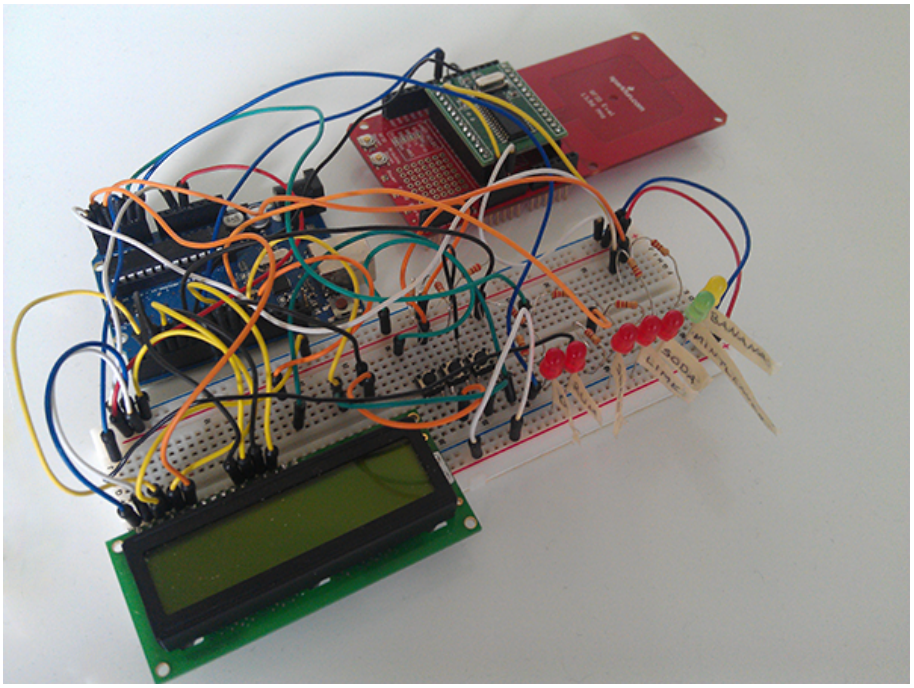
P4 Project



SPLAD

P4 project

Group SW407F13

Software

Department of Computer Science

Aalborg University

Spring 2013

AALBORG UNIVERSITY

STUDENT REPORT

AALBORG UNIVERSITY
STUDENT REPORT

Title:

SPLAD - Simple Programming Language for Arduino Drinks-mixer

Project period:
P4, spring 2013

Project group:
SW407F13

Group members:
Aleksander Sørensen Nilsson
Christian Jødal O'Keeffe
Kasper Plejdrup
Mette Thomsen Pedersen
Niels Brøndum Pedersen
Rasmus Fischer Gadensgaard

Supervisor:
Ricardo Gomes Lage

Total number of pages:
109

Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg East
http://www.cs.aau.dk/en

Synopsis:

This report contains a description of the environment where drinks are an essential part, this leads to the following problem statement: "How can a programming language be developed, which makes it suitable for the hobbyist programmer to program drink machines based on Arduino platforms?" The product of this project is a programming language with a compiler which can make it easier to program a machine that can mix and serve drinks. The program language's syntax is written in BNF and the compiler is developed using the parser and lexer generator ANTLR. Lastly unit testing is used to test certain parts of the compiler to see if they work as intended. It has been concluded that this project gives a fulfilling answer to the problem stated in this project.

Prolog

This report is written by Aleksander S. Nilsson, Christian J. O'Keeffe, Kasper Plejdrup, Niels B. Pedersen, Mette T. Pedersen and Rasmus F. Gadensgaard as a 4th semester software project. We are a group of students from the Department of Computer Science at Aalborg University (AAU). This report documents and describes the process of designing and implementing a compiler.

The references in the report will be in the format [Example, year] with a corresponding entry in the bibliography in the back of the report just before the appendix. Figures and tables will be referred to in this manner: Table 3.5, where the first number is the chapter and the second number is the index of the figure or table in that chapter.

The DVD included on the last page of the report, see appendix C contains the complete source code of the compiler, a PDF of the rapport, the compiled compiler and a sample program.

Aalborg May 27, 2013

Aleksander Sørensen Nilsson	Christian Jødal O'Keeffe
Kasper Plejdrup	Mette Thomsen Pedersen
Niels Brøndum Pedersen	Rasmus Fischer Gadensgaard

Contents

1 Introduction

The purpose of this report is to document the design and implementation of the programming language SPLAD[1] and its compiler. SPLAD is a programming language designed for programming a drink-mixer based on the Arduino platform. The goal is to make a simple imperative programming language, fitted for the hobbyist programmer who wants to build a drink-mixer. The syntax of SPLAD will be specified in a way, so it has a high degree of readability. The semantics of SPLAD will be defined, and based on the syntax and semantics, a compiler will be made. The compiler will compile the SPLAD program to the C-like programming language for Arduino.

1.1 Environment for this Project

The environment in this project is an Arduino-based drinks-machine. The basic idea behind this environment, is that the drinks-machine resides in a bar or at a party, where customers will buy RFID-tags[2], which contains the identifier of a specific drink. These RFID-tags will be read by the drinks-machine by using a RFID-reader. The machine will then automatically mix the drink, stored on the RFID-tag.

1.1.1 Solution in Bars

Currently bars and clubs often have multiple bartenders who mix the drinks and serve the customers. The bartender handles both receiving the order, the mixing the drink and the payment of the drink. This process can be done more efficiently. If a bar had a drinks-machine, the bar would require fewer bartenders instead of multiple bartenders. The cashier would handle selling and programming the RFID-tags. The customers themselves then place the RFID-tag on the RFID-reader on the drinks-machine. The machine mixes the appropriate drink, and either decreases the count on the RFID-tag. It then displays an appropriate message to the customer on the display of the machine. The RFID-tags are programmed by the cashier which encodes a drink id and a drink count onto the tag. This allows the customer to buy for example 10 mojitos on the same tag. The machine will simply check if the drink-count on the tag is above zero, and display an error if it is not.

The system can also be used in many other systems, like an ice-cream machine, juice machine, or even a food dispenser in a restaurant to help with self-service. It can also be

[1] **S**imple **P**ramming **L**anguage for **A**rduino **D**rink-mixer

[2] RFID is a system that uses radio waves to read different encoded tags. See section 3.2.2 for more information.

used in a cinema for regular customers to get popcorn, coke, or others forms of sweets for the movies. There are many possibilities on how to use such a system, but in this project a drink-mixer will be the focus.

1.2 Problem Statement

In this section a problem statement will be presented, which will be used as a basis for this project. In this project it has been decided to examine, how a drinks machine could be programmed using Arduino as a platform for the processing. The programming language usually used for Arduino is based on C and C++, which is not aimed at programming a drinks machines, see section 3.2.1. It could be useful to have a niche programming language aimed directly at programming drinks machines on an Arduino platform. This will be the goal of this project.

The programming language in this project is aimed at the hobbyist programmer who wants to program his own drinks machine. Because of this, the programs written in this language must be simple to understand and maintain. This however sacrifices some write-ability of the programs, because of constraints imposed to ensure programs are easier to understand. These trade-offs will be further discussed in section 2.3.3. A hobbyist programmer is defined as a programmer who knows the basic structure of programming, but does not have an education in programming or work with software development.

This leads to the following problem statement:

- **How can a programming language be developed, which makes it suitable for the hobbyist programmer to program drinks machines based on Arduino platforms?**

The purpose of this problem statement is to guide the development of the programming language for this project, so when it reaches its final state, it is as simple as possible for hobbyist programmers to program when using the language.

1.2.1 Sub Statements

On the basis of the problem statement, a number of sub-statements arises:

- **How can a programming language be specified, which makes it suitable for novice programmers?** Because the language of this project is aimed at hobbyist programmers, the programming language should be specified in a way which is suited for the target group.
- **How can a compiler be developed, which recognizes the language, and translates the source program into suitable code for Arduino?** Of course it is not enough to have a simple-to-understand language, if no compiler exists for that language. The language would then be rendered useless for the purpose. This is the reason why a compiler must be developed, either by compiling the program code directly to Arduino machine code, or by first compiling the program code to an intermediate language, and then use the Arduino compiler to compile that code further.

1.3 Report Structure

The structure of the report is as follows:

- In chapter 2 the language specifications will be described which contains the design criteria for the language as well as the syntax and semantics of SPLAD.
- In chapter 3 the implementation of SPLAD will be described which contains the different compiler phases, the criteria for the compiler, and the code of the compiler.
- Chapter 4, contains a discussion about the project, and the project is reflected upon.
- In chapter 5, the project it is concluded how the project fulfills the problem statement.

Language Specification 2

In this chapter the four paradigms in programming languages are described. The paradigm which is the focus of the programming language in this project is described in more thoroughly. Moreover, the prioritization in the design criteria for this project is also described. The theory and design of syntax and semantics is also described. Lastly this chapter contains a few code examples to demonstrate the SPLAD.

2.1 Programming Language Paradigms

In computer science a paradigm means "A pattern that serves as a *school of thoughts* for programming of computers" [Nørmark, 2010b]. There are four main paradigms of programming languages [Nørmark, 2010a]. In this section these paradigms will be briefly described followed by a subsection, explaining the choice of programming paradigm of the language in this project.

2.1.1 Imperative Programming

Imperative programming is a sequential or procedural way to program, in the sense that a step is performed, then another step and so on. These steps are controlled by control-structures for example an if-statement. An example of an imperative programming language is C. Imperative programming languages describe programs in terms of statements which alter the state of a program. This makes imperative languages simple, and a good starting point for new programmers [Nørmark, 2010a].

2.1.2 Functional Programming

Functional programming originates from the theory of functions in mathematics. In functional programming all computations are done by calling functions. The function calls of a functional programming language will always yield the same result, if the function is called with the same parameters as input. This is in contrast to imperative programming where function calls can result in different values depending on the state of the program at the given time. Some examples of functional programming languages are Haskell and OCaml. Example 2.1 is a piece of pseudo code which shows that a function could return a different result, if it is written in a functional programming language or not.

```
1     foo(x) + foo(x) = 2*foo(x)
```

Listing 2.1: A pseudo code in functional programming [Popplestone, 1999].

The code on listing 2.1 will always be true in functional programming paradigm, but in other paradigms such as the imperative programming paradigm, there can be a global variable which is used in the function "foo()" that can change on runtime and it could thereby return false [Nørmark, 2010a].

2.1.3 Object-oriented Programming

Object-oriented programming is based on the idea of data encapsulation, and grouping of logical program aspects. The concept of parsing messages between objects is also a very desirable feature when programs reach a certain size. In object-oriented programming, each class of objects can be given methods, which is a kind of function that can be called on that object. For example the expression "foo.Equals(bar)", would call the Equals-method in the class of 'foo' with the parameter 'bar'. It is also relatively simple in object-oriented languages to specify access-levels of classes, and thereby protect certain classes from external exposure. Classes can inherit from other classes. For example one could have a 'Car'-class, which inherits all properties and methods of a 'Vehicle'-class. This allows for a high degree of code-reuse [Nørmark, 2010a].

2.1.4 Logic Programming

Logic programming is fundamentally different from the imperative-, functional-, and object-oriented programming languages. In logic programming, it cannot be stated how a result should be computed, but rather the form and characteristics of the result. An example of a logic programming language is Prolog [Nørmark, 2010a]. An example of a hello world program written in Prolog can be seen on listing 2.2

```
1 :- write('Hello, world!'),nl.
```

Listing 2.2: Hello world program written in Prolog

2.1.5 Choice of Paradigm in this Project

For this project, an imperative approach has been chosen. The reason for this, is that the programming language of this project should be simple to both understand and program for beginners. To provide this friendliness towards beginners, the programming language will likely have a relatively simple syntax, meaning that it should have short simple commands that are easy to understand. This makes object-orienting undesirable because these type of languages has a tendency to provide a lot of functions and allows users to call commands upon commands which can result in long sequences of commands.

2.2 Design Criteria

In this section the criteria for making the programming language is described, and which of these criteria are the focus point in SPLAD.

2.2.1 Programming an Arduino-based Drinks-mixer

To make it easier to program a drinks machine, it would be nice to have a programming language aimed specifically at this. See chapter 1 for a description of the problem. Based on this it has been decided to make a programming language aimed at the stated problem. The is called SPLAD which stands for: **S**imple **P**rogramming **L**anguage for **A**rduino **D**rink-mixer. The SPLAD is described more thoroughly in section 1.2.

2.2.2 Design Criteria in this Project

To determine how a programming language should be syntactically described, the trade-offs of designing a programming language must be taken into account. The different criteria of a programming language, that will be used to evaluate trade-offs can be seen in table 2.1.

Readability:	How easy it is to understand and comprehend a computation.
Writability:	How easy it is for the programmer to write a computation clearly, correctly, concisely and quickly.
Reliability:	Assures a program behaves the way it is suppose to do.
Simplicity:	When a program e.g. has a small number of basic constructs, and it is simple in the way that it has only one way to accomplish a particular operation.
Orthogonality:	A relatively small set of primitive constructs can be combined legally in a relatively small number of ways.
Data types:	The presence of appropriate data types in the language.
Syntax design:	The syntax, or form, of the elements of a language. E.g. special words like "while" and "if".
Support for abstraction:	Abstraction means the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored.
Expressivity:	A language can refer to several different characteristics. It means that a language has a relatively convenient, rather than cumbersome, way of specifying computations.
Type Checking:	Testing for type errors in a given program, either by the compiler or during program execution.
Exception handling:	The ability of a program to intercept run-time errors, take corrective measures, and then continue.
Restricted aliasing:	Aliasing is having two or more distinct names that can be used to access the same memory cell.

Table 2.1: Brief explanation of language characteristics [Sebesta, 2009].

These criteria are used to evaluate the trade-offs of programming languages. An overview of these can be seen on table 2.2.

Criteria	Readability	Writability	Reliability
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Table 2.2: Overview of trade-offs [Sebesta, 2009].

In table 2.2 • means that the characteristic affects the feature of the programming language where the • is placed. If there is no • in front of a feature, it means that this particular criteria does not directly affect the feature.

Based on these trade-offs, it is clear that having a simple programming language affect both readability, write-ability and reliability. This is because having a simple-to-understand language, might not make it very write-able, since code that are simple-to-understand could take longer to write. On the other hand, having a simple-to-write programming language, might not make it very readable. An example of this is the if-statement in C, which can be written both with the 'if'-keyword, or more compact. This can be seen by comparing listing 2.3 with listing 2.4, which both yield the same result. It is then clear, that the compact if-statement might be faster to write, but slower to read and understand, and opposite with the if-statement.

```
1     if (x > y)
2     {
3         res = 1;
4     }
5     else
6     {
7         res = 0;
8     }
```

Listing 2.3: Simple example of if-statement in C using the 'if'-keyword.

```
1     res = x > y ? 1 : 0;
```

Listing 2.4: Simple example of if-statement in C without using the 'if'-keyword.

When defining the syntax of a programming language, the criteria should be balance these to achieve the right amount of trade-offs for that particular language. For the language of this project, it is important that the language is simple to read and understand, because the target group is the hobbyist-programmer, who might not have much experience in programming.

Characteristic	Chosen focus for this project
Simplicity	high
Orthogonality	medium
Data types	medium
Syntax design	high
Support for abstraction	medium
Expressivity	low
Type checking	high
Exception handling	low
Restricted aliasing	low

Table 2.3: Overview of choice of focus.

Table 2.3 shows what paradigms that is in the focus for the project.

- **Simplicity:** The language of this project will not have multiple ways of doing the same simple instructions. This is to make writing code simpler for beginners. An example is that it will not be possible to increase a variable by one by writing "A++;". Instead it has to be written as "$A <- -A + 1;$".
- **Orthogonality:** It will not be possible to make classes or constructs, this is to simplify types for beginners. However it will be possible to combine various other constructs, for example calling a function in an expression.
- **Data types:** The language of this project will have five primitive types, bool, double, int, char and string and two special types, container and drink. The two special types are described in section 2.3.4.
- **Syntax design:** It will not be possible to make classes in the language of this project but there will be both "for" and "while" loops because they are relatively simple to use and understand for beginners. All functions and loops will start with "begin" and end with "end" to make it clear where blocks starts and ends.
- **Support for abstraction:** Our language will be an abstraction of the Arduino language. This is done to simplify the programming for the Arduino platform and help the user by making it more simple.

- **Expressivity:** Shorthands will not be implemented in SPLAD, this is to increase the readability of the programs, as described in listing 2.3.
- **Type checking:** This is a major part of the project. The type checking will be done on compile time rather then run time, to help beginners see their errors in their code.
- **Exception handling:** Because the Arduino will execute the program-code in the end, the compiler will handle scope, type and code generation exceptions.
- **Restricted aliasing:** Since it is known that using aliasing is a dangerous feature in a programming language [Sebesta, 2009], the language of this project will not have this feature, to help beginners.

2.3 Syntax

In this section the four grammar types in the Chomsky hierarchy [Martin, 2003] are described and the syntax we have chosen for our programming language is also outlined. A deeper explanation of the grammar for our programming language is also presented.

2.3.1 Grammartypes

The Chomsky hierarchy is a hierarchy of formal grammars. There are four types of grammar in the Chomsky hierarchy, where type 0 is the most unrestricted grammar, and type 3 is the most restricted grammar. These types of grammar are described below. The Chomsky hierarchy is used to divide the grammars into different types. All the grammars in a given type is a subset of the less restricted types. This means that if a language is a type 2, it is a subset of the grammars of type 1 and 0, but it is not equal to any of these grammars [Chomsky, 1959].

2.3.1.1 Type - 3: Regular Grammar

Regular grammars can be described by finite automata or regular expressions. Regular grammars are meant to be used on computers with an extremely limited amount of memory, because regular languages do not need to use a lot of memory to recognize a language [Sipser, 2013].

2.3.1.2 Type - 2: Context-free Grammar

Context-free grammars are described by substitution rules, also called productions. Substitution rules for context-free grammars can make the grammar ambiguous. This is a problem since different computers might yield different output for the same grammar. Context-free grammars can be described in Backus Naur form and recognized by a Push-down automata (PDA). PDA's works almost in the same way as finite automata. The difference is that a PDA uses a stack as memory to help create the output [Sipser, 2013].

2.3.1.3 Type - 1: Context-sensitive Grammar

The substitution rules of context-sensitive grammars have nearly the same rules as those used in Context-free grammar. In context-sensitive the right side of the production can have more then one terminal and there can be non-terminals on the right side of the production. A context-sensitive grammar can be recognized by a linear-bounded automaton [Martin, 2003].

2.3.1.4 Type - 0: Recursively Enumerable

Recursively enumerable or unrestricted grammar is a type of grammar, where there is no restrictions on the left and right hand side of the grammars productions. On top of that, a language is recursively enumerable if it is recognized by some Turing machine [Sipser, 2013].

2.3.2 Theory of Context-free Grammars

By looking at the different types of grammars, it has become clear that a context-free grammar will be sufficient for this project language because we will be able to describe a language out of context. A grammar is used to define the syntax of a language. A context-free grammar (CFG) is a 4-tuple (V, Σ, R, S) [Sipser, 2013] finite language defined by:

1. V is a finite set called the variables
2. Σ is a finite set, disjoint from V called the terminals
3. R is a finite set of rules, with each rule being a variable and a string or variables and terminals
4. $S : S \in V$ is a start variable

The most common way of writing a CFG is by using Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF). When writing EBNF you enhance the descriptive power of the grammar. This only enhances readability and writeability [Sebesta, 2009]. BNF is named after John Backus who presented the notation, and Peter Naur who modified Backus' method of notation slightly [Sebesta, 2009]. By using the BNF-notation it is possible to describe a CFG. It is preferred to have an unambiguous grammar. A CFG is ambiguously if a string derived in the grammar has two or more different leftmost derivations [Sipser, 2013]. An unambiguous grammar will ensure that a program reading a string using CFG can only read the string in one way. It is worth spending time making a grammar unambiguous, because if the grammar is ambiguous, multiple compilations of a program using that grammar can result in different programs with different meanings and programs yielding different results [Sebesta, 2009]. But it also worth noting that some grammars are inherently ambiguous, meaning that no matter what, it can not become unambiguous.

A CFG is a part of the $LL(k)$ grammar class if it is possible to produce the leftmost derivation of a string by looking at most k tokens ahead in the string. LL algorithms works on the same subset of free grammars which means that LL parsers works on $LL(k)$ grammars. $LL(k)$ means that the grammar needs to be free of left-recursion which makes it possible to create a top-down leftmost derivation parser. The $LL(1)$ have proprieties that makes the grammar attractive for simple compiler construction. One property is that $LL(1)$ grammars are fairly easy to implement compared to $LL(k)$ where $k > 1$ to implement because the parser analyzer only has to look one element ahead in order to determine the appropriate parser action. $LL(1)$ is also relatively faster than $LL(k)$ where $k > 1$ because of the same reason: The parser only has to look one element ahead. A disadvantage of the LL grammars is that the parser finds syntax errors towards the end of its parsing process because it starts from the root, where a LR parser detects the syntax errors faster. LL is also inferior compared to LR in terms of describing a languages based on the idea

that LL is a subclass of the bigger grammar class LR. That means with a LR grammar it is possible to describe aspects of a language that might not been possible in a LL grammar [Fischer et al., 2009] [Sebesta, 2009].

A CFG is a part of the $LR(k)$ grammar classes if it is possible to produce the rightmost derivation in reverse of a string by looking at most k tokens ahead in the string. LR grammars are a superset for the LL grammars meaning that LR covers a larger variety of programming languages than LL. LR parsers are bottom-up parsers meaning that they begin constructing the abstract syntax tree (AST) from its leaf and works its way to the root. Abstract syntax trees are described in section 3.4.6 LR parsers are generally harder to implement by hand than LL parsers but there exists tools which automatically generates LR parsers for a given grammar. $LR(k)$ grammars allows left recursion which means that the LR grammars are a bigger grammar class than LL. $LALR$ and $SLAR$ is subclasses of the $LR(k)$ grammars which means that $LR(k)$ describes a larger class of languages at the cost of a bigger parser table in comparison to $SLAR$ and $LALR$. The balance of power and efficiency makes the $LALR(1)$ a popular table building method compare to LR building method [Fischer et al., 2009] [Sebesta, 2009].

2.3.3 Choice of Grammar

The programmer using the language of this project, could be a hobbyist programmer, who wants to program a custom drinks machine, but does not possess a high level of experience in programming. This is the reason why it was decided that the grammar should have a high level of readability (see section 2.2.2), because this in turn will ensure that it is easier for the programmers to read and understand their programs - this is also useful if the code has to be maintained later on. This on the other hand can decrease the level of write-ability because the programs have to be written in a specific way, and will need to contain some overhead in form of extra words and symbols to mimic a language easier for the hobbyist programmer to comprehend.

The method to assign a value to a variable is by typing "$variable <--$ 'value to assign'", without the quotes. This approach has been chosen instead of the more commonly used "=" symbol, because a person not accustomed to programming might confuse which side of the "=" is assigned to the other. Thus by using the arrow, it is clearly indicated that the value is assigned to the variable, and therefore providing more readability.

