A Formal Description of NoBeard

v 1.2

P. Bauer

HTBLA Leonding
Limesstr. 14 - 18
4060 Leonding
Austria

Revisions

Date	Author	Change
March 6, 2019	P. Bauer	Updated naming schemes for NoBeard identifiers and Assembler labels. Documentation of assembler instruction `in` is corrected.
September 21, 2016	P. Bauer	Added instructins in.
September 2, 2016	P. Bauer	Added instructins lci, lvi.
August 15, 2016	P. Bauer	Added introduction to assembler.
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	**Released v. 1.1**
June 5, 2012	P. Bauer	**Released v. 1.0**

Contents

Chapter 1

Introduction

1.1 The NoBeard Project

This project aims to provide students who want to dig into the fields of assembler programming, system programming and compiler construction a playground where they can experiment in a pretty small and clean environment. The components developed here do not have the quirks and compromises real machines, assemblers, and compilers have to make in order to keep up with their real life requirements. In particular the following parts are provided:

- The *NoBeard Machine*. A virtual machine with an instruction set of less than 30 instructions which is pretty easy to grasp compared to the instruction set sizes of real life machines. The machine is purely stack based such that the structure of each instruction is again easy to understand and to follow.
- The *NoBeard Assembler*. To write programs for the NoBeard Machine an assembler is provided. In the published version of the project no support of labels, symbolic variables, etc. is given. Here the focus is to direct the students to the very basics of how machine programs for the NoBeard Machine look like and how they work. Extensions as the ones mentioned above can be done by the interested student as an exercise.
- The *NoBeard Compiler*. To facilitate programming for the NoBeard Machine and to give insights into the basic techniques of compiler construction a dedicated language is defined and the corresponding compiler is implemented. We did not go for the automated construction of compilers with compiler generators since we wanted to emphasize the direct relationship of formal grammars to parsers using the recursive descending method. For students interested in compiler construction using compiler generators the great book of Pat Terry [Ter04] is recommended.

1.2 Trivia and Acknowledgements

The NoBeard tools are used to support the courses *Technical Informatics* and *Programming and Software Engineering* (in particular the part *Theoretical Informatics*) held for third grade students of the Department of Informatics at the HTBLA Leonding. Therefore the students of these grades are forced to go through this material in more detail. All remarks from students of the former years helped to improve the project and to bring it into the shape as it is today. A big Thank You at this point is the least one can do in return for that. Since the whole project is still ongoing: In case of typos, misleading wording or other problems, please feel free to contact the author. Any help in this direction is highly appreciated.

According to a web article (see [Kha08]) the popularity of programming languages is strongly related to the fact whether its inventor(s) is a / 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 assembler programming, system programming, and compiler construction are. This is the background of the name of the whole project. Thanks to my former colleagues Peter Penz, Hugo Koblmüller, and Harald Wick for their constant friendship and creativity which not only helped me to find a proper name for this project.

When developing this project we had an extremely helpful guideline in the *MiniModula2* project developed by Hanspeter Mössenböck at the Johannes Kepler University in Linz [MRR91]. Actually all basic concepts used here are stolen from this project.

1.3 Structure

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

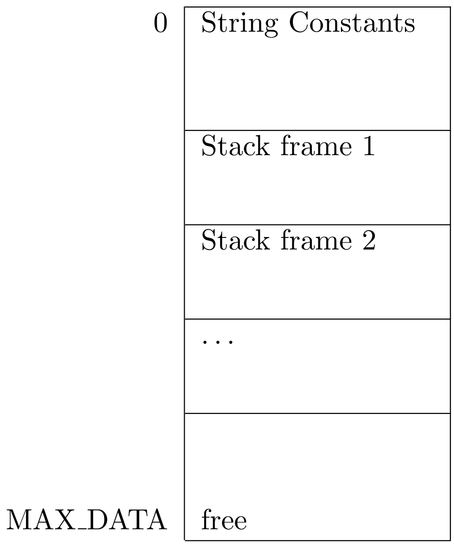

Figure 2.1: Data Memory of the NoBeard Machine

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:

Address	Content	Remark
0	0	frame pointer of frame 0
	...	
32	13	local int in frame 0
36	0	static link to frame 0 (start of frame 1)
	...	
68	17	local int in frame 1
72	42	local int in frame 1
76	36	static link to frame 1 (start of frame 2)
	...	
108	'D'	
109	61	
MAX_DATA	free	

Figure 2.2: Snapshot of a call stack with three frames

`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

2.2 Stack Frames

Considering a concrete scenario as given in figure 2.2 we have three frames. Frame 0 starts at address 0. The first 32 bytes of each frame are reserved for administrative data like the static link and the dynamic link to the surrounding frame, the return value, etc. Address 32 holds the value of a local variable in frame 0.

