Befunge-93 interpreter in Haskell

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

Staffan Annerwall, Patrik Johansson, Erik Rimskog

February 19, 2017

Abstract

empty

Contents

1 Introduction

We introduce the programming language Befunge-93 and briefly mention its expansion Funge-98.

1.1 What is Befunge-93?

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, or matrix (in this document also referred to as the "program space" or simply the "space"), where each cell can hold 1 byte of data. There are no variables available, but a LIFO-stack (last-in-first-out) at which values can be pushed onto and popped off of.

Befunge-93 was the first iteration of several Befunge-specifications which eventually lead to the specification of Funge-98 (see 1.2).

1.1.1 Program flow and the Program Counter

Program flow is determined by a unique Program Counter ("PC") and its position in the program space as well as direction, which at any given time is one of North, East, South, or West. The PC starts at position (0, 0) which is the upper-left corner of the program space and has an initial direction of East.

When executing, the PC first reads the character at its current position in the program space, executes its corresponding command, and then moves to a new position.

1.1.2 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.

1.1.3 The program space

The program space in which program data is stored can topologically be thought of as a torus. 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 PC is at position (6, 24) and attempts to move South, its new position would be (6, 0), still moving South.

1.2 Funge-98

Funge-98 is a generalization of Befunge-93 in which program spaces can be either one-, two-, or three-dimensional. The specification 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.

2 Running the program

empty

2.1 Linux

empty

2.2 Windows

empty

2.3 Befunge instruction characters

Befunge-93 allows for a total of 35 actions, 9 of which simply push numeric values 0 – 9.

See Table 1 for a complete list of all instructions that the interpreter handles. If the PC encounters an unrecognized character (i.e. a character that is not in the table), 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` off the stack, then push $a - b$.
`*`	Pop `b` and `a` off the stack, then push $a \times b$.
`/`	Pop `b` and `a` off the stack, then push a/b.
`%`	Pop `b` and `a` off the stack, then push `a` mod `b`.
`` ` ``	Pop `b` and `a` off the stack. 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 $a \neq 0$, otherwise East.
`|`	Pop `a` and instruct the PC to move North if $a \neq 0$, 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).
`&`	Wait for user value input and push it.
`~`	Wait for user character input and push its ASCII value.
`.`	Pop a value and print it.
`,`	Pop a value and print its ASCII character.
␣	Spaces are ignored, the PC continues and no modifications are made.
`@`	Terminate the program.

Table 1: A list of all Befunge-93 instructions; the interpreter handles all of them correctly.

2.4 Example programs

empty

2.4.1 Basic arithmetic

3 Implementation

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

3.1 Data structures

empty

3.1.1 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. Should `s` be empty, a tuple `(BStack [], 0)` is returned.

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

All three functions have precondition `True`.

3.1.2 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.

3.2 Algorithms

empty

3.3 Major functions

empty (remember specifications!)

3.3.1 Program flow

also empty

4 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.