When declaring a function it has to be written on the form "function $functionname$ return $type$ using ($parameter(s)$) begin $statements$ return $expression$ end". Where $function\text{-}name$ is the name of the function that is about to be declared, $type$ is the type of value that is returned by the function. $Parameter(s)$ are used to parse the function some input values from its call(s). $Statements$ is where the function can call other functions, declare variables, calculate and assign values. $Expression$ is where the value of the right type is returned or an expression which result is of the correct type. An example of this can be seen on listing 2.5.

```
1     function DoSomething return int using (int x)
2     begin
3       x <-- x + 1;
4       return x;
5     end
```

Listing 2.5: Example of function declaration in SPLAD.

To get more continuity in the structure of the code the functions must always return something, but it can return the value "nothing". This will ensure a better understanding and readability of the code because the programmer can see what it returns, even if no parameter was provided. To indicate that *return* is the last thing that will be executed in a function, the *return* must always be at the end of a function. To indicate that a function is called "call *functionname*" must be written. Words are used instead of symbols, when suitable, to improve the understanding of the program(compared to most other programming languages). "begin" and "end" are used to indicate a block (e.g. an "if" statement). To combine logical expression the words "AND" and "OR" are used. The ";" symbol is used to improve readability by making it easier to see when the end of a statement has been reached.

It would be appropriate to design a grammar that is a subset of $LL(1)$ grammars. This is based on the idea that it is easier to implement a parser for $LL(1)$ grammars by hand compared to LR grammars [Fischer et al., 2009]. This approach means it would be possible to both implement a parser by hand or use some of the already existing tools. This way both approaches are possible which is a suitable solution for the project because it allows the project group to later go back and make the parser by hand instead of using a parser generator if so desired.

If the purpose was to create an efficient compiler it would be more appropriate to design the grammar as a subset of the $LALR$ grammar class. A parser for $LALR$ is balanced between power and efficiency which makes it more desirable than LL and other LR grammars, see section 2.3.2 for more on the grammars.

2.3.4 Specification of the Language Compared to the Purpose

To specify the language, so it will make the program as suitable as possible for writing code to be used in a drink machine, we looked at what the central aspects of a drink machine could be:

- **A drink:** A drink is central to this machine. A drink should be the product made by the machine, defined by a number of ingredients. A drink should be like a recipe.
- **An ingredient:** An ingredient is an element of a drink. They will be stored in a container within the machine.
- **An RFID-tag:** An RFID-tag will contain a drink ID and an amount of how many drinks there are left for the holder of the tag.

•**An RFID-RW:** An RFID-reader and writer for Arduino, to read and write the content of an RFID-tag. See section 2.3.4.4 for details on how an RFID works.

•**An LCD:** For communicating with the user, an LCD is in some situations preferable.

•**Buttons:** For getting input from users, buttons are a possibility for example a button to allow the user to confirm the making of a drink.

•**Mechanism for pouring ingredients:** A mechanism for pouring the right amount of ingredients into the drink.

These are by our assessment the most central aspects of the drink machine system. We will now judge each each of the listed aspects, and see if it is possible to make a structure in the language which will support the programmer or in any other way make it easier to implement this aspect in the system.

2.3.4.1 A Drink

The concept of a drink is one of the most central aspects of a drink machine system. A drink should contain a recipe as a list of ingredients, and how much of the ingredients to pour. A drink also has a name, and should have some sort of ID which can be stored on an RFID-tag. We should make a structure which can implement a drink type and assign the recipe to the drink. The structure should follow the same design criteria as the rest of the language, and should be inspired by the other structures in the language. To fulfill these requirements, the declaration of a drink should have a block with "begin" and "end", and in the block have the "recipe". An example can be seen on listing 2.6.

```
1 drink [drinkname] is
2 begin
3   [recipe]
4 end
```

Listing 2.6: The first example of a declaration of a drink.

We have now decided how to define an element of the type drink. We must now look at the body of the block in the declaration (the recipe). To make it as readable as possible, it should be written as close to a normal recipe as possible. Because of that, we have decided to state it in the form "add [number] of [ingredient]", where [number] is a number representing the amount of the ingredient to add, though this will be up to the programmer to decide what type of measurement it actually is. The declaration of a drink will now be defined as seen on listing 2.7.

```
1 drink [drinkname] is
2 begin
3   add [number] of [ingredient];
4   add [number] of [ingredient];
5   ⋮
6   add [number] of [ingredient];
7 end
```

Listing 2.7: The final structure of how to declare a drink.

In some situations, a drink could be similar to another drink, with only a few changes. An example could be a drink with a double shot of alcohol. In this case, it would be the exactly same drink, but with more of one ingredient. It could also be an ingredient that should be removed, for example alcohol to make it non-alcohol, or if someone is allergic to some of the elements in a drink. Because of these situations and many other, it could be preferable to have a way to inherit the recipe from another drink, and then modify it. The declaration of a drink which inherits from another drink should be very much like the normal drink-declaration as seen on listing 2.7, but also with some significant modifications, so it is easy to see that this drink inherits from another drink. The block structure should be the same, and the way to add an amount of an ingredient should be the same. We have to add a new command to the block statement: "remove". With the remove statement, it should be possible to completely remove an ingredient from a drink. It should be easy to read, and therefore, we have decided that the declaration statement before the block should be "drink [drinkname] as [drinkname of drink to inherit] but" to say that the drink is as the other drink, but with changes. The final structure will therefore be as seen on listing 2.8.

```
1 drink [drinkname] as [drinkname of drink to inherit] but
2 begin
3   add [number] of [ingredient];
4   remove [ingredient];
5   ⋮
6   add [number] of [ingredient];
7 end
```

Listing 2.8: The structure of how to declare a drink which inherits the recipe from another drink

We have now defined how the structure of the type drink should be. To formally define the syntax, we use BNF which can be seen in grammar 2.1.

⟨*drinkdcl*⟩ → drink ⟨*id*⟩ is begin ⟨*drinkstmts*⟩ end
| drink ⟨*id*⟩ as ⟨*id*⟩ but begin ⟨*changedrinkstmts*⟩ end

⟨*drinkstmts*⟩ → ⟨*drinkstmt*⟩ ⟨*drinkstmtsend*⟩

⟨*drinkstmt*⟩ → add ⟨*numeric*⟩ of ⟨*id*⟩

⟨*drinkstmtsend*⟩ → ; ⟨*drinkstmts*⟩
| ;

⟨*changedrinkstmts*⟩ → ⟨*changedrinkstmt*⟩ ⟨*changedrinkstmtsend*⟩

⟨*changedrinkstmt*⟩ → ⟨*drinkstmt*⟩
| remove ⟨*id*⟩

⟨*changedrinkstmtsend*⟩ → ; ⟨*changedrinkstmts*⟩
| ;

Grammar 2.1: The grammar for the drink declaration.

2.3.4.2 Ingredient

With the drink type defined above, we should now focus on how to handle ingredients. In the physical machine, an ingredient will in most cases be a liquid in a container. It could also be leaves or fruit slices, also in a container. When pouring an ingredient, no matter what ingredient it is, the drink machine should in some way (different for each type of ingredient) send a signal to the container to open and pour the wanted amount of the ingredient. When sending a signal in Arduino, you change the output voltage on a pin, so to make it easier for the programmer, we could implement a type container which hold the pinnumber for the signal to the container. The structure of this can be seen on listing 3.24.

```
1 container [containername] <-- [containerpin]
```

Listing 2.9: The structure of how to declare a container.

2.3.4.3 An RFID-tag

For a customer to use the drink machine, they should buy an RFID-tag with information of which drink and how many of the drink they have bought. A solution for these RFID-tags, is to take care of this for the programmer, so they should not deal with how to store the information and read it. It should therefore be provided by functions built into the language. This can be seen in section 2.3.4.4.

2.3.4.4 An RFID-RW

To read and write the RFID-tags, an RFID reader and writer is required. To simplify the communication with the RFID-RW, the language could, as mentioned in section 2.3.4.3,

take care of it. When writing information to a tag, you should call a function with two parameters: The drink ID and the number of drinks. A call to the function RFIDWrite could be as the example seen on listing 2.10.

```
1 call RFIDWrite([drinkID], [Amount]);
```

Listing 2.10: An example of the call to RFIDWrite.

2.3.4.5 An LCD

For communicating with the user of the drink machine, an LCD would be preferable like the one seen in figure 2.1.

Figure 2.1: An example of a LCD to use in a drink machine [let elektronik].

Like the function provided for the RFID-RW (see section 2.3.4.4), it could simplify the programmers job if the language provided a function to print text on the LCD. The function should take two parameters: the string to print on the display and an integer which indicates which line to print the string on. This function will be called LCDPrint. It should also be possible to clear the LCD. The function to do this will be called LCDClear. An example of a call to the functions can be seen on listing 2.11.

```
1 call LCDPrint("[text to print]", [linenumber]);
2 call LCDClear();
```

Listing 2.11: An example of how to call the LCD functions.

2.3.4.6 Button

To get input from either the staff or the customer, buttons could be used. It could be possible to make a button type in the language, and make some operations for the button easier. This could be making a listener, which will call a function when a button is pressed, or other button-functions to be built into the language. This is considered a good idea,

but because it is not essential for the project it will not be implemented, but it could be made if the language were to be developed further.

2.3.4.7 Mechanism for Pouring Ingredients

The mechanism to pour an ingredient would be very different. A liquid will for example not be put into the drink the same way as leaves would be put into it. For this reason, it has been decided not to support the pouring mechanism in any special way in the language, because there are too many possible ways to make a pouring mechanism. Example of these can be seen in section 3.8.

2.3.5 The BNF of SPLAD

This section contains the BNF for the programming language of this project. It is clear that the BNF begins with the non-terminal "program". The "program" non-terminal can then be derived in a number of ways, to represent a specific program. The grammar for SPLAD is presented in the following sections.

2.3.5.1 Grammar - Types

This section contains the grammar related to types. This can be seen on grammar 2.2.

⟨*program*⟩ → ⟨*roots*⟩

⟨*roots*⟩ → ε
| ⟨*root*⟩ ⟨*roots*⟩

⟨*root*⟩ → ⟨*dcl*⟩;
| ⟨*assign*⟩;
| ⟨*function*⟩
| ⟨*COMMENT*⟩
| ⟨*drinkdcl*⟩

⟨*dcl*⟩ → ⟨*type*⟩ ⟨*assign*⟩

⟨*type*⟩ → ⟨*constant*⟩ ⟨*primitivetype*⟩
| ⟨*specialtype*⟩

⟨*constant*⟩ → const
| ε

⟨*primitivetype*⟩ → bool
| double
| int
| char
| string

⟨*specialtype*⟩ → drink
| container

⟨*id*⟩ → ⟨*LETTER*⟩

Grammar 2.2: Grammar for types.

2.3.5.2 Grammar - Drinks

The grammar for the drinks is described in section 2.1.

2.3.5.3 Grammar - Assign

This section contains the grammar related to assignments. This can be seen on grammar 2.3.

⟨*assign*⟩ → ⟨*callid*⟩ ⟨*assignend*⟩

⟨*assignend*⟩ → <-- ⟨*expr*⟩
| ε

⟨*expr*⟩ → ⟨*term*⟩ ⟨*exprend*⟩

⟨*term*⟩ → ⟨*comp*⟩ ⟨*termend*⟩

⟨*comp*⟩ → ⟨*addsub*⟩ ⟨*compend*⟩

⟨*addsub*⟩ → ⟨*muldiv*⟩ ⟨*addsubend*⟩

⟨*muldiv*⟩ → ⟨*factor*⟩ ⟨*muldivend*⟩

⟨*factor*⟩ → (⟨*expr*⟩)
| !(⟨*expr*⟩)
| ⟨*callid*⟩
| ⟨*numeric*⟩
| A ⟨*DIGIT*⟩
| ⟨*string*⟩
| ⟨*functioncall*⟩
| ⟨*cast*⟩
| LOW
| HIGH
| true
| false
| INPUT
| OUTPUT

⟨*callid*⟩ → ⟨*id*⟩ ⟨*arrayidend*⟩

⟨*arrayidend*⟩ → ⟨*arraycall*⟩ [⟨*expr*⟩]
| ε

⟨*arraycall*⟩ → [⟨*expr*⟩] ⟨*arraycall*⟩
| [ ] ⟨*arraycall*⟩
| ε

⟨*numeric*⟩ → ⟨*plusminusorempty*⟩ ⟨*DIGIT*⟩ ⟨*numericend*⟩

Grammar 2.3: Grammar related to assignments.

2.3.5.4 Grammar - Expressions

This section contains the grammar related to expressions. This can be seen on grammar 2.4.

⟨*addsubend*⟩ → ⟨*plusminus*⟩ ⟨*addsub*⟩
| *ε*

⟨*muldivend*⟩ → ⟨*timesdivide*⟩ ⟨*muldiv*⟩
| *ε*

⟨*timesdivide*⟩ → *
| /

⟨*plusminusorempty*⟩ → *ε*
| ⟨*plusminus*⟩

⟨*plusminus*⟩ → -
| +

⟨*numericend*⟩ → *ε*
| . ⟨*DIGIT*⟩

⟨*string*⟩ → ⟨*STRINGTOKEN*⟩

⟨*functioncall*⟩ → call ⟨*id*⟩ (⟨*callexpr*⟩)

⟨*callexpr*⟩ → ⟨*subcallexpr*⟩
| *ε*

⟨*subcallexpr*⟩ → ⟨*expr*⟩ ⟨*subcallexprend*⟩

⟨*subcallexprend*⟩ → , ⟨*subcallexpr*⟩
| *ε*

⟨*compend*⟩ → ⟨*comparisonoperator*⟩ ⟨*comp*⟩
| *ε*

⟨*comparisonoperator*⟩ → >
| <
| <=
| >=
| !=
| =

⟨*termend*⟩ → ⟨*termsymbol*⟩ ⟨*term*⟩
| *ε*

⟨*termsymbol*⟩ → AND

⟨*exprend*⟩ → ⟨*exprsymbol*⟩ ⟨*expr*⟩
| *ε*

⟨*exprsymbol*⟩ → OR

Grammar 2.4: The grammar related to expressions.

2.3.5.5 Grammar - Functions

This section contains the grammar related to functions. This can be seen on grammar 2.5.

⟨*cast*⟩ → ⟨*type*⟩ (⟨*expr*⟩)

⟨*function*⟩ → function ⟨*id*⟩ return ⟨*functionmid*⟩

⟨*functionmid*⟩ → ⟨*type*⟩ ⟨*functionend*⟩ ⟨*expr*⟩; end
| nothing ⟨*functionend*⟩ nothing; end

⟨*functionend*⟩ → using (⟨*params*⟩) begin ⟨*stmts*⟩ return

⟨*params*⟩ → ⟨*subparams*⟩
| ε

⟨*subparams*⟩ → ⟨*type*⟩ ⟨*callid*⟩ ⟨*subparamsend*⟩

⟨*subparamsend*⟩ → , ⟨*subparams*⟩
| ε

⟨*stmts*⟩ → ε
| ⟨*stmt*⟩ ⟨*stmts*⟩

⟨*stmt*⟩ → ⟨*assign*⟩;
| ⟨*nontermif*⟩
| ⟨*nontermwhile*⟩
| ⟨*from*⟩
| ⟨*dcl*⟩;
| ⟨*functioncall*⟩;
| ⟨*nontermswitch*⟩
| ⟨*COMMENT*⟩

Grammar 2.5: The grammar related to functions.

2.3.5.6 Grammar - Loops

This section contains the grammar related to loops. This can be seen on grammar 2.6.

⟨*nontermif*⟩ → if(⟨*expr*⟩) ⟨*block*⟩ ⟨*endif*⟩

⟨*endif*⟩ → else ⟨*nontermelse*⟩
| ε

⟨*nontermelse*⟩ → ⟨*nontermif*⟩
| ⟨*block*⟩

⟨*nontermwhile*⟩ → while(⟨*expr*⟩) ⟨*block*⟩

⟨*from*⟩ → from ⟨*assign*⟩ to ⟨*expr*⟩ step ⟨*plusminusorempty*⟩ ⟨*DIGIT*⟩ ⟨*block*⟩

⟨*block*⟩ → begin ⟨*stmts*⟩ end

⟨*nontermswitch*⟩ → switch (⟨*expr*⟩) begin ⟨*cases*⟩ end

⟨*cases*⟩ → case ⟨*expr*⟩: ⟨*stmts*⟩ break; ⟨*endcase*⟩

⟨*endcase*⟩ → ⟨*cases*⟩
| default: ⟨*stmts*⟩ break;

Grammar 2.6: The grammar related to loops.

2.3.6 Lexicon

The lexicon is used by *ANTLR*. In the lexicon tokens are written with capital letters. The tokens are expressed with regular expressions, where "." means any character. This can be seen on grammar 2.7.

⟨*STRINGTOKEN*⟩ → " .*? "

⟨*LETTER*⟩ → [a - zA - Z] $^+$

⟨*DIGIT*⟩ → [0 - 9] $^+$

⟨*NOTZERODIGIT*⟩ → [1-9][0-9] *

⟨*COMMENT*⟩ → /* .*? */

Grammar 2.7: The grammar for the lexicon.

2.4 Semantics

In this section the semantics of SPLAD is described.

2.4.1 Theory of Scope Rules

The scope of a variable is the block of the program in which it is accessible. A variable is local to a block, if it is declared in that block. A variable is non-local to a block if it is not declared in that block, but is still visible in that block (ex. global variables) [Sebesta,

2009]. The scope rules a the language determines how a variable name is associated with a value in a particular occurrence. When working with a functional language, the program needs to know how a name is associated with an expression when a variable is declared in a program unit or block. When a variable is declared in a block it is local to that part. The non-local variables are visible within the block if they are not declared there. Lastly global variables are a special category of non-local variables.

2.4.1.1 Static Scope

Static scoping was introduced in ALGOL 60 [Sebesta, 2009]. This type of scoping binds names to non-local variables. There are two categories of statically-scoped languages. The first one is where sub-programs can be nested and where sub-programs creates nested static scopes. The other is static scoping where sub-programs cannot be nested, but sub-programs do create static scopes. In the latter nested scopes are only created by blocks or class definitions.

Blocks are used to define new static scopes in many languages. The idea is that it allows a section of code to have its own local variables.

An example on the use of blocks can be seen on listing 2.12. Before the block, the variable x is declared and set to the value 5. In the block, x is set to 10 and an extra variable y is declared and is set to the value 15. y is only visible inside the block, thereby it can not be called outside of the block. After the block, x still has the value 10 which was given inside the block.

```
1 int x = 5
2 {
3     int y = 15
4     x = 10
5 }
```

Listing 2.12: A simple code example of the use of blocks.

2.4.1.2 Dynamic Scope

With dynamic scope, the scope is determined at run time, because it is based on the calling sequence of sub-programs and not their spatial relationship to one another.

2.4.1.3 Declaration Order

The main thing about declaration order is how the data declarations are made. They can be before functions, like C89, before they are used, like C#, or they can be anywhere in the code, like C99, C++, Java and JavaScript [Sebesta, 2009].

2.4.1.4 Global Scope

Some languages allow a program structure to be a sequence of function definitions, like C, C++, PHP. Definitions outside functions in a file creates global variables, which makes it visible to those functions [Sebesta, 2009].

2.4.1.5 Dynamic vs. Static Scope

To better understand the difference between dynamic and static scope a code example can be seen in listing 2.13.

```
1 int b = 5;
2
3 int foo()
4 {
5   int a = b + 5;
6   return a;
7 }
8
9 int bar()
10 {
11   int b = 2;
12   return foo();
13 }
14
15 int main()
16 {
17   foo();
18   bar();
19   return 0;
20 }
```

Listing 2.13: A simple code example showing the difference between static and dynamic scoping [msujaws, 2011].

Listing 2.13 will in both cases return 10 in the foo() function, but in bar() the result will differ. With static scoping the bar() function will return 10 because at compile time b was set to 5 while with dynamic scoping it will return 7 because at run time b is set to 2.

2.4.2 Scoping in this Project

In SPLAD, static scoping is used. This means that scopes are computed at compile time, based on the program code. The main reason for this, is that programs for the Arduino platform are mainly written in C, which also uses static scoping [Sebesta, 2009]. This makes the compilation from SPLAD to C simpler for the compiler. Static scoping means that a hierarchy of scopes are maintained during compilation. To determine the name of used variables, the compiler must first check if the variable is in the current scope. If it is, the value of the variable is found, and the compiler can proceed, else it must recursively search the scope hierarchy for the variable. When done, if the variable is still not found, the compiler returns an error, because an undeclared variable is used [Sebesta, 2009].

2.4.3 Symbol Tables

Symbol tables are used to store information like type and attributes about names in the program to be compiled. Generally there are two approaches to symbol tables: One symbol table for each scope, or one global symbol table [Sebesta, 2009].

2.4.3.1 Multiple Symbol Tables

In each scope, a symbol table exists, which is an ADT (Abstract Data Type), that stores the name of the identifiers and relate each identifier to its attributes. The general opera-

tions of a symbol table is: Empty the table, add entry, find entry, open and close scope [Sebesta, 2009].

It can be useful to think of the structure of static scoping and nested symbol tables as a kind of tree structure. Then when the compiler analyzes the tree, only one branch/path is available at a time. This exactly creates these features of e.g. local variables.

A stack might intuitively make sense because of the way scopes are defined by "begin" and "end". When a scope begins a symbol table is simply pushed onto the stack, and when the scope ends, the symbol table is popped from the stack. This also accounts for nested scopes. Searching for a non-local variable would require searching the entire stack [Sebesta, 2009].

2.4.3.2 One Symbol Table

To maintain one symbol table for a whole program, each name will be in the same table. The names must therefore be named appropriately by the compiler, so that each name also contain information about nesting level. Various approaches to maintain one symbol table exists, for example maintaining a binary search tree might seem like a good idea, because it is generally searchable in $O(lg(n))$. The fact that programmers generally do not name variables and functions at random, causes the search to take as long as linear search. Therefore hash-tables are generally used. This is because of hash-tables perform excellent, with insertion and searching in $O(1)$, if a good hash function and a good collision-handling technique is used [Sebesta, 2009].

2.4.4 Transition Rules

In this section some of the transition rules in SPLAD will be explained. The complete list of all the rules can be seen in appendix A. In the following text we use the following names to represent different syntactic categories.

