Befunge-93—A Formal Semantics

Abstract

This is a formal semantics for the language "Befunge-93". Befunge-93 was designed by Chris Pressey in 1993. It is a reflective, stack-based, two-dimensional language. The language is not turing complete (there are a finite number of allowable input programs). More information can be found at `http://en.wikipedia.org/wiki/Befunge` and `http://catseye.tc/projects/befunge93/`. Some comments below are adapted from the Wikipedia article and from the catseye webpage.

K formal semantics for Befunge-93 written by Chucky Ellison (celliso2@illinois.edu).

Befunge is a two-dimensional language, meaning the programs are not merely a linear sequence of commands, but an 80×25 grid of commands laid out on a plane. Instructions are each a single character, and are laid out in a grid where the top left corner is coordinate $(0, 0)$, the top right is $(79, 0)$, the bottom left is $(0, 24)$, and the bottom right is $(79, 24)$. The instruction pointer, or "program counter" is not just a scalar offset, but is actually an ordered pair (x, y) with a direction (right, left, up, or down). If programs are smaller than 80×25, they are assumed to be placed in the upper left corner with the rest of the grid being composed of spaces. Finally, the space is assumed to be a torus, where running off of the right or left sides would bring the program counter back to the left or right sides respectively, and similarly for the top and bottom.

The program starts with the right-pointing program counter at $(0, 0)$. As evaluation continues, certain commands can cause the program counter to point in a different direction. For example, executing the commands `>`, `<`, `^`, or `v` cause the program counter to change direction to go right, left, up, or down, respectively. This means the program:

```
>v
^<
```

is a simple infinite loop, where the program counter actually moves around in a tight circle.

The language is stack based. There are a number of commands that push data onto the stack. For example, executing "0"–"9", pushes the corresponding decimal number onto the stack. Most of the remaining commands offer ways of manipulating the stack.

The following is a "Hello world!" program. We denote explicit spaces with ␣. As mentioned above, missing characters are also considered to be spaces.

```
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣v
>v"Hello␣world!"0<
,:
^_25*,@
```

MODULE BEFUNGE
IMPORTS K
IMPORTS PL-BUILTINS
SYNTAX *Bag* ::= eval(*K*)
| eval-inp(*K* , *K*)
SYNTAX *KLabel* ::= injectM(*Map*)
SYNTAX *K* ::= *Bool*
| *Char*
| *Int*
| *String*
| coord(*K* , *K*)
| load
| movePC
| push(*K*)
| defaultmode
| stringmode

CONFIGURATION:
⟨T⟩ ⟨k⟩ • ⟨stack⟩ • ⟨direction⟩ ">" ⟨pc⟩ coord(0 , 0) ⟨mode⟩ defaultmode ⟨program⟩ • ⟨output⟩ "" ⟨input⟩ • ⟨result⟩ "" ⟨resultProgram⟩ ""

RULE eval(injectM(*M*) (•)) ⇒ eval-inp(injectM(*M*) (•) , "")

RULE eval-inp(injectM(*M*) (•) , *Input*) ⇒ ⟨T⟩ … ⟨program⟩ *M* ⟨input⟩ *Input* …

`@` (end): This command ends the program. Because the language is reflective and can manipulate the program during runtime, we return an ascii version of the program as well. Many Befunge programmers manipulate the program as a way of providing output.

RULE ⟨T⟩ *S* ⟨k⟩ "@" … ⟨output⟩ *S* ⟨program⟩ *M* ⇀ ⟨resultProgram⟩ printProgram(*M*) ⟨result⟩ *S*

`"` (stringmode): This command toggles "stringmode" (vs. "defaultmode"). When in stringmode, any character encountered (other than `"`) will have its ascii value pushed onto the stack. When in defaultmode, commands are interpreted normally.

RULE ⟨k⟩ "\"" ⇒ • … ⟨mode⟩ defaultmode ⇒ stringmode

RULE ⟨k⟩ "\"" ⇒ • … ⟨mode⟩ stringmode ⇒ defaultmode

Any character in stringmode: If the current mode of execution is "stringmode", then any character other than `"` will have its ascii value pushed onto the stack.

RULE ⟨k⟩ *C* ⇒ push(charToAscii (*C*)) … ⟨mode⟩ stringmode when $C \neq_{Bool}$ "\""

Any number: In defaultmode, if the next "command" to execute is a digit, then the value of that digit should be pushed onto the stack.

