

Befunge-93 interpreter in Haskell

A project for the course Program Design and Data Structures (1DL201) at Uppsala University

Staffan Annerwall, Patrik Johansson, Erik Rimskog

February 24, 2017

Abstract

This document serves as documentation for the interpreter as well as an introduction to Befunge-93. We describe how to run the interpreter and give a comprehensive list of all supported instruction characters. We give examples of Befunge programs and discuss the implementation, specifically data structures and algorithms. Finally we mention shortcomings and caveats of the interpreter.

Contents

1	Introduction	3
1.1	What is Befunge-93?	3
1.1.1	The stack	3
1.1.2	The program space	3
1.1.3	Program flow and the Program Counter	3
2	Running the interpreter	4
2.1	Dependencies	4
2.2	Linux	4
2.3	Windows	4
2.4	Command-line arguments	4
3	Instruction list	5
4	Example programs	5
4.1	Basic arithmetic	5
4.2	Moving around	6
4.3	Input and output	6
4.4	If-statements	6
4.5	getting and putting values	7
4.6	Determining prime numbers	7
5	Implementation	8
5.1	Data structures	8
5.1.1	Stack	8
5.1.2	Program space	9
5.2	Algorithms	9
5.3	Major functions	9
5.3.1	Program flow	9
6	Shortcomings and caveats	9

1 Introduction

We introduce the programming language Befunge-93 as well as its stack, program space and program flow.

1.1 What is Befunge-93?

Befunge-93 (sometimes in this document simply “Befunge”) is a two-dimensional esoteric programming language developed by Chris Pressey in 1993 [1]. The program to be executed is stored in a 80 by 25 grid, the “program space”, where each cell can hold 1 byte of data. There are no variables available but data can be stored in either a LIFO-stack (last-in-first-out) or, by using the `p` instruction, in the program space itself. This allows for the program to modify itself while running. See 4.5 “getting and putting values”.

Befunge-93 was the first iteration of several Befunge-specifications which eventually lead to Funge-98, a generalization of in which program spaces can be either one-, two-, or three-dimensional. Funge-98 also provides a paradigm for program spaces in an arbitrary number of dimensions [2]. Besides new dimensions, Funge-98 also specifies several new instructions as well as concurrent program flow and a stack of stacks.

1.1.1 The stack

The stack has the property that it is never truly empty. Instead, if an attempt to pop a value of the stack is made when the stack is thought to be empty, a value of 0 is returned. The stack is in all but this one regard a regular LIFO-stack.

When referring to the stack this document will sometimes use the notation $\langle a_n, \dots, a_1 \rangle$ where a_1 is the top-most value of the stack.

1.1.2 The program space

The program space consists of 2,000 cells arranged in a 80 by 25 grid. The upper left corner of this grid is given the coordinate position (0,0) and the lower right corner is identified as (79,24), see Figure 1.

The grid can topologically be thought of as a torus (the surface of a dough-nut); should the PC at any point try to move outside the bounds of the program space it “wraps around” to the other side. For example, assuming that indexing starts at 0, if the Program Counter (see section 1.1.3) is at position (6, 24) and attempts to move South, its new position would be (6, 0), still facing South.

In this document we will use “program space”, “space”, “program memory” and “memory” synonymously.

1.1.3 Program flow and the Program Counter

Program flow is determined by the position (in the program space) and direction of a unique Program Counter, often simply called the “PC”. The position is usually represented as a pair (x, y) of coordinates and the direction is always one of either East,

(0, 0)	(1, 0)	...	(79, 0)
(0, 1)	(1, 1)	...	(79, 1)
⋮	⋮	⋱	⋮
(0, 24)	(1, 24)	...	(79, 24)

Figure 1: A visualization of the program space in Befunge where each cell is labeled with its coordinates.

South, West or North. Traveling East increases the x -component of the PC's position, and traveling West decreases it. Traveling South or North works analogously.

The PC starts at position (0, 0) – the upper-left corner of the program space – and has an initial direction of East. Execution of any Befunge program then consists of three simple steps:

1. Read the character at the PC's position in the program space.
2. Execute the instruction corresponding to the character that was read.
3. Step the PC one step in its direction.

These three steps are repeated until an @-character is read at which point the program immediately terminates. See section 3 for more information on instructions.

2 Running the interpreter

empty

2.1 Dependencies

empty

2.2 Linux

empty

2.3 Windows

empty

2.4 Command-line arguments

empty

3 Instruction list

Befunge-93 specifies a total of 35 actions, 10 of which simply push the numeric values 0 – 9 to the stack, each represented by an ASCII character. Arithmetic is executed using Reverse Polish Notation (RPN).

Table 1 contains a complete list of all characters that the interpreter recognizes. If the PC encounters an unrecognized character, it ignores it completely – as if it were a space (␣).