- $n \in \mathbf{Num}$ - Numerals
- $x \in \mathbf{Var}$ - Variables
- $r \in \mathbf{Arrays}$ - Array names
- $a \in \mathbf{A_{exp}}$ - Arithmetic expressions
- $b \in \mathbf{B_{exp}}$ - Boolean expressions
- $S \in \mathbf{Stm}$ - Statements
- $R \in \mathbf{Stm}$ - Root statements
- $p \in \mathbf{Pnames}$ - Procedure names
- $D_V \in \mathbf{DecV}$ - Variable declarations
- $D_P \in \mathbf{DecP}$ - Procedure declarations
- $D_A \in \mathbf{DecA}$ - Array declarations

FiXme Fatal: der skal tilføjes noget med astrakt syntax, præcedens og sådan noget, kig i Hans' bog

2.4.4.1 Abstract Syntax

In order to describe the behavior of a program, one must first account for its syntax. We use the notion of abstract syntax, since it will allow us to describe the essential structure of a program. In other words, the abstract syntax are not concerned with operator precedence etc. The abstract syntax for SPLAD is defined as follows:

$$
\begin{aligned}
R &::= D_P\ D_A\ D_V\ |R_1\ R_2 \\
S &::= x := a \mid r[a_1] := a_2 \mid S_1;\ S_2 \mid \text{if } b \text{ begin } S \text{ end} \mid \text{if } b \text{ begin } S_1 \text{ end else begin } S_2 \text{ end} \\
&\quad \text{while } b \text{ begin } S \text{ end} \mid \text{from } x := a_1 \text{ to } a_2 \text{ step } a_3 \text{ begin } S \text{ end} \mid \text{call } p(\vec{x}) \mid D_V \mid D_A \\
&\quad \mid \text{switch}(a) \text{ begin case } a_1 :\ S_1 \text{ break; } \ldots \text{ case } a_k :\ S_k \text{ break; default} :\ S \text{ break end} \\
a &::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1/a_2 \mid (a) \mid r[a_i] \\
b &::= a_1 = a_2 \mid a_1 > a_2 \mid a_1 < a_2 \mid \neg b \mid b_1\ \wedge\ b_2 \mid b_1\ \vee\ b_2 \mid (b) \\
D_V &::= \text{var } x := a \mid \varepsilon \\
D_P &::= \text{ func } p(\vec{x}) \text{ is begin } S \text{ end} \mid \varepsilon \\
D_A &::= \text{array } r[a_1] \mid \varepsilon
\end{aligned}
$$

We assume a collection of syntactic categories and for each syntactic category we give a finite set of formation rules which defines how the inhabitants of the category can be built [Hüttel, 2010].

2.4.4.2 Transition System

A transition system is used to describe how the constructions of a programming language behaves given a state and a cause. The cause is represented as a transition and given a state it will result in a movement in the system.

A transition system is defined as a 3-tuple, hence a triple. These three sets are: A set of configurations (Γ), a set of transition relations ($\rightarrow$) and a set of terminal configurations (T). A configuration is a description of a program and the state of it. Transition relations describes how the program moves from state to state. The terminal configuration describes the states where the program ends, meaning that when one of these are reached the program terminates [Hüttel, 2010].

We use the transition system to describe the semantics of our programming language. It gives us a tool to formally describe precisely how the language should behave. This results in a more precise way of implementing the rules we want, compared to if it was described in a informal way. There are two ways of using a transition system, big-step and small-step.

2.4.4.3 Big-step vs. Small-step Semantics

There are two ways to describe the computation of transition systems, big-step and small-step. In Big-step-semantics a transition describes the full computation, meaning there are not multiple steps in the computation - the whole computation is done in one step. To describe a computation step by step, small-step-semantics are used. These allow each step in the computation itself to be described [Hüttel, 2010].

The property of big-step-semantics make it easier to formulate the transition rules because they do not have to describe the steps in the computations. The downside of big-step-semantics is that it makes it almost impossible to describe parallelism in a language.

The reason for this, is that to describe parallelism in a language, it will be necessary to describe each step of the computations so that the system for example is allowed to switch between two statements. This is also the reason why small-step-semantics is the most reasonable choice when describing parallelism. On the other hand small-step-semantics is not as easy to describe as big-step-semantics [Hüttel, 2010].

We have decided to use big-step-semantic because we do not wish to describe parallelism in this project. Therefore big-step-semantic is a more appealing choice because it is easier to formulate compared to small-step-semantics.

2.4.4.4 Environment-store Model

In this project we use the *environment-store model* to represent how a variable is bound to a storage cell (called a *location*), in a computer, and that the value of the variable is the content of the bound location. All of the possible locations are denoted by **Loc** and a single location as $l \in \mathbf{Loc}$. We assume all locations are $\mathbf{Loc} = \mathbb{N}$. Since all locations are natural numbers we can define a function to find the next location: $\mathbf{Loc} \to \mathbf{Loc}$, where $l = l + 1$ [Hüttel, 2010].

We define the set of stores to be the mappings from locations to values $\mathbf{Sto} = \mathbf{Loc} \rightharpoonup \mathbb{N}$, where *sto* is a single element in **Sto**.

A variable-environment is like a symbol table containing each variable and storing the variables address. The store then describes which value that is on each address.

The following names represent the different environments.

- $env_V \in Env_V$ - Variable environment
- $env_A \in Env_A$ - Array environment
- $env_P \in Env_P$ - Procedure environment

2.4.4.5 Statements

The transition rules for the statements are on the form: $env_V, env_P \vdash \langle S, sto\rangle \to sto'$. The transition system is defined by: $(\Gamma_{\mathbf{Stm}}, \to, \mathrm{T}_{\mathbf{Stm}})$ and the configurations are defined by $\Gamma_{\mathbf{Stm}} = \mathbf{Stm} \times \mathbf{Sto} \cup \mathbf{Sto}$. The end configurations are defined by $\mathrm{T}_{\mathbf{Stm}} = \mathbf{Sto}$.

On table 2.4 the assignment rule for variables can be seen. The rule states that if a evaluates to v and x points to the location l, then v is stored in l. The rest of the transition rules for statements can be seen in appendix A

$$\text{[VAR-ASS]} \quad env_V, env_P \vdash \langle x <\!\!- a, sto\rangle \to sto[l \mapsto v]$$

where $env_V, sto \vdash a \to_a v$
and $env_V\ x = l$

Table 2.4: Transition rule for variable assignment.

2.4.4.6 Arithmetic Expressions

The transition rules for the arithmetic expressions are on the form: $env_V, sto \vdash a \rightarrow_a v$. The transition system is defined by: $(\Gamma_{\mathbf{Aexp}}, \rightarrow_a, \mathrm{T}_{\mathbf{Aexp}})$ and the configurations are defined by $\Gamma_{\mathbf{Aexp}} = \mathbf{Aexp} \cup \mathbb{Z}$. The end configurations are defined by $\mathrm{T}_{\mathbf{Aexp}} = \mathbb{Z}$.

The transition rule for multiplication in SPLAD can be seen on table 2.5. The rule states that if a_1 evaluates to v_1 and a_2 evaluates to v_2, using any of the rules from the arithmetic expressions, then $a_1 \cdot a_2$ evaluates to v where $v = v_1 \cdot v_2$.

$$\text{[MULT]} \quad \frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 \cdot a_2 \rightarrow_a v}$$

$$\text{where } v = v_1 \cdot v_2$$

Table 2.5: The transition rule for the arithmetic multiplication expression.

2.4.4.7 Boolean Expressions

The transition rules for boolean expressions are on the form: $env_V, sto \vdash b \rightarrow_b t$. The transition system is defined by: $(\Gamma_{\mathbf{Bexp}}, \rightarrow_b, \mathrm{T}_{\mathbf{Bexp}})$ and the configurations are defined by $\Gamma_{\mathbf{Bexp}} = \mathbf{Bexp} \cup \{tt, ff\}$. The end configurations are defined by $\mathrm{T}_{\mathbf{Bexp}} = \{tt, ff\}$.

The transition rule for logical-or in SPLAD can be seen on table 2.6. The rules have two parts: [OR-TRUE] and [OR-FALSE]. The [OR-TRUE] rule states that either b_1 or b_2 evaluates to *TRUE*, using any of the rules from the boolean expressions, then the expression b_1 OR b_2 evaluates to *TRUE*. [OR-FALSE] states that if both b_1 and b_2 evaluate to *FALSE* then the expression b_1 OR b_2 evaluates to *FALSE*.

$$\text{[OR-TRUE]} \quad \frac{env_V, sto \vdash b_i \rightarrow_b \text{TRUE}}{env_V, sto \vdash b_1 \text{ OR } b_2 \rightarrow_b \text{TRUE}}$$

$$\text{where } i \in 1, 2$$

$$\text{[OR-FALSE]} \quad \frac{env_V, sto \vdash b_1 \rightarrow_b \text{FALSE} \quad env_V, sto \vdash b_2 \rightarrow_b \text{FALSE}}{env_V, sto \vdash b_1 \text{ OR } b_2 \rightarrow_b \text{FALSE}}$$

Table 2.6: Transition rule for the boolean expression logical-or.

2.4.4.8 Variable Declarations

The transition rules for the variable declarations are on the form: $\langle D_V, env_V, sto \rangle \rightarrow_{DV} (env'_V, sto')$. The transition system is defined by: $\Gamma_{\mathbf{DecV}}, \rightarrow_{DV}, \mathrm{T}_{\mathbf{DecV}}$ and the configurations are defined by $\Gamma_{DV} = (\mathbf{DecV} \times \mathbf{EnvV} \times \mathbf{Sto}) \cup (\mathbf{EnvV} \times \mathbf{Sto})$ and $\mathrm{T}_{DV} = (\mathbf{EnvV} \times \mathbf{Sto})$. The end configurations are defined by $\mathrm{T}_{\mathbf{DecV}} = \mathbf{EnvV} \times \mathbf{Sto}$.

The transition rules for variable declarations can be seen on table 2.7. It is done by binding l to the next available location and binding x to this location. The function *new* is then used to point at the next available location. Then env_V is updated to include the new variable.

$$\text{[VAR-DEC]} \quad \frac{\langle D_V, env''_V, sto[l \mapsto v]\rangle \rightarrow_{DV} (env'_V, sto')}{\text{var } x <-- a; D_V, env_V, sto\rangle \rightarrow_{DV} (env'_V, sto')}$$

where $env_V, sto \vdash a \rightarrow_a v$
and $l = env_V$ next
and $env''_V = env_V[x \mapsto l][\text{next} \mapsto \text{new } l]$

Table 2.7: Transition rules for the variable declarations.

2.4.4.9 Procedure Declarations

The transition rules for the procedure declarations are on the form: $env_V \vdash \langle D_P, env_P\rangle \rightarrow_{DP} env'_P$. The transition system is defined by: $(\Gamma_{\mathbf{DecP}}, \rightarrow_{DP}, \mathrm{T}_{\mathbf{DecP}})$ and the configurations are defined by $\Gamma_{DP} = (\mathbf{DecP} \times \mathbf{EnvP}) \cup \mathbf{EnvP}$ and $\mathrm{T}_{DP} = \mathbf{EnvP}$. The end configurations are defined by $\mathrm{T}_{\mathbf{DecP}} = \mathbf{EnvP}$.

The transitions rules for procedure declarations with none to multiple parameters can be seen on table 2.8. The rule states that the new procedure is stored in the procedure environment along with the statement, formal parameters, and procedure- and variable-bindings from the time of declaration.

$$\text{[PROC-PARA-DEC]} \quad \frac{env_V \vdash \langle D_P, env_P[p \mapsto (S, \vec{x}, env_V, env_P)]\rangle \rightarrow_{DP} env'_P}{env_V \vdash \langle \text{func } p(\text{var } \vec{x}) \text{ is begin } S \text{ end}, env_P\rangle \rightarrow_{DP} env'_P}$$

Table 2.8: Transition rules for the procedure declarations.

2.4.4.10 Array Declarations

The transition rules for the array declarations are on the form: $\langle D_A, env_V, sto\rangle \rightarrow_{DA} (env'_V, sto')$. The transition system is defined by: $(\Gamma_{\mathbf{DecA}}, \rightarrow_{DA}, \mathrm{T}_{\mathbf{DecA}})$ and the configurations are defined by $\Gamma_{DA} = (\mathbf{DecA} \times \mathbf{EnvV} \times \mathbf{Sto}) \cup (\mathbf{EnvV} \times \mathbf{Sto})$ and $\mathrm{T}_{DA} = \mathbf{EnvV} \times \mathbf{Sto}$. The end configurations are defined by $\mathrm{T}_{\mathbf{DecA}} = \mathbf{EnvV} \times \mathbf{Sto}$.

The transitions rules for array declaration can be seen on table 2.9. The rule states that a number of locations equal to the length of the array plus one is allocated, and the pointer is set to point at the next available location. The length of the array is then stored in the first location of the array.

$$\text{[ARRAY-DEC]} \quad \frac{\langle D_A, env_V[r \mapsto l, \text{next} \mapsto l + v + 1], sto[l \mapsto v]\rangle \rightarrow_{DA} (env'_V, sto')}{\langle \text{array } r[a_1], env_V, sto\rangle \rightarrow_{DA} (env'_V, sto')}$$

where $env_V, sto \vdash a_1 \rightarrow_a v$
and $l = env_V$ next
and $v > 0$

Table 2.9: Transition rules for the array declarations.

2.4.5 Informal Type Rules

This section contains the informal type rules. In the type rules, E is an expression and S is a statement.

- Type rule for <, >, <=, >= :

  "E_1 (<, >, <=, >=) E_2" is type correct and of type boolean if E_1 and E_2 are type correct and of type integer or double.

- Type rule for !=, = :

  "E_1 (!=, =) E_2" is type correct and of type boolean if E_1 and E_2 are type correct and of type integer or double, or if E_1 and E_2 are of the same type of either char, bool or string.

- Type rule for + :

  "E_1 (+) E_2" is type correct and of type integer, string or double if E_1 and E_2 are type correct and of type integer, string or double.

- Type rule for -, * :

  "E_1 (-, *) E_2" is type correct and of type integer or double if E_1 and E_2 are type correct and of type integer or double.

- Type rule for / :

  "E_1 / E_2" is type correct and of type double if E_1 and E_2 are type correct and of type integer or double.

- Type rule for <-- (assign) :

  "E_1 <-- E_2" is type correct if E_1 and E_2 are type correct and E_1 and E_2 are of the same of type or if E_1 and E_2 is of type integer or double.

- Type rule of 'while'-statement:

  "while (E) begin S end" is type correct if E is of type boolean and type correct and S is type correct.

- Type rule of 'from to step'-statement:

  "from E_1 to E_2 step E_3 begin S end" are type correct if E_1, E_2 and E_3 are type correct and of type integer, and S is type correct.

- This is the type rules for 'if-else'-statement:

  "if(E) begin S_1 end else S_2" is type correct if E is type correct and of type boolean, and S_1 and S_2 is type correct.

- Here the type rules for switch/case is described:

  "switch (E) begin case E_1: S_1 break; ... case E_n: S_n break; default: S_d break; end" is type correct if E, $E_1 ... E_n$ are type correct and of type integer, bool, double, char or string and are of the same type, and $S_1 ... S_n$ and S_d are type correct.

2.4.6 Formal Type Rules

This section will follow up on the informal description of the type rules. Expressions can be seen on table 2.10, declarations can be seen in table 2.11 and statements can be seen in table 2.12. The types rules for arrays are not written because an array is just a sequence of variables and therefore works like the variable assignments.

$$[NUM_{EXP}] \quad E \vdash n : T$$

$$[SUB_{EXP}] \quad \frac{E \vdash e_1 : T \;\; E \vdash e_2 : T}{E \vdash e_1 - e_2 : T}$$

$$[ADD_{EXP}] \quad \frac{E \vdash e_1 : T \;\; E \vdash e_2 : T}{E \vdash e_1 + e_2 : T}$$

$$[VAR_{EXP}] \quad \frac{E(x) = T}{E \vdash x : T}$$

$$[PAR_{EXP}] \quad \frac{E \vdash e_1 : T}{E \vdash (e_1) : T}$$

$$[MULT_{EXP}] \quad \frac{E \vdash e_1 : T \;\; E \vdash e_2 : T}{E \vdash e_1 * e_2 : T}$$

$$[DIV_{EXP}] \quad \frac{E \vdash e_1 : T \;\; E \vdash e_2 : T}{E \vdash e_1 / e_2 : \text{Double}}$$

$$[EQUAL_{EXP}] \quad \frac{E \vdash e_1 : T \;\; E \vdash e_2 : T}{E \vdash e_1 = e_2 : \text{Bool}}$$

$$[GRT_{EXP}] \quad \frac{E \vdash e_1 : T \;\; E \vdash e_2 : T}{E \vdash e_1 > e_2 : \text{Bool}}$$

$$[AND_{EXP}] \quad \frac{E \vdash e_1 : \text{Bool} \;\; E \vdash e_2 : \text{Bool}}{E \vdash e_1 \; AND \; e_2 : \text{Bool}}$$

$$[NOT_{EXP}] \quad \frac{E \vdash e_1 : \text{Bool}}{E \vdash !e_1 : \text{Bool}}$$

$$[OR_{EXP}] \quad \frac{E \vdash e_1 : \text{Bool} \;\; E \vdash e_2 : \text{Bool}}{E \vdash e_1 \; OR \; e_2 : \text{Bool}}$$

Table 2.10: Type rules for expressions.

$$[VAR_{DEC}] \quad \frac{E[x \mapsto T] \vdash D_V : \text{ok} \;\; E \vdash a : T}{E \vdash \text{var } T \; x := a; D_V : \text{ok}}$$

$$[EMPTY_{DEC}] \quad E \vdash \varepsilon : \text{ok}$$

$$[PROC_{DEC}] \quad \frac{E[f \mapsto (\vec{x} : \vec{T} \to \text{ok})] \vdash D_F : \text{ok}}{E \vdash \text{function } f(T_1 \; x_1, \; T_2 \; x_2 \ldots T_k \; x_k) \; is \; S, D_F : \text{ok}}$$
where $0 \leq k$

Table 2.11: Type rules for declarations.

$[WHILE_{STM}]$	$\frac{E \vdash e : \text{Bool} \quad E \vdash S : \text{ok}}{E \vdash \text{while } e \text{ do } S : \text{ok}}$
$[FROM_{STM}]$	$\frac{E \vdash e_1 : \text{Int} \quad E \vdash e_2 : \text{Int} \quad E \vdash e_3 : \text{Int} \quad E \vdash S : \text{ok}}{E \vdash \text{from } e_1 \text{ to } e_2 \text{ step } e_3 \text{ do } S : \text{ok}}$
$[IF_{STM}]$	$\frac{E \vdash e : \text{Bool} \quad E \vdash S_1 : ok \quad E \vdash S_2 : ok}{if(e) \text{ then } S_1 \; else \; S_2 : \text{ok}}$
$[SWITCH_{STM}]$	$\frac{E \vdash x : T \quad E \vdash E_1 : T \ldots \quad E \vdash E_k : T \quad E \vdash S_1 : \text{ok} \ldots \quad E \vdash S_k : \text{ok} \quad E \vdash S : \text{ok}}{\text{switch } (x) \text{ begin case } E_1 : \; S_1 \text{ break; } \ldots \text{ case } E_k : \; S_k \text{ break; default: } S \text{ break; end} : \text{ok}}$
$[ASS_{STM}]$	$\frac{E \vdash x : T \quad E \vdash a : T}{E \vdash x <-- a : \text{ok}}$
$[COMP_{STM}]$	$\frac{E \vdash S_1 : \text{ok} \quad E \vdash S_2 : \text{ok}}{E \vdash S_1; S_2 : \text{ok}}$
$[BLOCK_{STM}]$	$\frac{E \vdash D_V \; : \; \text{ok} \quad E_1 \vdash D_F \; : \; \text{ok} \quad E_2 \vdash S \; : \; \text{ok}}{E \vdash \text{begin } D_V \; D_F \; S \text{ end} \; : \; \text{ok}}$ where $E_1 = E(D_V, E)$ and $E_2 = E(D_F, E_1)$
$[CALL_{STM}]$	$\frac{E \vdash f : (\vec{x} : \vec{T} \rightarrow \text{ok}) \quad E \vdash \vec{e} : \vec{T}}{E \vdash \text{call } f(\vec{e}) : ok}$ where $\lvert\vec{x}\rvert = \lvert\vec{e}\rvert$

Table 2.12: Type rules for statements.

2.5 Requirements to the Programmer

There are certain requirements to the programmer, when they are writing in SPLAD. The programmer must implement the functions:

- function pour return nothing using(container cont, int Amount)
- function RFIDFound return nothing using(int DrinkID, int Amount)

These two functions must be implemented, so that the SPLAD framework can call the functions. The pour function is called when a drink is to be mixed with the pourDrink function. The programmer should then handle how to pour the ingredients from their containers. The RFIDFound function is called when an RFID-tag is found, and the programmer should then, with the contents of the RFID-tag as parameters, handle what to do with the tag.

2.6 Functions Provided by SPLAD

There are certain functions provided by SPLAD. The functions are:

- function LCDPrint return nothing using(string StringToPrint, int Line)
- function LCDClear return nothing using()
- function RFIDWrite return bool using(int DrinkID, int Amount)
- function pourDrink return nothing using(drink DrinkToPour)

The LCDPrint function is provided to the programmer, so they can print to the LCD. The LCDClear will clear both lines on the LCD. The RFIDWrite function is provided, so the programmer can write a DrinkID and an Amount to the RFIDTag. When a drink should be poured, the programmer can call pourDrink, which will call the pour function provided by the programmer for each ingredient in the drink.

2.7 SPLAD Code Examples

This section will contain code examples written in SPLAD. It should be noted, that these examples are written as if one would write a SPLAD program. Each example will be described line by line, and in the end put together to a working SPLAD program. When writing a SPLAD program, it is important to keep in mind that there are functions that must be implemented. These functions are "pour" and "RFIDFound". First we will see how an implementation of the "pour" function could look like. This can be seen on listing 2.14. On line 1 of the "pour" example, the function is declared. The function takes two parameters as input, a container and an amount. Amount is actually implemented as time because it represents how long a nozzle will have to be open for a given amount to be poured. So if "pour" is called with container x and amount 100 it would in principle open the nozzle on container x for 100 seconds. It is the function on line 3 on listing 2.14 that opens the nozzle. Then on line 4, the Arduino platform is told to wait for a given number of milliseconds times 1000, which if "amount" was 100, corresponds to the Arduino waiting for 100 seconds. On line 5, the container nozzle is closed again, and on line 6 the function returns "nothing". The example is very close to the implementation used in the test program which can be seen in appendix B.

```
1 function pour return nothing using(container c, int Amount)
2 begin
3   call digitalWrite(c, HIGH);
4   call delay(Amount*1000);
5   call digitalWrite(c, LOW);
6   return nothing;
7 end
```

Listing 2.14: Implementation of the pour function.

Now to give an example of the "RFIDFound" function, which is also required in a SPLAD program. An example of the RFIDFound function can be seen on listing 2.15. On line 1, the function is declared - it takes a drink id and an amount as input. In this simple example, the amount is not used. The RFIDFound functions is called when the RFID-reader reads an RFID-tag. In this example we assume two drinks, "Tom Collins", which has drink id 0 and "Tequila Sunrise" which has drink id 1. On line 3, it is checked

if the provided drink id is 0. If it is, the pourDrink function is called with TomCollins as parameter. The pourDrink function would then pour the drink. If the drink id is not 0, the Tequila Sunrise drink is poured.

```
1 function RFIDFound return nothing using(int DrinkID, int Amount)
2 begin
3   if(DrinkID = 0)
4   begin
5     pourDrink(TomCollins);
6   end
7   else
8   begin
9     pourDrink(TequilaSunrise);
10  end
11  return nothing;
12 end
```

Listing 2.15: Implementation of the RFIDFound function.

