A Formal Description of NoBeard

v 1.2

P. Bauer

HTBLA Leonding
Limesstr. 14 - 18
4060 Leonding
Austria

Revisions

Date	Author	Change
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	**Released v. 1.1**
June 5, 2012	P. Bauer	**Released v. 1.0**

Contents

1 Introduction **4**

2 The Programming Language **5**

2.1 Lexical Structure 5

2.1.1 Character Sets 5

2.1.2 Keywords 5

2.1.3 Token Classes 5

2.1.4 Single Tokens 5

2.1.5 Semantics 5

2.2 Sample Program 6

2.3 Syntax 6

2.4 Semantics 7

3 The NoBeard Machine **9**

3.1 Overview 9

3.2 Stack Frames 10

3.3 Runtime Structure of a NoBeard Program 10

3.4 Instructions 11

3.4.1 NOP 12

3.4.2 LIT 12

3.4.3 LA 13

3.4.4 LV 13

3.4.5 LC 13

3.4.6 STO 14

3.4.7 STC 14

3.4.8 ASSN 15

3.4.9 NEG 15

3.4.10 ADD 15

3.4.11 SUB 16

3.4.12 MUL 16

3.4.13 DIV 16

3.4.14 MOD 17

3.4.15 NOT 17

3.4.16 REL 17
3.4.17 FJMP 18
3.4.18 TJMP 19
3.4.19 JMP 19
3.4.20 PUT 19
3.4.21 INC 21
3.4.22 HALT 21

4 Symbol List **22**

5 Some Translations by Example **24**
5.1 Reserving Space for Local Variables 24
5.2 Assignments 24
5.3 Boolean Expressions 24

6 Error Handling **26**

7 Attributed Grammar **27**

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 Programming Language

2.1 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.

2.1.1 Character Sets

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

digit `'[0-9]'`

2.1.2 Keywords

There is only one keyword, namely `PUT`.

2.1.3 Token Classes

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

number `['digit digit*]`

2.1.4 Single Tokens

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

2.1.5 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.

2.2 Sample Program

```
unit ComplexExpr;
# ----------------- ComplexExpr.nb ----------------------
# --- A syntactically correct NoBeard program
# --------------------------------------------------------
do
    int l = 10;
    int b =5;
    int h= 170;
        int unused = l;
    int x=1001 + l * b - h / (b * h);

    put ("Evaluating 1001 + l * b - h / (b * h)");
    putln;
    put ("Result is ");
    put (x);          # result should be 1051
done ComplexExpr;
```

2.3 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                 =  “*” | “/” | “%”.
```

2.4 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 then *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 3

The NoBeard Machine

3.1 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 4 bytes.

`prog[MAX_PROG]` The *program memory* which is byte addressed.

`dat[MAX_DATA]]` The *data memory* is byte addressed and is separated into two parts:

- String constants
- Stack frames of the currently running functions

Figure 3.1 shows this. Before a program is started the string constants are stored in 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 3.2.
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.

`top` Address of the start of the last used word on the stack.

`db` *Data Base*: Address of the first byte of the currently running function's stack frame.

`pc` *Program Counter*: Start address of the next instruction in `prog` to be executed.

`ms` *Machine State*: The NoBeard machine may have three different states:

- `run`: The machine runs
- `stop`: The machine stops. Usually when the end of program is reached.
- `error`: Error state

3.2 Stack Frames

3.3 Runtime Structure of a NoBeard Program

The NoBeard Machine follows the following fixed execution cycle:

1. Fetch instruction
2. Decode instruction

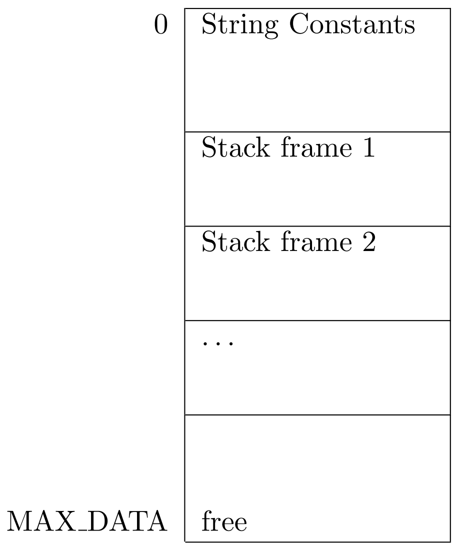

Figure 3.1: Data Memory of the NoBeard Machine

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

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

3.4 Instructions

NoBeard instructions have a variable length. Every instruction has one 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.

3.4.1 NOP

Instruction

Byte 0
0x00

Operation

Empty instruction. Does nothing