At address 36 frame 1 starts with the address to its statically surrounding frame, i.e, the function (or unit or block) represented by frame 0 is defining the function (or block) represented by frame 1. Frame 1 defines two local values at addresses 68 and 72. Now it can be easily verified that an assembler instruction `la 0 32` (see section 2.5) loads the

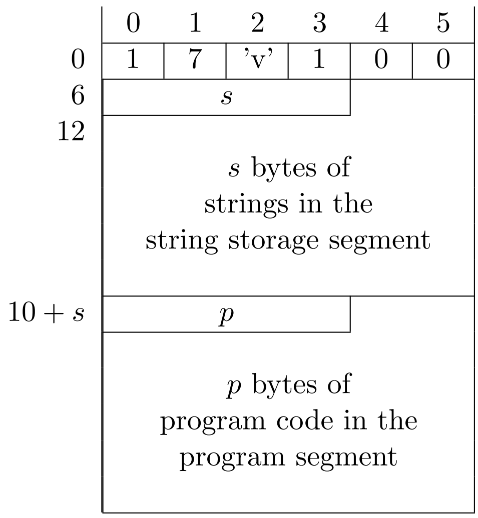

Figure 2.3: NoBeard Binary File Format

value of the local (i.e., relative to the closest frame pointer) address 32. In the concrete example this is address $36 + 32 = 68$.

2.3 Binary File Format

NoBeard binaries have the extension `.no` (from NoBeard object file) and consist of three segments, a header segment, a string storage segment and a program segment.

The header segment takes the first six bytes and contains a file identifier (the digits 1 and 7, each stored in one byte) followed by the file format version. This version information starts with the character `'v'` followed by three bytes. Figure 2.3 shows an example of such a file. From bytes 3 to 5 it can be seen that this is a file in version 1.0.0.

The string storage segment starts with a four byte number s holding the length of the string storage segment and then follows a bunch of s bytes holding the characters of the strings stored.

The final program segment is organized analogously to the string storage segment. Again a four byte number p is followed by p bytes of program code.

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[addr ... addr + 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 lvi

Instruction

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

Operation

Loads a value indirectly on the stack.

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

```
}
adr = base + DataAddress;
ra = dat[addr ... addr + 3];
push(ra);
```

2.5.7 lci

Instruction

Byte 0	Byte 1	Byte 2	Byte 3
0x05	Displacement	DataAddress	

Operation

Loads a character indirectly on the stack.

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

2.5.8 sto

Instruction

Byte 0
0x07

Operation

Stores a value on an address.

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

2.5.9 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;
```

2.5.10 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];
```

2.5.11 neg

Instruction

Byte 0
0x0B

Operation

Negates the top of the stack.

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

2.5.12 add

Instruction

Byte 0
0x0C

Operation

Adds the top two values of the stack.

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

2.5.13 sub

Instruction

Byte 0
0x0D

Operation

Subtracts the top two values of the stack.

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

2.5.14 mul

Instruction

Byte 0
0x0E

Operation

Multiplies the top two values of the stack.

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

2.5.15 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();
```

2.5.16 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);
```

2.5.17 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);
```

2.5.18 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;
}
```

2.5.19 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;
```

2.5.20 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;
```

2.5.21 jmp

Instruction

Byte 0	Byte 1	Byte 2
0x18	NewPc	

Operation

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

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

2.5.22 in

Instruction

Byte 0	Byte 1
0x19	Type

Operation

Reads data from the terminal. Depending on `Type` different data types are read:

- 0: Attempt to read an `int`. After execution the value 1 is stored to the address located on top of stack if reading was successful, otherwise 0 is stored. In case of success the value read is stored on the address located right under the top of stack.
- 1: A `char` is read and stored at the address on top of the stack. After execution the value 1 is pushed
- 2: a `string` with a specific length is read

```
in Type
switch(Type) {
    case 0:
        s = pop();
        a = pop();
        n = readInt(&readOk);
        if (readOk) {
            dat[a ... a + 3] = n;
            dat[s ... s + 3] = 1;
        } else {
            dat[s ... s + 3] = 0;
        }
        break;
```

```
        case 1:
            a= pop();
            ch = readChar();
            dat[a] = ch;
            break;
        case 2:
            a = pop();
            strLen = pop();

            break;
    }
```

2.5.23 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

```
    out 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;
   }
```

2.5.24 inc

Instruction

Byte 0	Byte 1	Byte 2
0x1D	Size	

Operation

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

```
inc Size
top += Size;
```

2.5.25 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	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 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 and labels. 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.