Now we have the required functions, but the two drinks in the above example has not been declared. The drink declarations can be seen on listing 2.16. On line 1, the declaration of the drink TomCollins begins, next, from line 3 to line 6 the various ingredients are added. The same principles are valid for the declaration of the TequilaSunrise. It is clear that the declaration of a drink is similar to a recipe for a drink.

```
1 drink TomCollins is
2 begin
3   add 3 of Gin;
4   add 2 of Sugar;
5   add 2 of Lime;
6   add 5 of Soda;
7 end
8
9 drink TequilaSunrise is
10 begin
11   add 2 of Tequila;
12   add 5 of OrangeJuice;
13   add 1 of RedGrenadine;
14 end
```

Listing 2.16: Declaring the drinks.

Lastly we need to declare the ingredients for the drinks. The ingredients is implemented as containers which holds the specific ingredients. This is even simpler than declaring the drinks. The declaration of the ingredients can be seen on listing 2.17. It is pretty straightforward to declare the various containers. The only requirement is that the containers are assigned to a pin on the Arduino board. These pins range from A0-A5 and 0-13 on an Arduino Uno board [Arduino, c].

```
1 container Gin <-- 2;
2 container Sugar <-- 3;
3 container Lime <-- 4;
4 container Soda <-- 5;
5 container Tequila <-- 6;
6 container OrangeJuice <-- 9;
7 container RedGrenadine <-- 10;
```

Listing 2.17: Declaring the containers.

Now that every part of the example program is defined, it can be put together, the whole example can be seen on listing 2.18. The RFIDFound function is called when the RFID reader reads an RFID-tag. The tag contains the id 0 or 1, depending on the drink. If the tag contains 0, the drink Tom Collins is poured. For a more complex example of a SPLAD program, see appendix B.

```
container Gin <-- 2;
container Sugar <-- 3;
container Lime <-- 4;
container Soda <-- 5;
container Tequila <-- 6;
container OrangeJuice <-- 9;
container RedGrenadine <-- 10;

drink TomCollins is
begin
  add 3 of Gin;
  add 2 of Sugar;
  add 2 of Lime;
  add 5 of Soda;
end

drink TequilaSunrise is
begin
  add 2 of Tequila;
  add 5 of OrangeJuice;
  add 1 of RedGrenadine;
end

function pour return nothing using(container c, int Amount)
begin
  call digitalWrite(c, HIGH);
  call delay(Amount*1000);
  call digitalWrite(c, LOW);
  return nothing;
end

function RFIDFound return nothing using(int DrinkID, int Amount)
begin
  if(DrinkID = 0)
  begin
    pourDrink(TomCollins);
  end
  else
  begin
    pourDrink(TequilaSunrise);
  end
  return nothing;
end
```

Listing 2.18: The example program put together.

3 Implementation

This chapter describes the process of implementing a compiler. First we evaluate the criteria of the compiler, then we look at the architecture in this case the hardware which we use in this project. A general overview of the compiler phases is described followed by the syntactic, contextual analysis and code generation. Lastly we show the setup of how a drinks machine could look like albeit with LEDs substituting the containers.

3.1 Evaluation Criteria for the Compiler

To express the priorities of this project, we use a table, see table 3.1 where we rate each criteria by the importance compared to the other.

Criteria	Very Important	Important	Less Important	Irrelevant
Reliable	•			
Effectiveness			•	
Correctness	•			
Testable			•	
Maintainable			•	
Moveable		•		
Recycle				•
Safe				•

Table 3.1: This table will be used to prioritize the different criteria compared to the compiler in this project.

We want the compiler and its output to be as reliable as possible because the compiler should not generate errors on its own. The programmer should be able to write a correct program, compile it and be able to run it, hence we do not want the compiler to be a source of errors. Therefore we have prioritized the criteria "reliable" as very important.

The effectiveness of the compiler is not as important as the compilers functionality. We have deemed that effectiveness should not be in focus, based on the idea that the

compiler will run on modern computers and is aimed at hobby programming. Therefore the effectiveness is rated as less important compared to the other criteria.

The compiler should fulfill the given criterion to achieve the properties set for this projects compiler. Therefore we have set the "correctness" criterion as very important.

We have deemed that we should be able to test the compiler to check its correctness but we do not deem the "testable" criterion as important as other properties. Therefore it has been rated as less important.

Because the compiler is meant as a product, the focus has not been to simplify the compiler so it is easier to maintain. This would be attractive if said compiler should be used for further development later but that is not the case. Therefore we have rated "maintainable" as less important.

We want the compiler to be able to run on as many operating systems as possible, so the users are not restricted by their systems. The compiler is written in Java, which makes it able to run on different operating systems as long as the Java virtual machine is implemented for their architecture. Therefore we have rated the "moveable" criterion as important.

The "recycle" criterion has been rated as irrelevant because its not in the intention to make parts of the compiler usable out of the context. Meaning different parts of the compiler is context dependent.

We have also rated the "safe" criterion as irrelevant because this compiler does not handle security critical data.

3.2 Hardware

This section is about the hardware components used in this project, describing them and the reasons they are used in this project. The description states the basic technical specification that will be relevant for this project.

3.2.1 Hardware Platform

Arduino UNO is a powerful micro controller board that provides the user with ways to communicate with other components such as LCDs, LEDs, sensors and other electronic bricks (building blocks) which is a desirable feature in this project. Arduino uses the ATmega328 chip which provides more memory than its predecessors [Arduino, c].

There exists alternatives to Arduino products which could be considered for this project. Teensy is similar to Arduino UNO in many ways, its only difference is in the actual size of the product. Teensy is also cheaper than Arduino but does require soldering for simple set-ups where Arduino UNO comes with a board and pin-ports which means that it requires little pre work before using it [PJRC]. Seeeduino is a near replica of Arduino, an example could be Seeeduino Stalker which offers features as SD-card slot, flat-coin battery holder and X-bee module-headers. X-bee is a module for radio communication between one or more of these modules. Seeeduino is compatible with the same components as Arduino that makes it suited for acting as a replacement [Studio]. Netduino is a faster version of Arduino but it comes at a higher cost. Netduino also require the .NET framework so it will only work together with Windows operating systems. Netduino uses a micro-USB instead of the USB-B Arduino uses [Walker, 2012].

Arduino UNO is more accessible because Aalborg University already has some in stock that could be used where the other alternatives have to be bought first. Arduino UNO, Teensy and Seeeduino are all compatible with the other equipments that will be used in this project. Netduino is limited to the .NET framework where Arduino UNO and the other alternatives are more flexible and therefore more ideal because they work with a broader range of platforms. So it comes down to that Arduino UNO have all the necessary features and is the most convenient one to acquire. The project group has deemed this platform suited for this project based on these reflections.

The Arduino UNO board has 14 digital input/output pins where six of them can emulate an analog output through PWM (Pulse-Width Modulation) which are available on the Arduino UNO board. The Arduino UNO board also provides the user with six analog inputs which enables the reading of an alternating current and provides the user with the currents voltage. These pins can be used to control or perform readings on other components and in that way provides interaction with the environment around the board. The Arduino board is also mounted with a USB-B-port and power jack socket. The board can be hooked up with a USB cable or an AC-to-DC adapter through the power jack socket to power the unit. Arduino UNO operates at 5v but the recommend range is 7-12v because lower current than 7v may cause instability if the unit needs to provide a lot of power to the attached electronic bricks. The USB is also used to program the unit with the desired program through a computer [Arduino, c].

Programs for Arduino are commonly made in Arduino's own language that are based on C and C++. The producers of the Arduino platform provides a development environment (Arduino IDE) that makes it possible to write and then simply upload the code to the connected Arduino platform. This process also provides a library with functions to communicate with the platform and compatible components [Arduino, b]. Arduino is suited for this project because it makes it possible to demonstrate the language and illustrate that the translation works.

3.2.2 RFID

To administrate the users collection of purchased drinks the plan is to store the number and the identifier of drinks on an RFID tag that the customer then use at the drinks machine to get their drinks served.

RFID (Radio Frequency IDentification) is used to identify individual objects using radio waves. The communication between the reader and the RFID-tag can go both ways, and it is possible to both read and write to most tag types. The objects that are able to be read differs a lot. It can be clothes, documents, packaging and a lot of other kinds of objects. All tags contain a unique ID which cannot directly be changed. This ID is used to identify an individual tag. Tags can be either passive or active. Passive tags do not do anything until a signal from a reader transfers energy to the tag. Once activated it sends a signal back in return. Active tags have a power source and therefore are able to send a signal on their own, making the read-distance greater. The tags can also be either *read only tag* or *read/write tag*. A *read only tag* only sends its ID back when it connects with a reader, while a *read/write tag* has a memory for storing additional information it then sends with the ID [Specialisten].

An alternative to RFID is NFC (Near Field Communication). NFC uses radio communication like the RFID, but unlike RFID the communication between two NFC devices is two-way. An NFC can also read passive NFC tags and could replace the RFID. We have

chosen not to use the NFC devices since we have no need for the two-way communication [Nosowitz, 2011].

The RFID-tags used in this project are passive, high frequency, *read/write* tags. The passive high frequency tags have a maximum read-distance of one meter [Specialisten], and that is far enough for this project. We are using *read/write* tags to store drink-ids and drink-counts on the tags and make the reader read and respond to the information.

An RFID-reader/writer can read and write RFID-tags.

3.2.3 Other Components

The demonstration of the product will require something to illustrate more advanced parts of a theoretical machine. The reason for making a whole drinks machine just to show that the project product works, will take too much time that instead could have been used to make the product better. We use LEDs (light emitting diode) to illustrate the different functions of the machine, when they are active or inactive. An LED is made of a semiconductor which produces light when a current runs through the unit.

LEDs are normally easy to use by simply running a current the correct way through the LED. The reason why LEDs are being using instead of making the machine is that there is neither time nor is it the main focus of this project.

It would also be good to be able to print a form of text to the customer. To do this there will be used an LCD 16-pin (Liquid Crystal Display). Arduino's Liquid Crystal library provides the functions to write to LCD so no low level code is needed to communicate with the LCD [Arduino, a].

Switches/buttons is used as input devices. This will allow interaction with the program at runtime. The switches will illustrate a more advanced control unit but in the project buttons will be sufficient.

3.3 Overview of the Compiler

Figure 3.1 shows an abstract overview of each of the different phases in a compiler, what each phase requires as input, and what each step returns to the next phase.

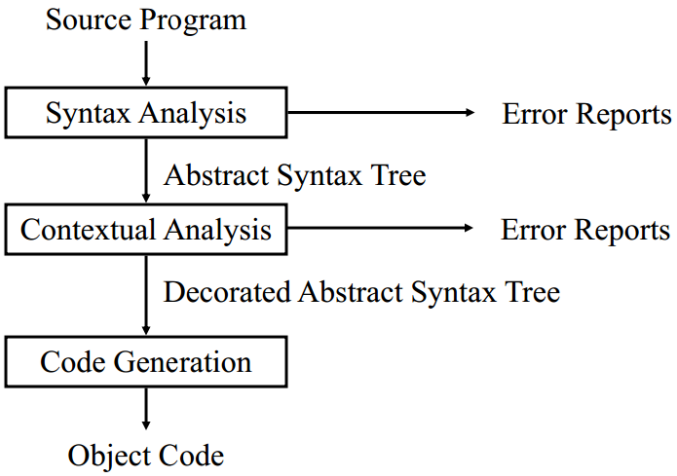


Figure 3.1: This is an abstract overview of how the compiler is structured [Bent Thomsen, 2013].

A compiler is a fundamental part of modern computing. Its purpose is to translate a programming language into a lower level programming language. A compiler can also compile source code into virtual instructions, which can be executed by virtual machines, as is the case for the Java-compiler. This makes the compiled program portable across different computers [Fischer et al., 2009].

A compiler consists mainly of three different phases. The different phases roughly correspond to the different parts in a language specification which can be seen on figure 3.1. The syntax analysis correspond to the syntax and the contextual analysis to the type and scope rules.

The three main phases are even in simple compilers implemented through more phases. This can be seen in figure 3.2. In the syntax analysis phase the compiler consists of a scanner and a parser. The scanner takes the source program and transforms it into a stream of tokens. The parser then use the tokens to create an abstract syntax tree (AST). In the contextual analysis a symbol table is created from the abstract syntax tree. At the end, the semantic analysis decorates the AST, and translates this into the targeted language.

The syntactic analysis is described in detail in section 3.5, the contextual analysis is described in section 3.6, and the code generation is described in section 3.7.

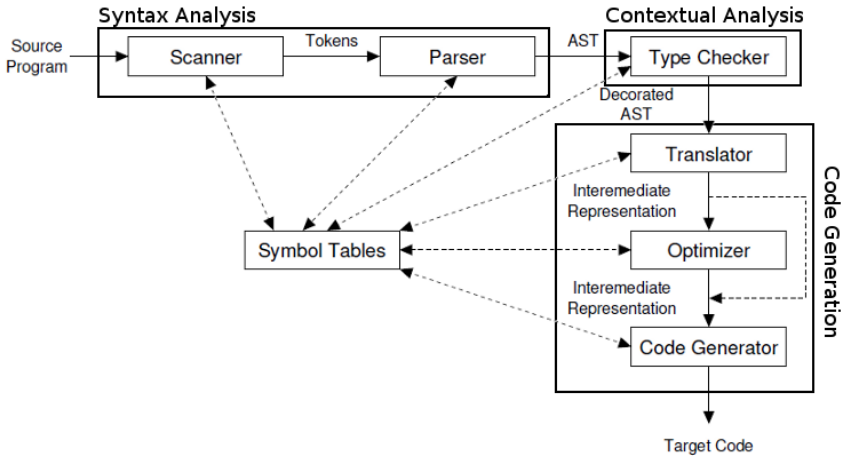


Figure 3.2: This is an more detailed overview of how the compiler is structured [Fischer et al., 2009].

3.4 Language Processor

This section introduces several different techniques for processing a language, and compare them briefly. This results in a section describing the choice of language processor for this project, and the reasoning for this choice.

3.4.1 Compiler

A compiler can be viewed as a translator for a programming language. The compiler translates one programming language (source code) into another (target code), typically from a high-level programming language like C to a low-level language like machine code. Generally compilers generate one of three types of target code: Pure Machine Code, Augmented Machine Code and Virtual Machine Code [Fischer et al., 2009]. An overview of the compiler is described in section 3.3.

3.4.1.1 Pure Machine Code

Pure Machine Code is code for a particular machines instruction set - this is excluding any operating system- or library functions. Pure machine code is rare because of the exclusion of such as the use of libraries and is mostly used in compilers for implementation languages, which are used for implementing e.g. operating systems [Fischer et al., 2009].

3.4.1.2 Augmented Machine Code

The most common approach when compiling, is having a compiler generate augmented machine code. This form of machine code includes operating system- and library routines, which allow the compiled program to be executed on any machine granted that the particular machine is equipped with the necessary operating system [Fischer et al., 2009].

3.4.1.3 Virtual Machine Code

Virtual machine code is a type of code which consists of *virtual* instructions. This approach is for example used by the Java compiler, which compiles Java-source programs to Java byte-code, which are virtual instructions. The byte-code can then be executed in the

Java Virtual Machine (JVM), which means that if an implementation of the JVM exists for a particular machine, the Java byte-code can be executed. This is a clever approach, because if the compiler compiles language L into virtual instructions, and the compiler is written in language L, the compiler can compile itself into virtual instructions. If the virtual machine is simple, it is relatively simple to port the virtual machine to a another architecture, allowing the compiler to work on many different architectures. The process of porting a compiler to another architecture this way is called bootstrapping [Fischer et al., 2009].

3.4.2 Interpreters

Interpreters differs from compilers because where compilers generate some target code, interpreters execute programs directly. This approach is desirable when a high degree of platform independency is needed, because no machine code is generated - instructions are simply performed inside the interpreter. This is achieved because porting an interpreter to another architecture merely requires a programmer to recompile the interpreter on a different machine. Another advantage is that programs can be modified on run time, which allows interactive debugging, where it is possible to see values of variables at any given time during the execution of a program.

3.4.3 Choice of Language Processor for this Project

Because the language of this project is targeted against the Arduino platform, it is desirable to either compile SPLAD into Arduino machine code, or compile SPLAD to the C/C++ like programming language that Arduino uses. Because it was deemed too time consuming to directly translate to Arduino machine code compared to other parts of the project, it was decided that the SPLAD-compiler should convert the provided source program written in SPLAD, to the Arduino programming language. This approach was also used in the first versions of the C++ compiler "CFront" [Sebesta, 2009], which compiled C++ into C, and then used a C compiler to compile the program to machine code instructions. Therefore the SPLAD compiler outputs a ".ino" file, which is the file format of Arduino code file, which can be compiled by the Arduino compiler, which also allows uploading to the Arduino platform.

3.4.4 Language Processing Strategy

This section will describe the language processing strategy in this project. To introduce what is called tombstone diagrams, a small example of a tombstone diagram for the Java-language is presented in figure 3.3. This figure shows that the Java-compiler takes some java-code and turns it into byte code on machine M. The Java-virtual machine (JVM) can then execute the byte code. This makes the bytecode platform-independent, because if there is a JVM-implementation for an architecture, Java byte code can be compiled on another architecture and be executed on the first architecture. The tombstone to the left should be read as: Compiler written in machine-code M, takes a Java-source program and turns it into byte-code while running on architecture M.

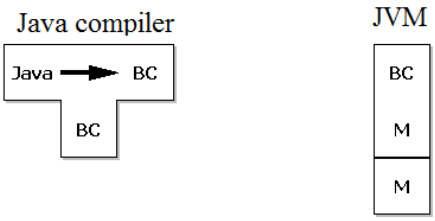


Figure 3.3: Tombstone diagram for the Java-compiler

The left part of figure 3.3, represent the JVM, which can actually execute the byte-code (BC) on architecture M.

Figure 3.4 illustrates the compilation of the SPLAD-compiler. The SPLAD-compiler is written in Java and converts the SPLAD source-program to Arduino-code. As the SPLAD-compiler is written in Java, it is therefore compiled into Java byte-code using the Java-compiler. It can be seen in figure 3.4, that everything is bootstrapped together. The SPLAD-compiler is bootstrapped to the Java-compiler which returns a SPLAD-to-Arduino compiler written in Java byte-code. The Java-compiler is also written in byte-code, and runs on the JVM, which is written in machine code for architecture M, and runs on architecture M.

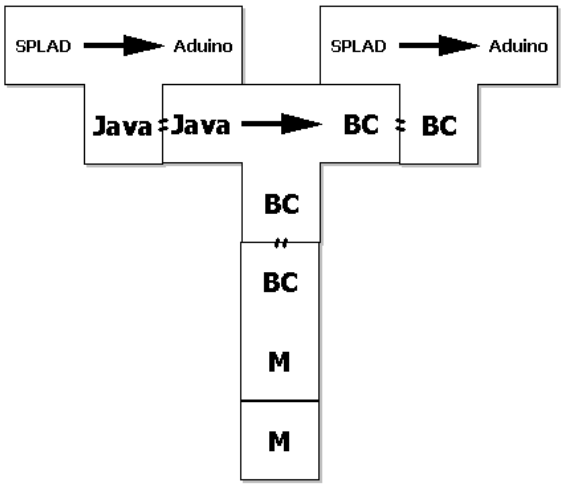


Figure 3.4: Tombstone diagram for the SPLAD-compilers compilation process

Now we have the SPLAD to Arduino compiler. Figure 3.5 illustrates how the compiler works. The SPLAD-compiler takes a SPLAD-source program, and compiles it to Arduino-code, while running on the JVM. Therefore the user must use the Arduino compiler to compile the generated Arduino program, to a Arduino machine code, and upload it to the Arduino platform.

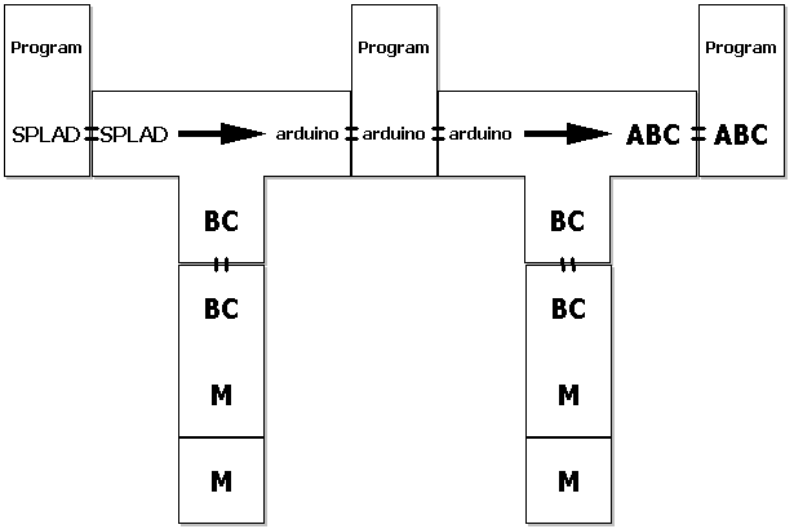


Figure 3.5: Tombstone diagram for the SPLAD-compiler compilation process (ABC is an acronym for Arduino Byte Code)

3.4.5 Compilation Passes

The compiler in this project consists of several passes. First, the lexer scans the program, and returns the result to the parser, which generates the parse tree. The lexer and parser is both generated from the BNF of the SPLAD-language (see section 2.3.5 for the BNF). The parse tree serves as input to the type checker, which checks that the source-program complies according to the type-rules. Next, the code-generator generates the code for the Arduino platform. It is also in the code-generating phase, the scope-rules are being checked. Due to lack of time, there is no optimizing phase.

3.4.6 Abstract Syntax Trees

The parser generates an abstract syntax tree (AST) [Fischer et al., 2009], which is an abstract data type describing the structure of the source program. This means that the AST contains information about which constructs the source program contains. More specifically, each node in the AST represent a construct in the source language, for example an 'if'-block.

When the AST has been generated, it is decorated with types by the type checker. The type checker traverses the AST, and checks the static semantics of each node, which means that it verifies that the node represent valid constructs. If each node is correct it is returned to the translator [Fischer et al., 2009]. The translator then uses the AST to an intermediate representation (IR code), which is used in the later phases of the compiler.

3.4.7 Parse Tree

A parse tree is nearly the same as an abstract syntax tree, but here all internal nodes are labeled with a non-terminal symbol and all leaves are labeled with a terminal symbol. A sub-tree describes one instance of an abstraction of a sentence.

To help understand the difference between an abstract syntax tree as seen on figure 3.6 and a parse tree as seen on figure 3.7, two trees are made, one of each, from the same code for a variable declaration seen on listing 3.1 in the projects language.

```
1     int x <-- 3+2;
```

Listing 3.1: A simple variable declaration in the project language.

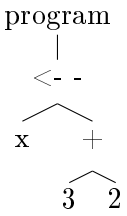


Figure 3.6: An abstract syntax tree.

