SPLAD
P4 PROJEKT
GROUP SW407F13
SOFTWARE
DEPARTMENT OF COMPUTER SCIENCE
AALBORG UNIVERSITY
MAY 2013

AALBORG UNIVERSITY
STUDENT REPORT

Titel:

　　SPLAD - Special 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:
　　84

Project end:
　　29th of May, 2013

The content of the report is freely available, but can only be published (with source reference) with an agreement with the authors.

AALBORG UNIVERSITY

S T U D E N T　R E P O R T

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

Synopsis:

FiXme Fatal: synopsis mangler

Prolog

This report was 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 project. We are a group of students from the Department of Computer Science at Aalborg University (AAU). This report will document and describe the process of designing and implementing a compiler.

The references in the report will be in the format [example] 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 number of the figure or table in that chapter.

The disc included on the last page of the report contains the complete source code of the compiler and a PDF version of the rapport.

<div align="center">Aalborg May 16, 2013</div>

<div align="center">
Aleksander Sørensen Nilsson   Christian Jødal O'Keeffe

Kasper Plejdrup   Mette Thomsen Pedersen

Niels Brøndum Pedersen   Rasmus Fischer Gadensgaard
</div>

Contents

Introduction 1

The purpose of this report is to document the design and implementation of the programming language SPLAD 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 hobby programmer who will build a drink-mixer. The syntax of SPLAD will be specified in a way, so it has a high grade of readability. The semantics of SPLAD will be defined, and based on the semantics and syntax a compiler will be made. The compiler will compile the SPLAD program to the C-like programming language for Arduino, a single-board microcontroller.

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 to a party, where customers will buy RFID-tags [1], which contains information about 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 on the RFID-tag.

1.1.1   Solution in Bars

Currently bars and clubs often have multiple bartenders who mixes the drinks and serves the customers. The bartender handles both receiving the order, mixing the drink and handles the payment of the drink. This process can be done more efficiently. If a bar had a drinks-machine, the bar would require only one cashier instead of multiple bartenders. The cashier would handle selling and programming the RFID-tags. The customers themselves then places the RFID-tag on the RFID-reader on the drinks-machine, and the machine mixes the appropriate drink, and either decreases the count on the RFID-tag, or disables the tag, and display an appropriate message to the customer on the display of the machine. The RFID-tags are programmed by the cashier which encodes in the tag a drink id, and a drink count. This allows the customer to buy for example 10 cosmopolitans 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 use in many other systems, like a ice-cream machine, juice machine, or even a food dispenser in a restaurant to help with self-service. It can also be use in a cinema for regular customers to get popcorn, coke, or others forms of candy

---

[1] RFID is a system that uses radio waves to read different encoded tags. See section 3.2.1.2 for more information.

for the movies. There might even be a refill machine where the film is shown. There are many possibilities 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 chosen to examine how drink machine could be programmed using Arduino as platform for the processing. As mentioned in section 3.2.1.1, the programming language usually used for Arduino is based on C and C++, which is not aimed at programming drink machines as programming purpose. It could be useful to have a niche programming language aimed directly at programming drink machines on a Arduino platform. This will be the goal of this project.

The programming language in this project is aimed at the hobby programmer who wants to program his own drink 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 make sure programs are easily understandable. These trade-offs will be further discussed in section 2.3.3. A hobby 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.

Based on the above, the following problem statement comes to light:

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

The purpose of this problem statement is to guide the programming language for this project, so when the programming language reaches a final state, it is simple for hobby programmers to program 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 hobby programmers, the programming language should be specified in a way which is suited for the programmer.

- **How can a compiler be developed, which recognizes the language, and translates the source program into Arduino suitable code?** Of course it is not enough to have a simple-to-understand language, if it does not have a compiler for that language. The language would then render useless. 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 c code, 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, containing the design criteria as well as the syntax and semantics of SPLAD.

- In chapter 3 the implementation of SPLAD will be described, containing the different compiler phases and the code of the compiler.

FiXme Fatal: Ma
konklusion og aj
men det venter
til vi er nået læi

Language Specification 2

2.1   Paradigms of Programming Language

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 language [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 the if-statement. An example of a imperative programming language is C. Imperative programming language describes programs in terms of statements which alter the state of the program. This makes imperative languages simple, and are also a good starting point for new programmers.

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. In functional programming languages calls to a function 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. An example 2.1 is a piece of pseudo code where there could return a different result, if it is written in a functional programming paradigm or not.

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

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

The code 2.1 will always be true in functional programming paradigm, but in others paradigms such as the imperative programming paradigm, there can be a global variable that is used in the function "foo()" that can change on runtime.

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 are 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 which can be called on that object. For example the expression "foo.Equals(bar)", would call the Equals-method in the class of 'foo', and evaluate if 'bar' equals 'foo'. 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.

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.

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 understand for newcomers to programming. Also the programs in this programming language will likely remain of a relatively small length, which does not make object-orienting desirable.

2.2   Design Criteria

2.2.1   Programming an Arduino-based Drinks-mixer

To make it easier to program the machine described above, it would be nice to have a programming language aimed specifically at programming drinks-machines. This would make it easier for programmers with little or no experience to program the machine, and thereby making it easier for e.g. bar-owners to program and install their own drinks-machines in their bar. Therefore it has been decided to make a programming language aimed at this problem. The SPLAD language: **S**imple **P**rogramming **L**anguage for **A**rduino **D**rink-mixer. The SPLAD are 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 characteristics of a programming language, that will be used to evaluate trade-offs can be seen on 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 type	The presence of adequate facilities for defining data types and data structures.
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 have 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 characteristics are used to evaluate the the trade-offs of programming language. An overview of these can be seen on table 2.2.

Characteristic	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 characteristic is not affected by the feature.

Based on these trade-offs, it is clear that having a simple programming language affects 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 2.2 with listing 2.3, 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 2.2: Simple example of if-statement in C using the 'if'-keyword.

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

Listing 2.3: Simple example of if-statement in C without using the 'if'-keyword.

When defining the syntax of a programming language, it should balance these characteristics 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	low
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 does not have multiple ways of doing the same simple instructions. This is to make writing code simpler for beginners. An example is that it is not possible to increase a variable by 1 by writing "A++;". Instead it has to be written as "A<−A+1;".

- **Orthogonality:** It is not possible to make classes or constructs, this is done to simplify types for beginners. However it is possible to combine various other constructs, for example calling a function in an expression.

- **Data types:** The language of this project has five primitive types, bool, double, int, char and string and 2 special types, container and drink.

- **Syntax design:** It is not possible to make classes in the language of this project but there is both "for" and "while" loops because they are relatively simple to use and understand for beginners. All functions and loops start with "begin" and end with "end" to make it clear where they start and end.

- **Support for abstraction:** This is close to non-existent in the project languages, because to help beginners with using the language, a simple type system is simpler to understand.

- **Expressivity:** Shorthands have not been implemented in SPLAD, this is to increase the readability of the program, as described in listing 2.2.

- **Type checking:** 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, instead of at run time.

- **Exception handling:** Because the Arduino will execute the program-code in the end, the handling of exceptions such as array overflows, is left to the Arduino.

