SPLAD

P4 projekt

Group SW407F13

Software

Department of Computer Science
Aalborg University

May 2013

AALBORG UNIVERSITY
STUDENT REPORT

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

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:
44

Project end:
29th of May, 2013

Synopsis:

FiXme Fatal: synopsis mangler

Prolog

Aalborg April 19, 2013

FiXme Fatal: pro
mangler

Aleksander Sørensen Nilsson

Christian Jødal O'Keeffe

Kasper Plejdrup

Mette Thomsen Pedersen

Niels Brøndum Pedersen

Rasmus Fischer Gadensgaard

Contents

Introduction 1

FiXme Fatal: inc mangler

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, which contains information about a specific drink. These RFID-tags can be read by the drinks-machine by 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 like the one described above, 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 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 and 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 an 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

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.

```
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 will be 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
Write-ability	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
Orthogonality	A relatively small set of primitive constructs can be combined legally in a relatively small number of ways
Uniformity	If some features are similar they should look and behave similar
Maintainability	Errors can be found and corrected and new features can be added easily
Generality	Avoid special cases in the availability or use of constructs and by combining closely related constructs into a single more general one
Extensibility	Provide some general mechanism for the programmer to add new constructs to a language
Standardability	Allow programs to be transported from one computer to another without significant change in language structure
Implementability	Ensure a translator or interpreter can be written

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, writability and reliability. This is because having a very simple-to-understand language, might not make it very writable. 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.

```c
if (x > y)
{
    res = 1;
}
else
{
    res = 0;
}
```

Listing 2.2: Simple example of if-statement in C using the 'if'-keyword.

```c
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.

