SPLAT

P4 projekt

Group SW407F13
Software
Department of Computer Science
Aalborg University

May 2013

AALBORG UNIVERSITY
STUDENT REPORT

Titel:
SPLAT - Special Programming Language for Arduino Tipple-mixer

Project period:
P4, spring 2012

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:
39

Project end:
29th of May, 2013

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

Synopsis:

FiXme Fatal: synopsis mangler

Prolog

Aalborg March 25, 2013

FiXme Fatal: pr
mangler

Aleksander Sørensen Nilsson

Christian Jødal O'Keeffe

Kasper Plejdrup

Mette Thomsen Pedersen

Niels Brøndum Pedersen

Rasmus Fischer Gadensgaard

Contents

Introduction 1

FiXme Fatal: ind mangler

1.1 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 **??**, 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 easy 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 **??**. 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 easy 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 easy for hobby programmers to program using the language.

1.1.1 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 easy for novice programmers to learn it?** 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 easy-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.

Problem statement 2

FiXme Fatal: pro
statement mang

Analysis 3

3.1 Current language

3.2 Embedded systems

3.3 Arduino platform

Therory 4

4.1 Language

4.1.1 Paradigms of Programming Language

In computer science, four main paradigms of programming languages exists [Nørmark, 2010]. In this section these paradigms will be shortly described followed by a subsection, explaining the choice of programming paradigm of the language in this project.

Imperative Programming

Imperative programming is a very 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 program state. This makes imperative languages very simple, and are also a good starting point for new programmers.

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 are 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 that given time. Some examples of functional programming languages are Haskell and OCaml.

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.

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 become of certain size. In object-oriented programming,

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 4.1: Brief explanation of language characteristics [Sebesta, 2009]

each class of object can be given methods, which is a kind of functions 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 easy in object-oriented languages to specify access-levels of classes, and thereby protect certain classes from external exposure. Classes can inherit form 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.

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 very easy 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 desired.

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 care. The different characteristics of a programming language, which will be used to evaluate trade-offs can be seen on table **??**.

These characteristics are used to evaluate the the trade-offs of programming language. An overview of these can be seen on table 4.2.

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 4.1 with listing 4.2, 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.

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

Table 4.2: Overview of trade-offs [Sebesta, 2009]

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

Listing 4.1: Simple example of if-statement in C using the 'if'-keyword

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

Listing 4.2: Simple example of if-statement in C without using the 'if'-keyword

When defining the syntax of a programming language, it should a 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 hobby-programmer, who might not have much experience in programming.

4.2 Compilers

4.3 Syntax analysis

4.4 Grammartypes

There are 4 types of grammar, where type 0 is the most unrestricted grammar, and type 3 is the more restricted grammar.

FiXme Fatal: KI MANGLER
FiXme Fatal: IN MANGLER til l grammar types c

4.4.1 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.

4.4.2 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.

4.4.3 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.

4.4.4 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].

4.4.5 choice of Grammar

The grammar which has been used to describe the language of this project, is a context-free grammar. It is relatively easy to understand a language described in context-free grammar, but it is still strict enough to allow a computer to work with it. Another reason for using context-free grammar is that most parser- and lexer generators require languages to be described in BNF (Backus–Naur Form) or EBNF (Extended Backus–Naur Form) which is a notation form for context-free grammar. A regular grammar would not have been sufficient, due to the complexity of the language.

4.5 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.

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.

4.5.1 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 4.3. 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 4.4 has the token type IDENT and the value 'c'.

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

Table 4.3: 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.

```
c = 42;
```

Listing 4.3: Simple example of code

As an example the lines of code seen on listing 4.3 might be read as the tokens seen on table 4.4.

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

Table 4.4: 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].

4.6 Semantics

4.7 Contextual analysis

4.8 Code generation

Design 5

5.1 Syntax design

5.2 Choice of grammar

The programmer, using the language of this project, could be a hobby programmer, who would wants to program a custom drink machine, but does not possess a high level of experience in programming. Therefore it was decided that the grammar should have a high level of readability because this will ensure that it is easier for the programmer to read and understand their program - 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 extra words or 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 more clearly indicated that the value is assigned to the variable, and therefore ensuring readability - especially for the hobby programmer.

When making a function it has to be on the form "function $functionname$ return *type* using ($parameter(s)$) begin *statements* return *expresion* end". $Functionname$ 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 a function some values from its call destination(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 a expression which result is of the right type. An example of this can be seen on listing 5.1.

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

Listing 5.1: Example of function declaration in SPLAT

To get a more symmetrical 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 when 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 the 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. 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 are a suited 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 4.5 for more on the grammars.