- **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

2.3.1 Grammartypes

The Chomsky hierarchy is a hierarchy of formal grammars. There are 4 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 or by a Pushdown 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

Context-sensitive grammars substitution rules have nearly the same rules as those used in Context-free grammar. But 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 sides 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 Grammar

By looking at the different types of grammars, it has become clearer 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) finite language defined by [Sipser, 2013]:

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). 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 a unambiguously grammar. A CFG is ambiguously if a string derived in the grammar has two or more different leftmost derivations [Sipser, 2013]. An unambiguously 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 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 parsing process 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 language that LL. LR parsers are bottom-up parsers meaning that they begin constructing the abstract tree from its leaf and works its way to the root. LR parsers are generally harder to implement by hand than LL parsers but there exists tools

which automatic 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].

Based on these understandings of grammars there will be a section were there will looked into which grammar that will be used in this project.

2.3.3 Choice of Grammar

The programmer, using the language of this project, could be a hobby 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 humans 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 ensuring readability - especially for the hobby programmer.

When declaring a function it has to be written on the form "function *functionname* return *type* using (*parameter(s)*) begin *statements* return *expresion* end". Where *functionname* 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. *Expresion* 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 2.4.

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

Listing 2.4: Example of function declaration in SPLAD.

To get a more continuity structure in 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 value was parsed. 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 (eg. an "if" statement). To combine logical operators 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 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 are:

- **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 with the machine.

- **A RFID-tag:** An RFID-tag will contain a drink ID and an amount of how many drinks there are left.

- **A RFID-RW:** An RFID-reader and writer for Arduino, to write and read the content of an RFID-tag.

- **A LCD:** For communicating with the user, an LCD is in most situations preferable.

- **Buttons:** For getting input from users, buttons are a possibility.

- **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 make a judgment of 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 have a name, and should have some sort of ID which can be stored on a RFID-tag. We should make a structure which can implement a drink type and assign the recipe to the drink. The structure should be following 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 2.5.

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

Listing 2.5: 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. 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 measure it actually is. The declaration of a drink will now be defined as seen on listing 2.6.

```
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 2.6: 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 more, 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 2.6, 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 2.7.

```
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 2.7: 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   An ingredient

With the drink type defined above, we should now focus on how to define an ingredient. 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 3.17.

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

Listing 2.8: The structure of how to declare a container

2.3.4.3   A 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 build into the language. This can be seen in section 2.3.4.4.

2.3.4.4   A 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, should 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 2.9.

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

Listing 2.9: An example of the call to RFIDWrite.

2.3.4.5   A 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 to 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 2.10.

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

Listing 2.10: An example of how to call the LCD functions.

2.3.4.6 Buttons

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 buttons functions to be build into the language. This is considered as an good idea, but because it is not essential for the project is will not be implemented, but it could be made if the language was developed further.

2.3.4.7 Mechanism for pouring ingredients

The mechanism to pour the 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 a 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.

2.3.5 The BNF of SPLAD

This section contains the BNF for the programming language of this project; SPLAD. 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:

2.3.5.1   Grammar - Drinks

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

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

⟨*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 2.3: Grammar related to drinks

2.3.5.3   Grammar - Assign

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

⟨*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 2.4: Grammar related to assignments

2.3.5.4   Grammar - Expressions

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

$\langle addsubend \rangle \rightarrow \langle plusminus \rangle\ \langle addsub \rangle$
  $\mid\ \ \varepsilon$

$\langle muldivend \rangle \rightarrow \langle timesdivide \rangle\ \langle muldiv \rangle$
  $\mid\ \ \varepsilon$

$\langle timesdivide \rangle \rightarrow\ *$
  $\mid\ \ /$

$\langle plusminusorempty \rangle \rightarrow\ \varepsilon$
  $\mid\ \ \langle plusminus \rangle$

$\langle plusminus \rangle \rightarrow\ \text{-}$
  $\mid\ \ +$

$\langle numericend \rangle \rightarrow\ \varepsilon$
  $\mid\ \ .\ \langle DIGIT \rangle$

$\langle string \rangle \rightarrow\ \langle STRINGTOKEN \rangle$

$\langle functioncall \rangle \rightarrow\ \text{call}\ \langle id \rangle\ (\langle callexpr \rangle)$

$\langle callexpr \rangle \rightarrow\ \langle subcallexpr \rangle$
  $\mid\ \ \varepsilon$

$\langle subcallexpr \rangle \rightarrow\ \langle expr \rangle\ \langle subcallexprend \rangle$

$\langle subcallexprend \rangle \rightarrow\ ,\ \langle subcallexpr \rangle$
  $\mid\ \ \varepsilon$

$\langle compend \rangle \rightarrow\ \langle comparisonoperator \rangle\ \langle comp \rangle$
  $\mid\ \ \varepsilon$

$\langle comparisonoperator \rangle \rightarrow\ \text{>}$
  $\mid\ \ \text{<}$
  $\mid\ \ \text{<=}$
  $\mid\ \ \text{>=}$
  $\mid\ \ \text{!=}$
  $\mid\ \ =$

$\langle termend \rangle \rightarrow\ \langle termsymbol \rangle\ \langle term \rangle$
  $\mid\ \ \varepsilon$

$\langle termsymbol \rangle \rightarrow\ \text{AND}$

$\langle exprend \rangle \rightarrow\ \langle exprsymbol \rangle\ \langle expr \rangle$
  $\mid\ \ \varepsilon$

$\langle exprsymbol \rangle \rightarrow\ \text{OR}$

Grammar 2.5: The grammar related to expressions

2.3.5.5   Grammar - Functions

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

$\langle cast \rangle \rightarrow \langle type \rangle\ (\langle expr \rangle)$

$\langle function \rangle \rightarrow$ function $\langle id \rangle$ return $\langle functionmid \rangle$

$\langle functionmid \rangle \rightarrow \langle type \rangle\ \langle functionend \rangle\ \langle expr \rangle$; end
$\quad |\quad$ nothing $\langle functionend \rangle$ nothing; end

$\langle functionend \rangle \rightarrow$ using $(\langle params \rangle)$ begin $\langle stmts \rangle$ return

$\langle params \rangle \rightarrow \langle subparams \rangle$
$\quad |\quad \varepsilon$

$\langle subparams \rangle \rightarrow \langle type \rangle\ \langle callid \rangle\ \langle subparamsend \rangle$

$\langle subparamsend \rangle \rightarrow\ ,\ \langle subparams \rangle$
$\quad |\quad \varepsilon$

$\langle stmts \rangle \rightarrow \varepsilon$
$\quad |\quad \langle stmt \rangle\ \langle stmts \rangle$

$\langle stmt \rangle \rightarrow \langle assign \rangle$;
$\quad |\quad \langle nontermif \rangle$
$\quad |\quad \langle nontermwhile \rangle$
$\quad |\quad \langle from \rangle$
$\quad |\quad \langle dcl \rangle$;
$\quad |\quad \langle functioncall \rangle$;
$\quad |\quad \langle nontermswitch \rangle$
$\quad |\quad \langle COMMENT \rangle$

Grammar 2.6: The grammar related to functions

2.3.5.6   Grammar - Loops

This section contains the grammar related to loops. This can be seen on 2.7

$\langle nontermif \rangle \rightarrow$ if($\langle expr \rangle$) $\langle block \rangle$ $\langle endif \rangle$

$\langle endif \rangle \rightarrow$ else $\langle nontermelse \rangle$
$\quad | \quad \varepsilon$