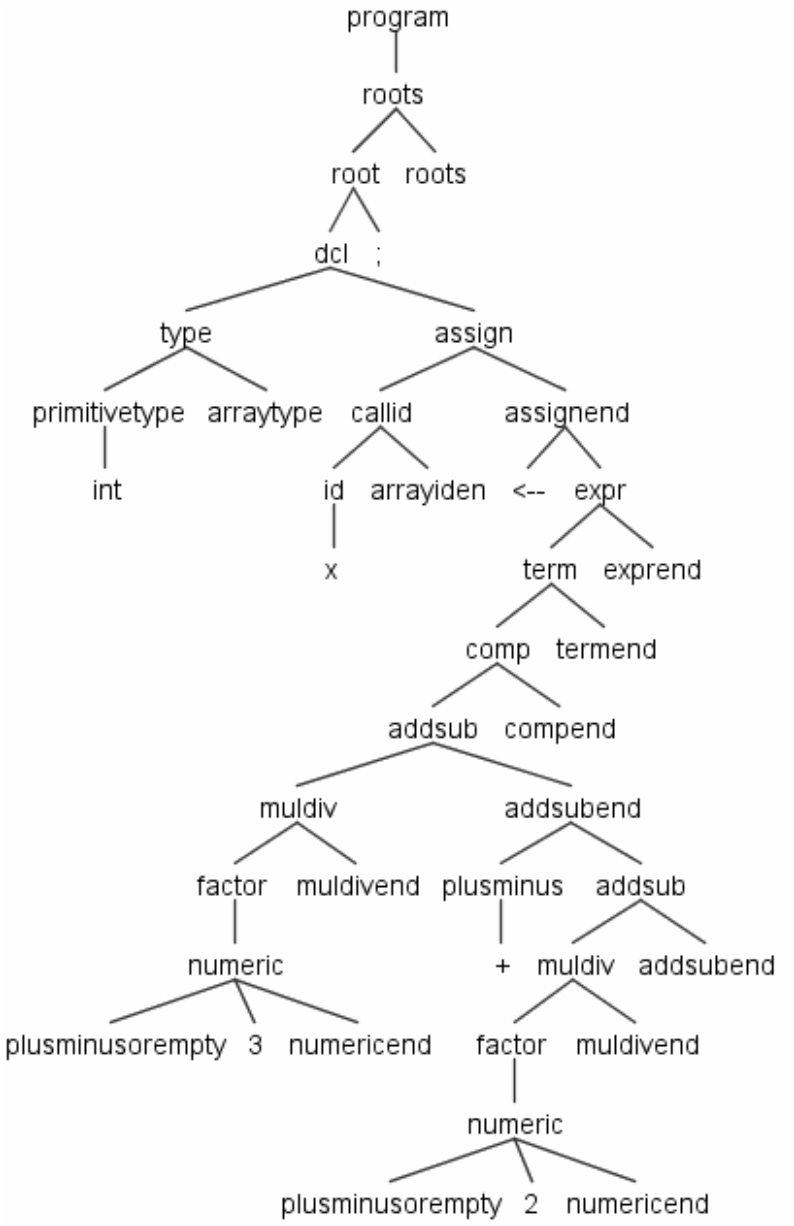


Figure 3.7: A parse tree made from the code on listing 3.1 from the BNF of SPLAD.

3.4.8 Visitor Pattern

The visitor pattern is ideal for traversing an abstract syntax tree or a parse tree during the semantic analysis and code generation, to help manage the large number of phase- and node interactions. A phase is started by visiting the first node in the AST, to reach every node there must be a call to *visit* in the preceding node.

Single dispatch is used by most object-oriented languages to determine which *visit* method that must be used. The method is based on the type of the parameter. There are a few problems with single dispatch, such as the fact that it finds a match based on the declared type of its parameters when it is called. This is where the visitor pattern comes in handy with a form of double dispatch, the idea is to make use of the abstract class like the one seen on listing 3.2 that all nodes implement.

```
1 class Visitor
2   procedure visit(AbstractNode n)
3     n.ACCEPT(this)
4   end
5 end
```

Listing 3.2: A generic Visitor.

If the visit method is called without the accept method, it will try to visit the abstract node. For example if a phase contained a method visit(*IfNode n*) like on listing 3.3, it will not invoke the actual *IfNode*, this is because the supplied parameter is based on the declared type (*AbstractNode*).

```
1 class TypeChecking extends Visitor
2   procedure VISIT(IfNode i)
3   end
4 end
```

Listing 3.3: A concrete Visitor.

Therefore the specific node accepts a visitor as seen on listing 3.4 and thus determines the type of the node which allows it to be visited because it now knows the actual *IfNode* and the code it contains can be executed.

```
1 class IfNode extends AbstractNode
2   procedure ACCEPT(Visitor v)
3     v.VISIT(this)
4   end
5   ...
6 end
```

Listing 3.4: A concrete Node.

3.5 Syntactic Analysis

Syntactic analysis is the first phase of the compilation process from SPLAD to the C/C++ like language Arduino uses. It consists of a lexer and a parser. These components are described in section 3.5.1. The syntactic analyzer reads through the SPLAD program and checks if the program complies with the syntax specification of SPLAD and if it does, it creates a parse tree which is used by the contextual analysis.

3.5.1 Known Lexers and Parsers

In this section some of the different lexer and parser generators, that are available through the internet, is described.

3.5.1.1 Lexer Generators

These programs generate a lexical analyzer also known as a scanner, that turns code into tokens which a parser uses.

Lex: Files are divided into three sections separated by lines containing two percent signs. The first is the "definition section" this is where macros can be defined and where headerfiles are imported. The second is the "Rules section" where regular expressions are read in terms of C statements. The third is the "C code section" which contains C statements and functions that are copied verbatim to the generated source file. Lex is not open source, but there are versions of Lex that are open source such as Flex, Jflex and Jlex [Lex].

Flex: Alternativ to lex [Flex].

An optional feature to flex is the REJECT macro, which enables non-linear performance that allows it to match extremely long tokens. The use of REJECT is discouraged by Flex manual and thus not enabled by default.

The scanner flex generates does not by default allow reentrancy, which means that the program can not safely be interrupted and then resumed later on.

Jflex: Jflex is based on Flex that focuses on speed and full Unicode support. It can be used as a standalone tool or together with the LALR parser generators Cup and BYacc/J [Jflex].

Jlex: Based on lex but used for java [Jlex].

3.5.1.2 Parser

Parsertools generate a parser, based on a formal grammar from a lexer, checks for correct syntax and builds a data structure (Often in the form of a parse tree, abstract syntax tree or other hierarchical structure).

Yacc: Generates a LALR parser that checks the syntax based on an analytic grammar, written in a similar fashion to BNF. Requires an external lexical analyser, such as those generated by Lex or Flex. The output language is C [Yacc].

Cup: More or less like Yacc, output language is in java instead [Cup].

3.5.1.3 Lexer and parser

Combines the lexer and parser in one tool.

SableCC: Using the CFG(Context Free Grammar) written in Extented Backus-Naur Form SableCC generates a LALR(1) parser, the output languages are: C, C++, C#, Java, OCaml, Python [SableCC].

ANTLR: *ANother Tool for Language Recognition* uses the CFG(Context Free Grammar) written in Extented Backus-Naur Form to generate an LL(*) parser. It has a wide variety of output languages, including, C, C++ and Java. ANTLR can also make a tree parsers

and combined lexer-parsers. It can automatically generate abstract syntax trees with a parser[Antlr]. Lexer rules is written with an upper-case beginnings letter so that ANTLR can distinguish between lexer rules and parser rules [Parr, 2012].

JavaCC: Javacc generate a parser from a formal grammar written in EBNF notation. The output is Java source code. JavaCC generates top-down parsers, which limits it to the LL(k) class of grammars (in particular, left recursion cannot be used). JavaCC also generates lexical analyzers in a fashion similar to lex[Norvell]. The tree builder that accompanies it, JJTree, constructs its trees from the bottom uplex[JJTree].

3.5.1.4 Comparison Tables

On table 3.2 a comparison between the different lexers can be seen. It has been used in the discussion about how to make the lexer in this project.

Name	Lexer algorithm	Output language
Lex	DFA	C
Flex	DFA table driven	C, C++
Jflex	DFA	Java
Jlex	DFA	Java

Table 3.2: Comparison between the different lexical analyzers.

Based on the different lexers and parsers attributes (seen on 3.3), compared to the expectations of this project, it has been decided that ANTLR best fit the project. The reason behind this is that ANTLR uses the LL(*) parser algorithm, this fits the structure of the CFG grammar for this project. Furthermore ANTLR's output language can be in Java, C or C++, this makes it easier to work on an Arduino. Another possibility could be to write the lexer and parser by hand, but many typingerrors are avoided by using a tool like ANTLR. Futhermore, it is easier to maintain the lexer and parser with a tool. When the grammar is changed, you can just generate a new lexer and parser with the tool. It has therefore been decided to use ANTLR for generating the lexer and parser in this project.

Name	Parsing algorithm	Input notation	Output language	Lexer
Yacc	LALR(1)	YACC	C	External
Cup	LALR(1)	EBNF	java	External
SableCC	LALR(1)	EBNF	C, C++, C#, java, OCaml,Python	Generated
ANTLR	LL(*)	EBNF	ActionScript, Ada95, C, C++, C#, Java, JavaScript,Objective-C, Perl, Python, Ruby	Generated
JavaCC	LL(k)	EBNF	Java, C++(beta)	Generated

Table 3.3: Comparison between the different parsers and lexer-parsers.

3.5.2 ANTLR

ANTLR (ANother Tool for Language Recognition) is a tool for generating a parser or lexer-parser from a given grammar. The ANTLR starts by generating the lexer based on the lexer rules that are define in the grammar [Parr, 2012]. In the SPLAD language it has been decided that a lexer rule should all be written in upper-case. This is done in order to better distinguish between lexer parser rules when writing or reading the grammar. Lexer starts from the top of the rules and work its way down through the rules, meaning that it will try to generate tokens from the very first rule and work its way down until it meets a possible match between a given input and a rule. Because of this the most complex rules should be placed first in the grammar in order for the lexer to generate the correct tokens. The token stream from the lexer are then parsed following the parser rules that have been defined in the grammar. Parser rules are all written with lower-case in contrast to the lexer rules. ANTLR works with $LL(*)$ grammars which means that the parser uses left-most derivation to parse the token stream. ANTLR can generate a abstract syntax tree for the grammar by incorporating specific operators in the grammar that tells if an element should be a root node of a subtree with its children or if an element should be left out of the tree construction. ANTLR also allows the use of rewritten rules to generate a tree from the given grammar [Parr].

```
1 function setup return nothing using()
2 begin
3   /*Do something*/
4   return nothing;
5 end
6
7 function LCDPrint return nothing using(string text)
8 begin
9   /*Function to write a string to the LCD connected to the arduino
        */
10
11   return nothing;
12 end
13
14 function makedrink return nothing using()
15 begin
16   string message <-- "Hello World!";
17   call LCDPrint(message);
18   return nothing;
19 end
```

Listing 3.5: Here the code for the simple "Hello world" program can be seen.

The ANTLR library comes with a tool for testing the generated lexer and parser. The tool, (test rig), allows for parsing some code and get it represented in a GUI or tree representation. This have been done for a simple "Hello world" program, see listing 3.6, to show how a program is parsed and represented using the GUI option.

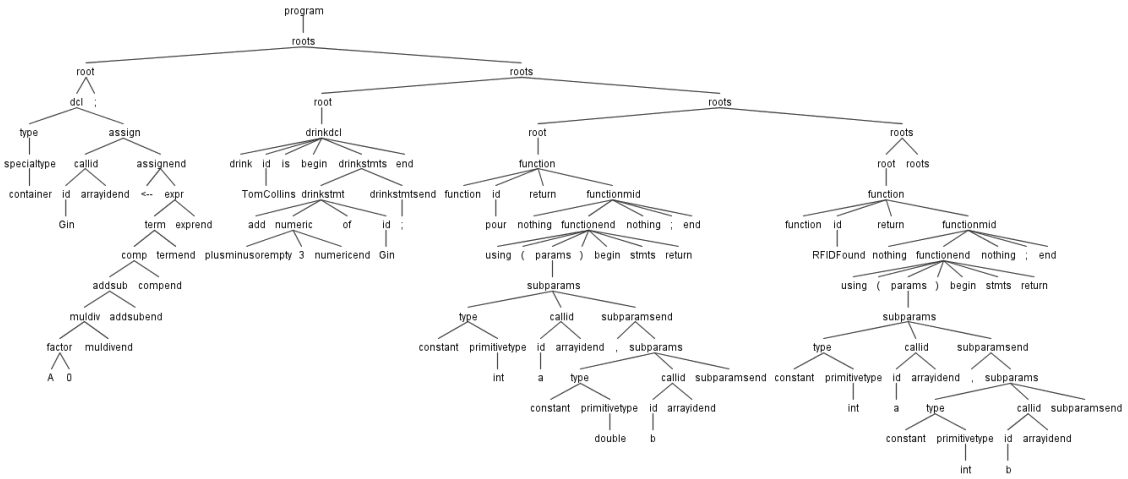


Figure 3.8: On this figure the parse tree for the parsed program "Hello world" can be seen.

On figure 3.8 a parse tree for the "Hello world" program can be seen. Derivations are illustrated by being children of the parent node, see section 3.4.7 for more about parse trees.

3.5.3 Lexical Analyzer

A lexical analyzer or "lexer" reads the input file, and returns a series of tokens based on the input [Fischer et al., 2009]. More specifically it is the scanner in the lexical analyzer which does this. These tokens are matched by rules, usually described by regular expressions. An example of such grammar rules can be seen on table 3.4. Formally a token consists of two parts: The token type, and the token value [Fischer et al., 2009]. As an example the IDENT token seen on 3.5 has the token type IDENT and the value 'c'.

Terminal	Regular expression
dcl	"$[a-z]$"
assign	"$=$"
digit	"$[0-9]^{+}$"
endassign	";"
blank	""+

Table 3.4: Sample token specification.

This specification of tokens, would be used by the scanner to determine how tokens looks, and thereby which text-elements are tokens.

```
1 c = 42;
```

Listing 3.6: Simple example of code.

As an example the lines of code seen on listing 3.7 might be read as the tokens seen on table 3.5.

Token	Lexeme
IDENT	c
ASSIGN	=
DIGIT	42
SEMICOLON	;

Table 3.5: Example of tokens.

The scanner produces a stream of tokens, which is returned to the parser. The parser checks if the tokens conforms to the language-specification [Fischer et al., 2009].

3.5.3.1 Tokens

For a compiler to able to distinguish between variables names and types the compiler will need some rules to describe the difference between them. This is done by reserving the words, called keywords, which are used to describe types, the beginnings and endings of blocks, and declaration of statements. A variable may not be named the same as any of the keywords since the compiler can not distinguish if it is a variable name or a reserved keyword.

Reserved Keywords

The reserved keywords for SPLAD can be seen on table 3.6.

AND	end	OR
begin	false	return
bool	from	step
break	function	string
case	HIGH	switch
char	if	to
container	int	true
default	LOW	using
double	nothing	while
else		

Table 3.6: The reserved keywords in SPLAD.

This list is used to keep track of which words are going to be reserved and in that way provide an overview for the programmer.

Token Specification

A parser needs a stream of tokens to parse a program correctly. These tokens are generated by a lexer which reads a stream of input symbols and from a given set of rules, makes the corresponding tokens. A token specification is used to describe the rules the lexer need in the construction of tokens. Token specification are expressed in way related to regular expressions [Sebesta, 2009]. Regular expressions are strong in describing patterns which

is the core of token production [Sipser, 2013]. The tokens used for this project can be seen on table 3.7.

STRINGTOKEN	" .*? "
LETTER	[a - zA - Z] $^{+}$
DIGIT	[0 - 9] $^{+}$
NOTZERODIGIT	[1-9][0-9] *
COMMENT	/* .*? */
WHITESPACE	\r | \n | \t
OTHER	ε

Table 3.7: The tokens in SPLAD.

Further work would be making a lexer to generate a token for the parser. Another options was to find a suited tool for generating a lexer for the given rules. This is a valid option because making a lexer can be automated and therefore already exists a lot of good lexer generators that can be used, see section 3.5.1.

3.5.4 Parser

A parser takes the tokens from the scanner and use them to create an abstract syntax tree. It also checks if the stream of tokens conforms to the syntax specification, usually written formal using context-free grammar (CFG).

The main purpose of the parser is to analyze the tokens and check if the source program is written in the correct syntax. If this is not the case the parser should show a message describing the error. The parser will at the end create an abstract syntax tree.

Generally there are two different approaches to parsing: top-down and bottom-up. Before describing the different approaches to parsing, it is worth to describe derivation shortly. Derivations is how the parser will create the abstract syntax tree. Either it will be built leftmost or it will be built rightmost. Leftmost-derivation is where the parser will take the terminal that is most to the left, and create a derivation for that. A rightmost-derivation is the opposite: The parser chooses the first terminal from the right, and creates a derivation for that.

3.5.4.1 Top-down Parsers

Then top-down parser starts at the root and works its way to the leaves in a depth-first manner, doing a pre-order traversal of the parse tree. This is done by reading tokens from left to right using a leftmost derivation. Furthermore top-down parsers can be split into table-driven LL and recursive descent parse algorithms.

***** Table-driven LL Parsers Uses a parse table to determine what to do next. The entries in the parse table is determined by the particular LL(k) grammar. The parser then searches the table to see what to do.

***** Recursive-descent Parser The recursive-descent parsers consists of mutually recursive parsing routines. Each of the non-terminals in the grammar has a parsing procedure that determines if the token stream contains a sequence of tokens derivable from that non-terminal.

3.5.4.2 Bottom-Up Parsers

A bottom-up parser has to do a post-order traversal of the parse tree, meaning that it starts from the leaves and works towards the root. A bottom-up parser is more powerful and efficient than a top-down parser, but not as simple.

***** LR A LR parser reads from left to right and because it is a bottom-up parser it uses a reversed rightmost derivation which means it takes terminals and turn them into non-terminals. It is as the LL parser driven from a parse table. The biggest difference is how it is derived and how the parse table is handled.

***** LALR A LALR(Lookahead Ahead LR) parser is one of the most commonly used algorithms today, because it is a powerful algorithm but do not need a very large parse table. It works like the LR parser.

3.6 Contextual Analysis

Because we are using ANTLR, we get an parse tree, with a basic visitor, to work with. By expanding the basic visitor, three modified visitors have been made. The first visitor is for checking if the scope rules are in order, the errors that are found is put into an error list which will be shown to the programmer to indicate what is wrong with the code. The second visitor is for type checking, it will visit the abstract syntax tree to see if all the types are used together correctly, again errors will be put into a list and shown to the programmer. The last visitor will generate code from our language into a language that can be used by Arduino.

3.6.1 Scope Checking

In our project we have decided to use static scoping. Because the program language made in the project group is a imperative, it makes more sense to use static scoping. The language for Arduino uses static scoping, see section 2.4.1. Static scoping are used by well known program languages like C, C# and Java [Softwere, 2013]. When a variable is used in SPLAD, the variable will need to be declared in the scope or in an outer scope before it can be used.

When scope checking is started, a list called "scopecontrol" is made which can hold other lists. A list called "listOfErrors" is also created which holds the errors that are found. Each list in scopecontrol is a scope. The scope lists are used to store variable names in the given scopes. An example of this can be seen in figure 3.9. To show the errors that are related to the scope checking, the errors will be saved in the list listOfErrors. Scope checking is split into nine different places in the parse tree node visitors: visitProgram, visitBlock, visitCases, visitEndcase, visitFunction, visitDcl, visitSubparams and visitCallid.

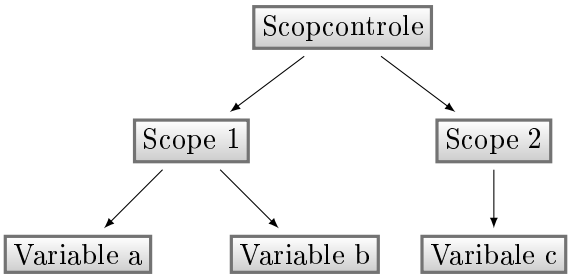


Figure 3.9: A visual diagram of the structure of scopecontrol.

For making sure that global variables are saved in scopecontrol, a global scope is added to scopecontrol in the visitProgram function. The global scope is removed when we are done with visiting the program.

There are three ways of creating a scope in the SPLAD language, not counting the one that makes the global scope. The first way is through visitBlock function, which is visited through if, while and from statements. In visitBlock a new list of strings(a scope) is made and then added to scopecontrol. Thereafter all statements in the block is visited. When all the statements are visited, the scope is removed from scopecontrol. An example of this can be seen in listing 3.8.

```
1 List<String> Templist = new ArrayList<String>();
2 Scopecontrol.add(Templist);
3
4 String Temp = "{" + visit(ctx.stmts()) + "}";
5
6 Scopecontrol.remove(Scopecontrol.size()-1);
7 return Temp;
```

Listing 3.7: Visitor for blocks with scope checking implemented.

The second way to create a scope is by making a function, and thereby visit the visitor visitFunction. A functions name is irrelevant for the scope checker, because a function can only be declared in the global scope. The visitBlock cannot be used to make scopes for functions, since they can have a number parameters from where they were called. The scope checker must therefore make a scope and add it to the scopecontrol before the parameter are declared. After the parameters are declared, in the newly made scope, all statements in the function will be visited and it will then remove the scope from the scopecontrol before returning.

The third way to create a scope is by a making a switch. It will then visit visitCases and visitEndcase. When visiting visitEndcase it can either visit visitCases or construct a default case and return it. The scope checker will have to create a scope for every case in the switch and remove the scope after the case is finished, this includes the default case.

VisitDcl and visitSubparams is where variables names are added to the innermost scope, meaning the scope with the last index in scopecontrol. This can seen in listing 3.9 for VisitDcl.

```
1 String Temp = ctx.assign().callid().id().getText();
2 Temp = Temp.replaceAll("\\s","");
3 int ScopeRange = Scopecontrol.size()-1;
4 Scopecontrol.get(ScopeRange).add(Temp);
```

Listing 3.8: The visitor, visitDCL, with the scope checking implemented

The main part of the scope checker is located in visitCallid. Here the scope checker needs to look through all known scopes to see if a variable name exist. This is done by using two for-loops, which can be seen in listing 3.10. The first one goes through scopecontrol and the second one is for each element in the list that are in scopecontrol. In this for-loop there is a if-statement that inspects if the called variable name is in one of the scopes. If the variable name exists in one of the scopes things are fine, but if it does not exists in one of the scopes, the scope checker will add it as an error in the list, listOfErrors.

```
1 for(int i = Scopecontrol.size()-1; i >= 0; i--)
2 {
3   for(int j = Scopecontrol.get(i).size()-1; j >= 0; j--)
4   {
5     Temp2 = Scopecontrol.get(i).get(j).toString();
6     if(Temp.equals(Temp2))
7     {
8       return visit(ctx.id()) + visit(ctx.arrayidend());
```

Listing 3.9: The scope checker checks a variable by looking through all scopes, the list scopecontrol

After the scopecheck is finished, the compiler will print the errors in the list, listOfErrors.

3.6.2 Type Checking

In this section we will describe how we type-check the code using the visitor pattern generated by ANTLR [Antlr].