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

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

$\langle root \rangle \rightarrow \langle dcl \rangle;$
$\quad | \quad \langle function \rangle$
$\quad | \quad \langle comment \rangle$

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

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

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

$\langle arraytype \rangle \rightarrow \langle type \rangle$ [ ]
$\quad | \quad \varepsilon$

$\langle id \rangle \rightarrow \langle letter \rangle\ \langle idend \rangle$

$\langle letter \rangle \rightarrow$ [a - zA - Z]

$\langle idend \rangle \rightarrow \langle letter \rangle \ \langle idend \rangle$
$\quad | \quad \langle digit \rangle \ \langle idend \rangle$
$\quad | \quad \varepsilon$

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

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

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

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

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

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

$\langle callid \rangle \rightarrow \langle id \rangle \ \langle arraycall \rangle$

$\langle arraycall \rangle \rightarrow [\langle notnulldigits \rangle]$
$\quad | \quad \varepsilon$

$\langle notnulldigits \rangle \rightarrow \langle notnulldigit \rangle \ \langle digits \rangle$

$\langle notnulldigit \rangle \rightarrow [1 \text{ - } 9]$

$\langle digits \rangle \rightarrow \varepsilon$
$\quad | \quad \langle digit \rangle \ \langle digits \rangle$

$\langle digit \rangle \rightarrow [0 \text{ - } 9]$

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

$\langle plusminus \rangle \rightarrow \varepsilon$
$\quad | \quad \text{-}$

$\langle digitsnotempty \rangle \rightarrow \langle digit \rangle \ \langle digits \rangle$

$\langle numericend \rangle \rightarrow \varepsilon$
$\quad | \quad . \ \langle digitsnotempty \rangle$

$\langle string \rangle \rightarrow \text{"}\langle stringmidt \rangle\text{"}$

$\langle stringmidt \rangle \rightarrow \langle letter \rangle \ \langle stringmidt \rangle$
$\mid \ \langle symbol \rangle \ \langle stringmidt \rangle$
$\mid \ \langle digit \rangle \ \langle stringmidt \rangle$
$\mid \ \varepsilon$

$\langle symbol \rangle \rightarrow \ !$
$\mid \ \%$
$\mid \ \hat{}$
$\mid \ \&$
$\mid \ ($
$\mid \ )$
$\mid \ \_$
$\mid \ +$
$\mid \ |$
$\mid \ \tilde{}$
$\mid \ \texttt{-}$
$\mid \ =$
$\mid \ \grave{}$
$\mid \ \{$
$\mid \ \}$
$\mid \ [$
$\mid \ ]$
$\mid \ :$
$\mid \ ;$
$\mid \ ?$
$\mid \ ,$
$\mid \ .$
$\mid \ /$
$\mid \ ,\ ,$
$\mid$

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

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

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

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

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

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

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

$\langle \mathit{termend} \rangle \rightarrow$ * $\langle \mathit{term} \rangle$
| / $\langle \mathit{term} \rangle$
| AND $\langle \mathit{term} \rangle$
| ε

$\langle \mathit{exprend} \rangle \rightarrow + \langle \mathit{expr} \rangle$
| - $\langle \mathit{expr} \rangle$
| OR $\langle \mathit{expr} \rangle$
| ε

$\langle \mathit{function} \rangle \rightarrow \langle \mathit{functionstart} \rangle \langle \mathit{functionmidt} \rangle$

$\langle \mathit{functionstart} \rangle \rightarrow$ function $\langle \mathit{id} \rangle$ return

$\langle \mathit{functionmidt} \rangle \rightarrow \langle \mathit{type} \rangle \langle \mathit{functionend} \rangle \langle \mathit{expr} \rangle$; end
| nothing $\langle \mathit{functionend} \rangle$ nothing; end

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

$\langle \mathit{params} \rangle \rightarrow \langle \mathit{subparams} \rangle$
| ε

$\langle \mathit{subparams} \rangle \rightarrow \langle \mathit{type} \rangle \langle \mathit{id} \rangle \langle \mathit{subparamsend} \rangle$

$\langle \mathit{subparamsend} \rangle \rightarrow$, $\langle \mathit{subparams} \rangle$
| ε

$\langle \mathit{stmts} \rangle \rightarrow \varepsilon$
| $\langle \mathit{stmt} \rangle \langle \mathit{stmts} \rangle$