$\langle nontermelse \rangle \rightarrow \langle nontermif \rangle$
$\quad | \quad \langle block \rangle$

$\langle nontermwhile \rangle \rightarrow$ while($\langle expr \rangle$) $\langle block \rangle$

$\langle from \rangle \rightarrow$ from $\langle assign \rangle$ to $\langle expr \rangle$ step $\langle plusminusorempty \rangle$ $\langle DIGIT \rangle$ $\langle block \rangle$

$\langle block \rangle \rightarrow$ begin $\langle stmts \rangle$ end

$\langle nontermswitch \rangle \rightarrow$ switch ($\langle expr \rangle$) begin $\langle cases \rangle$ end

$\langle cases \rangle \rightarrow$ case $\langle expr \rangle$: $\langle stmts \rangle$ break; $\langle endcase \rangle$

$\langle endcase \rangle \rightarrow \langle cases \rangle$
$\quad | \quad$ default: $\langle stmts \rangle$ break;

Grammar 2.7: 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 2.8

$\langle STRINGTOKEN \rangle \rightarrow$ ” .*? ”

$\langle LETTER \rangle \rightarrow$ [a - zA - Z] $^+$

$\langle DIGIT \rangle \rightarrow$ [0 - 9] $^+$

$\langle NOTZERODIGIT \rangle \rightarrow$ [1-9][0-9] *

$\langle COMMENT \rangle \rightarrow$ /* .*? */

Grammar 2.8: The grammar for the lexicon

2.3.7 EBNF?

2.4 Semantics

In this section the semantics of SPLAD will be described.

2.4.1 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

FiXme Fatal: Sk med?

it is not declared in that block, but is still visible in that block (ex. global variables) [Sebesta, 2009]. The languages scope rules determine how a variable name is associated with a variable in a particular occurrence or when working with a functional language, it 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 program unit or block it is local for that part. Then the non-local variables are visible within the program unit or block if they are not declared there. Lastly there are global variables these are a special category of non-local variables.

2.4.1.1 Static Scope

Static scoping method was introduced in ALGOL 60, which is the method of binding names to non-local variables. There are two categories of static-scoped languages. First one is which sub-programs can be nested, this creates nested static scopes. And the other is static scopes which is also created by sub-programs but nested scopes are created only by nested class definitions and blocks.

Blocks 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 in code example 2.11. Before the block {} the variable x is initialized and set to integer 5, in the block, x is set to 10 and a extra variable y is initialized and is set to integer 15, y is only visible inside block, thereby it can not be call outside of the block, but after the block, x still have the value 10 that was given inside the block.

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

Listing 2.11: a simple code with 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.

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 make it visible to those functions.

2.4.1.5   The difference between dynamic and static scope

To help better understand the difference between dynamic and static Scope a larger code example is being shown here 2.12.

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

Listing 2.12: A simple code showing the difference between static and dynamic scoping. [msujaws, 2011]

The code 2.12 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 text input. The main reason for this, is that programs for the Arduino platform is mainly written in C, which also uses static scoping. This makes the compilation from SPLAD to C simpler for the compiler [Arduino, e]. 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].

Multiple Symbol Tables

In each scope, a symbol table exists, which is an ADT (Abstract Data Type), that stores identifier names and relate each identifier to its attributes. The general operations 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 this 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. A begin scope would simply push a symbol table scope to the stack, and when the scope ends, the symbol table is popped from the stack. This also accounts for nested scopes. But searching for a non-local variable would require searching the entire stack [Sebesta, 2009].

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 $O(lg(n))$. But the fact that programmers generally does 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.1. In the following text we use the following names to represent different syntactic categories.

- $n \in$ **Num** - Numerals

- v - Values

- $x \in$ **Var** - Variables

- $r \in$ **Arrays** - Array names

- $a \in Aexp$ - Arithmetic expression

- $b \in Bexp$ - Boolean expression

- $e \in A_{exp} \cup B_{exp}$ - expressions

- $C \in$ **Com** - Commands

2.4.5 Environment-Store Model

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

$$[\text{MUL}] \qquad \frac{env_E,\ sto \vdash a_1 \to_a v_1 \quad env_E,\ sto \vdash a_2 \to_a v_2}{env_E,\ sto \vdash a_1 * a_2 \to_a v}$$

where $v = v_1 * v_2$

Table 2.4: The transition rule for the arithmetic multiplication expression.

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

The following names represent the different environments.

- $env_V \in Env_V$ - Variable declarations

- $env_A \in Env_A$ - Array declarations

- $env_C \in Env_C$ - Constant declarations

- $env_E \in Env_E$ - Expressions declarations

2.4.6 Arithmetic Expressions

The transition rule for multiplication in SPLAD can be seen on table 2.4. The rule states, that if a_1 evaluates to v_1 and a_2 evaluates to v_2, using any of the rules from A_{exp}, then $a_1 * a_2$ evaluates to v where $v = v_1 * v_2$.

2.4.7 Boolean Expressions

The transition rule for logical-or in SPLAD can be seen on table 2.5. The rules have to 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 B_{exp} then the expression $b_1 OR b_2$ evaluates to *true*. [OR-FALSE] states that if both b_1 and b_2 evaluates to *false* then the expression $b_1 OR b_2$ evaluates to *false*.

$$[\text{OR-TRUE}] \qquad \frac{env_E,\ sto \vdash b_1 \vee b_2 \to_b \text{true}}{env_E,\ sto \vdash b_1 \text{ OR } b_2 \to_b \text{true}}$$

$$[\text{OR-FALSE}] \qquad \frac{env_E,\ sto \vdash b_1 \wedge b_2 \to_b \text{false}}{env_E,\ sto \vdash b_1 \text{ OR } b_2 \to_b \text{false}}$$

Table 2.5: Transition rule for the boolean expression logical-or.

2.4.8 Declarations

2.4.9 Assignments

FiXme Fatal: Ka
laves siden at tr
rules til dette pu
er lavet endnu

The transition rule for variable assignment in SPLAD can be seen on table 2.6. When a variable is assigned the contents of l is updated to v, where l is the location of x found in the env_V and v is the result of evaluation e.

$$[\text{VAR-ASS}] \qquad env_C, \vdash \langle x = e,\ sto \rangle \to sto[l \mapsto v]$$

where $env_C,\ sto \vdash e \to_e v$
and $env_V\ x = l$

Table 2.6: Transition rule for variable assignment.

2.4.10   Commands

The transition rule for the while statement in SPLAD can be seen on table 2.7. The rule have to parts: [WHL-TRUE] and [WHL-FALSE]. If the condition b evaluates to *true* then the [WHL-TRUE] states that C will be executed which will update the *store* (sto) and again call the expression and evaluate the new b. If the condition b evaluates to *false* then C is <u>not</u> executed and the *store* is not updated. The program exits the while statement. [Hüttel, 2010]

[WHL-TRUE] $$\dfrac{env_C \vdash \langle C,\, sto \rangle \to sto''\ env_C \vdash \langle \mathbf{while}(b)\ \text{begin}\ C\ \text{end},\, sto'' \rangle \to sto'}{env_C \vdash \langle \mathbf{while}(b)\ \text{begin}\ C\ \text{end},\, sto \rangle \to sto'}$$

if $env_C,\ sto \vdash b \to_b$ true

[WHL-FALSE] $env_C \vdash \langle \mathbf{while}(b)\ \text{begin}\ C\ \text{end},\, sto \rangle \to sto$

if $env_C,\ sto \vdash b \to_b$ false

Table 2.7: Transition rules for the while statement.

