Note: The content came from <http://quadium.net/funge/spec98.html>

# Funge-98 Final Specification

Chris Pressey, Sept 11, 1998  
revised for clarity: Sept 30 1998

### Table of Contents

#### [Introduction](http://quadium.net/funge/spec98.html" \l "Introduction)

* + [What is a Funge?](http://quadium.net/funge/spec98.html#Whatis)
  + [About this Document](http://quadium.net/funge/spec98.html#About)

#### [The Funge Virtual Machine](http://quadium.net/funge/spec98.html#Machine)

* + [Code and Data](http://quadium.net/funge/spec98.html#Code_Data)
  + [Funge-Space](http://quadium.net/funge/spec98.html#Space)
  + [Stack Stack](http://quadium.net/funge/spec98.html#Stack_Stack)
  + [Funge Source File Format](http://quadium.net/funge/spec98.html#Format)

#### [Code: Program Flow](http://quadium.net/funge/spec98.html#Code)

* + [Instruction Pointer](http://quadium.net/funge/spec98.html#IP)
  + [Instructions](http://quadium.net/funge/spec98.html#Instructions)
  + [Direction Changing](http://quadium.net/funge/spec98.html#Direction)
  + [Wrapping](http://quadium.net/funge/spec98.html#Wrapping)
  + [Flow Control](http://quadium.net/funge/spec98.html#Flow)
  + [Decision Making](http://quadium.net/funge/spec98.html#Decision)

#### [Data: Cell Crunching](http://quadium.net/funge/spec98.html#Data)

* + [Integers](http://quadium.net/funge/spec98.html#Integers)
  + [Strings](http://quadium.net/funge/spec98.html#Strings)
  + [Stack Manipulation](http://quadium.net/funge/spec98.html#Stack_Manipulation)
  + [Stack Stack Manipulation](http://quadium.net/funge/spec98.html#Stack_Stack_Manipulation)

#### [Media: Communications and Storage](http://quadium.net/funge/spec98.html#Media)

* + [Funge-Space Storage](http://quadium.net/funge/spec98.html#Storage)
  + [Standard Input/Output](http://quadium.net/funge/spec98.html#Stdio)
  + [File Input/Output](http://quadium.net/funge/spec98.html#Fileio)
  + [System Execution](http://quadium.net/funge/spec98.html#System)
  + [System Information Retrieval](http://quadium.net/funge/spec98.html#Sysinfo)

#### [Scale: Extension and Customization](http://quadium.net/funge/spec98.html#Scale)

* + [Handprints](http://quadium.net/funge/spec98.html#Handprints)
  + [Fingerprints](http://quadium.net/funge/spec98.html#Fingerprints)
  + [Funge Central Registry](http://quadium.net/funge/spec98.html#Registry)

#### [Appendix](http://quadium.net/funge/spec98.html#Appendix)

* + [Instruction Quick Reference](http://quadium.net/funge/spec98.html#Quickref)
  + [Concurrent Funge-98](http://quadium.net/funge/spec98.html#Concurrent)
  + [Lahey-Space](http://quadium.net/funge/spec98.html#Lahey)
  + [Other Topologies](http://quadium.net/funge/spec98.html#Topologies)

## Introduction

### What is a Funge?

Funges are programming languages whose programs are typically expressed in a given topological pattern and number of dimensions.

Funge-98 is currently an official prototype standard for Funges. Funge-98 has evolved from Funge-97, which was a generalization of Befunge-97, which was an improvement over Befunge-96, which was an update of the original Befunge-93 language definition.

Funge-98 is a *class* of three real and officially sanctioned programming languages (Unefunge, Befunge, and Trefunge) and provides a paradigm for describing any number of imaginary ones with different topologies and any number of dimensions.

The most popular Funge by far is Befunge, which is two-dimensional and based on a Cartesian Lahey-Space (or Cartesian Torus, in Befunge-93 only) topology. Other Cartesian Lahey-Space Funges include Unefunge (one-dimensional) and Trefunge (three-dimensional.) Since not all Funge instructions are appropriate in all Funges, comparison to Befunge is often used to clarify points in this document.

### About this Document

This is a final document. The information it contains has been formally approved and it is endorsed by its supporters as the 'official' technical specification of the Funge language family.

This document is suitable for an audience not already familiar with any Funge of any kind or year.

## The Funge Virtual Machine

### Code and Data

Any given piece of code or data in a Funge can be stored in one of two places (called a *cell*):

* *Funge-Space*, a matrix appropriate to the dimensionality, topology and tiling pattern of the Funge, where each *node* in its topological net contains a cell; or
* the *stack* in Befunge-93 or the *stack stack* in Funge-98; either way, it's often called *the stack* and it's accessed as a last-in, first-out (LIFO) stack of cells.

Befunge-93 defines signed 32-bit stack cells and unsigned 8-bit Funge-Space cells. In Funge-98, stack and Funge-Space cells alike should be treated as signed integers of the same size.

What size exactly is left up to the implementer. 32 bits is typical. 16 bit and 8 bit versions are discussed as separate variations on Funge-98. More than 32 bits is just fine. The important thing is that the stack cells have the same memory size as the Funge-Space cells.

