



# A Formal Description of NoBeard

v 1.2

P. Bauer

HTBLA Leonding  
Limesstr. 14 - 18  
4060 Leonding  
Austria

## Revisions

Date	Author	Change
August 9, 2016	P. Bauer	Explained call stack and control unit of the machine.
June 25, 2016	P. Bauer	Added more detailed description of the machine and added all assembler instructions used so far.
June 12, 2016	P. Bauer	Changed environment for grammar and formatting of source code.
June 5, 2014	P. Bauer	<b>Released v. 1.1</b>
June 5, 2012	P. Bauer	<b>Released v. 1.0</b>

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>The NoBeard Machine</b>	<b>5</b>
2.1	Overview . . . . .	5
2.1.1	Program Memory . . . . .	5
2.1.2	Data Memory . . . . .	5
2.1.3	Call Stack . . . . .	6
2.1.4	Control Unit . . . . .	6
2.2	Stack Frames . . . . .	7
2.3	Binary File Format . . . . .	7
2.4	Runtime Structure of a NoBeard Program . . . . .	7
2.5	Instructions . . . . .	8
2.5.1	nop . . . . .	8
2.5.2	lit . . . . .	9
2.5.3	la . . . . .	9
2.5.4	lv . . . . .	9
2.5.5	lc . . . . .	10
2.5.6	sto . . . . .	10
2.5.7	stc . . . . .	11
2.5.8	assn . . . . .	11
2.5.9	neg . . . . .	11
2.5.10	add . . . . .	12
2.5.11	sub . . . . .	12
2.5.12	mul . . . . .	12
2.5.13	div . . . . .	13
2.5.14	mod . . . . .	13
2.5.15	not . . . . .	13
2.5.16	rel . . . . .	14
2.5.17	fjmp . . . . .	15
2.5.18	tjmp . . . . .	15
2.5.19	jmp . . . . .	16
2.5.20	out . . . . .	16
2.5.21	inc . . . . .	17

2.5.22	halt . . . . .	17
<b>3</b>	<b>Symbol List</b>	<b>18</b>
<b>4</b>	<b>NoBeard Assembler</b>	<b>20</b>
4.1	Assembler File Structure . . . . .	20
4.2	Examples . . . . .	20
4.3	Formal Description . . . . .	21
<b>5</b>	<b>The Programming Language</b>	<b>22</b>
5.1	Lexical Structure . . . . .	22
5.1.1	Character Sets . . . . .	22
5.1.2	Keywords . . . . .	22
5.1.3	Token Classes . . . . .	22
5.1.4	Single Tokens . . . . .	22
5.1.5	Semantics . . . . .	22
5.2	Sample Program . . . . .	23
5.3	Syntax . . . . .	23
5.4	Semantics . . . . .	24
<b>6</b>	<b>Some Translations by Example</b>	<b>26</b>
6.1	Reserving Space for Local Variables . . . . .	26
6.2	Assignments . . . . .	26
6.3	Boolean Expressions . . . . .	26
<b>7</b>	<b>Error Handling</b>	<b>28</b>
<b>8</b>	<b>Attributed Grammar</b>	<b>29</b>

# Chapter 1

## Introduction

According to a web article (see [Kha08]) the popularity of programming languages is strongly related to the fact whether its inventor(s) is/are are bearded m[ae]n or not. Well, the main aim of the programming language NoBeard is not to be popular, moreover it should give the reader a clear insight how the main principles of compiler construction are.

This report aims to give a formal description of the programming language NoBeard. Please note that only the parts necessary for your work can be trusted. In the next versions, more and more information relevant for your assignments will be available.

In case of typos, misleading wording or other problems, please feel free to contact me. Thanks for your help. Some more text to read [Ter04].

## Chapter 2

# The NoBeard Machine

### 2.1 Overview

The virtual machine being target for NoBeard programs is a stack machine with instructions of variable length and has the following components. The word width of the NoBeard machine is four bytes.

#### 2.1.1 Program Memory

The program memory is further denoted as `prog[MAX_PROG]`. It is byte addressed with a maximum size of `MAX_PROG`. Attempts to access addresses outside the range of 0 to `MAX_PROG - 1` result in a `ProgramAddressError`.