3.6.2.1 Tree traversal

To traverse our parse tree we override the basevisitors and at each node we visit, we check which node to visit next and this is done with every node that follows it until we

reach a leaf. As we reach a node we isolate important key data like; variables, types and operators. These are bubbled up to their respective node where they are used. For instance a variable is sent up from a leaf node to the declaration node and paired with a type to see if they are of the same type. When we meet a variable for the first time we store it along with its value(s) using a memory function called "variablememory". This allows us to replace the variable name with its value or a value of the given type if it has not been assigned a value, see section 3.6.2.4 for the explanation of what this is used for. We also use a memory function called "functionmemory" where we store our functions so we can easily call them later, see section 3.6.2.3 for more regarding the class used in the memory function.

3.6.2.2 Value

We use a class called "Value" to be able to return almost any type of data through the visitor pattern.

```
1 public class Value {
2  public static Value VOID = new Value(new Object());
3
4  final Object value;
5
6  public Value(Object value) {
7    this.value = value;
8  }
9
10 //Default constructor returns empty but not null value object
11 public Value()
12 {
13   this.value = "";
14 }
```

Listing 3.10: Value type

The class, as can be seen on listing 3.11, allows us to write expressions like "Value p = new Value(5)", thus we can return the value 5. This can also be used to combine different types and return them as a value. It is useful because we do not always know what type will be returned and this makes it possible to convert everything into the same type which we can evaluate later.

```
1 //Check if value is double
2 public boolean isDouble()
3 {
4   //If value matches the regular expressions, it must a double
5   if(this.toString().matches("((-)?[0-9]+)\\.([0-9]){1,2}"))
6   {
7     return true;
8   }
9   return false;
10 }
11
12 //Check if value is type int
13 public boolean isInt()
14 {
15   //The vale must match the regular expression, or be HIGH, LOW,
       OUTPUT or INPUT
16   if(this.toString().matches("(-)?[0-9]+") || this.toString().
       equals("HIGH") || this.toString().equals("LOW") || this.
       toString().equals("INPUT") || this.toString().equals("OUTPUT")
       )
17   {
18     return true;
19   }
20   return false;
21 }
```

Listing 3.11: How to evaluate Value

When we evaluate the type value we use functions like those seen on listing 3.12 which returns "true" or "false". In the code example we make use of regular expressions to determine if it is an integer or double by looking at what it contains like numbers and symbols in the defined order. We use similar functions for each type that exists in the language.

3.6.2.3 Function

We have made a class "function" as seen on listing 3.13 to make it easier to store the name, parameters and return value of a function. This will be used to retrieve the specific data that we need. Every time a function is found it is added to our function memory. This allows us to find it when for instance the function is called later in the code. We can then check if the call have the right setup simply by comparing the type and amount of parameters from our memory to the ones that are being used in the call.

```
1 public class Function {
2  String Name;
3  ArrayList<ParamsType> Params;
4  String ReturnType;
5
6  @Override
7  public String toString() {
8    return this.Name;
9  }
10 }
```

Listing 3.12: Function class

The code that the programmer writes is intended to be compiled to an Arduino platform and thus needs to have certain functions to work properly, these are the functions "setup" and "loop" and so we check if they are present. Some commands like LCDPrint and RFIDWrite are predefined in the Arduino language and can be used without a declaration, so we have also defined them in our language as seen on listing 3.14

```
1 ArrayList<Function> Functions = new ArrayList<Function>();
2
3 Function LCDPrint = new Function();
4
5 ParamsType LCDPrintParamsString = new ParamsType();
6 ParamsType LCDPrintParamsLine= new ParamsType();
7
8 LCDPrintParamsString.id = "x";
9 LCDPrintParamsString.type = "\"string\"";
10 LCDPrintParamsLine.id = "y";
11 LCDPrintParamsLine.type = "3";
12
13 ArrayList<ParamsType> LCDPrintParams = new ArrayList<ParamsType>();
14 LCDPrintParams.add(LCDPrintParamsString);
15 LCDPrintParams.add(LCDPrintParamsLine);
16 LCDPrint.Params = LCDPrintParams;
17 LCDPrint.Name = "LCDPrint";
18 LCDPrint.ReturnType = "nothing";
19 Functions.add(LCDPrint);
```

Listing 3.13: Defining LCDPrint

We will not implement all Arduino functions because it would take too much time and effort, so we have decided to only take the ones involved with the LCD and RFID and a couple of the most common used Arduino functions. Furthermore we have defined our own functions such as 3.15 so we can type check it.

```
1 Function PourDrink = new Function();
2 ParamsType PourDrinkParam = new ParamsType();
3
4 PourDrinkParam.id = "x";
5 PourDrinkParam.type = "drink";
6 PourDrink.Name = "pourDrink";
7 PourDrink.ReturnType = "nothing";
8 ArrayList<ParamsType> PourDrinkParams = new ArrayList<ParamsType>()
      ;
9 PourDrinkParams.add(PourDrinkParam);
10 PourDrink.Params = PourDrinkParams;
11 Functions.add(PourDrink);
```

Listing 3.14: Predefining function PourDrink

3.6.2.4 Variable

We have made a "variable" class to be make it easier to store and retrieve the necessary data as it is put into our "variablememory". This class can be seen on listing 3.16. In this class we have added two boolean expressions to make it easier to handle constants and arrays, so if we meet a constant or an array we set the appropriate expression to true. This will allows us to make a simple check to see if it is allowed to assign a new value to the variable.

```
1 public class Variable {
2  String Id;
3  String Type;
4  String Value;
5  boolean constant = false;
6  boolean isArray = false;
7 }
```

Listing 3.15: Variable class

In our language "INPUT" and "OUTPUT" would be seen as variables and by default be undefined so we predefined them like seen on listing 3.17. This is also done for "A0" to "A5" since these are used when determining input and output for the Arduino board. We have defined them with the type "container", because a container can have one of these values as its output.

```
1 Variable Output = new Variable();
2 Output.Id = "OUTPUT";
3 Output.Type = "int";
4 Output.Value = "0";
5 Variables.add(Output);
```

Listing 3.16: Predefining OUTPUT

When we make a declaration we first find the ID and then the type of the ID, then we visit the node "assign" that traverses the tree all the way down to the factor node and returns the value.

3.6.2.5 Drink

Our special type "drink" has its own separate declaration method, the initial part can be seen on listing 3.18. First we check if it is a new drink will or will not inherit from another drink. This is done be looking at how many IDs are present, if there are two then it means that will inherit from an existing drink. If there is only one ID then that means a new drink will not inherit from another drink.

```
1 //Handle the declaration of drinks
2 @Override public Value visitDrinkdcl(SPLADParser.DrinkdclContext
      ctx)
3 {
4   Value ID  = visit(ctx.id(0));
5
6   if(ctx.id(1) != null)
7   {
8     visit(ctx.id(1));
9     visit(ctx.changedrinkstmts());
10  }
11  else
12  {
13    visit(ctx.drinkstmts());
14  }
15
16  Variable var = new Variable();
17  var.Id = ID.toString();
18  var.Type = "drink";
19  var.Value = "drink";
20
21  //Add drink to variable memory
22  VariableMemory.put(ID.toString(), var);
23
24  return null;
25 }
```

Listing 3.17: Drink declaration

Whether or not it inherits from an existing drink, the type checker needs to check if the expressions used in the body are valid containers. If any of the expressions are invalid then an error for the given drink will be added to the list.

3.6.2.6 Error Handling

We have made an "error" class to notify the programmer of the mistakes the type checker have found while checking the program. When an error is encountered we send a number and an id to our "error" class as seen on listing 3.19.

```
1 //If the type is int or container, it the expression must not
      contain "."
2 if(type.toString().equals("int") || type.toString().equals("
      container"))
3 {
4   if(str.contains("."))
5   {
6     Error err = new Error(1, str);
7     ErrList.add(err);
8   }
9 }
```

Listing 3.18: Int Error

Our "error" class contains a switch case an excerpt of this can be seen on listing 3.20. The error found is saved in a list that will be shown, when all the nodes have been visited.

```
1 public String GetErrorMessage()
2 {
3   String ErrorMessage;
4
5   if(this.ErrorBody.equals("Syntax Error"))
6   {
7     ErrorMessage = "Syntax Error";
8   }
9   else
10  {
11   switch (this.ErrorNumber) {
12     case 1:  ErrorMessage = "Value \"" + this.ErrorBody +"\" is
           not of type int\n";
13     break;
14     case 2:  ErrorMessage = "Value \"" + this.ErrorBody +"\" is
           not of type numeric\n";
15      break;
```

Listing 3.19: Switch case error handling

3.7 Code Generation

Code generation is the last phase of a compiler. It is in this phase that the target code is generated. An overview of all the phases in a compiler can be seen on figure 3.2.

Traditionally a compiler generates machine code targeted at a specific architecture. This means that machine code compiled for the Intel x86 architecture will not be able to run on a RISC architecture. The reason why a program compiled for one architecture

will not run on another architecture, is that each architecture have slight differences in the set of instructions. The first phase in the code generation is the instruction selection. Instruction selection is needed exactly because of these different instruction sets. Another thing the code generator must take into account is the register allocation and code scheduling, which might be implemented differently on different architectures, hence the different instruction set. It is also obvious, that register allocation is not needed on a machine which has no registers. Here other techniques for storing intermediate values can be used depending on what the specific architecture provides.

Recall that the target code need not be machine code or byte code. A compiler can be used to compile a high-level language to a lower-level language, which means that the target code for example can be C. The main task of a *machine code* generator is to choose the right instructions, and to insert the right instructions at the right places, the latter is called the code scheduling. This task is important because some pieces of code might be dependent of other pieces of code. The code generator can then place these pieces of code close together, so no very long jumps occur. Long jumps are more expensive than shorter jumps. The code scheduler uses various algorithms to ensure the best position for a given piece of code, these algorithms includes topological sort, and shortest path algorithms.

Depending on how the compiler is implemented, the code generator can either generate code from a provided intermediate representation, which is supplied by a translator. This is typically used when the compiler also includes an optimizer, which optimizes the source program to obtain greater execution speed. If the compiler includes no optimizer, the code-generator can be implemented as a visitor, which traverses the decorated abstract syntax tree provided directly by the type checker.

3.7.1 Code Generation of SPLAD

The idea of using a high-level language is that it should be easier and faster to write programs. But by using a high-level language or any other kind of language, a compiler is needed to produce object code that should result in a running program, if the code is without errors. This section will be about how the code generation is implemented in this project and will also describe the choices that have affected the code generator.

The code produced by the code generator will need to run on an Arduino platform before it satisfies the project formulation, hence that the target code should be runnable on the targeted platform.

Arduino's IDE makes changes to the original code to ensure that the C/C++ code is correct. After this it calls avr-gcc which complies C/C++ code to object files and links to the necessary libraries. The object files are then uploaded to the Arduino unit using AVRDUDE [Arduino, d]. AVRDUDE is a tool used to upload to an AVR micro-controller [AVRDUDE, 2010]. For our code generator it would be the most correct way to implement these functions into the compiler itself, but because of other more critical tasks at hand these features have been substituted by using the Arduino IDE to compile and upload the C/C++ code to the Arduino board. Therefore the target code for our compiler should be Arduino C/C++ code after which we use Arduino IDE for further compilation and uploading.

```
1 public final static SPLADParser generatedParseTree(){
2   CharStream program = null;
3   try {
4     program = new ANTLRFileStream(file);
5   } catch (IOException e) {
6     e.printStackTrace();
7   }
8   //makes a lexer based on the source program
9   SPLADLexer lexer = new SPLADLexer(program);
10  //makes a stream of tokens based on the lexer
11  CommonTokenStream tokens = new CommonTokenStream(lexer);
12  //makes the parse based on the stream of tokens
13  SPLADParser parser = new SPLADParser(tokens);
14  //builds the parse tree
15  parser.setBuildParseTree(true);
16  return parser;
17 }
```

Listing 3.20: Here, the function to generate parse trees can be seen.

The code generation is done by using a visitor pattern to go through the parse tree provided by the generated lexer and parser. This is done by creating an "ANTLRfilestream" of the file that should be compiled. This ANTLRfilestream are then given as parameters for the lexer generation, which are then used to generate a token stream by using the CommonTokenStream constructor. ANTLRfilestream and CommonTokenStream are provided by the antlr-4.0-complete.jar which is a library provided by ANTLR at [ANTLR]. A parse tree is then generated by using a parser based on the token stream. This procedure has been made as a function that returns a parse tree of the type SPLADParser to make it easier to generate parse trees. This function can be seen in code example 3.21.

Before any of the actual code generation can begin we have to check scope rules, see section 3.6.1, and type check the program, see section 3.6.2. If any of the two checks returns any errors there will not be generated any code based on the idea that non-functional code is a waste of resources. The compiler then print these errors to the user as information regarding what did not go well during compilation. If the two checks do not result in any errors the compiler will generate code for the given program.

```
1 public String visitRoots(SPLADParser.RootsContext ctx) {
2   if (ctx.root() != null){
3     return visit(ctx.root()) + visit(ctx.roots());
4   }
5   else {
6     return "";
7   }
8 }
```

Listing 3.21: An example of how a visitor for at node is constructed.

The code generation uses a visitor pattern that extends the AbstractParseTreeVisitor which is provided by the ANTLR tool when generating the lexer and parser. By extending the AbstractParseTreeVisitor we only need to override the methods that we are interested in. The visitor is implemented as a class which we then create an object of. The tree traversing is started by using the objects visit method with the root node of the parse tree as parameter. This rode node then visits method corresponding to our BNF rules, see section 2.3.5, this way the parse tree is traversed all the way through. Each visitor then generates corresponding code depending on the type of node that have been traversed, an example of this can be seen in code example 3.22. All the generated code is stored in StringBuffers which are used to write all the code to a file. To illustrate the structure of a node thoroughly, we will describe and show code of how we have implemented the type container, drink and how some standard functions are provided to the users.

```
1 private String PrintContentofFile(String path){
2   InputStream in;
3   StringBuffer fileintxt = new StringBuffer();
4   try{
5     ClassLoader CLoader = this.getClass().getClassLoader();
6     in = CLoader.getResourceAsStream(path);
7     Scanner test = new Scanner(in,"UTF-8");
8     while(test.hasNext()){
9       fileintxt.append(test.useDelimiter("\\A").next());
10    }
11    in.close();
12    test.close();
13    return fileintxt.toString();
14  }
15  catch (IOException IOerror){
16    System.out.println("Could not read the file");
17    CodeGeneratorErrors.add("Syntax error");
18    return "";
19  }
20 }
```

Listing 3.22: Here the function for printing the content of a file can be seen.

Arduino uses a "setup" function for assigning values to global variables or starting modules like LCD or RFID. The "setup" function is called once before Arduino calls the "loop" function. The "loop" functions main purpose is to keep the program running. To provide some standard function for the users, it has been necessary to write these functions and stored them inside the compiler. The chosen stored method is a simple text file which can be opened and read from. To open and read files a simple function has been made which returns a string with the content of the file hence the name "PrintContentofFile". The function can be seen in code example 3.23. These additional functions are added to the "headerbuffer" StringBuffer before any of the users code. The translated user code is then added to the "headerbuffer" afterwards.

```
1 ContentBuffer.append("String ContainersnameSW407F13[" +
      ListOfContainers.size() + "];\n");
2 ContentBuffer.append("int ContainerspinSW407F13[" +
      ListOfContainers.size() + "];\n");
3
4
5 //Add content to the arrays in the setupfirstbuffer.
6 for (int i = 0; i < ListOfContainers.size(); i++){
7   setupfirstbuffer.append("ContainersnameSW407F13[" + i + "] = \""
        + ListOfContainers.get(i).containername + "\";\n");
8   setupfirstbuffer.append("ContainerspinSW407F13[" + i + "] = " +
        ListOfContainers.get(i).pinid + ";\n");
9 }
10
11 //Add the content of the program to the ContentBuffer
12 ContentBuffer.append(HeaderBuffer);
```

Listing 3.23: In this code example the creation and assigning to the container arrays can be seen

Special measurements was needed to implement our types drink and containers. To handle the type container a list of them is generated while visiting the parse tree. Two arrays of the types string and integer are then created based on the size of these lists and added to a "contentbuffer" of the type StringBuffer. The string array will contain the containers name and the integer array will contain the output pin, which the containers are associated with. The containers name and output pins are then added to the two arrays and stored in the "setupfirstbuffer". This can be seen in code example 3.24.

```
1 if (ctx.drinkstmts() != null){
2   visit(ctx.drinkstmts());
3 }
4 //Else it must be a drink which inherits from an other drink
5 else {
6   //Find the drink to inherit from
7   int k = 0;
8   while(!ListOfDrinks.get(k).drinkid.equals(visit(ctx.id(1)))){
9     k++;
10  }
11
12  //Add every ingredient from the inherited drink to the new empty
        drink in the drinkHolder.
13  for(Iterator<Ingredients> j = ListOfDrinks.get(k).
        ListOfIngredient.iterator(); j.hasNext();){
14    Ingredients tempingredient = new Ingredients();
15    Ingredients tempnextingredient = j.next();
16    tempingredient.Ingredientid = tempnextingredient.Ingredientid;
17    tempingredient.IngredientAmount = tempnextingredient.
          IngredientAmount;
18    drinkHolder.ListOfIngredient.add(tempingredient);
19  }
20
21  //Visit the statements
22  visit(ctx.changedrinkstmts());
```

Listing 3.24: In this example the code for handling normal declaration and declaration with inheritances can be seen.

The drink type is implemented in a similar way as the container. When a drink is declared the code generator will create an new object of the class "Drinks" for the global variable "drinkHolder". A drink can be declare normally or inherit a recipe from another drink, which the users then can alter to his liking. To consider these options we check if there are any "drinkstmts", if there are then no inheriting takes place and code generator goes through the "drinkstmts" to compose the recipe list. This recipe list is stored as a list of ingredients in "drinkHolder" which at the end of the "visitDrinkdcl" function is added to the global list of "Drinks". If there were no "drinkstmts", the code generator will look up the drink to inherit from in the global list of drinks. If the drink it is to inherit from exists it takes that drinks recipe and copies it into the new drinks "listofingredients". The code generator then goes through the "changedrinkstmts" and adds the new ingredients which are to be added or removed. "Drinkstmts" and "changedrinkstmts" are nodes that takes care of adding or removing ingredients from drinks. The difference between "drinkstmts" and "changedrinkstmts" is the context they are used in. "Drinkstmts" can only be used in a non-inheriting declaration while "changedrinkstmts" only works in declaration with inherits. This can be seen in code example 3.25.

```
1 tempreturnstring.append("double " + drinkHolder.drinkid + "[" + (
      drinkHolder.getIngredientcount()+1) + "][2];\n");
2 //The first element in the array will hold the size of the array.
3 setupfirstbuffer.append(drinkHolder.drinkid + "[" + 0 + "][0] =" +
      drinkHolder.getIngredientcount() + ";\n" + drinkHolder.drinkid +
       "[" + 0 + "][1] =" + drinkHolder.getIngredientcount() + ";\n");
4
5 //Add the ingrediens of the drink to the array.
6 for (int i = 0; i < drinkHolder.getIngredientcount(); i++){
7   int counter = 0;
8   while (ListOfContainers.size() > counter && !ListOfContainers.get
        (counter).containername.equals(drinkHolder.ListOfIngredient.
        get(i).Ingredientid)){
9     counter++;
10  }
11  int place = i + 1;
12  setupfirstbuffer.append(drinkHolder.drinkid + "[" + place + "][0]
         =" + counter + ";\n" + drinkHolder.drinkid + "[" + place + "
        ][1] =" + drinkHolder.ListOfIngredient.get(i).IngredientAmount
         + ";\n");
13 }
14
15 //add the drink to the list of drinks
16 ListOfDrinks.add(drinkHolder);
```

Listing 3.25: In this example it can be seen how the code generator generates code for the two-dimensional array and assigns content to it.

After the code generator has composed the drink recipe it appends the code to a StringBuffer, "tempreturnstring", to declare a two-dimensional array of the type double, named after the drink. The sizes are the size of the list of ingredients in the drink and two. The is done so each ingredient has an reference to container arrays and an amount of the given ingredient. The locations 0,0 and 0,1 are reserved to hold the number of the given ingredients for the given drink. This is done so one of the pre-made functions, "pourDrink", can get the array size. The code generator then runs through the list of ingredients and appends code for assigning the array with the id of the ingredient container and the amount to the StringBuffer "setupfirstbuffer", this can be seen in code example 3.26.

Lastly all the translated code is appended to "contentbuffer" which is used to store the translated program.

3.7.2 Unit Testing

This section will describe how unit tests have been used to ensure a higher rate of reliability to the compiler of this project.

3.7.2.1 Unit Testing in General

The basic idea in unit testing is to test small parts of your program. Suppose one had a custom class Foo, with the method Bar, which does some calculations, and returns them. One might want to test this method to ensure that it works as expected - this can be done using unit tests. A unit test for the above situation, could then be the following:

1. Create an object of the "Foo" class.
2. Call the "Bar" method with appropriate parameters.
3. Check that the result it what is expected.

Now this might seem simple, but unit testing is a very powerful tool for ensuring reliability in programs. Suppose that you have a very large and complex project, creating unit test while developing the project, ensures that each part of the project works as expected.

A unit test can essentially be 1 very simple line. The test-framework JUnit for Java is used in this project for unit testing. A unit test can be seen on listing 3.27, which checks that a method returns true. The annotation "@Test" tells JUnit that the following method is a unit test. The function "assertTrue" is a part of a unit test - here we assert that the method with the supplied parameters *must* return true.

```
1  @Test
2  public void testIsGreater() {
3      assertTrue("10 > 0 must return true", Value v = new Value(10)
           .GreaterThan(0));
4  }
```

Listing 3.26: Simple unit test

Test-frameworks like JUnit can automatically create test-skeletons, which then can be implemented in a desired way. When some unit tests have been implemented, most test-frameworks can then tell how much of the program code is covered by these tests. This is called the code-coverage, and can be used to check that a desirable amount of the program has been tested.

3.7.2.2 Unit Testing in This Project

In this project only the Value-class has been properly unit tested. This is because of the difficulties arising when trying to unit test methods in the visitor pattern, which requires nodes as input - it would require building a lot of parse trees for testing method under different conditions. However, every method in the type-checker uses the Value-class, which is very essential. Therefore it was decided to focus on having a decent code-coverage of the Value-class, which has been achieved with a code-coverage at 100%. JUnit, which is the test-framework used in this project, does not include a way to see the code-coverage. Therefore an additional plugin "CodeCover" [Schmidberger] to Eclipse has been used. This plugin is not only able to tell the code-coverage percentage. But it is also able to tell exactly which parts of the program code is covered, which is not covered, and which is partially covered by unit tests. By partially covered, CodeCover means methods which have only been tested in one direction. For example a method is partially covered, if there is a test for it, where the method returns true, but not a test where the method returns false. Additionally CodeCover is able to identify different kinds of code-coverage like these that are used in this project: Statement-, branch-, and term-coverage. Statement-coverage covers the very simplest type of tests. For example ensuring that "i–" is actually executed successfully and is considered a successful statement-test. Branch-coverage covers conditional statements, for example if-else statements. CodeCover can

complete a test, and tell which branch of the if-else statement has been taken. This is useful for discovering parts of a program which is never executed. Suppose one had a condition in an if-statement, which always returned true - the entire if-statement would then be useless, and could perhaps be removed, or corrected, if this behavior was a result of a programming error. Conditional-coverage covers the basic boolean terms in a conditional expression.