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

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 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 were the function can call other functions, declare variables, calculate and assign values. *Expresion* is were 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.

```
function DoSomething return int using (int x)
begin
  x <-- x + 1;
  return x;
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 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:

$\langle program \rangle \rightarrow \langle roots \rangle$

$\langle roots \rangle \rightarrow \varepsilon$
$\quad | \quad \langle root \rangle \; \langle roots \rangle$

$\langle root \rangle \rightarrow \langle dcl \rangle;$
$\quad | \quad \langle callid \rangle \; \langle assign \rangle;$
$\quad | \quad \langle function \rangle$
$\quad | \quad \langle COMMENT \rangle$

$\langle dcl \rangle \rightarrow \langle type \rangle\ \langle id \rangle\ \langle dclend \rangle$

$\langle type \rangle \rightarrow \langle primitivetype \rangle\ \langle arraytype \rangle$

$\langle primitivetype \rangle \rightarrow$ bool
| double
| int
| char
| container
| string

$\langle arraytype \rangle \rightarrow [\langle NOTZERODIGIT \rangle]$
| ε

$\langle id \rangle \rightarrow \langle LETTER \rangle$

$\langle dclend \rangle \rightarrow \varepsilon$
| $\langle assign \rangle$

$\langle assign \rangle \rightarrow$ `<--` $\langle expr \rangle$

$\langle expr \rangle \rightarrow \langle term \rangle\ \langle exprend \rangle$

$\langle term \rangle \rightarrow \langle comp \rangle\ \langle termend \rangle$

$\langle comp \rangle \rightarrow \langle factor \rangle\ \langle compend \rangle$

$\langle factor \rangle \rightarrow (\ \langle expr \rangle\ )$
| $!(\langle expr \rangle)$
| $\langle callid \rangle$
| $\langle numeric \rangle$
| $\langle string \rangle$
| $\langle functioncall \rangle$
| $\langle cast \rangle$
| LOW
| HIGH
| true
| false

$\langle callid \rangle \rightarrow \langle id \rangle\ \langle arrayiden \rangle$

$\langle arrayiden \rangle \rightarrow \langle ARRAYCALL \rangle$
| ε

$\langle ARRAYCALL \rangle \rightarrow [\langle NOTZERODIGIT \rangle]$

$\langle numeric \rangle \rightarrow \langle plusminus \rangle\ \langle DIGIT \rangle\ \langle numericend \rangle$

$\langle plusminus \rangle \rightarrow \varepsilon$
| -
| $+$

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

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

$\langle functioncall \rangle \rightarrow$ call $\langle id \rangle$ ($\langle callexpr \rangle$)

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

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

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

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

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

$\langle comparisonoperator \rangle \rightarrow$ `>`
$\mid$ `<`
$\mid$ `<=`
$\mid$ `>=`
$\mid$ `!=`
$\mid$ `=`

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

$\langle termsymbol \rangle \rightarrow$ *
$\mid \ /$
$\mid$ AND

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

$\langle exprsymbol \rangle \rightarrow +$
$\mid$ `-`
$\mid$ OR

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

$\langle functionmidt \rangle \rightarrow \langle type \rangle \ \langle functionend \rangle \ \langle expr \rangle$; end
$\mid$ nothing $\langle functionend \rangle$ nothing; end

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

$\langle params \rangle \rightarrow \langle subparams \rangle$
$\mid \ \varepsilon$

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

$\langle subparamsend \rangle \rightarrow$, $\langle subparams \rangle$
$\mid \ \varepsilon$

$\langle \textit{stmts} \rangle \rightarrow \varepsilon$
$\quad | \quad \langle \textit{stmt} \rangle \ \langle \textit{stmts} \rangle$

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

$\langle \textit{nontermif} \rangle \rightarrow \text{if}(\langle \textit{expr} \rangle) \text{ begin } \langle \textit{stmts} \rangle \text{ end } \langle \textit{endif} \rangle$

$\langle \textit{endif} \rangle \rightarrow \text{else } \langle \textit{nontermelse} \rangle$
$\quad | \quad \varepsilon$

$\langle \textit{nontermelse} \rangle \rightarrow \langle \textit{nontermif} \rangle$
$\quad | \quad \text{begin } \langle \textit{stmts} \rangle \text{ end}$

$\langle \textit{nontermwhile} \rangle \rightarrow \text{while}(\langle \textit{expr} \rangle) \text{ begin } \langle \textit{stmts} \rangle \text{ end}$

$\langle \textit{from} \rangle \rightarrow \text{from } \langle \textit{callid} \rangle \ \langle \textit{assign} \rangle \text{ to } \langle \textit{expr} \rangle \text{ step } \langle \textit{plusminus} \rangle \ \langle \textit{DIGIT} \rangle \text{ begin } \langle \textit{stmts} \rangle \text{ end}$

$\langle \textit{nontermswitch} \rangle \rightarrow \text{switch } (\langle \textit{expr} \rangle) \text{ begin } \langle \textit{cases} \rangle \text{ end}$

$\langle \textit{cases} \rangle \rightarrow \text{case } \langle \textit{expr} \rangle\text{: } \langle \textit{stmts} \rangle \ \langle \textit{endcase} \rangle$

$\langle \textit{endcase} \rangle \rightarrow \langle \textit{cases} \rangle$