Labels are identifiers to name jump addresses and follow the following naming scheme:
`(letter | '.'){letter | digit | '.' | '_'}`

4.2 Examples

4.2.1 First Example

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.

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

Listing 4.1: Small and useless assembler program

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. Therefore the above program can be rewritten in the following way.

```
nop halt
```

Listing 4.2: Shorter way to write a small and useless program

4.2.2 String Handling and Output

A NoBeard assembler program writing “Hello World” into the console could look like as follows. The strings `Hello` and `World` must be specified at the very beginning of the program. We simply write the strings needed in a sequence, without any separator or similar. The instruction `out` is used to produce output. When calling `out` with operand `2` (see line 6) a string is printed. The instruction expects to have the address of the string, its length and the column width on the stack.

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

Listing 4.3: “Hello World” in NoBeard Assembler

Note that the width of the column when writing “Hello” is set one character wider than the length of the string (line 5). 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. Further details about outputting can be found in section 2.5.23.

4.2.3 Variables, Reading Integers from Terminal and Basic Arithmetic

As already described in section 2.2 local variables of the currently running function are stored on the stack frame. In order to be able to do so, we have to reserve memory for local variables. This is done by the `inc` instruction. Additionally we have to keep in mind that for each stack frame the NoBeard machine reserves 32 bytes for frame house keeping tasks. This is the reason why the following program accesses then the local variable at address 32. For further details about the semantics of the assembler instructions used here we refer to section 2.5.

```
# Define one variable and output it
inc 4   # Reserve 4 bytes for local integer value
la 0 32 # Load address 32 where the 4 bytes were reserved
lit 17  # Load the number 17
sto     # Store the number 17 at address 32
lv 0 32 # Load the value stored at address 32
lit 1   # Load the column width
out 0   # Output an integer value
out 3   # Output a new line
halt
```

Listing 4.4: A program defining one local variable

Maybe it appears superfluous to reload the value from address 32 before it can be printed to the output. The reason for this extra load instruction becomes clear if one studies the semantics of the `sto` instruction. Since it removes the value to be stored from the stack it has to be reloaded before it can be printed.

Since doing calculations is only of limited fun if the corresponding values cannot be entered by the user we show here how to read values provided by the user via the `stdin`. In order to do so, NoBeard assembler provides an instruction `in`. This instruction expects two addresses on the stack, one for holding the value inputted by the user and another one holding a success flag indicating whether reading the value succeeded or not. The following program reserves space for these two variables, reads an integer value from the user and finally outputs this value and the success flag.

```
# Reads an integer value from the
# stdin and prints it to the stdout.
"Enter an int:You entered:Read operation successful?"
inc 8     # reserve space for two variables

# print "Enter an int: "
lit 0     # address
lit 13    # string length
lit 14    # col width
out 2     # out string

# read int from stdin
la 0 32   # addr to store int
la 0 36   # addr to store success flag
in 0      # input int

# print "You entered: "
lit 13
lit 12
```

```
lit 13
out 2

# print value entered
lv 0 32    # load int inputted before
lit 1      # load column width
out 0      # out int
out 3      # new line

# print "Read operation successful? "
lit 25
lit 26
lit 27
out 2

# print success flag
lv 0 36
lit 1
out 0
out 3
halt
```

Listing 4.5: A program reading an input value

The value of the success flag can be verified by providing a valid integer value on a first run and entering an invalid integer (e.g., an alphanumeric character) on a second run.

The assembler provides a complete set of arithmetic operations such that one can express arbitrary arithmetic expressions. To illustrate this we want to write an assembler program which calculates the expression `((b + 11)/a) % 2` for some previously stored values `b` and `a != 0`. The result will be stored in some extra variable `c` which will then finally be printed.

```
# Doing calculations
# Define three variables a, b, c
# Store the values 17 and 42 in a and b
# Calculate ((b + 11) / a) % 2
# Store result of above calculation to c
inc 12      # reserve space for variables a, b, c
la 0 32     # load address of a
lit 17      # load value for a
sto         # store a

la 0 36
lit 42
sto         # b
```

```

la 0 40     # load address of c
lv 0 36     # load value of b
lit 11      # load 11
add         # b + 11
lv 0 32     # load value of a
div         # (b + 11) / a
lit 2       # load 2
mod         # ((b + 11)/a)%2
sto         # store result to c

lv 0 40     # load value of c
lit 1       # load column width for out
out 0       # ouput int value
out 3       # output newline
halt
```

Listing 4.6: Basic Arithmetic

Regarding the arithmetic instructions it can be seen that all of these expect their operand(s) to be pushed on the stack. During their calculation they remove these from the stack and push the result of the calculation back to the stack.