### Funge-Space

In Befunge-93, Funge-Space is restricted to 80 cells in the *x* dimension and 25 cells in the *y* dimension. No such limits are imposed on Funge-98 programs. A Funge-98 interpreter, ideally, has an addressing range equal to that of its cell size. i.e. A 32-bit implementation of Funge-98 uses signed 32-bit integers as each of its coordinate indices.

With such a large typical addressing range, the Funge-98-Space is generally considered to be dynamically allocated by some behind-the-scenes mechanism in the compiler. It *needn't* be, of course, but in practice, it usually is.

So, the storage mechanism has be consistently trustworthy about how it provides Funge-Space to the running program. A Funge-98 program should be able to rely on all code and data that it puts in Funge-Space through this mechanism not disappearing. A Funge-98 program should also be able to rely on the memory mechanism acting as if a cell contains blank space (ASCII 32) if it is unallocated, and setting memory to be full of blank space cells upon actual allocation (program load, or p instruction). If the underlying memory mechanism cannot provide this (e.g. no more memory is available to be allocated,) the interpreter should complain with an error and do what it can to recover, (but not necessarily gracefully).

The co-ordinate mapping used for both Befunge-93 and Funge-98 reflects the "Computer Storage" co-ordinate system used in screen graphics and spreadsheets; a larger *y* coordinate means further down the page. Compared to a standard mathematical representation of the usual Cartesian co-ordinate system, it is upside-down.

|  |
| --- |
| Befunge-93 32-bit Befunge-98  ========== =================  0 *x* 79 |-2,147,483,648  0+-------------+ |  | | *x*  | -----+-----  *y*| -2,147,483,648 | 2,147,483,647  | |  | *y*|2,147,483,647  24+ |

### Stack Stack

The Funge stack stack is a LIFO stack of typical LIFO stacks of cells. In Befunge-93, only two operations are possible on only one stack (referred to as *the stack*): to *push* a cell onto the top of the stack, and to *pop* a cell off the top of the stack.

In the case of Funge-98, however, *the stack* refers to the topmost stack on the stack stack. The push and pop operations are possible on the stack stack as well, but they push and pop entire stacks.

There is also a Funge-98 instruction to rid the stack (that is, the topmost stack of the stack stack) of cells, completely emptying it.

If a program attempts to pop a cell off the stack when it is empty, no error occurs; the program acts as if it popped a 0.

In this document, short stacks are generally notated left to right to mean **bottom to top**. The **leftmost** values listed in the documentation are the **bottommost** and the **first** to be pushed onto the stack. Long stacks are notated top to bottom, to mean precisely that, **top to bottom.**.

### Funge Source File Format

A Befunge-93 source (program) file name, by common convention, ends in the extension .bf. There is no enforced convention for what any given Funge-98 source file name ends in (e.g. you could easily write a C-Befunge polyglot whose file name ends in .c), but .b98 is a good choice for Befunge-98 sources - "standard" example programs use this suffix.

Befunge-93 source files are plain text files containing only printable ASCII characters and the end-of-line controls described below.

Funge-98 source files are made up of Funge characters. The Funge-98 character set overlays the ASCII subset used by Befunge-93 and may have characters greater than 127 present in it (and greater than 255 on systems where characters are stored in multiple bytes; but no greater than 2,147,483,647.) The Funge character set is 'display-independent.' That is to say, character #417 may look like a squiggle on system Foo and a happy face on system Bar, but the meaning is always the same to Funge, 'character #417', regardless of what it looks like.

In other words, what Funge characters look like on a particular computer or OS depends entirely on that computer or OS. However, when characters are not generally considered to be printable, they can have special meaning to Funge-98:

* 0..31 : "ASCII controls" (only 10 is currently defined to mean EOL)
* 32..126 : "ASCII printable characters" (all are input/output and fixed-width)
* 127 : "delete control" (undefined)
* 128..2bil: "extended printable characters" (machine and font specific)

In Befunge-93, each line ends with the current operating system's "end of line" character, which may be a line feed (10) (Linux), carriage return (13) (MacOS), or carriage return-line feed (13, 10) (MS-DOS).

In Funge-98, however, *any* of the following sequences should, ideally, be recognized by the interpreter as an end-of-line marker, no matter *what* operating system it's running on:

* Line Feed (10)
* Carriage Return (13)
* Carriage Return, Line Feed (13, 10)

If an interpreter cannot support all three varieties of end-of-line marker, it should be clearly noted in that interpreter's documentation.

End-of-line markers do **not** appear in Funge-Space once the program is loaded.

In Befunge-93, each line can contain up to 80 significant characters before the "End of Line" marker. There can be up to 25 such lines in the source file. There are no such restrictions on Befunge-98 and the user can reasonably expect to be able to have as many lines of as many characters as they want, for non-contrived cases.

Before load, every cell in Funge-Space contains a space (32) character. These default contents are written over by characters in the program source when it is loaded. However, spaces in the program source do not overwrite anything in Funge-Space; in essence the space character is transparent in source files. This becomes important when the i "Input File" instruction is used to include overlapping files.