Character	Action
0 – 9	Push the desired value onto the stack.
+	Pop b and a off the stack, then push a + b .
-	Pop b and a . Push a – b .
*	Pop b , a . Push a · b .
/	Pop b , a . Push 0 if b = 0, otherwise the integer part of a/b .
%	Pop b , a . Push a mod b .
`	Pop b and a . If a > b then push 1, otherwise push 0.
!	Pop a and if a is non-zero then push 0, otherwise push 1.
>	Instruct the PC to move East.
v	Instruct the PC to move South.
<	Instruct the PC to move West.
^	Instruct the PC to move North.
?	Instruct the PC to move in a random cardinal direction.
#	Skip the next instruction; the PC moves twice.
_	Pop a and instruct the PC to move West if a ≠ 0, otherwise East.
	Pop a and instruct the PC to move North if a ≠ 0, otherwise South.
:	Duplicate the top value of the stack.
\	Swap the top two values of the stack.
\$	Pop a value off the stack and discard it.
g	Pop y and x , then push the value of the character at position (x , y).
p	Pop y , x and v , then insert v at position (x , y) in the memory.
&	Wait for user value input and push it.
~	Wait for user character input and push its ASCII value.
.	Pop a value and print it, followed by a space (␣).
,	Pop a value and print its ASCII character.
␣	Spaces are ignored, the PC continues and no other modifications are made.
@	Terminate the program.

Table 1: All instructions supported by the interpreter.

4 Example programs

4.1 Basic arithmetic

Addition (+), subtraction (-), multiplication (*), division (/) and modulo (%) execute using Reverse Polish Notation (RPN). That is, the program 65- . prints 1 (and not -1).

Divison by 0 pushes 0 back to the stack regardless of the value of the numerator.

An example that utilizes all of these instructions is the program `329*+2*7/4%.@`. It first pushes 3, 2 and 9 to the stack followed by a multiplication of the top two values making the stack look like $\langle 3, 18 \rangle$, where the right-most value is the top of the stack. The `+` instruction adds the top two values on the stack; the stack now contains a single value of 21. This is then multiplied by 2 and divided by 7 resulting in the stack $\langle 6 \rangle$. Finally, a 4 is pushed and the modulo operation `%` calculates $6 \bmod 4 = 2$ and pushes it. The top value on the stack, now 2, is printed by `.` and the program stops after reading the `@`-character.

4.2 Moving around

There are 7 instructions that can change the direction of the Program Counter: `>`, `v`, `<`, `^`, `?`, `_` and `|`. We explain what `_` and `|` do in section 4.4 “If-statements”. The first four instructions should be thought of as arrows; they change the direction of the PC so that it moves in the direction indicated by these “arrows”. The questionmark symbol `?` randomizes the direction of the PC to one of four cardinal directions (i.e. one of East, North, South or West).

An example of the four basic move-instructions is shown in Figure 2. It makes use the the wrap-around property of the program space and never terminates, despite having a termination symbol in the center.

```
< ^  
@  
v >
```

Figure 2: A never-ending Befunge program.

Infinite loops are indeed easy to create; simply force the PC in a path that does not contain any termination symbol.

4.3 Input and output

Input can be asked for by `&` for numerical values and `~` for single-character inputs. Output is given by `.` and `,` for numeric and ASCII-characters respectively. The program `~.@` asks for a character input and prints its numerical value before terminating. The program `&: !#@_.` asks for numerical inputs and echoes whatever the user entered until the users enters 0.

When an input is asked for, the program pauses and `>>` is printed to the console to indicate that a value (or character) should be entered. After a value (or character) is entered, the program resumes. Should more than one value be entered, everything but the first value is ignored.

4.4 If-statements

The characters `_` and `|` are conditionals; one value is popped off the stack the PC changes direction to West and North respectively if the value is non-zero, otherwise East and South, respectively.

If-statements can be used either horizontally using `_` or vertically using `|`. Both instructions pop one value off the stack and instructs the PC to move West/North if the value is non-zero, and otherwise East/South.

Discarding all values from the stack until a 0 is on top is hence very easy: `>_@`. If a string is pushed onto the stack it is also easy to print it, as shown in Figure 3.

```
"egnufeB">:#v_@
^ ,<
```

Figure 3: Printing a string in Befunge. The program prints **Befunge**. Note that the string is pushed reversely onto the stack and must end with a 0 on the stack.

4.5 getting and putting values

Values from the program space can be retrieved using the `g`-command `g`; it pops the values `y` and `x` from the stack and pushes whichever character is at position `(x, y)`. The put-instruction `p` pops `y` and `x` as well as a third value `v`, and then inserts `v` at position `(x, y)` in the program space.