#### 2.1.2 Data Memory

The *data memory* is byte addressed and storage is done in the following way:

- Characters are one-byte values and are stored byte-wise into the data memory.
- Integers are four-byte values and are stored in *little endian order* i.e., the lowest significant byte is stored first. Negative integer values are stored as *Two's Complement* [Wik16].
- Booleans are four-byte values and are stored as the integer 0 for false and the integer 1 for true.

The data memory is separated into two parts:

- String constants
- Stack frames of the currently running functions

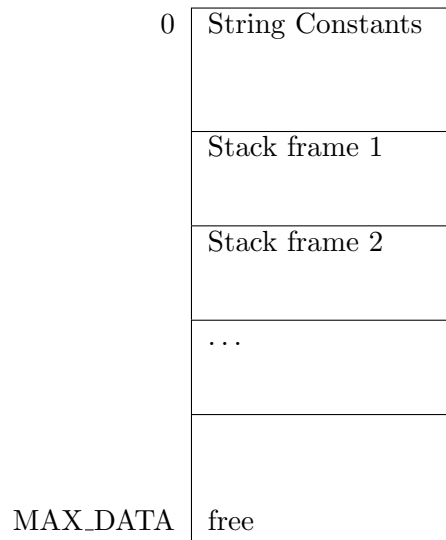


Figure 2.1: Data Memory of the NoBeard Machine

Figure 2.1 shows this. Before a program is started the string constants are stored in the constant memory. On top of this the stack frames are maintained as follows: Every time a function is called a frame is added. It holds data for the function arguments, local variables, some auxiliary data and its expression stack (shortly called stack in the sequel). As soon as the function ends, its frame is removed. A more detailed description of stack frames is given in section 2.2.

### 2.1.3 Call Stack

Since most of the data in the data memory is organized as a stack the *call stack* as an abstraction to the data memory is defined. It provides functions to add and remove frames from the stack and to maintain the expression stack. The expression stack is used to store data needed for each statement. It grows and shrinks as needed and is empty at the end of each statement. The stack is addressed word-wise only. The functions `push()` and `pop()` are used to add and remove values to and from the stack, respectively. It has the following components:

- `top` Address of the start of the last used word on the stack.
- `fp` *Frame Pointer*: Address of the first byte of the currently running function's stack frame.

### 2.1.4 Control Unit

The control unit is responsible for the proper execution of programs. It executes one machine cycle, i.e., it fetches, decodes and executes the current instruction. In order to do this it uses the following components:

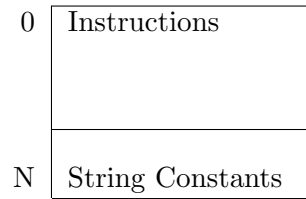


Figure 2.2: NoBeard Binary File Format

<b>pc</b>	<i>Program Counter:</i> Start address of the next instruction in <b>prog</b> to be executed.
<b>ms</b>	<i>Machine State:</i> The NoBeard machine may have three different states: <ul style="list-style-type: none"> <li>• <b>run:</b> The machine runs</li> <li>• <b>stop:</b> The machine stops. Usually when the end of program is reached.</li> <li>• <b>error:</b> Error state</li> </ul>

## 2.2 Stack Frames

## 2.3 Binary File Format

NoBeard binaries have the extension `.no` (from NoBeard object file) and have a format as shown in figure 2.2. From byte 0 onwards the machine instructions are stored in a continuous stream. After the final `halt` instruction the stream of string constants follows immediately.

## 2.4 Runtime Structure of a NoBeard Program

The NoBeard Machine follows the following fixed execution cycle:

1. Fetch instruction
2. Decode instruction
3. Execute instruction

The very first instruction is fetched from `prog[startPc]` where `startPc` has to be provided as an argument when starting the program. From this point of time onwards the program is executed until the machine state changes from *run*.



```

runProg(startPc) {
    fp = start byte of first free word in dat;
    top = fp + 28;
    pc = startPc;
    ms = run;

    while (ms == run) {
        fetch instruction which starts at prog[pc];
        pc = pc + length of instruction;
        execute instruction
    }
}

```

## 2.5 Instructions

NoBeard instructions have a variable length. Every instruction has an opcode and either zero, or one, or two operands. When describing the instructions we use the following conventions:

Each instruction is described in one of the following subsections. The title of the subsection is the mnemonic by which the instruction is identified on assembler level. Then a table follows which shows the size of the instruction and which bytes carry which information. For all instructions the first byte is dedicated to the op code, which is the id by which the instruction is identified on machine language level.