RULE ⟨k⟩ *C* ⇒ push(charToNumber(*C*)) … ⟨mode⟩ defaultmode when isDigit(*C*)

`>`, `<`, `^`, `v`, (right, left, up, down respectively): These commands change the direction of execution. For example, if the program counter was moving right and encountered a `^`, it would begin to go up.

RULE ⟨k⟩ *C* ⇒ • … ⟨direction⟩ _ ⇒ *C* ⟨mode⟩ defaultmode when $C =_{Bool}$ "<" $\vee_{Bool}$ $C =_{Bool}$ ">" $\vee_{Bool}$ $C =_{Bool}$ "^" $\vee_{Bool}$ $C =_{Bool}$ "v"

? (random): This command changes the direction of the program counter to be going in a random direction.

RULE ⟨k⟩ "?" ⇒ ">" …

RULE ⟨k⟩ "?" ⇒ "<" …

RULE ⟨k⟩ "?" ⇒ "^" …

RULE ⟨k⟩ "?" ⇒ "v" …

`#` (bridge): This command causes the next command which would normally be executed to be skipped. That is, it causes the program counter to continue moving in the direction it was already moving, but without executing the next command.

RULE ⟨k⟩ "#" ⇒ movePC … ⟨mode⟩ defaultmode

`:` (dup): This command duplicates the top element of the stack.

RULE ⟨k⟩ ":" ⇒ • … ⟨stack⟩ *Li* • ⇒ *Li* … ⟨mode⟩ defaultmode

`_` (horizontal if): This command acts like `<` if the value on the stack is true (non-zero) or `>` if it is false. It also pops the value off of the stack.

RULE ⟨k⟩ "_" ⇒ "<" … ⟨stack⟩ *I* ⇒ • … ⟨mode⟩ defaultmode when $I \neq_{Bool} 0$

RULE ⟨k⟩ "_" ⇒ ">" … ⟨stack⟩ *I* ⇒ • … ⟨mode⟩ defaultmode when $I =_{Bool} 0$

`|` (vertical if): This command acts like `^` if the value on the stack is true (non-zero) or `v` if it is false. It also pops the value off of the stack.

RULE ⟨k⟩ "|" ⇒ "^" … ⟨stack⟩ *I* ⇒ • … ⟨mode⟩ defaultmode when $I \neq_{Bool} 0$

RULE ⟨k⟩ "|" ⇒ "v" … ⟨stack⟩ *I* ⇒ • … ⟨mode⟩ defaultmode when $I =_{Bool} 0$

␣ (null command): This command does nothing.

RULE ⟨k⟩ "" ⇒ • … ⟨mode⟩ defaultmode

`$` (pop): This command pops a value off of the stack and throws it away.

RULE ⟨k⟩ "$" ⇒ • … ⟨stack⟩ _ ⇒ • … ⟨mode⟩ defaultmode

`\` (swap): This command swaps the top two elements of the stack.

RULE ⟨k⟩ "\\" ⇒ • … ⟨stack⟩ Li_1 Li_2 ⇒ Li_2 Li_1 … ⟨mode⟩ defaultmode

`!` (not): This command replaces the top element of the stack with a 1 if that element is 0, and with a 0 otherwise.

RULE ⟨k⟩ "!" ⇒ • … ⟨stack⟩ *I* ⇒ if $I =_{Bool}$ 0 then 1 else 0 fi … ⟨mode⟩ defaultmode

`` ` `` (greater): This command pops the top two elements B and A from the stack and pushes a 1 if $A > B$, and a 0 otherwise.

The rule below should show a backtick `` ` `` character, but due to a bug in the latex generator, it does not.

RULE ⟨k⟩ "" ⇒ • … ⟨stack⟩ *B A* ⇒ if $A >_{Int} B$ then 1 else 0 fi … ⟨mode⟩ defaultmode

`+` (add): This command pops the top two elements of the stack and replaces them with their sum.

RULE ⟨k⟩ "+" ⇒ • … ⟨stack⟩ *B A* ⇒ $A +_{Int} B$ … ⟨mode⟩ defaultmode

`-` (subtract): This command pops elements b and a from the stack and replaces them with the difference $a - b$.

RULE ⟨k⟩ "-" ⇒ • … ⟨stack⟩ *B A* ⇒ $A -_{Int} B$ … ⟨mode⟩ defaultmode

`*` (multiply): This command pops the top two elements of the stack and replaces them with their product.

RULE ⟨k⟩ "*" ⇒ • … ⟨stack⟩ *B A* ⇒ $A *_{Int} B$ … ⟨mode⟩ defaultmode

`/` (divide): This command pops elements b and a from the stack and replaces them with the integer quotient $\frac{a}{b}$.