$\langle \mathit{stmt} \rangle \rightarrow \langle \mathit{callid} \rangle \langle \mathit{assign} \rangle$;
| $\langle \mathit{nontermif} \rangle$
| $\langle \mathit{nontermwhile} \rangle$
| $\langle \mathit{from} \rangle$
| $\langle \mathit{dcl} \rangle$;
| $\langle \mathit{functioncall} \rangle$;
| $\langle \mathit{nontermswitch} \rangle$
| $\langle \mathit{comment} \rangle$

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

$\langle \mathit{endif} \rangle \rightarrow$ else $\langle \mathit{nontermelse} \rangle$
| ε

$\langle \mathit{nontermelse} \rangle \rightarrow \langle \mathit{nontermif} \rangle$
| begin $\langle \mathit{stmts} \rangle$ end

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

$\langle \mathit{from} \rangle \rightarrow$ from $\langle \mathit{expr} \rangle$ to $\langle \mathit{expr} \rangle$ step $\langle \mathit{assign} \rangle$ begin $\langle \mathit{stmts} \rangle$ end

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

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

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

$\langle breakend \rangle \rightarrow \langle cases \rangle$
  | default: $\langle stmts \rangle$ break;
  | ε

$\langle comment \rangle \rightarrow$ /* $\langle stringmidt \rangle$ */

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.

5.2.1 Reserved Keywords

The reserved keywords for SPLAT are:

- 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
- HIGH
- LOW

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.

5.2.2 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 [?]. Regular expressions are strong in describing patterns which is the core of token production [Sipser, 2013].

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

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 6.1.

5.3 Semantic design

In this section the semantics of SPLAT will be described.

5.3.1 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).

In SPLAT 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 SPLAT to C simpler for the compiler [ard]. 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.

Symbol tables

Generally there are two approaches to symbol tables: One symbol table for each scope, or one global symbol table [Sebesta, 2009].

ne Fatal: Vi skal besluttet hvilken symbol table vi bruger

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.

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.

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.

5.3.2 Type Rules

This section contains the type rules for the comparison operator.

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, 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, 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 integer or 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.

Here the type rules of assign will be described:

"E_1 <– E_2" is type correct if E_1 and E_2 are of the same of type integer, double, char or string.

Here the type rules of loops will be described.

Type rule of 'while'-statement: "while E begin C end" is type correct if E of type boolean and C are type correct.

Type rule of 'from to'-statement: "from E_1 to E_2 begin C end" are type correct if E_1 and E_2 are type correct and of type integer, and C are type correct.

This is the type rules for 'if'-statement: "if(E) begin C end" is type correct if E are type correct and of type boolean, and C are 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, double, char or string and are the same type, and C_1...C_n and C_d are type correct.

5.4 Code examples

Implementation 6

6.1 Known lexers and parsers

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

6.1.1 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]

6.1.2 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]

6.1.3 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]

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].

6.1.4 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.

6.2 Scanner class creation

6.3 Parser generation

6.4 Class generator classgen

6.5 Scope and type checking

6.6 Code generation

6.7 Test/evaluation

Conclusion 7

FiXme Fatal: ko
mangler

Bibliography

JJTree. *Arduino Build Process*. Web. URL `http://arduino.cc/en/Hacking/BuildProcess`.

Antlr. Antlr. *Theory behind Antlr*. URL `http://www.antlr.org/about.html`.

Cup. Cup. *Theory behind CUP*. URL `http://www2.cs.tum.edu/projects/cup/manual.html`.

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*. URL `http://flex.sourceforge.net/manual/`.

Jflex. Jflex. *Theory behind Jflex*. URL `http://jflex.de/manual.html#SECTION00040000000000000000`.

JJTree. JJTree. *JJTree to JavaCC*. URL `http://tomcopeland.blogs.com/juniordeveloper/2007/10/better-jjtree-v.html`.

Jlex. Jlex. *Theory behind Jlex*. URL `http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html`.

Lex. Lex. *Theory behind Lex*. URL `http://dinosaur.compilertools.net/lex/index.html`.

Norvell. Theodore S. Norvell. *The JavaCC FAQ*. URL `http://www.engr.mun.ca/~theo/JavaCC-FAQ/javacc-faq-moz.htm#jjtree-and-jtb`.

Nørmark, July 7 2010. Kurt Nørmark. *Overview of the four main programming paradigms*. web, 2010. URL `http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html`.

SableCC. SableCC. *SableCC homesite*. URL `http://sablecc.org/wiki`.

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.

Yacc. Yacc. *Theory behind Yacc*. URL `http://dinosaur.compilertools.net/yacc/index.html`.

List of Corrections

Appendix 8