The remaining bytes, if any, are dedicated to the operands of the instruction. When describing these we use the following conventions:

Name	Range	Size	Description
Literal	0 ... 65535	2 Bytes	Unsigned integer number.
Displacement	0 ... 256	1 Byte	Static difference in hierarchy between declaration and usage of an object.
DataAddress	0 ... 65535	2 Bytes	Data address relative to the start of its stack frame.

Each of these subsections ends with a description of its operation: First a description in human language is given which is then followed by a formal definition.

### 2.5.1 nop

#### Instruction

Byte 0
0x00

## Operation

Empty instruction. Does nothing

```
nop
```

### 2.5.2 lit

#### Instruction

Byte 0	Byte 1	Byte 2
0x01	Literal	

## Operation

Pushes a value on the expression stack.

```
lit Literal
push(Literal);
```

### 2.5.3 la

#### Instruction

Byte 0	Byte 1	Byte 2	Byte 3
0x02	Displacement	DataAddress	

## Operation

Loads an address on the stack.

```
la Displacement DataAddress
base = fp;
for (i= 0; i < Displacement; i++) {
    base = dat[base ... base + 3];
}
push(base + DataAddress);
```

### 2.5.4 lv

#### Instruction

Byte 0	Byte 1	Byte 2	Byte 3
0x03	Displacement	DataAddress	

## Operation

Loads a value on the stack.

```
lv Displacement DataAddress
base = fp;
for (i = 0; i < Displacement; i++) {
    base = dat[base ... base + 3];
}
adr = base + DataAddress;
push(dat[adr ... adr + 3]);
```

### 2.5.5 lc

#### Instruction

Byte 0	Byte 1	Byte 2	Byte 3
0x04	Displacement	DataAddress	

## Operation

Loads a character on the stack.

```
lc Displacement DataAddress
base = fp;
for (i = 0; i < Displacement; i++) {
    base = dat[base ... base + 3];
}
// fill 3 bytes of zeros to get a full word
lw = 000dat[base + Address];
push(lw);
```

### 2.5.6 sto

#### Instruction

Byte 0
0x07

## Operation

Stores a value on an address.

```
sto
x = pop();
a = pop();
dat[a ... a + 3] = x;
```

### 2.5.7 stc

#### Instruction

Byte 0
0x08

#### Operation

Stores a character on an address.

```
stc
x = pop();
a = pop();
// Only take the rightmost byte
dat[a] = 000x;
```

### 2.5.8 assn

#### Instruction

Byte 0
0x0A

#### Operation

Array assignment.

```
assn
n = pop();
src = pop();
dest = pop();
for (i = 0; i < n; i++)
    dat[dest + i] = dat[src + i];
```

### 2.5.9 neg

#### Instruction

Byte 0
0x0B

#### Operation

Negates the top of the stack.

```
neg
x = pop();
push(-x);
```

### 2.5.10 add

#### Instruction

Byte 0
0x0C

#### Operation

Adds the top two values of the stack.

```
add
push(pop() + pop());
```

### 2.5.11 sub

#### Instruction

Byte 0
0x0D

#### Operation

Subtracts the top two values of the stack.

```
sub
y = pop();
x = pop();
push(x - y);
```

### 2.5.12 mul

#### Instruction

Byte 0
0x0E

#### Operation

Multiplies the top two values of the stack.

```
mul
push(pop() * pop());
```

### 2.5.13 div

#### Instruction

Byte 0
0x0F

#### Operation

Divides the top two values of the stack.

```
div
y = pop();
x = pop();
if (y != 0)
    push(x / y);
else
    throwDivByZero();
```

### 2.5.14 mod

#### Instruction

Byte 0
0x10

#### Operation

Calculates the remainder of the division of the top values of the stack.

```
mod
y = pop();
x = pop();
push(x % y);
```

### 2.5.15 not

#### Instruction

Byte 0
0x11

## Operation

Calculates the remainder of the division of the top values of the stack.

```
not
x = pop();
if (x == 0)
    push(1);
else
    push(0);
```

### 2.5.16 rel

#### Instruction

Byte 0	Byte 1
0x12	RelOp

#### Operation

Compares two values of the stack and pushes the result back on the stack. The operand RelOp can have six different values:

- 0 for encoding < (smaller than)
- 1 for encoding <= (smaller or equal than)
- 2 for encoding == (equals)
- 3 for encoding != (not equals)
- 4 for encoding >= (greater or equal than)
- 5 for encoding > (greater than)

```
rel RelOp
y = pop();
x = pop();
switch(RelOp) {
    case 0:
        if (x < y) push(1); else push(0);
        break;
    case 1:
        if (x <= y) push(1); else push(0);
        break;
    case 2:
        if (x == y) push(1); else push(0);
```

```

        break;
    case 3:
        if (x != y) push(1); else push(0);
        break;
    case 4:
        if (x >= y) push(1); else push(0);
        break;
    case 5:
        if (x > y) push(1); else push(0);
        break;
}

```

### 2.5.17 fjmp

#### Instruction

Byte 0	Byte 1	Byte 2
0x16	NewPc	

#### Operation

Sets `pc` to `newPc` if stack top value is false.

```

fjmp newPc
x = pop();
if (x == 0)
    pc = NewPc;

```

### 2.5.18 tjmp

#### Instruction

Byte 0	Byte 1	Byte 2
0x17	NewPc	

#### Operation

Sets `pc` to `newPc` if stack top value is true.

```

tjmp newPc
x = pop();
if (x == 1)
    pc = NewPc;

```



### 2.5.19 jmp

#### Instruction

Byte 0	Byte 1	Byte 2
0x18	NewPc	

#### Operation

Unconditional jump: Sets **pc** to **newPc**.

```
jmp newPc
pc = NewPc;
```

### 2.5.20 out

#### Instruction

Byte 0	Byte 1
0x1A	Type

#### Operation

Writes data to the terminal. Depending on **Type** different data types are printed:

- 0: An **int** with a specific column width is printed
- 1: A **char** with a specific column width is printed
- 2: a **string** with a specific column width is printed
- 3: a new line is printed

```
put Type
switch(Type) {
    case 0:
        width = pop();
        x = pop();
        // + means string concatenation in the next line
        formatString = "%" + width + "d";
        printf(formatString, x);
        break;
    case 1:
        width= pop();
        x = pop();
        printf("%c", x);
        for (i = 0; i < width - 1; i++)
```

```

        printf(" ");
        break;
    case 2:
        width = pop();
        strLen = pop();
        strAddr = pop();
        printf("%s", dat[strAddr ... strAddr + strLen - 1]);
        for (i = n; i < width - 1; i++)
            printf(" ");
        break;
    case 3:
        printf("\n");
        break;
}

```

### 2.5.21 inc

#### Instruction

Byte 0	Byte 1	Byte 2
0x1D	Size	

#### Operation

Increases the size of the stack frame by **Size**.

```

inc Size
top += Size;

```

### 2.5.22 halt

#### Instruction

Byte 0
0x1F

#### Operation

Halts the machine.

```

halt
ms = stop;

```

## Chapter 3

# Symbol List

```
1 unit M;
2   function A(int a);
3     int b;
4
5     int function B(char c);
6       int d;
7     do
8       # some code
9     done B;
10
11    char function C;
12      int e;
13    do
14      # some more code
15    done C;
16  do
17    # some code on A
18  done A;
19 do
20   # this is the main of unit M
21 done M;
```

After parsing line 1 the symbol list looks as follows:

name	kind	type	size	addr	level
M	PROCKIND	UNITTYPE	0	0	0

After parsing line 2 a snapshot on the symbol list looks like

name	kind	type	size	addr	level
M	PROCKIND	UNITTYPE	0	0	0
A	PROCKIND	UNITTYPE	0	0	1
a	PARKIND	SIMINT	4	32	2

After parsing line 3

name	kind	type	size	addr	level
M	PROCKIND	UNITTYPE	0	0	0
A	PROCKIND	PROCTYPE	4	0	1
a	PARKIND	SIMINT	4	32	2
b	VARCKIND	SIMINT	4	36	2

After parsing line 6 being somewhere between line 7 and the end of line 9.

name	kind	type	size	addr	level
M	PROCKIND	UNITTYPE	0	0	0
A	PROCKIND	PROCTYPE	4	0	1
a	PARKIND	SIMINT	4	32	2
b	VARCKIND	SIMINT	4	36	2
B	PROCKIND	PROCTYPE	0	0	2
c	PARKIND	SIMCHAR	1	32	3
d	VARCKIND	SIMINT	4	36	3

## Chapter 4

# NoBeard Assembler

### 4.1 Assembler File Structure

NoBeard Assembler files are text files which contain two blocks, namely the string constants and the assembler program. The files have the extensions `.na` for NoBeard Assembler.