2.4.11   Informal Type Rules

This section contains the type rules for the comparison operator. In the type rules, E is an expression and C 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 or string.

- 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 and E_2 does not have the value of zero.

- Type rule for $<-$ (assign):

  "E_1 $<-$ E_2" is type correct if 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 C end" is type correct if E is of type boolean and C is type correct.

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

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

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

  "if(E) begin C end" is type correct if E is type correct and of type boolean, and C is type correct.

- Here the type rules for switch/case will be described:

  "switch (E) begin case E_1: C_1 break; ... case E_n: C_n break; default: C_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 $C_1...C_n$ and C_d are type correct.

2.4.12 Formal Type Rules

This section will follow up on the informal description of our type rules. Expressions can be seen on table 2.8, declarations can be seen on table 2.9 and statements can be seen on table 2.10.

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

$$[ADD_{EXP}] \qquad \frac{E \vdash e_1 : T \;\; E \vdash e_2 : T}{E \vdash e_1 + e_2 : T} \qquad\qquad [VAR_{EXP}] \quad \frac{E(x) = T}{E \vdash x : T}$$

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

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

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

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

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

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

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

Table 2.8: Type rules for expressions.

Continued on the next page

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

$$[FUNC_{DEC}] \quad \frac{E[f \mapsto (x : T \to \text{ok})] \vdash D_F : \text{ok}}{E \vdash \text{func } f(T \ x) \ is \ S, D_F : \text{ok}}$$

Table 2.9: Type rules for declarations.

Fatal: Husk at
op på function
ons parametere,
vist være flere

$$[WHILE_{STM}] \quad \frac{E \vdash e : \text{Bool} \quad E \vdash S : \text{ok}}{E \vdash \text{while } e \text{ do } S : \text{ok}}$$

$$[FROM_{STM}] \quad \frac{E \vdash e_1 : \text{Int} \quad E \vdash e_2 : \text{Int} \quad E \vdash e_3 : \text{Int} \quad E \vdash S : \text{ok}}{E \vdash \text{from } e_1 \text{ to } e_2 \text{ step } e_3 \text{ do } S : \text{ok}}$$

$$[IF_{STM}] \quad \frac{E \vdash e : \text{Bool} \quad E \vdash S_1 : ok \quad E \vdash S_2 : ok}{if(e) \text{ then } S_1 \text{ else } S_2 : \text{ok}}$$

$$[SWITCH_{STM}] \quad \frac{E \vdash x : T \quad E \vdash E_1 : T \dots \quad E \vdash E_k : T \quad E \vdash S_1 : \text{ok} \dots \quad E \vdash S_k : \text{ok} \quad E \vdash S : \text{ok}}{\text{switch } (x) \text{ begin case } E_1 : \ S_1 \text{ break;} \dots \text{ case } E_k : \ S_k \text{ break; default: } S \text{ break; end : ok}}$$

$$[ASS_{STM}] \quad \frac{E \vdash x : T \quad E \vdash a : T}{E \vdash x < - - a : \text{ok}}$$

$$[COMP_{STM}] \quad \frac{E \vdash S_1 : \text{ok} \quad E \vdash S_2 : \text{ok}}{E \vdash S_1; S_2 : \text{ok}}$$

$$[BLOCK_{STM}] \quad \frac{E \vdash D_V : \text{ok} \quad E_1 \vdash D_F : \text{ok} \quad E_2 \vdash S : \text{ok}}{E \vdash \text{begin } D_V \ D_F \ S \text{ end : ok}}$$
where $E_1 = E(D_V, E)$
and $E_2 = E(D_F, E_1)$

$$[CALL_{STM}] \quad \frac{E \vdash f : (\vec{x} : \vec{T} \to \text{ok}) \quad E \vdash \vec{e} : \vec{T}}{E \vdash \text{call } f(\vec{e}) : ok}$$
where $x_i : T_i$ and $e_i : T_i$
for all $1 < i < |\vec{e}|$

ELLER:

$$[CALL_{STM}] \quad \frac{E \vdash f : (\vec{x} : \vec{T}) \quad E \vdash \vec{e} : \vec{T}}{E \vdash \text{call } f(\vec{e}) : ok}$$
where $x_i : T_i \to \text{ok}$ and $e_i : T_i$
for all $1 < i < |\vec{e}|$

Table 2.10: Type rules for statements.

2.5 Code Examples

Implementation 3

3.1 Design Criteria

3.2 Architecture

In this section, we will look at the hardware components used in this project. The Arduino is described together with the RFID and other components in section 3.2.1. After that section, an overview of the compiler will be presented in section 3.2.2. Then the language processing strategy will be presented in section 3.2.3, and the compilation passes in section 3.2.4. Then the abstract syntax tree will be described in section 3.2.5. At last the visitor pattern used in this project will be described in section **??**.

3.2.1 Hardware

This section will be 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.

3.2.1.1 Hardware platform

Arduino UNO is a powerful micro controller board that provides the user with ways to communicate with other components such as LCDs, diodes, 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 product 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 regular USB 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 found 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-port and jack socket. The board can be hooked up with a USB cable or an AC-to-DC (Alternating Current to Direct Current) adapter through the jack socket to power the unit. Arduino UNO operates at 5v (volts) 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 brick. 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.

3.2.1.2 RFID

To administrate the users collection of purchased drinks the plan is to store the number and the kind of drinks on an RFID tag that the customer then use at the drink 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, food, documents, pets, packaging and a lot of other kinds objects. All tags contain a unique ID that can in no way be changed once made. 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* have 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 1 meter [Specialisten], and that is far enough for this project. We are using a *read/write* tag to store drink-ids and drink-counts on the tags and make the reader read and respond to the information.

3.2.1.3 Other components

The demonstration of the product will require something to illustrate more advanced parts of a theoretical machine. This is because making a whole drink-mixer just to show that the project product works, will take too much time that instead could have been used to make the product better. The plan is to use LEDs (light emitting diode) to illustrate the different function of the machine, when they are active or inactive. The LED is made of a semiconductor which produces a 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 will be used as input this will allow interaction with the program at runtime. The switches will illustrate a more advanced control unit but in the project switches will be sufficient.