$\quad | \quad \text{break; } \langle \textit{breakend} \rangle$
$\quad | \quad \text{default: } \langle \textit{stmts} \rangle \text{ break;}$

$\langle \textit{breakend} \rangle \rightarrow \langle \textit{cases} \rangle$
$\quad | \quad \text{default: } \langle \textit{stmts} \rangle \text{ break;}$
$\quad | \quad \varepsilon$

$\langle \textit{STRINGTOKEN} \rangle \rightarrow$ " .*? "

$\langle \textit{LETTER} \rangle \rightarrow \text{[a - zA - Z]}^{+}$

$\langle \textit{DIGIT} \rangle \rightarrow \text{[0 - 9]}^{+}$

$\langle \textit{NOTZERODIGIT} \rangle \rightarrow \text{[1-9][0-9]}^{*}$

$\langle \textit{COMMENT} \rangle \rightarrow$ /* .*? */

e Fatal: Skal det med?

2.3.5 EBNF?

e Fatal: Skal det med?

2.3.6 Lexicon

2.4 Semantics

In this section the semantics of SPLAD will be described.

2.4.1 Scoping

The scope of a variable is the block of the program in which it is accessible. A variable is local to a block, if it is declared in that block. A variable is non-local to a block if it is not declared in that block, but is still visible in that block (ex. global variables) [Sebesta, 2009].

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, d]. 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].

2.4.1.1 Symbol Tables

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].

2.4.2 Transition Rules

2.4.3 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.

  Here the type rules of loops will be described:

- 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.

2.5 Code Examples

Implementation 3

3.1 Design Criteria

3.2 Architecture

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. In the description there will be looked at a basic technical specification that will be relevant for this project.

3.2.1.1 Hardware platform

Arduino UNO is a powerful micro controller board which 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 in many ways but Teensy differences in the actual size of the product. Teensy is also cheaper than Arduino but does require soldering for simple set-ups where Arduino 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 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 regular USB as Arduino does [Walker, 2012].

Arduino 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, Teensy and Seeeduino are all compatible with the other equipments that will be used it this project. Netduino is limited to the .net framework were Arduino and the other alternatives are more flexible and therefore more ideal because they work with more platforms. So it comes down to that Arduino have all the necessary features and is the most convenient one to obtain. The project group has found this platform suited for this project based on these reflections.

Arduino UNO board has 14 digital input/output pins where six of them can emulate an analogy output through PWM (Pulse-Width modulation) which are available on the Arduino board. The Arduino board also provides the user with six analogy inputs which enables the reading of a alternating current and provides the user with the currents voltages. 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 an 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 produces 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 kind of drinks on an RFID tag that the customer then can 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. All tags contains 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 is 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].

3.2.1.3 Other components

The demonstration situation will require something to illustrate more advanced parts of theoretical machine. This is because making a whole drink-mixer just to show that the project product works, will take to 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 are neither time for it 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].

As input switches/buttons will be used that 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 **??** 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.

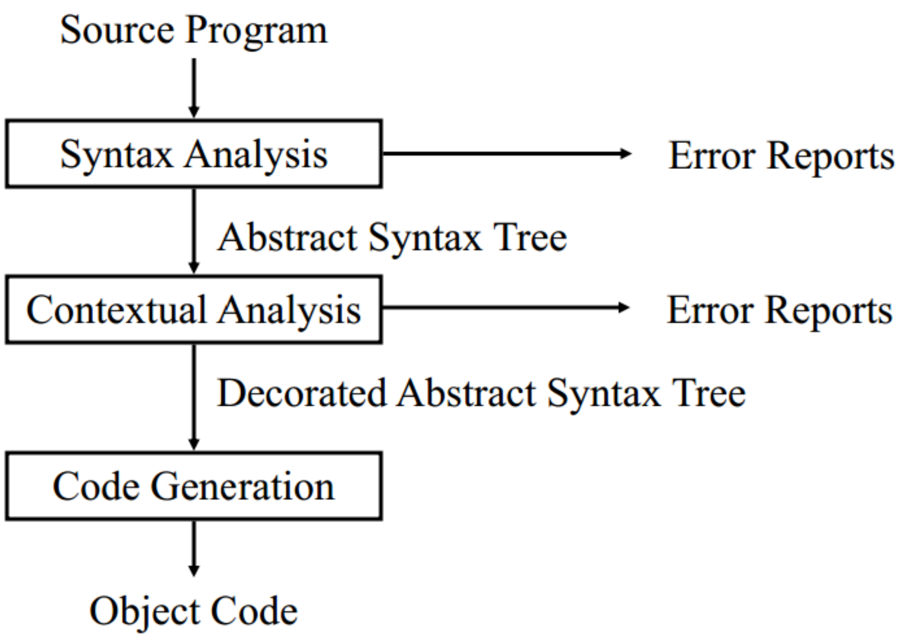

Figure 3.1: This is an abstract overview of how the compiler is structured. The figure is from [Bent Thomsen, 2013].

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.

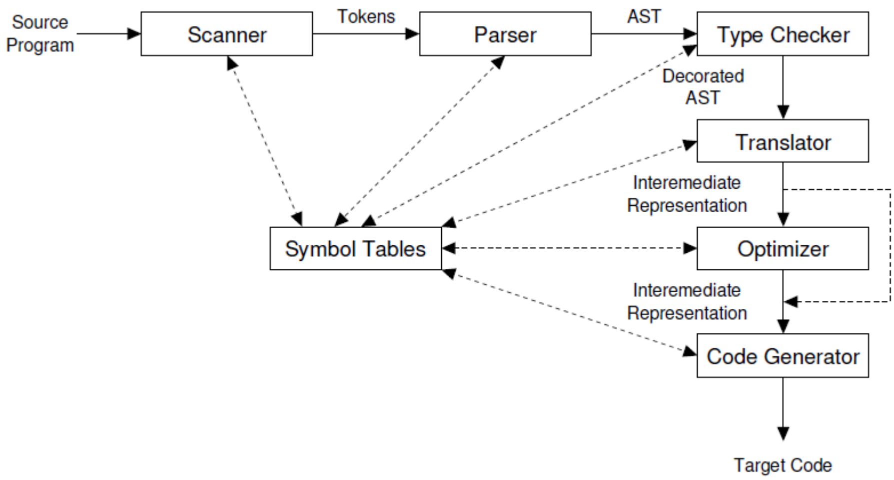

Figure 3.2: This is an more detailed overview of how the compiler is structured. The figure is from [Fischer et al., 2009].

e Fatal: skal det med?

3.2.3 Language Processing Strategy

e Fatal: Skal det med?

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.

atal: Eksempler hvordan parsing g hvordan træet bliver bygget

3.2.6 The Visitor Pattern

3.3 Syntactic Analysis

3.3.1 Semantic Analysis

3.3.2 Known lexers and parsers

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

3.3.2.1 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]

3.3.2.2 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]

3.3.2.3 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].

3.3.2.4 Comparison Table

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

Based on the different lexers and parsers attributes, 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.

3.3.3 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].

