0.1 Choice of grammar

It was decided that the grammar for this project should have a high level of readability. This was decided because the programmer could be a hobby programmer, who want to make a drink machine, but does not have a high level of education in programming. A high readability will ensure that this person easily can read and understand their program - also if they have to edit it later on. The method to assign a value to a variable is by typing "*variable <– valuetoassign*". By using the arrow, it is clearly indicated that the value is assign to the variable, and therefore ensuring readability - especially for the hobby programmer. The functions must always return something, but it can return the value "nothing". This will ensure the understanding and readability, when the programmer can see that it returns, but no value i parsed. To indicate that *return* is the last thing which will be executed in a function, the *return* must always be at the end the function. To indicate that a program is called there must be written "call *functionname*". There are used words instead of symbols (compared to most other programming languages) were suitable to improve the understanding of the program. To indicate a block (eg. a if statement), there are used "begin" and "end". To combine logical operators the words "AND" and "OR" is used. To end a line ";" is used, also to improve readability.

0.2 BNF

⟨*program*⟩ → ⟨*roots*⟩

⟨*roots*⟩ → *ε*
| ⟨*root*⟩ ⟨*roots*⟩

⟨*root*⟩ → ⟨*dcl*⟩;
| ⟨*function*⟩
| ⟨*comment*⟩

⟨*dcl*⟩ → ⟨*type*⟩ ⟨*id*⟩ ⟨*dclend*⟩

⟨*type*⟩ → ⟨*primitivetype*⟩ ⟨*arraytype*⟩

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

⟨*arraytype*⟩ → ⟨*type*⟩ [ ]
| *ε*

⟨*id*⟩ → ⟨*letter*⟩ ⟨*idend*⟩

⟨*letter*⟩ → [a - zA - Z]

⟨*idend*⟩ → ⟨*letter*⟩ ⟨*idend*⟩
| ⟨*digit*⟩ ⟨*idend*⟩
| *ε*

⟨*dclend*⟩ → *ε*
| ⟨*assign*⟩

⟨*assign*⟩ → <-- ⟨*expr*⟩

⟨*expr*⟩ → ⟨*term*⟩ ⟨*exprend*⟩

⟨*term*⟩ → ⟨*comp*⟩ ⟨*termend*⟩

⟨*comp*⟩ → ⟨*factor*⟩ ⟨*compend*⟩

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

⟨*callid*⟩ → ⟨*id*⟩ ⟨*arraycall*⟩

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

⟨*notnulldigits*⟩ → ⟨*notnulldigit*⟩ ⟨*digits*⟩

⟨*notnulldigit*⟩ → [1 - 9]

⟨*digits*⟩ → *ε*
| ⟨*digit*⟩ ⟨*digits*⟩

⟨*digit*⟩ → [0 - 9]

⟨*numeric*⟩ → ⟨*plusminus*⟩ ⟨*digitsnotempty*⟩ ⟨*numericend*⟩

⟨*plusminus*⟩ → *ε*
| -

⟨*digitsnotempty*⟩ → ⟨*digit*⟩ ⟨*digits*⟩

⟨*numericend*⟩ → *ε*
| . ⟨*digitsnotempty*⟩

⟨*string*⟩ → "⟨*stringmidt*⟩"

⟨*stringmidt*⟩ → ⟨*letter*⟩ ⟨*stringmidt*⟩
| ⟨*symbol*⟩ ⟨*stringmidt*⟩
| ⟨*digit*⟩ ⟨*stringmidt*⟩
| *ε*

⟨*symbol*⟩ → !
| %
| ^
| &
| (
|)
| _
| +
| |
| ~
| -
| =
| `
| {
| }
| [
|]
| :
| ;
| ?
| ,
| .
| /
| ' '

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

⟨*callexpr*⟩ → ⟨*subcallexpr*⟩
| *ε*

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

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

⟨*cast*⟩ → ⟨*type*⟩ (⟨*expr*⟩)