3.2.2 Overview of the Compiler

Figure 3.1 shows an abstract overview of each of the different phases in the compiler, what each phase requires as input, and what each step returns to the next phase.
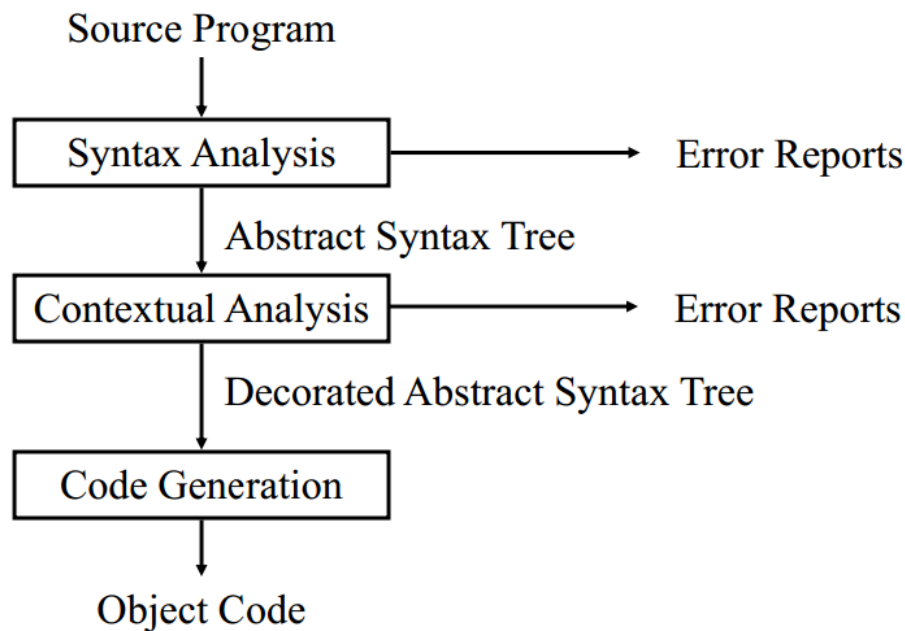
Figure 3.1: This is an abstract overview of how the compiler is structured. The figure is from [Bent Thomsen, 2013].

A compiler is a fundamental part of modern computing. Their job is to translate programming language into machine language. A compiler allows programs to make a virtual computer to ignore the machine-dependency details of machine language and therefor be portable across different computers [Fischer et al., 2009].

A compiler consists of 3 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, the contextual analysis to the contextual constraints and the code generation phase roughly corresponds to the semantics.

Given a simple compiler it will go through more than three phases. This can be seen on 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 uses 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 decorate the AST, and translates this into the target language.

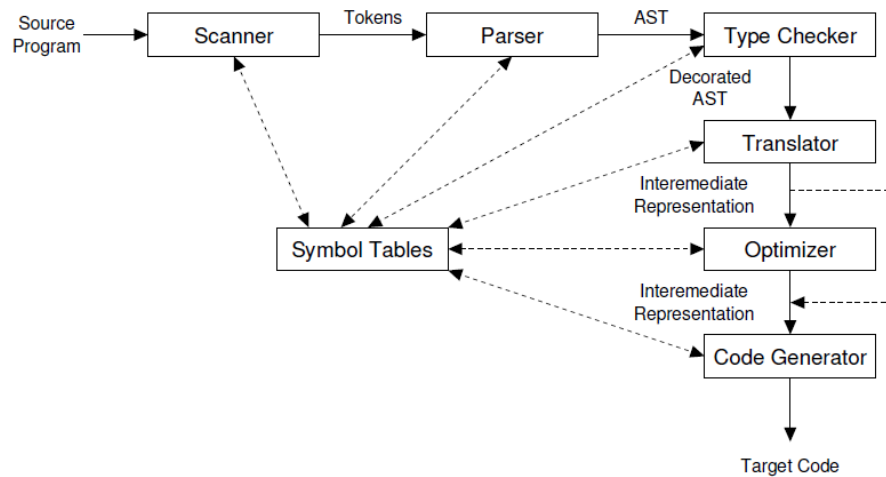The different phases will be described more thorough later in the rapport.

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

3.2.3   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 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.
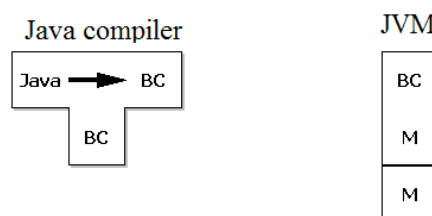
Figure 3.3: Tombstone diagram for the Java-compiler

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 mentioned, the SPLAD-compiler is written in Java, and is therefore compiled into Java byte-code using the Java-compiler. As it can be seen in figure 3.4, 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.
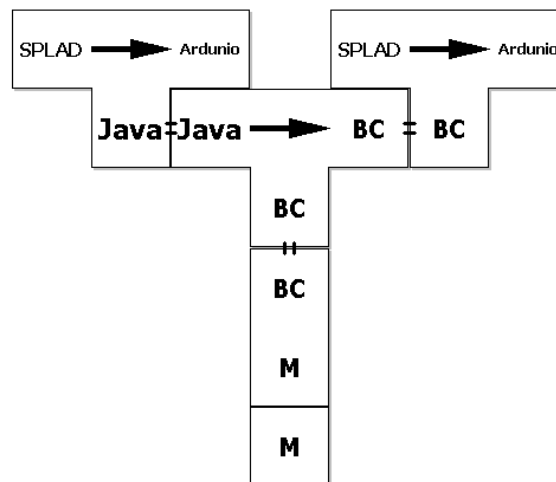
Figure 3.4: Tombstone diagram for the SPLAD-compiler compilation process

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.
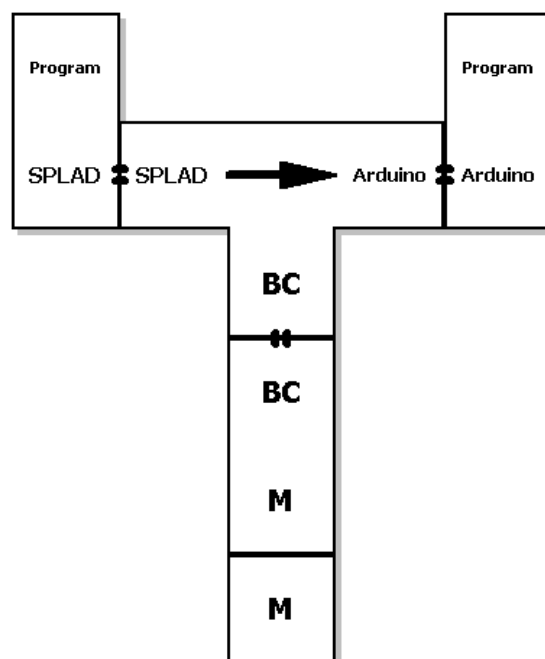
Figure 3.5: Tombstone diagram for the SPLAD-compiler compilation process

3.2.4   Compilation Passes

3.2.5   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.