The source file begins at the *origin* of Funge-Space. Subsequent columns of characters increment the *x* coordinate, and subsequent lines increment the *y* coordinate (if one is present) and reset the *x* coordinate to zero. Subsequent lines in Unefunge are simply appended to the first, and the end of the source file indicates the end of the (single) line. End-of-line markers are never copied into Funge-Space.

In Trefunge-98, the Form Feed (12) character increments the *z* coordinate and resets the *x* and *y* coordinates to zero.

## Code: Program Flow

### Instruction Pointer

The *instruction pointer* (IP) can be thought of as a *vector* (set of co-ordinates) which represents the "current position" of a running Funge program. It holds the same function as the instruction pointer (or *program counter* (PC)) in any other language or processor - to indicate where the currently executing instruction is located.

In most other languages and machines (both virtual and real,) the IP/PC is restricted to unidirectional travel in a single dimension, with random jumps. However, in Funge, the IP keeps track of another vector called the *delta*. Every *tick*, the IP executes its current instruction (that is, the instruction at the location pointed to by the IP), then travels to a new location, by adding its delta vector to its position vector.

At the beginning of a program, in Funge-98 as in Befunge-93, the IP always begins at the origin and starts with a delta of (1, 0). The origin is (0, 0) in Befunge, (0) in Unefunge, and (0, 0, 0) in Trefunge.

In two dimensions, we have the following terminology.

If the IP's delta is either (0,-1) (*south*), (1,0) (*east*), (0,1) (*north*), or (-1,0) (*west*), it is said to be traveling *cardinally*. This is the same as how a rook moves in chess and this is in fact the only way the IP can move in Befunge-93.

Any IP with a nonzero delta is considered *moving*. Any IP with a zero delta is said to be *stopped*. Any moving IP that is not traveling cardinally is said to be *flying*.

### Instructions

All standard instructions are one character long and range from ASCII 32 (space) to ASCII 126 (~). There are no multicharacter instructions in Funge.

An instruction is executed by an IP every tick. The IP executed is the one at the current position of the IP. Only after that does the IP moves by its delta to a new position.

Instructions A to Z all initially act like the r "Reflect" instruction. However, other instructions assign semantics to these instructions dynamically, allowing the Funge programmer to use libraries of both standard and proprietary instruction sets tagged with unique ID's (or *fingerprints*.)

However, a Funge-98 interpreter may also expose any number of proprietary instructions above ASCII 127 or below ASCII 0.

For all these reasons, when encountering any unimplemented instruction (this includes instructions like | in Unefunge,) the Funge interpreter should at least provide an option for informing the user that it was told to execute an instruction that isn't implemented, and possibly warning the user that this file might be an incorrect language or version.

An unimplemented instruction must otherwise act as if r was executed, and must not touch the stack. All undefined or otherwise unimplemented instructions must be considered unimplemented.

Also, any of the instructions t (concurrent execution,) = (execute,) i (input-file,) and o (output-file) may be unavailable in different interpreters for many reasons, and are routinely bound to (i.e. act just like) r as well. However, they may also act like r when they fail to execute. To test if they are actually supported, execute 1y and examine the cell it produces.

### Direction Changing

A few instructions are essential for changing the delta of the IP. The > "Go East" instruction causes the IP to travel east; the < "Go West" instruction causes the IP to travel west. These instructions are valid in all Funges.

The ^ "Go North" instruction causes the IP to travel north; the v "Go South" instruction causes the IP to travel south. These instructions are not available in Unefunge.

The h "Go High" instruction causes the IP to travel up (delta <- (0,0,1)); the l "Go Low" instruction causes the IP to travel down (delta <- (0,0,-1)). These instructions are not available in Unefunge or Befunge.

The ? "Go Away" instruction causes the IP to travel in a random cardinal direction appropriate to the number of dimensions in use: east or west in Unefunge; north, south, east or west in Befunge, etc.

The following instructions are not in Befunge-93, but they are in Funge-98.

The ] "Turn Right" and [ "Turn Left" instructions rotate by 90 degrees the delta of the IP which encounters them. They always rotate on the *z* axis. These instructions are not available in Unefunge.

To remember which is which, visualize yourself on the seat of a bicycle, looking down at the handlebars:

|  |  |  |
| --- | --- | --- |
| +- |  +- | +-+ | | | -+  | -+ |
| [ |  | ] |
| Turn Left | Go Forward | Turn Right |

The r "Reverse" instruction multiplies the IP's delta by -1. In two dimensions, this is the equivalent of reflecting the delta of the IP about the z-axis.

The x "Absolute Vector" instruction pops a vector off the stack, and sets the IP delta to that vector.

A vector on the stack is stored bottom-to-top, so that in Befunge, x (and all other vector-popping instructions) pops a value it calls *dy*, then pops a value it calls *dx*, then sets the delta to (*dx*, *dy*).

### Wrapping

Befunge-93 handles the case of the IP travelling out of bounds (off the map of Funge-Space) by treating the space as a torus. If the IP leaves the west edge, it reappears on the east edge at the same row; if it leaves the south edge, it reappears at the north edge at the same column, and vice versa in both cases.