These instructions can make for particularly confusing code, for example the program `444**00p` (or equivalently `4: **0:p`) which at first glance does not seem to terminate. However, `444**` pushes 64 (ASCII character `@`) to the stack which is then put into the program space at position `(0, 0)` by `00p`. The program then wraps around the right “edge” of the program space and appears at `(0, 0)` where there is now a termination symbol, `@`, and hence does terminate.

4.6 Determining prime numbers

A more complicated example is one that takes a number and determines whether it is a prime number or not. One solution, designed to be obscure and hard to read, is shown in Figure 4.

The very first row asks for a numeric input, say n , and outputs 0 (indicating “False”) if $n \leq 1$. The value is then compared to 3 on line two and if $n \leq 3$ (i.e. n is either 2 or 3) then 1 (“True”) is printed. If $n > 3$ the PC now travels South in the left-most column. $r_2 = n \bmod 2$ is calculated and if $r_2 = 0$ (meaning n is an even number) then the program terminates after printing 0.

If the program has not yet terminated we know that n is an odd integer such that $n \geq 5$. The value n is stored at position `(0, 0)` in the program space (replacing the initial `&`-character) and then the main primality testing loop begins (at position `(4, 4)`) with a single value of 1 on the stack (pushed at position `(3, 3)`).

When the loop starts it increments whatever is on the top of the stack by 2 and duplicates the new value, call it k . It then uses the `g`-command to get the value on position `(0, 0)`, i.e. the number n we stored earlier. The two top-most values on the stack are swapped (the stack is now equal to $\langle k, n, k \rangle$) and a comparison $n > k$ is performed using the ```-character. If the comparison $n > k$ is false (i.e. $n \leq k$) then 1 is printed and the program terminates (after wrapping around to position `(0, 10)`).

```

&::1`#v_0.@
v_v#`3<
: >1.@
2>p1v
%:v2<#<
#0+
>^:
|>0$%#^_ $
$\:
.^g:0:<
@ >\`#^_1.$

```

Figure 4: A primality test in Befunge, designed to be obscure.

Otherwise, if $n > k$, then the program duplicates the stack’s top value, pushes the stored n again (using `g`) and swaps the stack’s top elements (making the stack look like $\langle k, n, k \rangle$ again). We now calculate $n \bmod k$ and terminate with a printed 0 if $n \equiv 0 \bmod k$. Otherwise the PC goes back to the beginning of the loop and increments the top value of the stack by 2 again.

Effectively what happens, after checking base cases, is that we calculate $r_k = n \bmod k$ and ensure that it is non-zero for all odd $3 \leq k < n$.

5 Implementation

In which we discuss how the interpreter is implemented, including the data structures and algorithms used.

5.1 Data structures

`empty`

5.1.1 Stack

The Befunge stack, defined in `BStack.hs`, is implemented using a list of integers. It exports one identifier (`empty`) and three functions (`push`, `pop`, `top`). The type `BStack` is defined as a list of integers: `newtype BStack = BStack [Int] deriving (Show)`. The identifier `empty` (of type `BStack`) is identified simply as `BStack []`.

`push (BStack -> Int -> Bstack)` takes a (possibly empty) stack `s` and an integer `n`, and returns `s` with `n` on top.

`pop (BStack -> (BStack, Int))` takes a stack `s` and returns a tuple `(s', t)` where `t` is the top element of `s` and `s'` is `s` with `t` removed. If `s` is empty, a tuple `(BStack [], 0)` is returned instead.

`top (BStack -> Int)` takes a stack and returns its top element without modifying the stack.

All three functions have precondition `True`.

5.1.2 Program space

The Befunge program space, the “memory”, is constructed using an array. `BMemory.hs` is a wrapper for an `IOArray` (from `Data.Array.IO`) and the type `BMemory` is synonymous to `IOArray Position Char`. It exports three functions:

`buildArray (BMemory -> [String] -> IO ())` allocates an array and populates it with the characters in a list of strings. Any characters outside the bounds (defaults to 80×25) are ignored.

`getValue (BMemory -> Position -> IO Char)` takes an array and returns the character at the given position.

`putValue (BMemory -> Position -> Char -> IO ())` takes an array, a position and a value. It inserts the value at the given position and returns nothing.

Both `getValue` and `putValue` wraps array indices using modulus.

5.2 Algorithms

empty

5.3 Major functions

empty (remember specifications!)

5.3.1 Program flow

also empty

6 Shortcomings and caveats

References

- [1] “Befunge.” <http://esolangs.org/wiki/Befunge>. Online; retrieved 2017-02-17.
- [2] C. Pressey, “Funge-98 final specification.” <https://github.com/catseye/Funge-98/blob/master/doc/funge98.markdown>. Online; retrieved 2017-02-17.