```
function setup return nothing using()
begin
  /*Do something*/
  return nothing;
end

function LCDPrint return nothing using(string text)
begin
  /*Function to write a string to the LCD connected to the arduino
      */

  return nothing;
end

function makedrink return nothing using()
begin
  string message <-- "Hello World!";
  call LCDPrint(message);
  return nothing;
end
```

Listing 3.1: 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 **??**, to show how a program is parsed and represented using the GUI option.

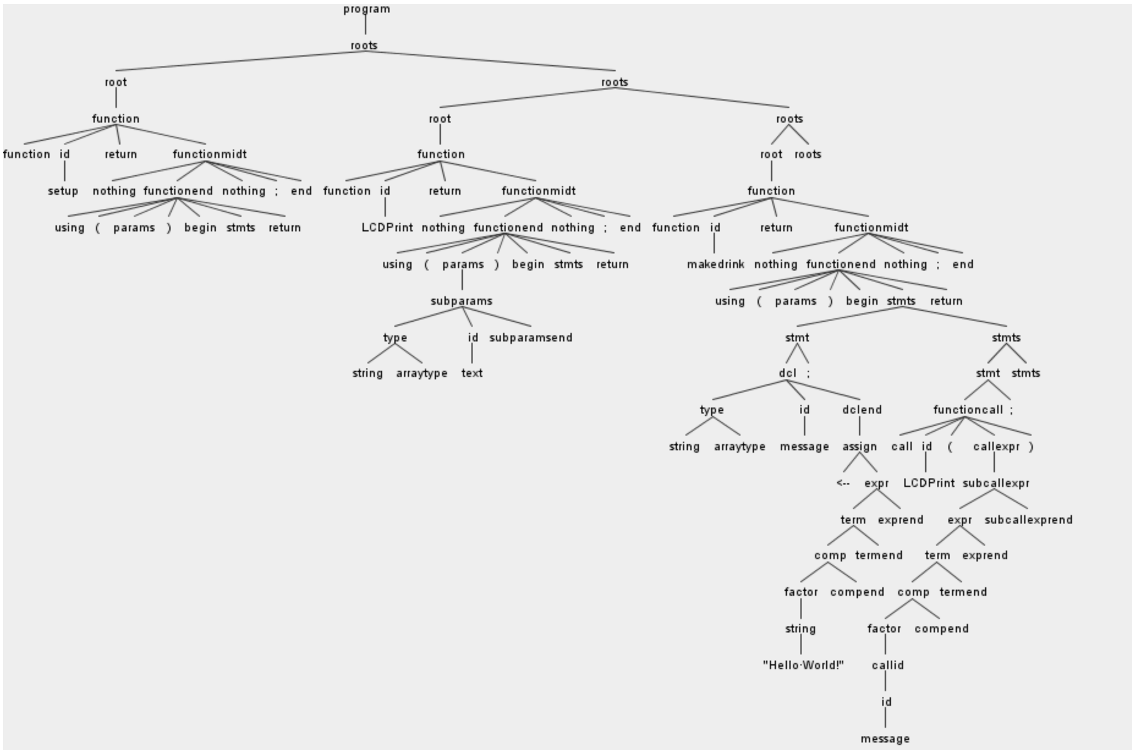

Figure 3.3: On this figure the parse tree for the parsed program "Hello world" can be seen.

On figure 3.3 a parse tree for the "Hello world" program can be seen. Derivations are illustrated by being children of the parent node, see section for more about parse trees.

me Fatal: ref til som parsetræer

3.3.4 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 3.1. 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 3.2 has the token type IDENT and the value 'c'.

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

Table 3.1: 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.