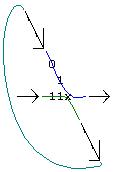
For various reasons, toroidal wrapping is problematic in Funge-98. Instead, we use a special wrapping technique that has more consistent results in this new, more flexible environment where Funge-Space can have an arbitrary size and the IP can fly. It is called *same-line wrapping*.

Same-line wrapping can be described in several ways, but the crucial idea it encompasses is this: unless the delta or position of the IP were to be changed by an intervening instruction, the IP will always wrap such that it would eventually return to the instruction it was on before it wrapped.

The mathematical description of same-line wrapping is known as *Lahey-space* wrapping, which defines a special topological space. It is generally of more interest to topologists and mathematicians than programmers. We won't cover it here, but it is included in the [Appendix](http://quadium.net/funge/spec98.html" \l "Lahey) for completeness.

The algorithmic description of same-line wrapping can be described as *backtrack wrapping*. It is more of interest to Funge interpreter implementers than Funge programmers. However, it does describe exactly how the wrapping *acts* in terms that a programmer can understand, so we will include it here.

When the IP attempts to travel into the whitespace between the code and the end of known, addressable space, it backtracks. This means that its delta is reversed and it ignores (skips over without executing) all instructions. Travelling thus, it finds the other 'edge' of code when there is again nothing but whitespace in front of it. It is reflected 180 degrees once more (to restore its original delta) and stops ignoring instructions. Execution then resumes normally - the wrap is complete.



It is easy to see at this point that the IP remains on the same line: thus the name. (Also note that this **never** takes any ticks in regards to multithreading, as would be expected from any wrapping process.)

Same-line wrapping has the advantage of being backward-compatible with Befunge-93's toroidal wrapping. It also works safely both when the IP delta is flying (non-cardinal), and when the size of the program changes.

As noted, by default every cell in Funge-Space contains a space (32) character. (This holds true with most decent Befunge-93 interpreters, too, although it was not in the original.)

In Befunge-93, when interpreted as an instruction, a space is treated as a "no operation" or *nop*. The interpreter does nothing and continues on its merry way.

Funge-98 acts much the same way, except technically, Funge-98 processes any number of spaces in "no time whatsoever", and this becomes important when you have more than one IP in Funge-Space at the same time (*multithreading*), which you'll read about later. For an explicit nop instruction in Funge-98, use z.

Space also takes on special properties (in Funge-98) with a special *mode* of the interpreter called stringmode, which you'll also read about later.

### Flow Control

The # "Trampoline" instruction moves the IP one position beyond the next Funge-Space cell in its path.

The @ "Stop" instruction kills the current IP. In non-Concurrent Funge, there is only a single IP. Either way, when no IP's are left alive, the program will subsequently end with no error (returning error code 0).

The following instructions and markers are not in Befunge-93, but they are in Funge-98.

The ; "Jump Over" marker causes the IP to jump over all subsequent instructions until the next ; marker. Like space, this takes zero ticks to execute, so that subroutines, comments, and satellite code can be insulated by surrounding it with ; markers, with no effect on multithreading.

; is truly ethereal; like space, it cannot ever be truly executed, in the sense of it taking up a tick and doing something.

The j "Jump Forward" instruction pops a value off the stack, and jumps over that many spaces. If there is a 1 on the stack, j will work like # does. e.g. 2j789. would print 9 and leave an empty stack. Negative values are legal arguments for j, such that 04-j@ is an infinite loop.

The q "Quit" instruction, only in Funge-98, ends the entire program immediately (regardless of the number of IPs active in Concurrent Funge). It also pops a cell off the stack and uses that value as the return value of the Funge interpreter to the operating system.

Note that most operating systems will only look at the least significant byte your return value, unsigned. But you can return a full cell, and the OS will interpret as much of it as it can handle, treating it as signed or unsigned as the OS demands.

The k "Iterate" instruction pops a value *n* off the stack. Then it finds the next instruction in Funge-space in the path of the IP (note that this cannot be a marker such as space or ;), treats it as an instruction, executing it *n* times. This takes only one tick with respect to concurrent operation.

Note that some instructions don't make much sense within the context of k unless you include zero as one of the possibilities for how many times the instruction is repeated. For example, no matter how many times after the first time k execute ^, the result is the same. However, you may pass a zero count to k, and the ^ instruction will not be executed; this can be a valuable behaviour.

Also, note k will never, ever actually execute instruction #32, space, or ;.

### Decision Making

The ! "Logical Not" instruction pops a value off the stack and pushes a value which is the logical negation of it. If the value is zero, it pushes one; if it is non-zero, it pushes zero.

The ` "Greater Than" instruction pops two cells off the stack, then pushes a one if second cell is greater than the first. Otherwise pushes a zero.

Funge has instructions that act like directional 'if' statements. The \_ "East-West If" instruction pops a value off the stack; if it is zero it acts like >, and if non-zero it acts like <.

The | "North-South If" instruction pops a value off the stack; if it is zero it acts like v, and if non-zero it acts like ^. | is not available in Unefunge.

The m "High-Low If" (think *middle*) instruction pops a value off the stack; if it is zero it acts like l, and if non-zero it acts like h. m is not available in Unefunge or Befunge.