FiXme Fatal: Ek på hvordan pars virker, og hvorda bliver bygget

3.2.6   Parse Tree

A parse tree are nearly the same as a 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 different between a abstract syntax tree seen on figure 3.6 and a parse tree seen on figure 3.7, two trees are made, one of each, over 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.
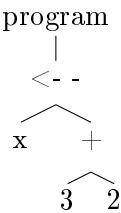
Figure 3.6: A abstract syntax tree
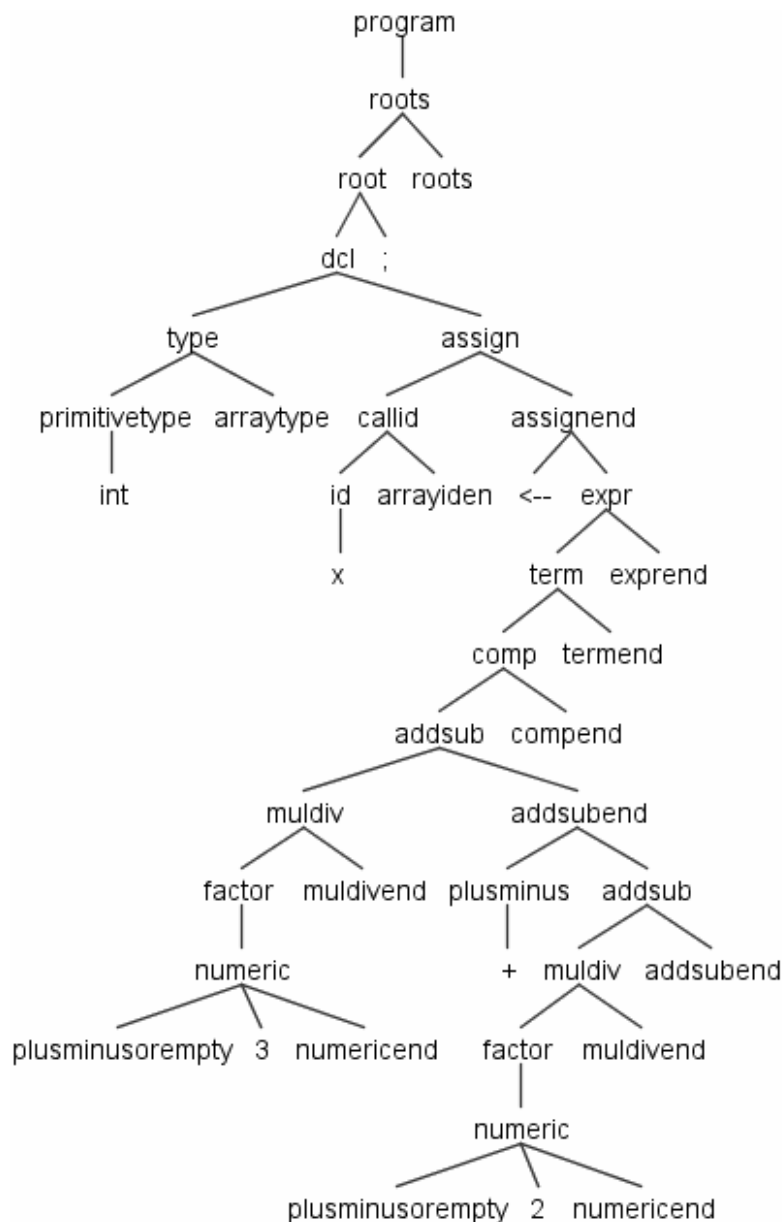
Figure 3.7: A parse tree that is made by using ANTLR.

3.2.7   The Visitor Pattern

3.3   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. The syntactic analyzer read 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.

The parser and lexer in the SPLAD-compiler is generated by ANTLR [Antlr], this can be seen on listing 3.14. The first function to be used is the lexer, which can be seen on line 8. Here it can be seen that the lexer needs the program code. Next, The tokens are generated, this can be seen on line 9. It is clear that the token generator needs the output

generated by the lexer when creating the tokens. When making the parser the tokens are needed, this can be seen on line 10. The tokens can be seen in section 3.6.

3.3.1   Known lexers and parsers

In this section some of the different lexers and parsers, that are available on the internet, will be described.

3.3.1.1   Lexer

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].

3.3.1.2   Parser

Parsertools generates 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].

3.3.1.3   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].

3.3.1.4 Comparison Tables

On table 3.1 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 3.1: Comparison between the different lexical analyzers.

Based on the different lexers and parsers attributes (seen on 3.2), 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 3.2: Comparison between the different parsers and lexer-parsers.

3.3.2   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 3.2: 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 3.2, to show how a program is parsed and represented using the GUI option.
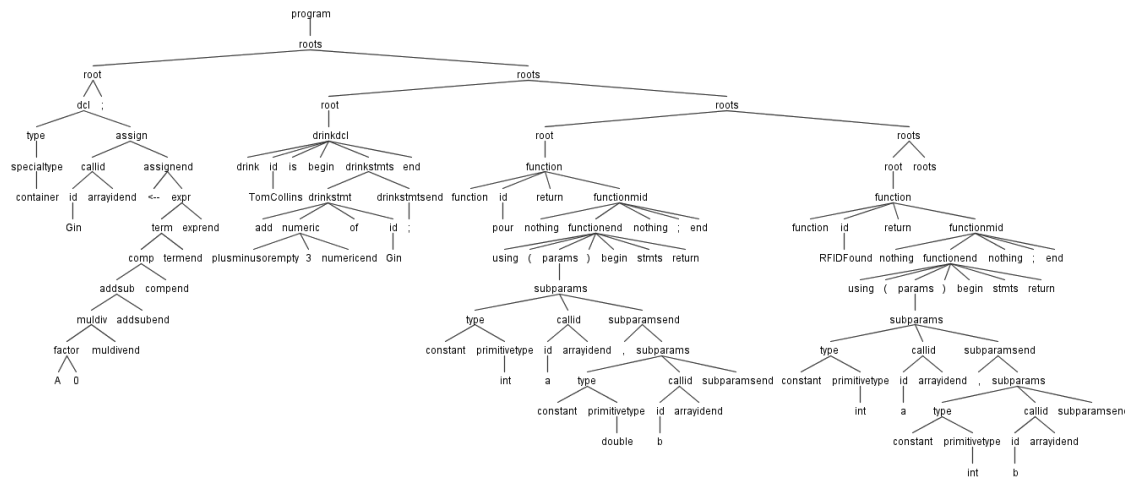
Figure 3.8: On this figure the parse tree for the parsed program "Hello world" can be seen.

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 3.2.6 for more about parse trees.

3.3.3 Lexical Analyzer

A lexical analyzer 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 3.3. 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 3.4 has the token type IDENT and the value 'c'.

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

Table 3.3: 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 3.3: Simple example of code.

As an example the lines of code seen on listing 3.3 might be read as the tokens seen on table 3.4.

Token	Lexeme
IDENT	c
ASSIGN	=
DIGIT	42
SEMICOLON	;

Table 3.4: 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].

3.3.3.1   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 3.5.

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 3.5: 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 3.6.

PRIMITIVETYPE	'int' | 'double' | 'bool' | 'char' | 'container' | 'string'
STRINGTOKEN	" ... "
DIGIT	$[0-9]^+$
NOTZERODIGIT	$[1-9][0-9]^*$
LETTER	$[A-Za-z]^+$
COMMENT	/* ... */
WHITESPACE	\r | \n | \t
OTHER	ε

Table 3.6: The tokens in SPLAD.

atal: Den passer
vedet ikke, og er
te mærkeligt det
mmer to gange?

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 3.3.1.

3.3.4   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.

Fatal: Den laver
åde parsetræ og
Der mangler at
krevet forskellen
arsetræ og AST

3.3.4.1   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.

3.3.4.2   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.

3.4   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.

3.4.1   Scope Checking

3.4.2   Type Checking

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

3.4.2.1   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 3.4: Value type

This class, as can be seen on listing 3.4, 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. This 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 3.5: How to evaluate Value

When we evaluate the type value we use functions like those seen on listing 3.5 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.

3.4.2.2   Functions

We have made a class "functions" as seen on listing 3.6 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 3.6: 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 3.7

```
 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 3.7: 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 3.8 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 3.8: Predefining function PourDrink

3.4.2.3   Variables

In our language "INPUT" and "OUTPUT" would be seen as variables and by default be undefined so we predefined them like seen on listing 3.9. 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 3.9: Predefining OUTPUT

We have made a "variable" class to be make it easier to store and retrieve the necessary data. This class can be seen on listing 3.10. 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 3.10: Variable class

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.

3.4.2.4 Drink

Our special type "drink" has its own separate declaration method, the initial part can be seen on listing 3.11. 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 3.11: Drink declaration

Whether or not its 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.

3.4.2.5 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 3.12.