The string constants are stored within one block of double quotes and can be organized by the programmer as (s)he wants. There is no possibility to address one single constant, i.e., when using a string constant in the assembler program one has to provide the start address of the string constant and the length needed in the program.

Assembler programs are texts holding a sequence of assembler instructions as described in section 2.5. The opcode has to be written in lower case letters. The programmer has to follow the instruction format, i.e., (s)he has to take care that the given operands fit into the required data format of the instruction.

### 4.2 Examples

The first example shows a non-empty assembler program which definitely does nothing. It is worth to be mentioned that `//` marks a comment which then lasts until the end of the line. Assembler instructions need not to start at every new line, they could also be written as a sequence in one line or one instruction could be broken up into several lines. Line breaks are only allowed between the opcode and operands or between the operands.

```
1 # a lazy program
2 nop    # This is an empty instruction which does nothing
3 halt   # halt is mandatory to properly finish the program
```

A NoBeard assembler program writing “Hello World” into the console could look as follows. Note that the width of the column when writing “Hello” is set one character wider than the length of the string (line 4). With this “trick” we get the blank between the two words. Of course, in this case, one could achieve the same result much easier

by specifying already the string constant as needed to output the required string in one `out` statement.

```
1 "HelloWorld"    # The string constants for output
2 lit 0           # load address of "Hello"
3 lit 5           # load length of "Hello"
4 lit 6           # load column width
5 out 2           # output string
6 lit 5           # load address of "World"
7 lit 5           # load lenght of "World"
8 lit 5           # load column width
9 out 2           # output string
10 out 3          # output new line
```

### 4.3 Formal Description

NoBeardAssembler	=	{AssemblerInstruction}.
AssemblerInstruction	=	opcode OneOperand   TwoOperands.
OneOperand	=	number.
TwoOperands	=	number number.

## Chapter 5

# The Programming Language

### 5.1 Lexical Structure

NoBeard programs are written in text files of free format, i.e., there is no restriction concerning columns or lines where the source text has to be. In this section the scanner relevant terms for NoBeard are denoted in the form of regular expressions with the extension that we allow "definitions" of non-terminals. This means in particular that if we define a term (e.g. *letter* as it can be seen in the next section) this term can be used in subsequent definitions and is rewritten as given in its original definition.

#### 5.1.1 Character Sets

**letter** '[A-Za-z]'

**digit** '[0-9]'

#### 5.1.2 Keywords

There is only one keyword, namely PUT.

#### 5.1.3 Token Classes

**ident** ['letter(letter | digit)\*']

**number** ['digit digit\*']

#### 5.1.4 Single Tokens

The characters "+", "-", "\*", "/", ":", ";", "(", and ")" are mapped to single tokens.

#### 5.1.5 Semantics

- NoBeard is a case sensitive language. For example, the names "myVar", "myvar", and "MYVAR" denote three different identifiers.

- Constants may only be between 0 and 65535 ( $2^{16} - 1$ ).
- No symbol may span over more than one line.

## 5.2 Sample Program

```

1 unit ComplexExpr;
2 # ----- ComplexExpr.nb -----
3 # --- A syntactically correct NoBeard program
4 # -----
5 do
6     int l = 10;
7     int b = 5;
8     int h = 170;
9     int unused = 1;
10    int x = 1001 + l * b - h / (b * h);
11
12    put ("Evaluating 1001 + l * b - h / (b * h)");
13    putln;
14    put ("Result is ");
15    put (x);           # result should be 1051
16 done ComplexExpr;

```

## 5.3 Syntax

The following context free grammar gives the syntax of NoBeard. The well-known EBNF notation [Wir77] is used.