It should be noted that CodeCover also supports loop-coverage, ?-operator-coverage and Synchronized-coverage. But since these types of constructs are not used in the Value-class, they are not described further.

As mentioned above, the Value-class in this project is 100% covered by unit tests. This includes 100% statement-coverage, 100% branch-coverage and 100% term-coverage. Some unit test for the Value-class is very simple. For example the test for the Value.isint() method is very simple, as can be seen on listing 3.28. The unit test for the Value.isint() method simply test three conditions: It assumes that the number 10 is an integer. It assumes that the number 10.0 is not integer, and it assumes that the string "int" also is not an int. These three tests all passes, and the code-coverage of the Value.isint() method becomes 100%.

```
1 @Test
2 public void testIsInt() {
3   Value test = new Value(10);
4   assertTrue("10 must be an int", test.isInt());
5   assertFalse("10.0 is not int", new Value(10.0).isInt());
6   assertFalse("string is not", new Value("int").isInt());
7   assertTrue("OUTPUT is int", new Value("OUTPUT").isInt());
8   assertTrue("INPUT is int", new Value("INPUT").isInt());
9   assertTrue("HIGH is int", new Value("HIGH").isInt());
10  assertTrue("LOW is int", new Value("LOW").isInt());
11 }
```

Listing 3.27: Simple unit test of the Value.isint() method

A more complex test is the unit test for the Value.GetType() method. This is a longer test, because this test must test for each type. The unit test for Value.GetType() method can be seen on listing 3.29. It should be noted that this is only a part of the Value.GetType() method, because of the length of the method. As it can be seen, each type must be checked, to see if the Value.GetType() returns the appropriate string, for each type. Therefore the test contains test-cases for both bool, int, double, string, char, container and drink.

```
1   @Test
2   public void testGetType() {
3       Value test = new Value(true);
4       //Test bool
5       assertTrue("true must be bool", test.GetType().equals("bool")
            );
6       assertTrue("bool is type bool", new Value("bool").GetType().
            equals("bool"));
7
8       //Test int
9       assertTrue("20 is int", new Value(20).GetType().equals("int")
            );
10      assertFalse("1.0 is not int", new Value(1.0).GetType().equals
            ("int"));
11      assertTrue("int is type int", new Value("int").GetType().
            equals("int"));
12
13    ...
14  }
```

Listing 3.28: Unit test for the Value.GetType() method

The use of unit tests did actually result in code-changes, because it was discovered that the method Value.IsNumericExpression(), which determine if the value is a numeric expression, for example "10+4-1*4+4.0", did not accept numeric expression containing parentheses, which, of course, are valid in numeric expression. This resulted in small modification, which essentially removes all parentheses. Recall that the goal of the type-checker is to type-check programs, *not* to evaluate expressions. Therefore parentheses can be completely ignored in numeric expressions.

3.8 Component Setup

To illustrate the drink mixer construction and the functions of the program language we have made a simple prototype of a drink mixer. The prototype serve as a test rig for the testing the program language. The whole construction is made out of the Arduino board, a breadboard, a LCD, the RFID module, three buttons, LEDs, resistors and wires. For more about the components see section 3.2.

The breadboard is used for constructing prototypes and is ideal for changes, because it allows easily replacement of components or restructuring of the set up. The breadboard is used for mounting LEDs, resistors, buttons, wires and the LCD. Normally the RFID module is mounted directly on the Arduino board as a shield. A shield extends the boards pins which allows for other modules to use the free pins, but because the RFID module caused some rather questionable result with the LCD, it have been decided not to mount the RFID module, but instead hook it up with wires as seen on figure 3.10.

The buttons are used for controlling the program at runtime. The LEDs are used to symbolize the containers, which holds the different ingredients that are used when making a drink. The LEDs will light up when their ingredient is being used by the drink mixer. The LCD is used to provide the user with information when using the drink mixer.

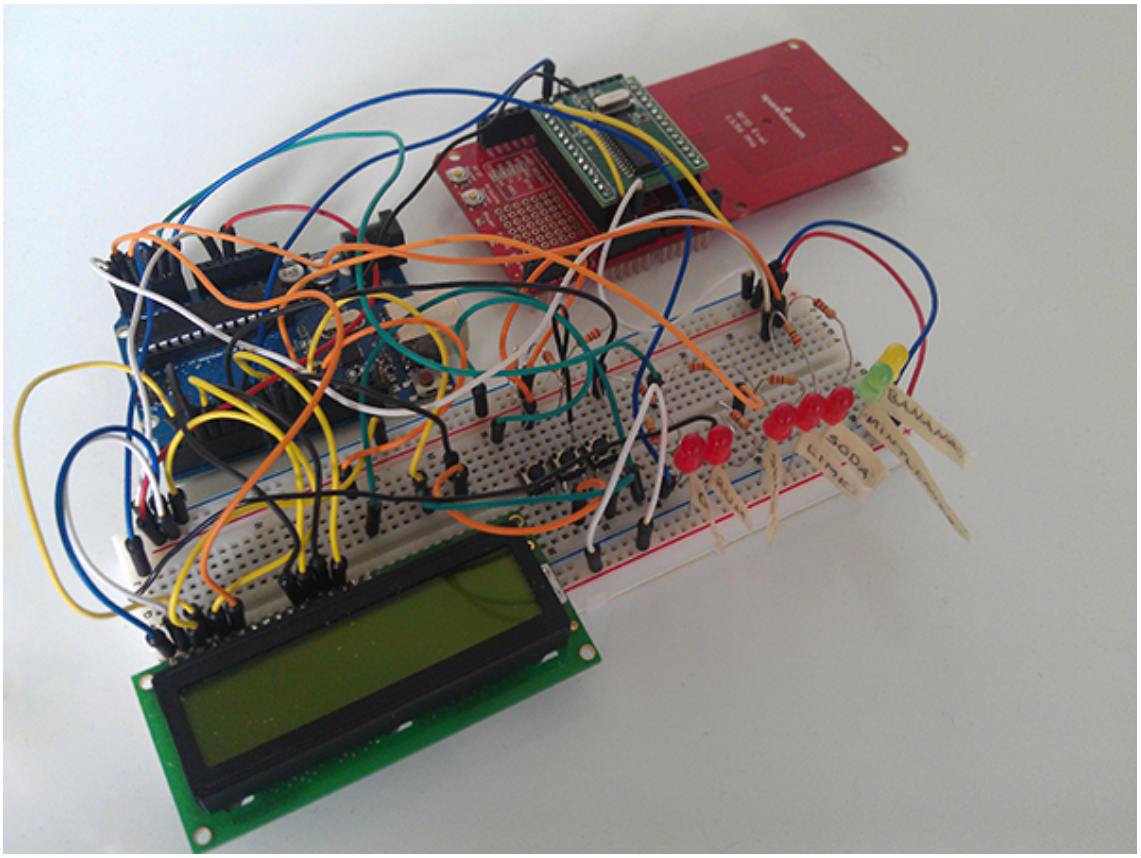


Figure 3.10: In this figure a illustrative resemble of a drink mixer can be seen.

The components are wired up to the Arduino board. The whole construction can be seen in figure 3.10.

3.9 Test of program written in SPLAD

This section contains a description of how the SPLAD compiler was tested in practice. The Arduino platform is built as seen in figure 3.10. This is the setup which was used as the test station. A basic test program (see appendix B) was written, which implemented various ingredients and drinks. As it can be seen in figure 3.10, no actual ingredients was used, instead a number of LED's were installed to act as containers of ingredients. When an LED is on, it means that the ingredient which is represented by the LED is added to the drink. The test was a success, and confirmed that all the input buttons, the RFID-reader/writer, the LED's as well as the LCD worked as intended, and the SPLAD compiler compiled the code, which could be compiled to the Arduino platform without problems.

4 Discussion and perspectivation

4.1 Discussion

This section contains a debate about the goals of this project, and if they have been met. There is also a section which contains the reflections about the project.

4.1.1 Characteristic of SPLAD

This section compares the design criteria set forward in section 2.2.2, and checks if these criteria have actually been met.

"Simplicity" was set to be high for SPLAD. This, as explained in section 4.1.2 has been achieved.

4.1.2 Simplicity of SPALD compared with C

This section will compare pieces of code written in SPLAD, with pieces of code written in C. The first code example, seen on listing 4.1, is an example of a simple SPLAD program, which prints "HelloWorld" on the LCD.

```
1 function pour return nothing using(int a, double b)
2 begin
3   return nothing;
4 end
5
6 function RFIDFound return nothing using(int a, int b)
7 begin
8   string message <-- "Hello World!";
9   /* Print message on line 1 on LCD */
10  call LCDPrint(message, 1);
11
12  return nothing;
13 end
```

Listing 4.1: Hello world program in SPLAD

Line 1-4 on listing 4.1 contains the pour function, which is required by the SPLAD compiler, but is irrelevant in this small example. The string "message" is declared on line 7, which is the string that will be printed to the LCD. On line 9, the function LCDPrint, which is provided by the SPLAD compiler is called, and the message is printed to the display. The message will be printed when an RFID-tag is found by the RFID-reader.

A hello world program written in Arudino C/C++, can be seen on listing 4.2. It should be noted that this is an example provided by the Arduino IDE [David A. Mellis]. Line 2 of this example includes the LCD Library. On line 5 the LCD display must be initialized with the pins on the Arduino. On line 9, the LCD display is setup, which means that the appropriate number of columns and rows is set depending on the particular model used. On line 11, the message is printed to the LCD.

```
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   // set up the LCD's number of columns and rows:
9   lcd.begin(16, 2);
10  char message[ ] = "Hello World";
11  // Print a message to the LCD.
12  lcd.print(message);
13 }
14
15 void loop() {
16
17 }
```

Listing 4.2: Hello world program in Arduino C/C++

The difference between the SPLAD program, and the Arduino program is clear: The SPLAD program is much more specialized with the target being drink machines, which means that there is an LCD print function provided by default. This is not the case in a normal Arduino program, because Arduino is aimed at a much more general purpose. This means that users of the SPLAD language do not need to think about which pins the LCD is connected to while writing their programs. Arduino does not provide a string-type, which means that strings are implemented by char arrays. In the SPLAD language there is a string type, which might seem more intuitive for novice programmers. The assignment is a bit different in SPLAD compared to C-notation. In SPLAD an assignment is denoted by '<–', which makes it completely clear what is assigned to what. In C, the assignment is denoted by '=', which might confuse novice programmers, because '=' generally is used to denote equality in for example mathematics. When using the '=', it can be unclear what is assigned to what because the are nothing indicating the direction.

4.1.3 Other Criteria

"Orthogonality" was set to medium for our language since it is not possible to create classes or custom constructs in our language. But it is possible to call a function in

an expression and to define functions. Therefore this criteria has also been met to an acceptable extend.

"Data types" was set to medium for our SPLAD language. Because SPLAD only have five primitive types and two special types, which makes it simple to keep an overview of them. On top of that, it is not possible to make classes. Therefore the criteria about data types is met.

"Syntax design" was set to be high for SPLAD. Since most of SPLAD has been created this way to give a better understanding of the different constructs, such as for-loops, and beginning and endings of block statements. Therefore this criteria has been achieved in SPLAD.

"Support for abstraction" was set to be medium for SPLAD, because our language is an abstraction of the Arduino programming language. Examples of abstraction can be seen in section 4.1.2. This abstraction make programming drink machines based on the Arduino platform easier.

The criteria "Expressivity" was set to low for SPLAD. Since SPLAD has a focus to be as simple as possible, the expressivity is not in focus for SPLAD.

"Type checking" was set to high for SPLAD. To make sure that all type errors that could be found, are found when a program is compiled, a lot of work and tests have been made while creating the type checker. There were some problems, like the build-in functions that an Arduino program has access to. This makes it hard to make a type checker that will find all errors. But all types, expressions, parameters which are in SPLAD are type-checked therefore this criteria is met.

4.2 Perspectivation

The idea of this project is, that any hobby programmer can program their own drink machine based on the Arduino platform. The purpose of the SPLAD language is to provide an easy alternative to the Arudino C/C++ like language, when programming a drinks machine. In the first place this was intended to be used by the owners of small bars, which also has a hobby of programming. Using the SPLAD language, they will be able to build their own drinks machines and program them to their liking.

The reason why bar owners would want this solution is that it could lessen the work load of the bartender, by replacing some of his or her tasks. For certain drinks, the bartender could simply sell a pre-programmed RFID-tag, which could be used on the drinks machine, and the bartender could then serve more customers. This is of course an advantage to both the bartenders and the bar owners. The bartenders can focus on other tasks than mixing the drinks, and for the bar owners, it would be like having an extra bartender without having to pay the extra wage. SPLAD could make this possible to a group of people, who would not have been able to make such a program otherwise.

4.3 Further Development

This section outlines which areas of the language, compiler and the Arduino component would be natural to develop further.

It would be interesting to actually build the drink machine, with appropriate containers with ingredients, hoses and valves, so it actually could mix drinks. It has not been possible

to build a complete drink machine because of limited funds. The construction of the drink machine would also require time to build which either must be taken from the actual project or some of the non-project time must be dedicate to the built. It could be interesting to pitch the idea of a drink machine to an actual bar, to see if it would be useful in a real context.

On the compiler side, it would be natural to develop the compiler so it do the producing of the Arduino code itself, and thereby skip the step for the user of compiling in the Arduino compiler. The reason why it was decided to compile into Arduino C/C++ is, that it was decided that it would be too time consuming to fully understand the Arduino machine-code, and then compile into machine-code.

When debugging code, it is important to have correct error messages. While this already is the case for the SPLAD compiler, it does not provide somewhat important information such as the line number the error happened. The reason why this was not in focus in the initial development of the compiler, is that it was simply deemed too time consuming, and not as important as getting the compiler working in the first place. The language could be extended with a construction to handle arrays of drinks, which would enable a simpler way to work with multiple drinks. Again the reason why this was not implemented in the first place, is that is would be too time consuming, compared to the deadline of this project.

Lastly, SPLAD should also handle different operations on the drink, such as stir, shake, cool and heat. This was not supported in the project, because there are too many ways to implement such features. In a future version of SPLAD, this should be supported.

Conclusion 5

This chapter concludes the report, and contains a round up of the most important aspects of the project. The problem statement was presented in section 1.2, and is as follows:

- **How can a programming language be developed, which makes it suitable for the hobby programmer to program drink machines based on Arduino platforms?**

To answer the problem statement, the SPLAD language was developed, which aimed at being a programming language that less experienced- and hobby programmers could program in. The formal specification of the SPLAD language can be seen in chapter 2. The SPLAD language is an abstraction of the C/C++ like language used for programming for the Arduino platform. The abstraction enables the programming of a drinks machine with less effort than if it was programmed directly in the Arduino language, this is discussed more thoroughly in 4.1. The main focus of the problem statement is, how a programming language suitable for a novice programmer can be specified.

This has been done by only including simple constructs, and not support more writeable statements, such as "i++" or "?" instead of "if else". This was decided to heighten readability, at the expense of decreasing writeability. These decisions and the trade-offs between the different criteria which can be seen in section 2.2. Another design decision that was made to heighten readability at the cost of writeability was the assignment of values to variables. In SPLAD this is denoted by "x <– 4", where most other programming languages simply allows "x = 4". This decision removes any doubt about what is assigned to what. These decisions makes the SPLAD language more suitable for novice programmers. Another part of developing a programming language, is making a compiler for the language. This process can be seen in chapter 3. The parser and lexical analyzer (lexer) of the SPLAD compiler is generated by ANTLR, which is a lexer and parser generator. The parser and lexer generates a parse tree, which is given to the type checker, which then traverses this tree and checks that the types fulfils the given rules for the SPLAD language. At last the code generator generates the target code, which can then be further compiled by the Arduino compiler. This answers the second sub-statement of the problem statement. Therefore it can be concluded, that this project gives a fulfilling answer to the problem stated in this project.

Bibliography

Antlr. Antlr. *Theory behind Antlr*. `http://www.antlr.org/about.html` [Last seen: 2013/03/15].

ANTLR. ANTLR. *Download ANTLR*. `http://www.antlr.org/download.html` [Last seen: 2013/05/12].

Arduino, a. Arduino. *Liquid Crystal*. `http://arduino.cc/en/Tutorial/LiquidCrystal` [Last seen: 2013/02/18].

Arduino, b. Arduino. *Language Reference*. `http://arduino.cc/en/Reference/HomePage` [Last seen: 2013/02/19].

Arduino, c. Arduino. *Arduino Uno*. `http://arduino.cc/en/Main/ArduinoBoardUno` [Last seen: 2013/02/18].

Arduino, d. Arduino. *Arduino Build Process*. `http://arduino.cc/en/Hacking/BuildProcess` [Last seen: 2013/05/12].

AVRDUDE, 2010. *AVRDUDE*. `http://www.nongnu.org/avrdude/` [Last seen: 2013/05/12], 2010.

Bent Thomsen, teacher at Department of Computer Science, 2013. Aalborg university Bent Thomsen, teacher at Department of Computer Science. *Languages and Compilers Lecture 1*, 2013. Slides of a Lecture given 7.2.13 by Bent Thomsen in Department of Computer Science, Aalborg university.

Chomsky, 1959. Noam Chomsky. *On certain formal properties of grammars*. Information and Control, 2(2), 137 – 167, 1959. ISSN 0019-9958.

Cup. Cup. *Theory behind CUP*. `http://www2.cs.tum.edu/projects/cup/manual.html` [Last seen: 2013/03/15].

David A. Mellis, Limor Fried. Tom Igoe David A. Mellis, Limor Fried. *LiquidCrystal - "Hello World!"*. web. URL `http://www.arduino.cc/en/Tutorial/LiquidCrystal`.

Fischer et al., 2009. Charles N. Fischer, K. Cyton Ron og J. LeBlanc. Jr. Richard. *Crafting a Compiler*. Pearson, 2009.

Flex. Flex. *Theory behind Flex*. `http://flex.sourceforge.net/manual/` [Last seen: 2013/03/15].

Hüttel, 2010. Hans Hüttel. *Transitions and Trees*. Cambridge University Press, 2010.

Jflex. Jflex. *Theory behind Jflex*. `http://jflex.de/manual.html#SECTION00040000000000000000` [Last seen: 2013/03/15].

JJTree. JJTree. *JJTree to JavaCC*. http://tomcopeland.blogs.com/juniordeveloper/2007/10/better-jjtree-v.html [Last seen: 2013/03/15].

Jlex. Jlex. *Theory behind Jlex*. http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html [Last seen: 2013/03/15].

elektronik. let elektronik. *LCD 16x2 Characters - Green Yellow back light*. http://www.let-elektronik.dk/lcd-16x2-characters-green-yellow-back-light.html [Last seen: 2013/05/12].

Lex. Lex. *Theory behind Lex*. http://dinosaur.compilertools.net/lex/index.html [Last seen: 2013/03/15].

Martin, 2003. John Martin. *Introducing to Language and the Theory of Computation - Third edition*. Elizabeth A. Jones, 2003.

msujaws, 3 May 2011. msujaws. *Static vs. Dynamic Scoping*. http://msujaws.wordpress.com/2011/05/03/static-vs-dynamic-scoping/ [Last seen: 2013/05/07], 2011.

Norvell. Theodore S. Norvell. *The JavaCC FAQ*. http://www.engr.mun.ca/~theo/JavaCC-FAQ/javacc-faq-moz.htm#jjtree-and-jtb [Last seen: 2013/03/15].

Nosowitz, 01 2011. Dan Nosowitz. *Everything You Need to Know About Near Field Communication*. http://www.popsci.com/, 2011.

Nørmark, July 7 2010a. Kurt Nørmark. *Overview of the four main programming paradigms*. http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html [Last seen: 2013/03/19], 2010.

Nørmark, July 7 2010b. Kurt Nørmark. *Overview of the Programming paradigms*. http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigms.html [Last seen: 2013/04/10], 2010.

Parr, Nov 2012. Terence Parr. *Lexer Rules*. http://www.antlr.org/wiki/display/ANTLR4/Lexer+Rules [Last seen: 2013/04/03], 2012.

Parr. Terence Parr. *Tree Construction*. http://www.antlr.org/wiki/display/ANTLR3/Tree%2Bconstruction [Last seen: 2013/04/03].

PJRC. PJRC. *Teensy USB Development Board*. http://www.pjrc.com/teensy/index.html [Last seen: 2013/03/11].

Popplestone, 1999. Robin Popplestone. *Lecture 1: What are Programming Paradigms?* http://www.cs.bham.ac.uk/research/projects/poplog/paradigms_lectures/lecture1.html [Last seen: 2013/04/17], 1999.

SableCC. SableCC. *SableCC homesite*. http://sablecc.org/wiki [Last seen: 2013/03/15].

Schmidberger. Rainer Schmidberger. *Codecover.org*. `http://codecover.org/` [Last seen: 2013/05/15].

Sebesta, **2009**. Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 9 udgave, 2009.

Sipser, **2013**. Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing, 3 udgave, 2013.

Softwere, **April 2013**. TIOBE Softwere. *Programming Community Index for April 2013*. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html` [Last seen: 2013/05/03], 2013.

Specialisten. RFID Specialisten. *RFID*. `http://www.rfid-specialisten.dk/rfid.asp` [Last seen: 2013/02/15].

Studio. Seeed Studio. *Seeduino Stalker V2*. `http://www.seeedstudio.com/depot/seeeduino-stalker-v2-p-727.html?cPath=80` [Last seen: 2013/03/11].

Walker, **April 2012**. Chris Walker. *Introducing Netduino Go*. `http://forums.netduino.com/index.php?/topic/3867-introducing-netduino-go/` [Last seen: 2013/03/11], 2012.

Yacc. Yacc. *Theory behind Yacc*. `http://dinosaur.compilertools.net/yacc/index.html` [Last seen: 2013/03/15].