```
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 3.12: Int Error

Our "error" class contains a switch case an excerpt of this can be seen on listing 3.13. 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 3.13: Switch case error handling

3.5 Code Generation

The idea by 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
9   SPLADLexer lexer = new SPLADLexer(program);
10  CommonTokenStream tokens = new CommonTokenStream(lexer);
11  SPLADParser parser = new SPLADParser(tokens);
12
13  parser.setBuildParseTree(true);
14  return parser;
15 }
```

Listing 3.14: 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 3.14.

Before any of the actual code generation can begin we have to check scope rules, see section ??, and type check the program, see section 3.4.2. 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 3.15: 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 3.15. 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 3.16: 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 3.16. 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 if (ListOfContainers.size() != 0){
2   ContentBuffer.append("String ContainersnameSW407F13[" +
        ListOfContainers.size() + "];\n");
3   ContentBuffer.append("int ContainerspinSW407F13[" +
        ListOfContainers.size() + "];\n");
4 }
5
6 //Add content to the arrays in the setupfirstbuffer.
7 for (int i = 0; i < ListOfContainers.size(); i++){
8   setupfirstbuffer.append("ContainersnameSW407F13[" + i + "] = \""
        + ListOfContainers.get(i).containername + "\";\n");
9   setupfirstbuffer.append("ContainerspinSW407F13[" + i + "] = " +
        ListOfContainers.get(i).pinid + ";\n");
10 }
11
12 //Add the content of the program to the ContentBuffer
13 ContentBuffer.append(HeaderBuffer);
```

Listing 3.17: 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 3.17.

```
 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 3.18: 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 3.18.

```
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 3.19: 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 3.19.

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

3.5.1 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.

3.5.1.1 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 3.20, 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 3.20: 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.

3.5.1.2 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 3.21. 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 3.21: 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 3.22. 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 3.22: 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.

3.6   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 3.2.1.

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 3.9.

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.
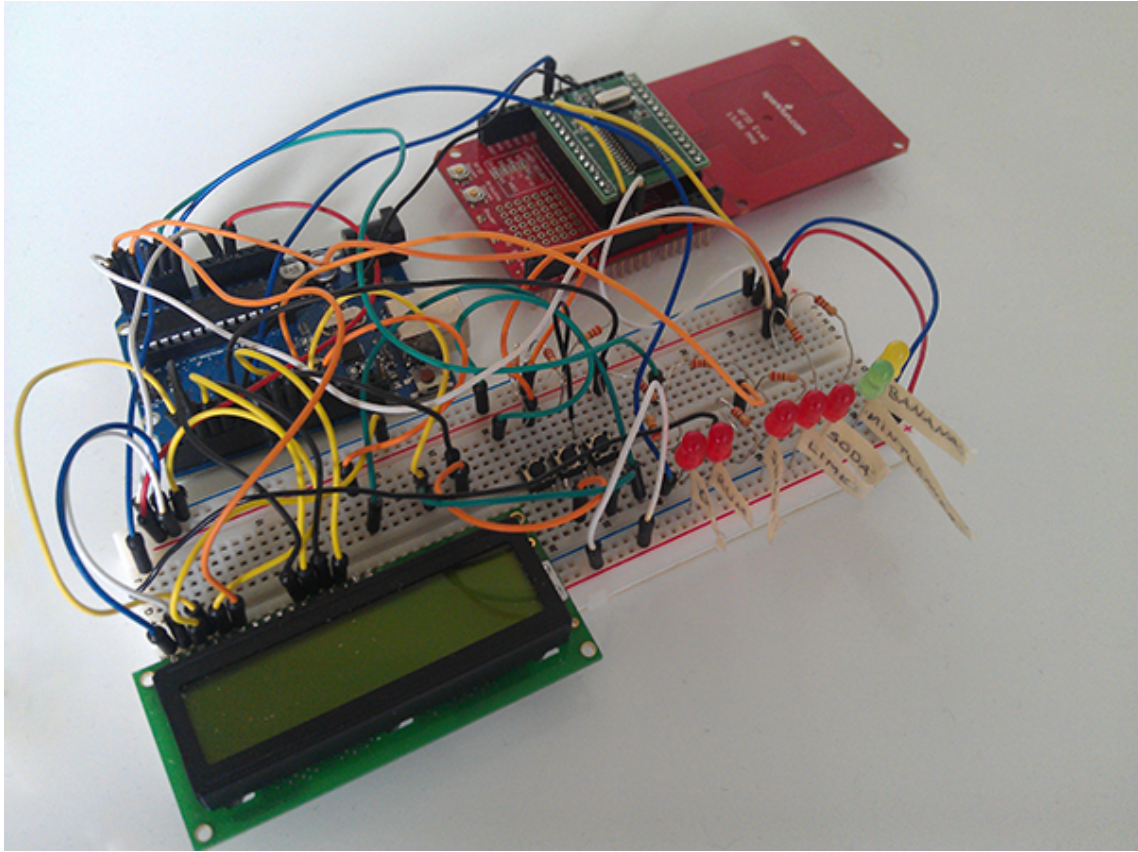
Figure 3.9: In this figure a illustrative resemble of a drink mixer can be seen.

The components are wired up to the Arduino board. The whole construction can be seen in figure 3.9.

Conclusion 4

FiXme Fatal: ko
mangler

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].

Arduino, **e**. Arduino. *Arduino Build Process.* `http://arduino.cc/en/Hacking/BuildProcess` [Last seen: 2013/03/22].

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].

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.

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

Semantics A

A.1 Transition Rules

[VAR-ASS] $\qquad env_C, \vdash \langle x = e,\ sto \rangle \to sto[l \mapsto v]$

$\qquad$ where $env_C,\ sto \vdash e \to_e v$
$\qquad$ and $env_V\ x = l$

[ARR-ASS] $\qquad env_C \vdash \langle r[a] = e,\ sto \rangle \to sto[l \mapsto v_2]$

$\qquad$ where $env_C,\ sto \vdash a \to_a v_1$
$\qquad$ and $env_C,\ sto \vdash e \to_e v_2$
$\qquad$ and $env_A\ r[v_1] = l$

[IF-TRUE] $\qquad \dfrac{env_C \vdash \langle C,\ sto \rangle \to sto'}{env_C\ \langle \mathbf{if}(b)\ \mathbf{begin}\ C\ \mathbf{end},\ sto \rangle \to sto'}$

$\qquad$ if $env_C,\ sto \vdash b \to_b$ true

[IF-FALSE] $\qquad env_C \vdash \langle \mathbf{if}(b)\ \mathbf{begin}\ C\ \mathbf{end},\ sto \rangle \to sto$

$\qquad$ if $env_C,\ sto \vdash b \to_b$ false

[IF-ELSE-TRUE] $\qquad \dfrac{env_C \vdash \langle C_1,\ sto \rangle \to sto'}{env_C \vdash \langle \mathbf{if}(b)\ \mathbf{begin}\ C_1\ \mathbf{end},\ \mathbf{else}\ \mathbf{begin}\ C_2\ \mathbf{end},\ sto \rangle \to sto'}$