The w "Compare" instruction pops a value *b* off the stack, then pops a value *a*, then compares them. (*a* is called *a* because it was the first of the two values to be *pushed* onto the stack.) If the *a* is smaller, w acts like [, and turns left. If the *a* is greater, w acts like ], and turns right. If *a* and *b* are equal, w does not affect the IP's delta. This instruction is not available in Befunge-93, nor Unefunge.

## Data: Cell Crunching

### Integers

Instructions 0 "Push Zero" through 9 "Push Niner" push the values zero through nine onto the stack, respectively. In Funge-98, a through f also push 10 through 15 respectively.

The + "Add" instruction pops two cells from the stack, adds them using integer addition, and pushes the result back onto the stack.

The \* "Multiply" instruction pops two cells from the stack, multiplies them using integer multiplication, and pushes the result back onto the stack. In this way numbers larger than 9 (or 15) can be pushed onto the stack.

For example, to push the value 123 onto the stack, one might write

99\*76\*+

or

555\*\*2-

Funge also offers the following arithmetic instructions:

* the - "Subtract" instruction, which pops two values, subtracts the first from the second using integer subtraction, and pushes the result;
* the / "Divide" instruction, which pops two values, divides the second by the first using integer division, and pushes the result (note that division by zero produces a result of zero in Funge-98, but Befunge-93 instead is supposed to *ask* the user what they want the result of the division to be); and
* the % "Remainder" instruction, which pops two values, divides the second by the first using integer division, and pushes the remainder, of those. Remainder by zero is subject to the same rules as division by zero, but if either argument is negative, the result is implementation-defined.

### Strings

The instruction " "Toggle Stringmode" toggles a special mode of the Funge interpreter called *stringmode*. In stringmode, every cell encountered by the IP (except " and in Funge-98, space) is not interpreted as an instruction, but rather as a Funge character, and is pushed onto the stack. A subsequent " toggles stringmode once more, turning it off.

In Funge-98 stringmode, spaces are treated "SGML-style"; that is, when any contiguous series of spaces is processed, it only takes one tick and pushes one space onto the stack. This introduces a small backward-incompatibility with Befunge-93; programs that have multiple spaces and/or wrap while in stringmode will have to be changed to work the same under Funge-98.

Befunge-93 Befunge-98

"hello world" "hello world"

"hello world" "hello "::"world"

There is also a ' "Fetch Character" instruction in Funge-98. This pushes the Funge character value of the next encountered cell (position + delta) onto the stack, then adds the delta to the position (like #), skipping over the character (in no ticks). For example, the following two snippets perform the same function, printing a Q:

"Q",

'Q,

s "Store Character" is the mirror image of the ' instruction: this instead pops a value off the stack and writes it into (position + delta).

Some instructions expect a Funge string on the stack as one of their arguments. The standard format for these strings is called *0"gnirts"* - that is, a null-terminated string with the start of the string at the top of the stack and the null termination at the bottom.

### Stack Manipulation

The $ "Pop" instruction pops a cell off the stack and discards it.

The : "Duplicate" instruction pops a cell off the stack, then pushes it back onto the stack twice, duplicating it.

The \ "Swap" instruction pops two cells off the stack, then pushes the first cell back on, then the second cell, in effect swapping the top two cells on the stack.

The n "Clear Stack" instruction (not available in Befunge-93) completely wipes the stack (popping and discarding elements until it is empty.)

### Stack Stack Manipulation

The stack stack transparently overlays the stack - that is to say, the top stack of Funge-98's stack stack is treated the same as Befunge-93's sole stack. The Funge programmer will never notice the difference unless they use the {, }, or u instructions of Funge-98.

When working with different stacks on the stack stack, though, it's useful to give two of them names: the *top of stack stack* or TOSS, which indicates the topmost stack on the stack stack, which works to emulate the sole stack of Befunge-93; and the *second on stack stack* or SOSS, which is the stack directly under the TOSS.

The { "Begin Block" instruction pops a cell it calls *n*, then pushes a new stack on the top of the stack stack, transfers *n* elements from the SOSS to the TOSS, then pushes the storage offset as a vector onto the SOSS, then sets the new storage offset to the location to be executed next by the IP (storage offset <- position + delta). It copies these elements as a block, so order is preserved.

If the SOSS contains *k* elements, where *k* is less than *n*, the *k* elements are transferred as the top *k* elements and the remaining bottom (*n-k*) elements are filled in with zero-value cells.

If *n* is zero, no elements are transferred.

If *n* is negative, |*n*| zeroes are pushed onto the SOSS.

The corresponding } "End Block" instruction pops a cell off the stack that it calls *n*, then pops a vector off the SOSS which it assigns to the storage offset, then transfers *n* elements (as a block) from the TOSS to the SOSS, then pops the top stack off the stack stack.

The transfer of elements for } "End Block" is in all respects similar to the transfer of elements for { "Begin Block", except for the direction in which elements are transferred. "Transfer" is used here in the sense of "move," not "copy": the original cells are removed.

If *n* is zero, no elements are transferred.

If *n* is negative, |*n*| cells are popped off of the (original) SOSS.