NoBeard	=	“unit” identifier “,” Block identifier “;”.
Block	=	“do” {Statement} “done”.
Statement	=	VariableDeclaration   Put   If   Assignment
VariableDeclaration	=	Type identifier [“=” Expression]“;”.
Type	=	SimpleType[ArraySpecification].
SimpleType	=	“int”   “char”   “bool”.
ArraySpecification	=	“[” number “]”.
Put	=	“put” “(” Expression [“,” Expression] “)” “;”   “putln” “;”.
If	=	“if” Expression Block [ “else” Block ].
Assignment	=	Reference “=” Expression “;”.
Reference	=	Identifier [ “[” Expression “]”].
Expression	=	AddExpression [RelOp AddExpression].
AddExpression	=	[AddOp] Term {AddOp Term}.
Term	=	Factor {MulOp Factor}.
Factor	=	Reference   number   string   “(” Expression “)”.
RelOp	=	“<”   “<=”   “==”   “>=”   “>”.
AddOp	=	“+”   “-”.
MulOp	=	“*”   “/”   “%”.

## 5.4 Semantics

Here a non-formal description of the semantics of NoBeard is given.

Put     =     “put” “(” Expression1  
              [“,” Expression2] “)” “;”.

Writes the value of *Expression1* to the output medium. If *Expression2* is given it defines the column width as follows: Integers are outputted as is. If the number of digits is less than *Expression2* the output is padded on the left. Characters are outputted as is. If *Expression2* is greater than 1 the output is padded on the right. Strings are outputted in the length of *Expression2*, i.e., they are truncated if longer than *Expression2* or padded on the right if shorter.

## Chapter 6

# Some Translations by Example

### 6.1 Reserving Space for Local Variables

### 6.2 Assignments

### 6.3 Boolean Expressions

We show the translation of a boolean expression  $a \parallel b \parallel c$  where  $a$ ,  $b$ , and  $c$  are variables of type `bool`. The sequence of several relational expressions or boolean variables connected via a boolean *or* is realized by a so-called or-chain. In particular, after evaluation of each single relational expression (or boolean variable) and this evaluation yields *true* all further evaluations are skipped and the program flow is continued at the end of the complete boolean expression. Figure 6.1 shows this principle. In order to keep the program flow simple, the load value parts in front of each evaluation are skipped. The more detailed NoBeard assembler code for this sequence is given in listing 6.1. Note that, for the sake of simplicity, the addresses given as operands to the `JMP` and `TJMP` instructions are the line numbers here. Of course, the “real” code generates the memory addresses of the targeted assembler instruction.

```
1  ...
2  LV 0, 32      ; load value a
3  TJMP 8        ; if true, jump to the end
4  LV 0, 36      ; load value b
5  TJMP 8        ; if true, jump to the end
6  LV 0, 40      ; load value c
7  JMP 9         ; result is determined by c only
8  LIT 1
9  ...
```

Listing 6.1: Assembler code of or-chain

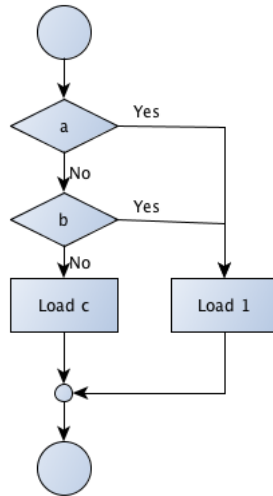


Figure 6.1: Program flow of an or chain

When generating this kind of code, we have to deal with the situation that the final addresses we have to jump to are not known in prior. Therefore, we have to construct a so-called or-chain, which work as follows. While parsing a conditional expression, we maintain an int variable holding the

The translation of *and*-expressions works analogously.

## Chapter 7

# Error Handling

```
ErrorHandler.getInstance().raise(new ...);
```

## Chapter 8

# Attributed Grammar

# Bibliography

- [Kha08] Tamir Khason. Computer languages and facial hair – take two, 2008.  
URL: <http://khason.net/blog/computer-languages-and-facial-hair-%e2%80%93take-two/>.
- [Ter04] Pat Terry. *Compiling With C# And Java*. Addison-Wesley Educational Publishers Inc, Harlow, England ; New York, 2004.
- [Wik16] Wikipedia. Two’s complement, July 2016. Page Version ID: 730322017. URL: [https://en.wikipedia.org/w/index.php?title=Two%27s\\_complement&oldid=730322017](https://en.wikipedia.org/w/index.php?title=Two%27s_complement&oldid=730322017).
- [Wir77] Niklaus Wirth. What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions? *Commun. ACM*, 20(11):822–823, November 1977.  
URL: <http://doi.acm.org/10.1145/359863.359883>, doi:10.1145/359863.359883.