RULE ⟨k⟩ "/" ⇒ • … ⟨stack⟩ *B A* ⇒ $A \div_{Int} B$ … ⟨mode⟩ defaultmode when $B \neq_{Bool} 0$

`%` (modulo): This command pops elements b and a from the stack and replaces them with the modulus $a\%b$.

RULE ⟨k⟩ "%" ⇒ • … ⟨stack⟩ *B A* ⇒ $A \%_{Int} B$ … ⟨mode⟩ defaultmode when $B \neq_{Bool} 0$

`g` (get): This command is a meta-operation that pops the top two elements of the stack and treats them as coordinates y and x into the program. The ascii value of the character that is at this position in the program is then pushed onto the stack. If there is no code at that position, the ascii value of a null command (a space) is pushed.

RULE ⟨k⟩ "g" ⇒ • … ⟨stack⟩ *Y X* ⇒ charToAscii (*K*) … ⟨program⟩ … coord(*X* , *Y*) ↦ *K* … ⟨mode⟩ defaultmode

RULE ⟨k⟩ "g" ⇒ • … ⟨stack⟩ *Y X* ⇒ charToAscii ("") … ⟨program⟩ *M* ⟨mode⟩ defaultmode when $\neg_{Bool}$ \$hasMapping(*M* , coord(*X* , *Y*))

`p` (put): This command is a meta-operation that pops three values from the stack: a coordinate pair y and x, and an additional value v. It treats x and y as coordinates into the program, and writes the ascii character corresponding to value v at that location in the program.

RULE ⟨k⟩ "p" ⇒ • … ⟨stack⟩ *Y X V* ⇒ • … ⟨program⟩ *M* ⇒ *M* [safeIntToChar(*V*) / coord(*X* , *Y*)] ⟨mode⟩ defaultmode when $X <_{Int} 80 \wedge_{Bool} Y <_{Int} 25 \wedge_{Bool} X \geq_{Int} 0 \wedge_{Bool} Y \geq_{Int} 0$

`.` (output value): This command pops a number off of the stack and outputs that number as a string, followed by a space. E.g., if the number 14 is on the stack, it will output "14 ".

RULE ⟨k⟩ "." ⇒ • … ⟨stack⟩ *I* ⇒ • … ⟨output⟩ *S* ⇒ $S +_{String}$ Int2String (*I*) $+_{String}$ "" ⟨mode⟩ defaultmode

`,` (output character): This command pops a number off of the stack, interprets that number as an ascii code, and outputs the corresponding character.

RULE ⟨k⟩ "," ⇒ • … ⟨stack⟩ *I* ⇒ • … ⟨output⟩ *S* ⇒ $S +_{String}$ safeIntToChar(*I*) ⟨mode⟩ defaultmode

`&` (input value): This command reads a numeric value (in decimal) from the standard input, and pushes it onto the stack. E.g., if the user types in "32 ", it will push 32 onto the stack.

RULE ⟨k⟩ "&" ⇒ readNumber(false , 0) … ⟨mode⟩ defaultmode

`~` (input character): This command reads a single character from the standard input, and pushes its ascii code onto the stack.

RULE ⟨k⟩ "~" ⇒ readCharacter … ⟨mode⟩ defaultmode

The remaining rules are used to define the helper operators and are more technical than the above rules.

RULE ⟨k⟩ push(*K*) ⇒ • … ⟨stack⟩ • ⇒ *K* …

RULE ⟨k⟩ • ⇒ load ↷ movePC

RULE ⟨stack⟩ • ⇒ 0

RULE ⟨stack⟩ _ • ⇒ 0

RULE ⟨k⟩ load ⇒ *C* … ⟨pc⟩ *Loc* ⟨program⟩ … *Loc* ↦ *C* …

RULE ⟨k⟩ load ⇒ "" … ⟨pc⟩ coord(*X* , *Y*) ⟨program⟩ *M* when $\neg_{Bool}$ \$hasMapping(*M* , coord(*X* , *Y*)) $\wedge_{Bool} X <_{Int} 80 \wedge_{Bool} Y <_{Int} 25 \wedge_{Bool} X \geq_{Int} 0 \wedge_{Bool} Y \geq_{Int} 0$

RULE ⟨k⟩ movePC ⇒ • … ⟨direction⟩ ">" ⟨pc⟩ coord(*I* ⇒ $I +_{Int} 1 \%_{Int} 80$, _)