{ makes the current TOSS the new SOSS. } makes the current SOSS the new TOSS.

{ may act like r if no more memory is available for another stack. } acts like r if a stack-stack underflow would otherwise occur (i.e. when there is only one stack on the stack-stack.)

The practical use of these instructions is to "insulate" and "localize" procedures or other blocks of Funge code.

The u "Stack under Stack" instruction pops a *count* and transfers that many cells from the SOSS to the TOSS. It transfers these cells in a pop-push loop. In other words, the order is not preserved during transfer, it is reversed.

If there is no SOSS (the TOSS is the only stack), u should act like r.

If *count* is negative, |*count*| cells are transferred (similarly in a pop-push loop) from the TOSS to the SOSS.

If *count* is zero, nothing happens.

## Media: Communications and Storage

### Funge-Space Storage

The g "Get" and p "Put" instructions are used to store and retrieve data and code in Funge-Space.

In Befunge-93, g pops a vector (that is, it pops a y value, then an x value,) and pushes the value (character) in the Befunge-Space cell at (x, y) onto the stack. p pops a vector, then it pops a value, then it places that value in the Funge-Space cell at (x, y).

In Funge-98, each IP has an additional vector property called the *storage offset*. Initially this vector is the set to the origin. As such, it works to emulate Befunge-93. The arguments to g and p are the same, but instead of pointing to absolute locations in Funge-Space, they reference a cell relative to the storage offset.

### Standard Input/Output

The . "Output Decimal" and , "Output Character" instructions provide numeric and Funge character/control output, respectively; they pop a cell off the stack and send it in numeric or Funge character format to the *standard output*, which is usually displayed by the interpreter in an interactive user terminal.

Outputting character number 10 will result in a new line being displayed on the standard output.

Numeric output is formatted as a decimal integer followed by a space.

These instructions will act as r does, should the standard output fail for any reason.

The & "Input Decimal" and ~ "Input Character" instructions provide numeric and Funge character/control input, respectively. They each suspend the program and wait for the user to enter a value in numeric or Funge character format to the *standard input*, which is usually displayed with the standard output. They then push this value on the stack.

An input of character number 10 indicates that the user pressed the 'Enter' key, or the equivalent key on their keyboard.

Decimal input reads and discards characters until it encounters decimal digit characters, at which point it reads a decimal number from those digits, up until (but not including) the point at which input characters stop being digits, or the point where the next digit would cause a cell overflow, whichever comes first.

Although the standard input and output are generally displayed in some sort of interactive user terminal, they needn't be; many operating systems support *redirection*. In the case of an end-of-file or other file error condition, the & and ~ both act like r.

### File Input/Output

File I/O is done with the i "Input File" and o "Output File" instructions, which are only available in Funge-98.

i pops a null-terminated 0"gnirts" string for the filename, followed by a flags cell, then a vector Va telling it where to operate. If the file can be opened for reading, it is inserted into Funge-Space at Va, and immediately closed. Two vectors are then pushed onto the stack, Va and Vb, suitable arguments to a corresponding o instruction. If the file open failed, the instruction acts like r.

i is in all respects similar to the procedure used to load the main Funge source code file, except it may specify any file, not necessarily Funge source code, and at any location, not necessarily the origin.

Also, if the least significant bit of the flags cell is high, i treats the file as a binary file; that is, EOL and FF sequences are stored in Funge-space instead of causing the dimension counters to be reset and incremented.

o first pops a null-terminated 0"gnirts" string to use for a filename. Then it pops a flags cell. It then pops a vector Va indicating a *least point* (point with the smallest numerical coordinates of a region; also known as the upper-left corner, when used in the context of Befunge) in space, and another vector Vb describing the size of a rectangle (or a rectangular prism in Trefunge, etc). If the file named by the filename can be opened for writing, the contents of the rectangle of Funge-Space from Va to Va+Vb are written into it, and it is immediately closed. If not, the instruction acts like r.

The first vectors popped by both of these instructions are considered relative to the storage offset. (The size vector Vb, of course, is relative to the least point Va.)

Note that in a greater-than-two-dimensional environment, specifying a more-than-two-dimensional size such as (3,3,3) is not guaranteed to produce sensical results.

Also, if the least significant bit of the flags cell is high, o treats the file as a linear text file; that is, any spaces before each EOL, and any EOLs before the EOF, are not written out. The resulting text file is identical in appearance and takes up less storage space.

### System Execution

The = "Execute" instruction pops a string off the stack, and attempts to execute it. How it executes it is implementation dependant. However, an implementation may support one of several standardized ways of interpreting the string, and which routine it uses can be determined by querying y. Typically methods include treating it as a C system() call would, or on a platform such as MacOS, for example, treating the string as AppleScript would be in order.

After execution, a failure value is pushed onto the stack. If this value is zero, everything went as expected. If the value is non-zero, it may be the return-code of the program that was executed; at any rate it means that the attempt to execute the program, or the program itself, did not succeed.

### System Information Retrieval

The y "Get SysInfo" instruction, only available in Funge-98, pops one value off the stack. If the value is zero or negative, y tells you far more than you ever really want to know about the underlying Funge interpreter, operating system, and computer (and is thus usually followed soon after by a n instruction).