```
NOP
```

3.4.2 LIT

Instruction

Byte 0	Byte 1	Byte 2
0x01	Literal	

Operation

Pushes a value on the expression stack.

```
LIT Literal
push(Literal);
```

3.4.3 LA

Instruction

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

Operation

Loads an address on the stack.

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

3.4.4 LV

Instruction

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

Operation

Loads a value on the stack.

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

3.4.5 LC

Instruction

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

Operation

Loads a character on the stack.

```
LC Displacement DataAddress
base = db;
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);
```

3.4.6 STO

Instruction

Byte 0
0x07

Operation

Stores a value on an address.

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

3.4.7 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;
```

3.4.8 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];
```

3.4.9 NEG

Instruction

Byte 0
0x0B

Operation

Negates the top of the stack.

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

3.4.10 ADD

Instruction

Byte 0
0x0C

Operation

Adds the top two values of the stack.

```
ADD
push(pop() + pop());
```

3.4.11 SUB

Instruction

Byte 0
0x0D

Operation

Subtracts the top two values of the stack.

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

3.4.12 MUL

Instruction

Byte 0
0x0E

Operation

Multiplies the top two values of the stack.

```
MUL
push(pop() * pop());
```

3.4.13 DIV

Instruction

Byte 0
0x0F

Operation

Divides the top two values of the stack.

```
DIV
y = pop();
x = pop();
push(x / y);
```

3.4.14 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);
```

3.4.15 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);
```

3.4.16 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)
- 0 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;
}
```

3.4.17 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;
```

3.4.18 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;
```

3.4.19 JMP

Instruction

Byte 0	Byte 1	Byte 2
0x18	NewPc	

Operation

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

```
JMP newPc
pc = NewPc;
```

3.4.20 PUT

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;
}
```

3.4.21 INC

Instruction

Byte 0	Byte 1	Byte 2
0x1D	Size	

Operation

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

```
INC Size
top += Size;
```

3.4.22 HALT

Instruction

Byte 0
0x1F

Operation

Halts the machine.

```
HALT
ms = stop;
```

Chapter 4

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	VARKIND	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	VARKIND	SIMINT	4	36	2
B	PROCKIND	PROCTYPE	0	0	2
c	PARKIND	SIMCHAR	1	32	3
d	VARKIND	SIMINT	4	36	3

Chapter 5

Some Translations by Example

5.1 Reserving Space for Local Variables

5.2 Assignments

5.3 Boolean Expressions

We show the translation of a boolean expression `a || b || 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 5.1 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 5.1. 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 5.1: Assembler code of or-chain

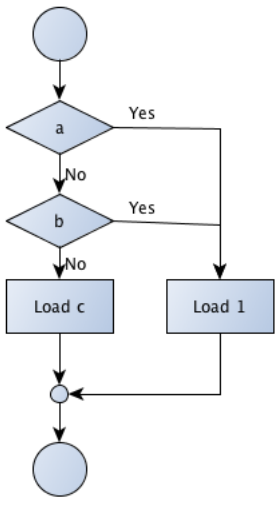

Figure 5.1: Program flow of an or chain

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 6

Error Handling

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

Chapter 7

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%93-take-two/`.

[Ter04] Pat Terry. *Compiling With C# And Java.* Addison-Wesley Educational Publishers Inc, Harlow, England ; New York, 2004.

[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`.