RULE ⟨k⟩ movePC ⇒ • … ⟨direction⟩ "<" ⟨pc⟩ coord(*I* ⇒ $80 +_{Int} I -_{Int} 1 \%_{Int} 80$, _)

RULE ⟨k⟩ movePC ⇒ • … ⟨direction⟩ "^" ⟨pc⟩ coord(_ , *I* ⇒ $25 +_{Int} I -_{Int} 1 \%_{Int} 25$)

RULE ⟨k⟩ movePC ⇒ • … ⟨direction⟩ "v" ⟨pc⟩ coord(_ , *I* ⇒ $I +_{Int} 1 \%_{Int} 25$)

SYNTAX *K* ::= readNumber(*K* , *K*)
| readCharacter

RULE ⟨k⟩ readNumber(*B* ⇒ true , *I* ⇒ $I +_{Int}$ charToNumber(firstChar(*S*))) … ⟨input⟩ *S* ⇒ butFirstChar(*S*) ⟨mode⟩ defaultmode when isDigit(firstChar(*S*))

RULE ⟨k⟩ readNumber(true , *I*) ⇒ • … ⟨stack⟩ • ⇒ *I* … ⟨input⟩ *S* when $\neg_{Bool}$ isDigit(firstChar(*S*))

RULE ⟨k⟩ readCharacter ⇒ • … ⟨stack⟩ • ⇒ asciiString (k2char(firstChar(*S*))) … ⟨input⟩ *S* ⇒ butFirstChar(*S*) when asciiString (k2char(firstChar(*S*))) $\geq_{Int}$ 0

SYNTAX *String* ::= printProgram(*Map*)
| printProgram-aux(*Int* , *Int* , *Map*)
RULE printProgram(*M*) ⇀ printProgram-aux(0 , 0 , *M*)

RULE printProgram-aux(*X* , *Y* , *M*) ⇀ k2string(*M* (coord(*X* , *Y*))) $+_{String}$ printProgram-aux($X +_{Int} 1$, *Y* , *M*) when \$hasMapping(*M* , coord(*X* , *Y*))

RULE printProgram-aux(*X* , *Y* , *M*) ⇀ "" $+_{String}$ printProgram-aux($X +_{Int} 1$, *Y* , *M*) when $\neg_{Bool}$ \$hasMapping(*M* , coord(*X* , *Y*)) $\wedge_{Bool} X <_{Int} 80 \wedge_{Bool} Y <_{Int} 25$

RULE printProgram-aux(80 , *Y* , *M*) ⇀ charString (10) $+_{String}$ printProgram-aux(0 , $Y +_{Int} 1$, *M*)

RULE printProgram-aux(0 , 25 , *M*) ⇀ ""

SYNTAX *String* ::= k2string(*K*)
RULE k2string(*S*) ⇀ *S*

SYNTAX *Char* ::= k2char(*K*)
RULE k2char(*C*) ⇀ *C*

SYNTAX *Bool* ::= isDigit(*K*)
RULE isDigit(*K*) ⇀ if $K =_{Bool}$ "0" $\vee_{Bool}$ $K =_{Bool}$ "1" $\vee_{Bool}$ $K =_{Bool}$ "2" $\vee_{Bool}$ $K =_{Bool}$ "3" $\vee_{Bool}$ $K =_{Bool}$ "4" $\vee_{Bool}$ $K =_{Bool}$ "5" $\vee_{Bool}$ $K =_{Bool}$ "6" $\vee_{Bool}$ $K =_{Bool}$ "7" $\vee_{Bool}$ $K =_{Bool}$ "8" $\vee_{Bool}$ $K =_{Bool}$ "9" then true else false fi

SYNTAX *String* ::= safeIntToChar(*K*)
RULE safeIntToChar(*I*) ⇀ charString ($256 +_{Int} I \%_{Int} 256 \%_{Int} 256$)

SYNTAX *K* ::= butFirstChar(*K*)
RULE butFirstChar(*S*) ⇀ substrString (*S* , 1 , lengthString (*S*))

SYNTAX *K* ::= firstChar(*K*)
RULE firstChar(*S*) ⇀ substrString (*S* , 0 , 1)

SYNTAX *KLabel* ::= charToAscii
RULE charToAscii (*C*) ⇀ asciiString (*C*)

SYNTAX *Int* ::= charToNumber(*K*)
RULE charToNumber(*C*) ⇀ asciiString (*C*) $-_{Int}$ asciiString ("0")

END MODULE