A Formal Description of NoBeard

v 01.02.00

P. Bauer

HTBLA Leonding
Limesstr. 14 - 18
4060 Leonding
Austria

Contents

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” StatementSequence “done”.
StatementSequence        =   {Statement}.

Statement                =   VariableDeclaration
                         |   Put
                         |   If
                         |   Assignment
                         |   ε.

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

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. The word width of the NoBeard machine is 4 bytes.

dat[**MAX_DATA**] The data memory is byte addressed and is separated into three parts: String constants, activation records, and expression stack. Figure

The expression stack is addressed word-wise only.

3.2 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) {
  db = start byte of first free word in dat;
  top = db + 28;
  pc = startPc;
  ms = run;

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

3.3 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:

Name	Range	Size	Description
n	0 ... 65535	2 Bytes	Unsigned number
b	0 ... 65535	2 Bytes	Code address (absolute number)

3.4 List of Assembler Instructions

Op code	Mnem-onic	ops	Description	Size
0	LIT	n	Load Literal `Push(n);`	3 Bytes

1	LA	d, a	Load Address
			`base = db;`
			`for (i= 0; i < d; i++) {`
			`  base = dat[base ... base + 3];`
			`}`
			`Push(base + a);`
				4 Bytes

2	LV	d, a	Load Value
			`base = db;`
			`for (i = 0; i < d; i++) {`
			`  base = dat[base ... base + 3];`
			`}`
			`adr = base + a;`
			`Push(dat[addr ... addr + 3]);`
				4 Bytes

3	LC	d, a	Load Character
			`base = db;`
			`for (i = 0; i < d; i++) {`
			`  base = dat[base ... base + 3];`
			`}`
			`lw = 000dat[base + a];`
			`Push(lw);`
			The local word to be pushed onto the stack is filled with three trailing binary zeros following the character to be loaded from `dat`.
				4 Bytes

Chapter 4

Symbol List

```
unit M;
  function A(int a);
    int b;

    int function B(char c);
      int d;
    do
        # some code
    done B;

    char function C;
      int e;
    do
      # some more code
    done C;
  do
    # some code on A
  done A;
do
  # this is the main of unit M
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

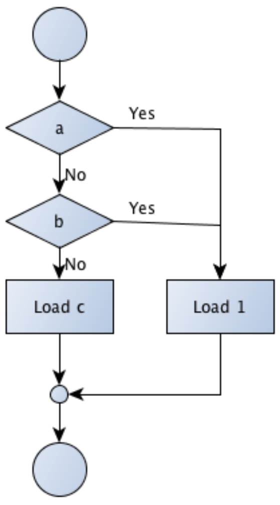

Figure 5.1: Program flow of an or chain

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

NoBeard	=		sem
			EmitOp(INC);
			Emit2(0);
			int inc_addr = 1;
			endsem
		Stat ";" {Stat ";"}.	
Stat	=	ident ":=" $\text{Expr}_{\uparrow \text{op}}$ |	
		"PUT" $\text{Expr}_{\uparrow \text{op}}$.	
$\text{Term}_{\uparrow \text{op}}$	=	$\text{Fact}_{\uparrow \text{op}}$	
		{("*"	sem opcode = mul endsem
		| "/"	sem opcode = div endsem
		)	semEmitOp(↓opcode)endsem
		$\text{Fact}_{\uparrow \text{op}}$	semLoadVal(↓op)endsem

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`.