Each cell of information retrieved by y, only applies to either the current IP, the current environment (that is, the interpreter of the current IP,) or the global environment (the environment of every IP in the same Funge-space, regardless of which interpreter it's running on.)

After an execution of y with a non-positive argument, the stack contains many more cells (listed from top to bottom:)

1. 1 cell containing flags (env).
   * Least Significant Bit 0 (0x01): high if t is implemented. (is this Concurrent Funge-98?)
   * Bit 1 (0x02): high if i is implemented.
   * Bit 2 (0x04): high if o is implemented.
   * Bit 3 (0x08): high if = is implemented.
   * Most Significant Bit 4 (0x10): high if unbuffered standard I/O (like getch()) is in effect, low if the usual buffered variety (like scanf("%c")) is being used.
   * Further more significant bits: undefined, should all be low in Funge-98
2. 1 cell containing the number of bytes per cell (global env).

aka cell size. Typically 4, could also be 2, 8, really really large, infinity, etc.

1. 1 cell containing the implementation's handprint (env).
2. 1 cell containing the implementation's version number (env).

If the version number contains points, they're stripped. v2.01 == 201, v1.03.05 = 10305, v1.5g = 1507. Don't use non-numbers in the version number to indicate 'personalizations' - change the handprint instead.

1. 1 cell containing an ID code for the Operating Paradigm (global env)
   * 0 = Unavailable
   * 1 = Equivalent to C-language system() call behaviour
   * 2 = Equivalent to interpretation by a specific shell or program

This shell or program is specified by the interpreter but should ideally be customizable by the interpreter-user, if applicable. Befunge programs that run under this paradigm should document what program they expect to interpret the string passed to =.

* + 3 = Equivalent to interpretation by the same shell as started this Funge interpreter, if applicable

If the interpreter supports this paradigm, then in this manner, the user executing a Befunge source can easily choose which shell to use for = instructions.

1. This value is included so the program can have a reasonable idea of what = will do. The values shown here are only the most basic set available at the time of publication. See the [Registry](http://quadium.net/funge/spec98.html" \l "Registry) for any late-breaking headway into further Operating Paradigms.
2. 1 cell containing a path seperator character (global env)

This is what path seperators for i and o filenames should look like.

1. 1 cell containing the number of scalars per vector (global env)

aka number of dimensions. 2 for Befunge, 1 for Unefunge, 3 for Trefunge.

1. 1 cell containing a unique ID for the current IP (ip)

Only significant for Concurrent Funge. This ID differentiates this IP from all others currently in the IP list.

1. 1 cell containing a unique team number for the current IP (ip)

Only significant for NetFunge, BeGlad, and the like.

1. 1 vector containing the Funge-Space position of the current IP (ip)
2. 1 vector containing the Funge-Space delta of the current IP (ip)
3. 1 vector containing the Funge-Space storage offset of the current IP (ip)
4. 1 vector containing the least point which contains a non-space cell, relative to the origin (env)
5. 1 vector containing the greatest point which contains a non-space cell, relative to the least point (env)

These two vectors are useful to give to the o instruction to output the entire program source as a text file.

1. 1 cell containing current ((year - 1900) \* 256 \* 256) + (month \* 256) + (day of month) (env)
2. 1 cell containing current (hour \* 256 \* 256) + (minute \* 256) + (second) (env)
3. 1 cell containing the total number of stacks currently in use by the IP (size of stack stack) (ip)
4. *size-of-stack-stack* cells containing size of each stack, listed from TOSS to BOSS (ip)

Stack sizes are pushed as if they were measured **before** y began pushing elements onto the stack.

1. a series of sequences of characters (strings), each terminated by a null, the series terminated by an additional double null, containing the command-line arguments. (env)

This means any isolated argument can be a null string, but no two consecutive arguments may be null strings - a rather contrived scenario, null string arguments being rare in themselves.

The first string is the name of the Funge source program being run.

1. a series of strings, each terminated by a null, the series terminated by an additional null, containing the environment variables. (env)

The format for each variable setting is NAME=VALUE.

If y is given a positive argument, all these cells are pushed onto the stack as if the argument was non-positive. However, y then goes on to copy the *argument*th stack cell (counting from the top) into a temporary location, subsequently removing all the cells it pushed onto the stack. It then pushes the temporary cell onto the stack. For example, 3y will act as if only the handprint was pushed onto the stack.

An interesting side-effect of this behaviour is that if y is given an argument that exceeds the number of cells it pushes onto the stack, it can act as a 'pick' instruction on data that was on the stack before y was even executed.

## Scale: Extension and Customization

Funge-98 is totally customizable and scalable in terms of functionality, and non-trivial programs can now be written portably in it without any sort of directives. The fingerprint mechanism allows for the definition of a virtually unlimited number of libraries in the form of fingerprinted extensions. The handprint mechanism allows a program to identify exactly what interpreter it's running on.

### Handprints

A handprint is a bitstring which uniquely identifies a particular implementation (interpreter, compiler, etc.) of Funge-98.