List of Corrections

Transition Rules A

A.1 Abstract Syntax

$$
\begin{aligned}
R &::= D_P\ D_A\ D_V \mid R_1\ R_2 \\
S &::= x := a \mid r[a_1] := a_2 \mid S_1;\ S_2 \mid \text{if } b \text{ begin } S \text{ end} \mid \text{if } b \text{ begin } S_1 \text{ end else begin } S_2 \text{ end} \\
&\quad \text{while } b \text{ begin } S \text{ end} \mid \text{from } x := a_1 \text{ to } a_2 \text{ step } a_3 \text{ begin } S \text{ end} \mid \text{call } p(\vec{x}) \mid D_V \mid D_A \\
&\quad \mid \text{switch}(a) \text{ begin case } a_1 :\ S_1 \text{ break; } \ldots \text{ case } a_k :\ S_k \text{ break; default :\ } S \text{ break end} \\
a &::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \mid (a) \mid r[a_i] \\
b &::= a_1 = a_2 \mid a_1 > a_2 \mid a_1 < a_2 \mid \neg b \mid b_1\ \wedge\ b_2 \mid b_1\ \vee\ b_2 \mid (b) \\
D_V &::= \text{var } x := a \mid \varepsilon \\
D_P &::= \text{ func } p(\vec{x}) \text{ is begin } S \text{ end} \mid \varepsilon \\
D_A &::= \text{array } r[a_1] \mid \varepsilon
\end{aligned}
$$

$$
\text{[BLOCK]} \quad \frac{\begin{array}{c}\langle D_V, env_V, sto\rangle \rightarrow_{DV} (env'_V, sto'') \\ \langle D_A, env'_V, sto''\rangle \rightarrow_{DA} (env''_V, sto') \\ env''_V \vdash \langle D_P, env_P\rangle \rightarrow_{DP} env'_P\end{array}}{env_V, env_P \vdash \langle\ D_V\ D_A\ D_P, sto\rangle \rightarrow sto'}
$$

$$
\text{[ROOT]} \quad \frac{\begin{array}{c} env_V, env_P \vdash \langle R_1, sto\rangle \rightarrow sto'' \\ env_V, env_P \vdash \langle R_2, sto''\rangle \rightarrow sto'\end{array}}{env_V, env_P \vdash \langle R_1; R_2, sto\rangle \rightarrow sto'}
$$

Table A.1: Root statements

Transitions are on the form: $env_V, env_P \vdash \langle S, sto\rangle \rightarrow sto'$

[VAR-ASS] $env_V, env_P \vdash \langle x <-- a, sto\rangle \rightarrow sto[l \mapsto v]$

where $env_V, sto \vdash a \rightarrow_a v$
and $env_V\ x = l$

Continued on the next page

[ARR-ASS] $$env_V, env_P \vdash \langle r[a_1] <-- a_2, sto\rangle \to sto[l_2 \mapsto v_2]$$

where $env_V, sto \vdash a_1 \to_a v_1$
and $env_V, sto \vdash a_2 \to_a v_2$
and $env_V\ r = l_1$
and $l_2 = l_1 + v_1 + 1$
and $v_3 = sto\ l_1$
and $0 \leq v_1 \leq v_3$

[COMP] $$\frac{env_V, env_P \vdash \langle S_1, sto\rangle \to sto'' \quad env_V, env_P \vdash \langle S_2, sto''\rangle \to sto'}{env_V, env_P \vdash \langle S_1; S_2, sto\rangle \to sto'}$$

[IF-TRUE] $$\frac{env_V, env_P \vdash \langle S, sto\rangle \to sto'}{env_V, env_P \vdash \langle \text{if } b \text{ begin } S \text{ end}, sto\rangle \to sto'}$$

if $env_V, sto \vdash b \to_b \text{true}$

[IF-FALSE] $$env_V, env_P \vdash \langle \text{if } b \text{ begin } S \text{ end}, sto\rangle \to sto$$

if $env_V, sto \vdash b \to_b \text{false}$

[IF-ELSE-TRUE] $$\frac{env_V, env_P \vdash \langle S_1, sto\rangle \to sto'}{env_V, env_P \vdash \langle \text{if } b \text{ begin } S_1 \text{ end else begin } S_2 \text{ end}, sto\rangle \to sto'}$$

if $env_V, sto \vdash b \to_b \text{true}$

[IF-ELSE-FALSE] $$\frac{env_V, env_P \vdash \langle S_2, sto\rangle \to sto'}{env_V, env_P \vdash \langle \text{if } b \text{ begin } S_1 \text{ end else begin } S_2 \text{ end}, sto\rangle \to sto'}$$

if $env_V, sto \vdash b \to_b \text{false}$

[WHILE-TRUE] $$\frac{env_V, env_P \vdash \langle S, sto\rangle \to sto'' \quad env_V, env_P \vdash \langle \text{while } b \text{ begin } S \text{ end}, sto''\rangle \to sto'}{env_V, env_P \vdash \langle \text{while } b \text{ begin } S \text{ end}, sto\rangle \to sto'}$$

if $env_V, sto \vdash b \to_b \text{true}$

[WHILE-FALSE] $$env_V, env_P \vdash \langle \text{while } b \text{ begin } S \text{ end}, sto\rangle \to sto$$

if $env_V, sto \vdash b \to_b \text{false}$

[FROM-TRUE] $$\frac{env_V, env_P \vdash \langle S, sto[l \mapsto v_1]\rangle \to sto'' \quad \langle \text{from } x <-- a_1 + a_3 \text{ to } a_2 \text{ step } a_3 \text{ begin } S \text{ end}, sto''\rangle \to sto'}{env_V, env_P \vdash \langle \text{from } x <-- a_1 \text{ to } a_2 \text{ step } a_3 \text{ begin } S \text{ end}, sto\rangle \to sto'}$$

where $env_V, sto \vdash a_1 \to_a v_1$
and $env_V, sto \vdash a_2 \to_a v_2$
and $env_V, sto \vdash a_3 \to_a v_3$

Continued on the next page

and $v_1 \leq v_2$
and $l = env_V\, x$

[FROM-FALSE] $env_V, env_P \vdash \langle \text{from } x := a_1 \text{ to } a_2 \text{ step } a_3 \text{ begin } S \text{ end}, sto\rangle \rightarrow sto$

where $env_V, sto \vdash a_1 \rightarrow_a v_1$
and $env_V, sto \vdash a_2 \rightarrow_a v_2$
and $env_V, sto \vdash a_3 \rightarrow_a v_3$
and $v_1 > v_2$

[CALL] $$\frac{env'_V[\vec{z} \mapsto \vec{l}], env'_P \vdash \langle S, sto[\vec{l} \mapsto \vec{v}]\rangle \rightarrow sto'}{env_V, env_P \vdash \langle \text{call } p(\vec{a}), sto\rangle \rightarrow sto'}$$

where $env_P\, p = (S, \vec{z}, env'_V, env'_P)$
and $|\vec{a}| = |\vec{z}|$
and $env_V, sto \vdash a_i \rightarrow v_i$ for each $1 \leq i \leq |\vec{a}|$
and $l_1 = env_V$ new
and $l_{i+1} = \text{new } l_i$ for each $1 \leq i < |\vec{a}|$

Table A.2: Statements

[SWITCH-1] $$\frac{env_V, env_P \vdash \langle S, sto\rangle \rightarrow (sto')}{env_V, env_P \vdash \langle \text{switch}(a) \text{ begin case } a_1 : \ S_1 \text{ break; default : } S \text{ break; end}, sto\rangle \rightarrow sto'}$$

Where $env_V, sto \vdash a \rightarrow_a v$
and $env_V, sto \vdash a_1 \rightarrow_a v_1$
and $v \neq v_1$

[SWITCH-2] $$\frac{env_V, env_P \vdash \langle S_1, sto\rangle \rightarrow sto'}{env_V, env_P \vdash \langle \text{switch}(a) \text{ begin case } a_1 : \ S_1 \text{ break; } \ldots \text{ case } a_k : \ S_k \text{ break; default : } S \text{ break; end}, sto\rangle \rightarrow sto'}$$

Where $k > 0$
and $env_V, sto \vdash a \rightarrow_a v$
and $env_V, sto \vdash a_1 \rightarrow_a v_1$
and $v = v_1$

[SWITCH-3] $$\frac{env_V, env_P \vdash \langle \text{switch}(a) \text{ begin case } a_2 : \ S_2 \text{ break; } \ldots \text{ case } a_k : \ S_k \text{ break; default : } S \text{ break; end}, sto\rangle \rightarrow sto'}{env_V, env_P \vdash \langle \text{switch}(a) \text{ begin case } a_1 : \ S_1 \text{ break; } \ldots \text{ case } a_k : \ S_k \text{ break; default : } S \text{ break; end}, sto\rangle \rightarrow sto'}$$

Where $k > 1$
and $env_V, sto \vdash a \rightarrow_a v$
and $env_V, sto \vdash a_1 \rightarrow_a v_1$
and $v \neq v_1$

Table A.3: Statements

Transitions are on the form: $env_V, sto \vdash a \rightarrow_a v$

[NUM] $env_V, sto \vdash n \rightarrow_a v$

if $\mathcal{N}[[n]] = v$

[VAR] $env_V, sto \vdash x \rightarrow_a v$

if $env_V\ x = l$
and $sto\ l = v$

[ADD] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 + a_2 \rightarrow_a v}$$

where $v = v_1 + v_2$

[SUB] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 - a_2 \rightarrow_a v}$$

where $v = v_1 - v_2$

[MULT] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 \cdot a_2 \rightarrow_a v}$$

where $v = v_1 \cdot v_2$

[DIV] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 / a_2 \rightarrow_a v}$$

where $v = v_1 / v_2$

[PAR] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1}{env_V, sto \vdash (a_1) \rightarrow_a v_1}$$

[ARR] $env_V, sto \vdash r[a_1] \rightarrow_a a_2$

where $env_V, sto \vdash a_1 \rightarrow_a v_1$
and $env_V, sto \vdash a_2 \rightarrow_a v_2$
and $env_V\ r = l$
and $sto\ l = v_3$
and $0 < v_1 \leq v_3$
and $sto(l + v_1) = v_2$

Table A.4: Arithmetic expressions

Transitions are on the form: $env_V, sto \vdash b \rightarrow_b \text{bool}$

[EQUAL-TRUE] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 = a_2 \rightarrow_b \text{true}}$$

Continued on the next page

if $v_1 = v_2$

[EQUAL-FALSE] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 = a_2 \rightarrow_b \text{false}}$$

if $v_1 \neq v_2$

[GRT-TRUE] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 > a_2 \rightarrow_b \text{true}}$$

if $v_1 > v_2$

[GRT-FALSE] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 > a_2 \rightarrow_b \text{false}}$$

if $v_1 \not> v_2$

[LESS-TRUE] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 < a_2 \rightarrow_b \text{true}}$$

if $v_1 < v_2$

[LESS-FALSE] $$\frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 < a_2 \rightarrow_b \text{false}}$$

if $v_1 \not< v_2$

[NOT-1] $$\frac{env_V, sto \vdash b \rightarrow_b \text{true}}{env_V, sto \vdash !b \rightarrow_b \text{false}}$$

[NOT-2] $$\frac{env_V, sto \vdash b \rightarrow_b \text{false}}{env_V, sto \vdash !b \rightarrow_b \text{true}}$$

[AND-TRUE] $$\frac{env_V, sto \vdash b_1 \rightarrow_b \text{true} \quad env_V, sto \vdash b_2 \rightarrow_b \text{true}}{env_V, sto \vdash b_1 \wedge b_2 \rightarrow_b \text{true}}$$

[AND-FALSE] $$\frac{env_V, sto \vdash b_i \rightarrow_b \text{false}}{env_V, sto \vdash b_1 \wedge b_2 \rightarrow_b \text{false}}$$

where $i \in 1, 2$

[OR-TRUE] $$\frac{env_V, sto \vdash b_i \rightarrow_b \text{true}}{env_V, sto \vdash b_1 \vee b_2 \rightarrow_b \text{true}}$$

where $i \in 1, 2$

[OR-FALSE] $$\frac{env_V, sto \vdash b_1 \rightarrow_b \text{false} \quad env_V, sto \vdash b_2 \rightarrow_b \text{false}}{env_V, sto \vdash b_1 \vee b_2 \rightarrow_b \text{false}}$$

Continued on the next page

[PAR-BOOL] $$\frac{env_V, sto \vdash b \rightarrow_b v}{env_V, sto \vdash (b) \rightarrow_b v}$$

Table A.5: Boolean expressions

Transitions are on the form: $\langle D_V, env_V, sto\rangle \rightarrow_{DV} (env'_V, sto')$

[VAR-DEC] $$\frac{\langle D_V, env''_V, sto[l \mapsto v]\rangle \rightarrow_{DV} (env'_V, sto')}{\text{var } x <-- a, env_V, sto\rangle \rightarrow_{DV} (env'_V, sto')}$$

where $env_V, sto \vdash a \rightarrow_a v$
and $l = env_V$ next
and $env''_V = env_V[x \mapsto l][\text{next} \mapsto \text{new } l]$

[EMPTY-VAR] $\langle \varepsilon, env_V, sto\rangle \rightarrow_{DV} (env_V, sto)$

Transitions are on the form: $env_V \vdash \langle D_P, env_P\rangle \rightarrow_{DP} env'_P$

[PROC-PARA-DEC] $$\frac{env_V \vdash \langle D_P, env_P[p \mapsto (S, \vec{x}, env_V, env_P)]\rangle \rightarrow_{DP} \ env'_P}{env_V \vdash \langle \text{function } p \text{ using(var } \vec{x}) \text{ begin } S \text{ end}, \ env_P\rangle \rightarrow_{DP} env'_P}$$

[EMPTY-PROC] $env_V \vdash \langle \varepsilon, env_P\rangle \rightarrow_{DP} env'_P$

Transitions are on the form: $\langle D_A, env_V, sto\rangle \rightarrow_{DA} (env'_V, sto')$

[ARRAY-DEC] $$\frac{\langle D_A, env_V[r \mapsto l, \text{next} \mapsto l + v + 1], sto[l \mapsto v]\rangle \rightarrow_{DA} (env'_V, sto')}{\langle \text{array } r[a_1], env_V, sto\rangle \rightarrow_{DA} (env'_V, sto')}$$

where $env_V, sto \vdash a_1 \rightarrow_a v$
and $l = env_V$next
and $v > 0$

[EMPTY-ARRAY] $\langle \varepsilon, env_V, sto\rangle \rightarrow_{DA} (env_V, sto)$

Table A.6: Declarations

Program written in SPLAD B

```
1 /*Buttons for input*/
2 int ButtonOne <-- 12;
3 int ButtonTen <-- 10;
4 int ButtonConfirm <-- 13;
5
6 /*Bool for checking if it is staffmode or not*/
7 bool IsStaffInput <-- false;
8
9 /*The containers for ingredients*/
10 container Gin <-- A0; /*Red*/
11 container Sugar <-- A2;/*Red*/
12 container Soda <-- A1;/*Red*/
13 container Lime <-- 11;/*Red*/
14 container Rum <-- 6;/*Red*/
15 container Mintleaves <-- A5; /*Green*/
16 container Banana <-- A3; /*Yellow*/
17
18 /*Tom collins drink*/
19 drink TomCollins is
20 begin
21  add 3 of Gin;
22  add 2 of Sugar;
23  add 2 of Lime;
24  add 5 of Soda;
25 end
26
27 /*A banana shake*/
28 drink BananaShake is
29 begin
30  add 5 of Banana;
31  add 5 of Rum;
32 end
33
34 /*A mojito drink*/
35 drink Mojito is
36 begin
37  add 2 of Soda;
38  add 2 of Rum;
39  add 2 of Lime;
40  add 2 of Sugar;
41  add 3 of Mintleaves;
42 end
43
44 /*A little bit of everything*/
```

```
45 drink AllTogether is
46 begin
47  add 2 of Gin;
48  add 2 of Sugar;
49  add 2 of Soda;
50  add 2 of Lime;
51  add 2 of Rum;
52  add 2 of Mintleaves;
53  add 2 of Banana;
54 end
55
56 /*A sweeter version of Tom Collins. Inherrits from the original Tom
       Collins.*/
57 drink SweetCollins as TomCollins but
58 begin
59  remove Lime;
60  add 1 of Lime;
61  add 1 of Sugar;
62 end
63
64 /*A software shot*/
65 drink SoftwareShot is
66 begin
67  add 3 of Gin;
68  add 3 of Sugar;
69  add 3 of Lime;
70 end
71
72 /*A startup message*/
73 function StartMsg return nothing using()
74 begin
75  call LCDPrint("Welcome to", 0);
76  call LCDPrint("      SPLAD     ", 1);
77  return nothing;
78 end
79
80 /*A function to wait for user input*/
81 function WaitOnInput return nothing using()
82 begin
83  call delay(300);
84  /*While no buttons is pressed, wait and try again*/
85  while(call digitalRead(ButtonOne) = LOW AND call digitalRead(
       ButtonTen) =  LOW AND call digitalRead(ButtonConfirm) = LOW)
86  begin
87    call delay(100);
88  end
89  return nothing;
90 end
91
92 /*Get a number from the user*/
93 function GetInputNumber return int using()
94 begin
95  int ReturnValue <-- 0;
96  /*While confirm not pressed, test for input and increas the input
       */
97  while(call digitalRead(ButtonConfirm) = LOW)
98  begin
99    call delay(100);
100   if(call digitalRead(ButtonOne) = HIGH)
101   begin
102     ReturnValue <-- ReturnValue + 1;
103   end
```

```
104     else if(call digitalRead(ButtonTen) = HIGH)
105     begin
106       ReturnValue <-- ReturnValue + 10;
107     end
108     call LCDPrint(string(ReturnValue), 1);
109   end
110   return ReturnValue;
111 end
112
113 /*The action performed when a RFID is found during staffmode*/
114 function StaffAction return nothing using()
115 begin
116   call LCDPrint("Type ID of the", 0);
117   call LCDPrint("drink to buy", 1);
118
119   /*Wait for user to press a button*/
120   call WaitOnInput();
121   call LCDClear();
122   call LCDPrint("Drink ID:", 0);
123
124   /*Get the Drink ID number to write to the RFID from the user*/
125   int drinkID <-- call GetInputNumber();
126
127   call LCDPrint("Number of drinks", 0);
128   call LCDPrint("",1);
129
130   /*Wait for user to press a button*/
131   call WaitOnInput();
132
133   int Amount <-- call GetInputNumber();
134
135   /*Check if write was succesful*/
136   if(call RFIDWrite(drinkID, Amount))
137   begin
138     call LCDPrint("", 1);
139     call LCDPrint("Amount: " + string(Amount), 1);
140     call LCDPrint("Drink ID: " + string(drinkID), 0);
141   end
142   else
143   begin
144     call LCDClear();
145     call LCDPrint("ERRROR", 0);
146   end
147   call delay(2000);
148   call StartMsg();
149   return nothing;
150 end
151
152 /*Implement the required pour function*/
153 function pour return nothing using(container cont, int Amount)
154 begin
155   call digitalWrite(cont, HIGH);
156   call delay(Amount*1000);
157   call digitalWrite(cont, LOW);
158   return nothing;
159 end
160
161 /*Pours the drink, and write message to the user*/
162 function readyToPour return nothing using(drink InputDrink, int
        drinksLeft)
163 begin
164   /*Call the pourDrink function implemented in SPLAD*/
```

```
165  call pourDrink(InputDrink);
166
167  /*Write messages to the user*/
168  call LCDPrint("Drinks left:", 0);
169  call LCDPrint(string(drinksLeft),1);
170  call delay(2000);
171  call LCDPrint("Drink is served", 0);
172  call LCDPrint("    by SPLAD    ", 1);
173  call delay(2000);
174  call StartMsg();
175  return nothing;
176 end
177
178 /*Called when a RFID with a valid Drink ID has been found*/
179 function DrinkFound return nothing using(int DrinkID, int Amount,
       drink DrinkToServe, string DrinkName)
180 begin
181  call LCDPrint("Confirm to make:", 0);
182  call LCDPrint(DrinkName, 1);
183  call delay(1000);
184  bool run <-- true;
185  while(run = true)
186  begin
187    /*If confirmed, try pouring drink*/
188    if(call digitalRead(ButtonConfirm) = HIGH)
189    begin
190      if(call RFIDWrite(DrinkID, Amount-1))
191      begin
192        call readyToPour(DrinkToServe, Amount-1);
193      end
194      else
195      begin
196        call LCDPrint("Error", 0);
197        call LCDPrint("writing to tag", 1);
198      end
199      run <-- false;
200    end
201    else if (call digitalRead(ButtonOne) = HIGH OR call digitalRead(
         ButtonTen) = HIGH)
202    begin
203      call LCDPrint("Drink making", 0);
204      call LCDPrint("cancelled",1);
205      run <-- false;
206    end
207  end
208  return nothing;
209 end
210
211 /*The action performed when a RFID is found during customermode*/
212 function CustomerAction return nothing using(int DrinkID, int
       Amount)
213 begin
214  /*If there balance on the RFID is high enough, find the drink*/
215  if(Amount > 0)
216  begin
217    /*Find the right drink, and call DrinkFound*/
218    switch(DrinkID)
219    begin
220      case 1:
221        call DrinkFound(DrinkID, Amount, TomCollins, "Tom Collins");
222        break;
223      case 10:
```

```
224         call DrinkFound(DrinkID, Amount, SweetCollins, "Sweet
                Collins");
225         break;
226       case 55:
227         call DrinkFound(DrinkID, Amount, BananaShake, "Banana Shake"
                );
228         break;
229       case 60:
230         call DrinkFound(DrinkID, Amount, Mojito, "Mojito");
231         break;
232       case 65:
233         call DrinkFound(DrinkID, Amount, AllTogether, "All Together"
                );
234         break;
235       case 101:
236         call DrinkFound(DrinkID, Amount, SoftwareShot, "Software
                Shot");
237         break;
238       default:
239         call LCDPrint("Error with RFID data", 0);
240         call LCDPrint("Contact staff", 1);
241         break;
242     end
243   end
244   else
245   begin
246     call LCDPrint("Your balance is", 0);
247     call LCDPrint("too low", 1);
248     call delay(3000);
249     call LCDClear();
250   end
251   return nothing;
252 end
253
254 /*The required function to implement. Called when a RFID is found.
       */
255 function RFIDFound return nothing using(int DrinkID, int Amount)
256 begin
257   if(IsStaffInput = true)
258   begin
259     call StaffAction();
260   end
261   else
262   begin
263     call CustomerAction(DrinkID, Amount);
264   end
265   return nothing;
266 end
267
268 /*Used to change mode between customer and staffmode*/
269 function ChangeMode return bool using(bool mode)
270 begin
271   string OutputString <-- "";
272   bool returnValue <-- false;
273   if(mode = true)
274   begin
275     OutputString <-- "staffmode";
276   end
277   else if(mode = false)
278   begin
279     OutputString <-- "customermode";
280   end
```

```
281
282  if(OutputString != "")
283  begin
284    call LCDPrint("Confirm change", 0);
285    call LCDPrint("to " + OutputString, 1);
286    call WaitOnInput();
287    if(call digitalRead(ButtonConfirm) = HIGH)
288    begin
289      returnValue <-- mode;
290      call LCDClear();
291      call LCDPrint(OutputString + " on", 0);
292      call delay(2000);
293      call StartMsg();
294    end
295    else
296    begin
297      call LCDPrint("Customer", 0);
298      call LCDPrint("mode on", 1);
299      call delay(2000);
300      call StartMsg();
301    end
302  end
303
304  return returnValue;
305 end
306
307 /* loop used to change from staffmode to customermode and visa
       versa*/
308 function loop return nothing using()
309 begin
310  if(call digitalRead(ButtonOne) = HIGH)
311  begin
312    IsStaffInput <-- call ChangeMode(true);
313  end
314  else if(call digitalRead(ButtonTen) = HIGH)
315  begin
316    IsStaffInput <-- call ChangeMode(false);
317  end
318  return nothing;
319 end
320
321 /*Setup to set the pinmode of the buttons*/
322 function setup return nothing using()
323 begin
324  call pinMode(ButtonOne, INPUT);
325  call pinMode(ButtonTen, INPUT);
326  call pinMode(ButtonConfirm, INPUT);
327  call StartMsg();
328  return nothing;
329 end
```

DVD - Source code C

The attached DVD contains:

- The source code of the compiler
- The compiled compiler
- The example code written in SPLAD
- The project report in PDF format.