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:
31

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 20, 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: inc
mangler

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

4.2 Compilers

4.3 Semantics

4.4 Syntax analysis

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 running through a string using CFG can only read the string in one way.

A CFG is a part of the $LL(k)$ grammar classes 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 grammes which means that LL parsers works on $LL(k)$ grammars. $LL(k)$ means that the grammar needs to be left-recursive free 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. A propriety is that $LL(1)$ grammars are fairly easy compare to $LL(k) where k > 1$ to implement because the parser analyser only have to look one element ahead in order to determine what parser action there should be taken. $LL(1)$ is also relatively faster than $LL(k) where k > 1$ based on the same reason, that the parser only have 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 is faster at detecting the syntax errors. LL is also inferior compare 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 parser is a bottom-up parser meaning that it start constructing the abstract trees from its leaf and works its way to the root. LR parsers are general harder to implement than LL parsers by hand but there exists tools that automatic can generate LR parsers. $LR(k)$ 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 language 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 of these understandings of grammars there will be a section were there will looked into which grammar that will be used in this project.

4.6 Contextual analysis

4.7 Code generation

Design 5

5.1 Syntax design

5.2 Choice of grammar

The programmer, using this projects language, could be a hobby programmer, who would want to program a custom drink machine, but does not possess a high level of education 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 person to read and understand their program - also useful if the code has to be edited later on. This on the other hand can decrease the level of write-ability because it has to be written in a specific way and will need to contain some extra words or symbols to mimic a language closer to human language rather than a computer language.

The method to assign a value to a variable is by typing "$variable <- valuetoassign$" this approach have 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.

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 is 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 program 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 line 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 tool 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. *LR* parsers can be made by hand but it is much more difficult than the *LL* parsers.

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

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

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

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

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

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

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

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

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

$\langle idend \rangle \rightarrow \langle letter \rangle$ $\langle idend \rangle$
| $\langle digit \rangle$ $\langle idend \rangle$
| ε

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

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

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

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

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

$\langle factor \rangle \rightarrow$ ($\langle expr \rangle$)
| !($\langle expr \rangle$)
| $\langle callid \rangle$
| $\langle numeric \rangle$
| $\langle string \rangle$

| $\langle functioncall \rangle$
| $\langle cast \rangle$
| LOW
| HIGH
| true
| false

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

$\langle arraycall \rangle \rightarrow$ [$\langle notnulldigits \rangle$]
| ε

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

$\langle notnulldigit \rangle \rightarrow$ [1 - 9]

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

$\langle digit \rangle \rightarrow$ [0 - 9]

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

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

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

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

$\langle string \rangle \rightarrow$ "$\langle stringmidt \rangle$"

$\langle stringmidt \rangle \rightarrow \langle letter \rangle \ \langle stringmidt \rangle$
| $\langle symbol \rangle \ \langle stringmidt \rangle$
| $\langle digit \rangle \ \langle stringmidt \rangle$
| ε

$\langle symbol \rangle \rightarrow$!
| %
| ^
| &
| (
|)
| _
| +
| |
| ~
| -
| =
| `
| {
| }

| [
|]
| :
| ;
| ?
| ,
| .
| /
| ' '

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

$\langle callexpr \rangle \rightarrow \langle subcallexpr \rangle$
| ε

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

$\langle subcallexprend \rangle \rightarrow$, $\langle subcallexpr \rangle$
| ε

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

$\langle compend \rangle \rightarrow \langle comparisonoperator \rangle \langle comp \rangle$
| ε

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

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

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

$\langle function \rangle \rightarrow \langle functionstart \rangle \langle functionmidt \rangle$

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

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

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

$\langle params \rangle \rightarrow \langle subparams \rangle$
| ε

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

$\langle subparamsend \rangle \rightarrow$, $\langle subparams \rangle$
| ε

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

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

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

$\langle endif \rangle \rightarrow$ else $\langle nontermelse \rangle$
| ε

$\langle nontermelse \rangle \rightarrow \langle nontermif \rangle$
| begin $\langle stmts \rangle$ end

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

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

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

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

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

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

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

5.3 Semantics of SPLAT

I 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).

ne Fatal: Er det compileren???

In SPLAT static scoping is used. This means that scopes are computed at compile time, based on the inputted program text. 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.

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 Checking

5.4 Code examples

6 Implementation

6.1 Scanner class creation

6.2 Parser generation

6.3 Class generator classgen

6.4 Scope and type checking

6.5 Code generation

6.6 Test/evaluation

Conclusion 7

FiXme Fatal: ko
mangler

Literature 8

Bibliography

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

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.

List of Corrections

Appendix 9