```
c = 42;
```

Listing 3.2: Simple example of code.

As an example the lines of code seen on listing 3.2 might be read as the tokens seen on table 3.2.

Token	**Lexeme**
IDENT	c
ASSIGN	$=$
DIGIT	42
SEMICOLON	;

Table 3.2: 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].

3.3.4.1 Scanner Class Generation

3.3.5 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.

3.3.5.1 Reserved Keywords

The reserved keywords for SPLAD can be seen on table 3.3.

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 3.3: 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.

3.3.5.2 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 3.4.

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 3.4: The tokens in SPLAD.

Further work would be making a lexer to generate a token for the parser. Another options was to find a suited tool for generating a lexer for the given rules. This is a valid option because making a lexer can be automated and therefore already exists a lot of good lexer generators that can be used, see section 3.3.2.

3.3.6 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

3.3.6.1 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.

3.3.6.2 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

3.4.1 Scope Checking

3.4.2 Type Checking

3.5 Code Generation

3.5.1 Declaration

3.5.2 Commands

3.5.3 Expressions

Conclusion 4

FiXme Fatal: ko
mangler

Bibliography

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

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: 2013/03/22].

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].

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].

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.

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].

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].

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

FiXme Fatal: Ma
for loop, call fur
declerations mås

[VAR-ASS]

$$env_C, \vdash \langle x = e,\ sto\rangle \rightarrow sto[l \mapsto v]$$

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

[ARR-ASS]

$$env_C\ \vdash \langle r[a] = e,\ sto\rangle \rightarrow sto[l \mapsto v_2]$$

where $env_C,\ sto \vdash a \rightarrow_a v_1$
and $env_C,\ sto \vdash e \rightarrow_e v_2$
and $env_A\ r[v_1] = l$

[IF-TRUE]

$$\frac{env_C\ \vdash \langle C,\ sto\rangle \rightarrow sto'}{env_C\ \langle \mathbf{if}(b) \text{ begin } C \text{ end},\ sto\rangle \rightarrow sto'}$$

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

[IF-FALSE]

$$env_C\ \vdash \langle \mathbf{if}(b) \text{ begin } C \text{ end},\ sto\rangle \rightarrow sto$$

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

[IF-ELSE-TRUE]

$$\frac{env_C\ \vdash \langle C_1,\ sto\rangle \rightarrow sto'}{env_C\ \vdash\ \langle \mathbf{if}(b) \text{ begin } C_1 \text{ end},\ \mathbf{else} \text{ begin } C_2 \text{ end},\ sto\rangle \rightarrow sto'}$$

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

[IF-ELSE-FALSE]

$$\frac{env_C\ \vdash \langle C_2,\ sto\rangle \rightarrow sto'}{env_C\ \vdash\ \langle \mathbf{if}(b) \text{ begin } C_1 \text{ end},\ \mathbf{else} \text{ begin } C_2 \text{ end},\ sto\rangle \rightarrow sto'}$$

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

[WHL-TRUE]

$$\frac{env_C\ \vdash \langle C,\ sto\rangle \rightarrow sto''\ env_c\ \vdash \langle \mathbf{while}(b) \text{ begin } C \text{ end},\ sto''\rangle \rightarrow sto'}{env_C\ \vdash\ \langle \mathbf{while}(b) \text{ begin } C \text{ end},\ sto\rangle \rightarrow sto'}$$

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

Continued on the next page

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

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

Table A.1: Commands

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

if $v_1 = v_2$

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

if $v_1 \neq v_2$

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

if $v_1 \neq v_2$

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

if $v_1 = v_2$

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

if $v_1 > v_2$

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

if $v_1 \leq v_2$

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

if $v_1 \geq v_2$

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

if $v_1 < v_2$

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

if $v_1 < v_2$

Continued on the next page

[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 $\mathcal{N}$ **: 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