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

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

⟨*termend*⟩ → * ⟨*term*⟩
| / ⟨*term*⟩
| AND ⟨*term*⟩
| *ε*

⟨*exprend*⟩ → + ⟨*expr*⟩
| - ⟨*expr*⟩
| OR ⟨*expr*⟩
| ε

⟨*function*⟩ → ⟨*functionstart*⟩ ⟨*functionmidt*⟩

⟨*functionstart*⟩ → function ⟨*id*⟩ return

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

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

⟨*params*⟩ → ⟨*subparams*⟩
| ε

⟨*subparams*⟩ → ⟨*type*⟩ ⟨*id*⟩ ⟨*subparamsend*⟩

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

⟨*stmts*⟩ → ε
| ⟨*stmt*⟩ ⟨*stmts*⟩

⟨*stmt*⟩ → ⟨*callid*⟩ ⟨*assign*⟩;
| ⟨*if*⟩
| ⟨*while*⟩
| ⟨*from*⟩
| ⟨*dcl*⟩;
| ⟨*functioncall*⟩;
| ⟨*switch*⟩
| ⟨*comment*⟩

⟨*if*⟩ → if(⟨*expr*⟩) begin ⟨*stmts*⟩ end ⟨*endif*⟩

⟨*endif*⟩ → else ⟨*else*⟩
| ε

⟨*else*⟩ → ⟨*if*⟩
| begin ⟨*stmts*⟩ end

⟨*while*⟩ → while(⟨*expr*⟩) begin ⟨*stmts*⟩ end

⟨*from*⟩ → from ⟨*expr*⟩ to ⟨*expr*⟩ step ⟨*assign*⟩ begin ⟨*stmts*⟩ end

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

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

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

⟨*breakend*⟩ → ⟨*cases*⟩
| default: ⟨*stmts*⟩ break;
| ε

⟨*comment*⟩ → /* ⟨*stringmidt*⟩ */

0.3 Hardware

This section will be about the hardware components used in this project, describing them and reasons they are used in this project. In the description there be looked at the more basic technical specification that will be relevant for this project.

0.3.1 Hardware platform

The Arduino UNO is a powerful microcontroller board which provides the user with ways to communicate with other components such as LCDs, diodes, sensors and other electronic bricks which is a desirable feature in this project. An Arduino uses the ATmega328 chip which provides more memory than it predecessors and provides the Arduino board the computing power its known for [**?**].

There exists different alternatives to the Arduino product which will looked and judge if they were more suited in this project. Teensy is very similar to the Arduino in many ways but Teensy differences in the actual size of product. The Teensy is also cheaper than the Arduino but does require soldering for simple set-ups were the Arduino comes with a broad and pin-ports which means that it require little pre work to start using it [**?**]. The Seeeduino is nearly a replica of the Arduino, an example could the 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. The Seeeduino is compatible with the same components as the Arduino that makes it suited for acting as a replacement [**?**]. The Netduino is a faster version of the Arduino but it comes at higher cost than the Arduino. The Netduino also require the .net framework so it will only work together with windows operating systems. The Netduino uses a micro-USB instead of regular USB as Arduino do [**?**].

The Arduino is more accessible because the Aalborg university already have some in stock that could be used were the others alternatives first must be brought. The Arduino, Teensy and Seseduino are all compatible with the other equipments that will be used it this project. The Netduino is limited to the .net framework were the Arduino and the other alternatives are more flexible and therefore more ideal because they work with more platforms. So it comes down to that the Arduino have all the necessary features and is the most convenience one to obtain. The project group have this platform suited for this project based on these reflection. To understand the Arduino and how it work there will be look into it and see who it works.

The Arduino UNO board have 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 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 other components and that way provides interaction with environment around the Arduino. The Arduino board is also mounted with an USB-port and jack socket. The board can be hooked up with an USB cable or a AC-to-DC (Alternating Current to Direct Current) adapter through the jack socket to power the unit. The 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 [?].

Programs for the 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. The produces also provides a library with functions to communicate with the platform and compatible components [?]. The Arduino is suited for this project because it makes it possible to demonstrate the language and illustrate that the translation works.

0.3.2 RFID

To administrate the users collection of purchased drinks the plan is to store the number and kind of drinks on a RFID tag that the customers then can use at the drink machine to get their drinks served.