4.2.4 Branching

The NoBeard assembler language provides three branch instructions, namely `tjmp`, `fjmp` and `jmp`, which jump to a destination address in case of the top value of the stack being non-zero, zero, or unconditionally, respecitvely. In order to prevent the programmer from counting the instruction sizes to get the correct jump destination fixed, the NoBeard assembler provides so-called labels as jump destinations. In the following example we want to show how to check for a given number whether it is odd or even.

```
# Shows a selection
# Prints whether given number is odd or even
"Please enter a number:The number is evenThe number is odd"
inc 8      # reserve space for local variables:
           # 32: value entered by the user;
           # 36: flag whether reading int was ok
# print "Please enter a number: "
lit 0
lit 22
lit 23
out 2

# read input from stdin
```

```
la 0 32
la 0 36
in 0

lv 0 32    # number to be checked
lit 2
mod        # calculate 17 % 2
lit 0      # check against value 0
rel 2      # 2->equals: check whether 17%2 equals 0
fjmp .output_odd   # if not jump down to output "odd"

# start of output "even"
lit 22
lit 18
lit 19
out 2
jmp .end     # the if branch is finished -> jump to end

.output_odd # jmp label where output "number is odd" starts
lit 40
lit 17
lit 18
out 2

.end
out 3     # output new line
halt      #
```

Listing 4.7: Odd/Even Checker

4.3 Formal Description

4.3.1 Opcodes

The opcodes are exhaustively described in section 2.5. They can be taken directly from the first word of the first line of the formal operation description.

4.3.2 Token Classes

Token classes are described in form of a regular expression.

letter `[a-zA-Z]`

digit `[0-9]`

label `\.letter(letter|digit|_)*`

number `digit digit*`

string : `'"' {nodoublequote} '"'`

4.3.3 Syntax

NoBeardAssembler	=	[string]{AssemblerInstruction}.
AssemblerInstruction	=	label | opcode [Operands].
Operands	=	label | 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 Token Classes

identifier `(letter | '_' | '$'){letter | digit | '_' | '$'}`

number `digit{digit}`

string `'"' {nodoublequote} '"'`

5.1.2 Keywords

1. `bool`
2. `char`
3. `else`
4. `false`
5. `if`
6. `int`
7. `true`

5.1.3 Single Tokens

1. `"&&"`
2. `"||"`
3. `"!"`
4. `"+"`
5. `"-"`
6. `"*"`
7. `"/"`
8. `"%"`
9. `"<"`
10. `"<="`
11. `"=="`
12. `"!="`

13. ">="
14. ">"
15. "&"
16. ";"
17. ","
18. "{"
19. "}"
20. "("
21. ")"
22. "="

5.1.4 Comments

Comments are treated like whitespace characters (space, tab, cr/lf). A comment starts with # and lasts until the end of the current line.

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

```
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;
```

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
                      |  If
                      |  Put
                      |  Get
                      |  Assignment

VariableDeclaration   =  Type identifier [“=” Expression]“;”.
Type                  =  SimpleType[ArraySpecification].
SimpleType            =  “int” | “char” | “bool”.
ArraySpecification    =  “[” number “]”.

If                    =  “if” Expression Block [ “else” Block ].

Put                   =  “put” “(” Expression [“,” Expression] “)”“;”
                      |  “putln” “;”.

Get                   =  “get” “(” identifier [“,” identifier] “)” “;”.
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 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.
Get	=	“get” “(” identifier1 [“,” identifier2] “)” “;”.	Reads an integer value from the stdin and stores it into *identifier1*. If *identifier2* is given it must be of type `bool` and gets `true` if an integer value has been read successfully (and, therefore it has been stored into *identifier1*) and `false` otherwise. In case *identifier2* is not given there is no feedback about success of the read operation and in case of no success *identifier1* has an undefined value.

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` || `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 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

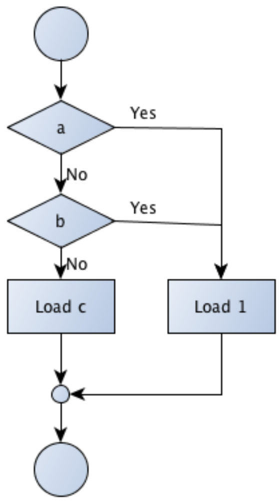

Figure 6.1: Program flow of an or chain

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%93-take-two/.

[MRR91] Hanspeter Mössenböck, Peter Rechenberg, and Christoph Reichenberger. Formale Beschreibung der Programmiersprache MiniModula2. Technical report, Johannes Kepler Universität Linz, Institut für Informatik, Abteilung Software, Linz, September 1991.

[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.

[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.