$\qquad$ if $env_C,\ sto \vdash b \to_b$ true

[IF-ELSE-FALSE] $\qquad \dfrac{env_C \vdash \langle C_2,\ sto \rangle \to sto'}{env_C \vdash \langle \mathbf{if}(b)\ \mathbf{begin}\ C_1\ \mathbf{end},\ \mathbf{else}\ \mathbf{begin}\ C_2\ \mathbf{end},\ sto \rangle \to sto'}$

$\qquad$ if $env_C,\ sto \vdash b \to_b$ false

[WHL-TRUE] $\qquad \dfrac{env_C \vdash \langle C,\ sto \rangle \to sto''\ env_c \vdash \langle \mathbf{while}(b)\ \mathbf{begin}\ C\ \mathbf{end},\ sto'' \rangle \to sto'}{env_C \vdash \langle \mathbf{while}(b)\ \mathbf{begin}\ C\ \mathbf{end},\ sto \rangle \to sto'}$

$\qquad$ if $env_C,\ sto \vdash b \to_b$ true

[WHL-FALSE] $$env_C \vdash \langle \mathbf{while}(b) \text{ begin } C \text{ end}, sto \rangle \rightarrow sto$$

$$\text{if } env_C, \ sto \vdash b \rightarrow_b \text{false}$$

<div align="center">Table A.1: Commands</div>

[EQL-TRUE] $$\frac{env_E, \ sto \vdash e_1 \rightarrow_e v_1 \quad env_E, \ sto \vdash e_2 \rightarrow_e v_2}{env_E, \ sto \vdash e_1 = e_2 \rightarrow_b \text{true}}$$

$$\text{if } v_1 = v_2$$

[EQL-FALSE] $$\frac{env_E, \ sto \vdash e_1 \rightarrow_e v_1 \quad env_E, \ sto \vdash e_2 \rightarrow_e v_2}{env_E, \ sto \vdash e_1 = e_2 \rightarrow_b \text{false}}$$

$$\text{if } v_1 \neq v_2$$

[NEQ-TRUE] $$\frac{env_E, \ sto \vdash e_1 \rightarrow_e v_1 \quad env_E, \ sto \vdash e_2 \rightarrow_e v_2}{env_E, \ sto \vdash e_1! = e_2 \rightarrow_b \text{true}}$$

$$\text{if } v_1 \neq v_2$$

[NEQ-FALSE] $$\frac{env_E, \ sto \vdash e_1 \rightarrow_e v_1 \quad env_E, \ sto \vdash e_2 \rightarrow_e v_2}{env_E, \ sto \vdash e_1! = e_2 \rightarrow_b \text{false}}$$

$$\text{if } v_1 = v_2$$

[GRT-TRUE] $$\frac{env_E, \ sto \vdash e_1 \rightarrow_e v_1 \quad env_E, \ sto \vdash e_2 \rightarrow_e v_2}{env_E, \ sto \vdash e_1 > e_2 \rightarrow_b \text{true}}$$

$$\text{if } v_1 > v_2$$

[GRT-FALSE] $$\frac{env_E, \ sto \vdash e_1 \rightarrow_e v_1 \quad env_E, \ sto \vdash e_2 \rightarrow_e v_2}{env_E, \ sto \vdash e_1 > e_2 \rightarrow_b \text{false}}$$

$$\text{if } v_1 \leq v_2$$

[GEQ-TRUE] $$\frac{env_E, \ sto \vdash e_1 \rightarrow_e v_1 \quad env_E, \ sto \vdash e_2 \rightarrow_e v_2}{env_E, \ sto \vdash e_1 >= e_2 \rightarrow_b \text{true}}$$

$$\text{if } v_1 \geq v_2$$

[GEQ-FALSE] $$\frac{env_E, \ sto \vdash e_1 \rightarrow_e v_1 \quad env_E, \ sto \vdash e_2 \rightarrow_e v_2}{env_E, \ sto \vdash e_1 >= e_2 \rightarrow_b \text{false}}$$

$$\text{if } v_1 < v_2$$

[LES-TRUE] $$\frac{env_E, \ sto \vdash e_1 \rightarrow_e v_1 \quad env_E, \ sto \vdash e_2 \rightarrow_e v_2}{env_E, \ sto \vdash e_1 < e_2 \rightarrow_b \text{true}}$$

$$\text{if } v_1 < v_2$$

<div align="center">*Continued on the next page*</div>

[LES-FALSE]
$$\frac{env_E, \; sto \vdash e_1 \rightarrow_e v_1 \quad env_E, \; sto \vdash e_2 \rightarrow_e v_2}{env_E, \; sto \vdash e_1 < e_2 \rightarrow_b \text{false}}$$

if $v_1 \geq v_2$

[LEQ-TRUE]
$$\frac{env_E, \; sto \vdash e_1 \rightarrow_e v_1 \quad env_E, \; sto \vdash e_2 \rightarrow_e v_2}{env_E, \; sto \vdash e_1 <= e_2 \rightarrow_b \text{true}}$$

if $v_1 \leq v_2$

[GEQ-FALSE]
$$\frac{env_E, \; sto \vdash e_1 \rightarrow_e v_1 \quad env_E, \; sto \vdash e_2 \rightarrow_e v_2}{env_E, \; sto \vdash e_1 <= e_2 \rightarrow_b \text{false}}$$

if $v_1 > v_2$

[NOT-TRUE]
$$\frac{env_E, \; sto \vdash b \rightarrow_b \text{true}}{env_E, \; sto \vdash !(b) \rightarrow_b \text{false}}$$

[NOT-FALSE]
$$\frac{env_E, \; sto \vdash b \rightarrow_b \text{false}}{env_E, \; sto \vdash !(b) \rightarrow_b \text{true}}$$

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

[AND-FALSE]
$$\frac{env_E, \; sto \vdash b_1 \vee b_2 \rightarrow_b \text{false}}{env_E, \; sto \vdash b_1 \text{ AND } b_2 \rightarrow_b \text{false}}$$

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

[OR-FALSE]
$$\frac{env_E, \; sto \vdash b_1 \wedge b_2 \rightarrow_b \text{false}}{env_E, \; sto \vdash b_1 \text{ OR } b_2 \rightarrow_b \text{false}}$$

[PAR]
$$\frac{env_E, \; sto \vdash b_1 \rightarrow_b v}{env_E, \; sto \vdash (b_1) \rightarrow_b v}$$

Table A.2: Boolean expressions

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

where $v = v_1 + v_2$

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

where $v = v_1 - v_2$

[MUL]
$$\frac{env_E, \; sto \vdash a_1 \rightarrow_a v_1 \quad env_E, \; sto \vdash a_2 \rightarrow_a v_2}{env_E, \; sto \vdash a_1 * a_2 \rightarrow_a v}$$

where $v = v_1 * v_2$

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

where $v = \frac{v_1}{v_2}$

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

where $v = \frac{v_1}{v_2}$

[NUM] $env_E, \; sto \vdash n \rightarrow_a v$

if $\mathcal{N}[n] = v$
where $\mathcal{N} : \textbf{Num} \rightarrow \mathbb{R}$

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

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

[ARR] $env_A, \; sto \vdash r[a_1] \rightarrow_a v_2$

if $env_A \; r[v_1] = l$ and $sto \; l = v_2$
where $a_1 \rightarrow_a v_1$

Table A.3: Aritmethic expressions

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
```