RFID (Radio Frequency IDentification) are used to identify individual objects using radio waves. The communication between the reader and the RFID tag can go both ways, and you are able to read and write to most tag types. The objects able to be read differ 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 from when they were made. This ID is used to identify an individual tag. Tags can be either passive to active tags. Passive tags do not do anything until their antenna catches a signal from a reader. This signal transfers enough energy to the tag for it to send a signal 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 [?].

0.3.3 Other components

The demonstration situation will require something to illustrate more advance parts of theoretical machine. The plan is to use LEDs (light emitting diode) to illustrate the different function of the machine, when their 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 run a current the correct way through the LED. The idea behind using LEDs is that there are not time for making the more advance parts of the machine nor is it the main focus of this project.

It is also desired that the is should be possible to print a form of text to the audience. To do this there will be used a LCD 16-pin (Liquid Crystal Display) which are compatible with the Hitachi44780 driver. Arduino's LiquidCrystal library provides the functions to write to LCD so that no low level code is needed to communicate with the LCD [?].

As input there will be used switches/buttons that will allow interaction with the program at runtime. The switches will illustrate a more advance control unit but in the project switches will be sufficient.

atal: indledning

0.3.4 symbols composition that can't be use for variables names (reserved keywords)

- bool
- int

- double
- char
- string
- <−−
- OR
- AND
- <
- >
- <=
- >=
- !=
- =
- true
- false
- begin
- end
- if
- else
- function
- using
- return
- nothing
- switch
- case
- break
- default
- from
- to
- step
- while
- container

FiXme Fatal: ov
til token specific

0.3.5 Token Specification

FiXme Fatal: mangler tekst

Terminal	Regular Expression
bool	"bool"
int	"int"
double	"double"
char	"char"
string	"string"
==	"< −−"
||	"OR"
&&	"AND"
<	"<"
>	">"
<=	"<="
>=	">="
!=	"!="
=	"="
{	"begin"
}	"end"
if	"if"
else	"else"
yy xx(zz)	"function xx returns yy using zz"
void	"nothing"
switch	"switch"
case	"case"
break	"break"
default	"default"
$for(i = xx; i <= yy; xx+ = zz)$	"from xx to yy step zz"
while	"while"
digitalWrite	"container"
m digit	$[0-9]^+$
numeric	$[0-9]^+.[0-9]^+$
id	$[A-Za-z] \circ ([A-Za-z] \cup [0-9])^*$
boolean	$[true] \cup [false]$

0.4 Parser

A parser takes the tokens from the scanner and use them to create a 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 analyse 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 a abstract syntax tree.

Generally there are two different approaches to parsing - top-down and bottom-up.

Top-down parsers.

Then top-down parser starts at the root and works it way to the leaves in a depth-first manner, doing a preorder traversal if the parse tree. This is done by reading tokens from left to right using a leftmost derivation. Top-down parsers can further more be split into table-drivel LL and recursive descent parse algorithms.

Table-driven LL Parsers: Use a parse table to determine what to do next. The entries in the parse table is determined by the particular LL(k) grammar.

Recursive-descent parser: The recursive-descent parsers consists of mutually recursive parsing routines. Each of the nonterminals in the grammar has an parsing procedure that determines if the token stream contains a sequence of tokens derivable from that nonterminal.

Bottom-Up Parsers.

A Bottom-up parser on the other had do a postorder traversal of the parse tree, meaning it starts from the leaves and works towards the root. A bottom-up parser are more powerful and efficient than a top-down parser, but not as simple

LR: A parser reads from left to right and uses a reversed rightmost derivation. It is as the LL parser driven from a parse table. Find mere her.

LALR: A LALR - se lecture 8 "!

0.5 Known lexers and parsers

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

0.5.1 Lexer

These programs generate a lexical analyser 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.

Flex

Alternativ to lex

An optional feature to flex is the REJECT macro, which enables non-linear perfomance 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

Jlex

Based on lex but used for java.

0.5.2 Parser

Parsers 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.

Cup

More or less like Yacc, output language is in java instead.

0.5.3 Lexer and parser

Combines the lexer and parser in one tool.

SableCC

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

ANTLR

ANother Tool for Language Recognition uses the CFG(Context Free Grammar) 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.

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. The tree builder that accompanies it, JJTree, constructs its trees from the bottom up.

0.5.4 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 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 C or C++, this makes it easier to work on an arduino.