These should really only be used by Funge-98 programs which know about **bugs** in a known interpreter, and want to work around them when their portable source is run on that interpreter. (*Features*, on the other hand, should be fingerprints: if everything is properly implemented, the current handprint **should** be irrelevant. Fingerprints should always be used in preference to handprints when making a design decision - handprints are a fallback for the less-than-ideal case.)

Handprints generally stay they same between revisions of an interpreter. They are accompanied by a version number when retrieved from y.

### Fingerprints

Extension and customization of Funge-98 are accomplished through the so-called "fingerprint mechanism". No such mechanism exists for Befunge-93.

To be more precise, a fingerprint is a unique ID code which indicates a library of routines (a *fingerprinted extension*) to be assigned temporarily to what the instructions A to Z do. Multiple loaded fingerprints can overlap and overload, so even object-oriented-like inheritance is possible.

Generally speaking, these new semantics and instructions are only available to and only apply to the IP which loaded them. The fingerprint spec itself may make exceptions to this rule, but it must clearly specify what they are.

The semantics of any given extension are generally coded directly into the interpreter. There is no reason, however, they may not be made available in a dynamically-loadable form; but there is no convenient, standard format for such a form.

Whether the semantics are dynamically loaded from a disk file containing some form of executable code, or hard-wired into the Funge interpreter, is really a moot point from the Funge programmer's perspective.

However, it is useful to explain at this point for the benefit of both Funge programmers and potential Funge extension writers, that there are two classes of fingerprinted extensions: those that interact with and change the behaviour of the underlying Funge Virtual Machine and/or are not re-entrant (*feral extensions*), and those which are self-contained and re-entrant (*tame extensions*).

The main difference is that a feral extension cannot simply or easily be "dropped into" a Funge interpreter which is not aware of it. When specifying fingerprinted extensions for public use, please try to make them as tame as possible - although many times it is unavoidable to have a feral fingerprint in order to accomplish a certain semantic goal, such as program control of proprietary interpreter features.

The ( "Load Semantics" instruction loads the semantics for a given fingerprint onto any or all of the instructions A to Z. ( pops a *count*. It then pops *count* cells. For each cell that it pops, it multiplies a temporary value (which is initially zero) by 256, and adds the cell value to it.

In this way, ( builds a fingerprint. This mechanism makes it simple to express large fingerprints like 0x452e472e in printable ASCII such as ".G.E"4( ... ), while not requiring the use of ASCII as the medium for all fingerprints.

( then uses this fingerprint to locate and load a set of semantics for the instructions A to Z. If the Funge implementation cannot find the correct library for the given fingerprint, ( acts like r. If, however, the semantic library load is successful, the new fingerprint, then a 1, are pushed onto the stack (to be accepted by a subsequent ) instruction.)

The corresponding ) "Unload Semantics" instruction unloads the semantics for a given fingerprint from any or all of the instructions A to Z (even if that fingerprint had never been loaded before). It pops a *count* and builds a fingerprint in exactly the same way.

() in Funge are kind of like "use x ... no x" in Perl. The interpreter-writer or interpreter-customizer is allowed to make fingerprints do anything they like, and they may implement fingerprints however they like, but they have to follow some general rules as a 'contract' between the fingerprint and the fingerprint user.

* A fingerprint should not affect the semantics of any instructions besides A to Z, except under exceptional circumstances which must be clearly documented in the fingerprint's spec.
* When loaded, a fingerprint which implements A and B semantics should act something like:

save(A); bind(A, myAsemantic()); save(B); bind(B, myBsemantic());

* When unloaded, the same fingerprint should act something like

restore(B); restore(A);

* In this sense, 'bind' means to change what the execution of an instruction does in the current Funge-98 program.
* 'save' means to save (push onto a stack) the semantics of an instruction for later use.
* 'restore' means to recall (pop from a stack) the semantics of an instruction from the last saved version.

This allows 'overloading' fingerprints. Say three fingerprints are implemented on your interpreter,

* E.G. which implements D as 'Destroy' and R as 'Restore'
* TEST which implements S as 'Send', R as 'Receive', and E as 'Encrypt'
* WORK which implements S as 'Slice', D as 'Dice', P as 'Puree'

With these, the Funge programmer ought to be able to make code like:

".G.E"4( ... "TSET"4( ... "KROW"4( ... S ... ) ... ) ... )

Here, the S instruction indicates the 'Slice' instruction in the WORK fingerprint, not 'Send' in TEST. But if it was written as:

"KROW"4( ... "TSET"4( ... ".G.E"4( ... S ... ) ... ) ... )

The S would indicate 'Send' from TEST, since there is no S in E.G., and a D instruction in the same location would indicate 'Destroy' in E.G., but P would be a 'Puree' from WORK, because it's not defined in either TEST or E.G.

### Funge Central Registry

The Funge Central Registry is an online database located at <http://www.cats-eye.com/funge/registry/>. Before developing and releasing your own Funge-98 implementation or extension, please register all applicable handprints and fingerprints you will require here.

This system is in place both to reduce conflicts between fingerprints and handprints world-wide, ensuring their uniqueness, and to provide a quick reference to all known extant handprints and fingerprints.

There really are no standard libraries in Befunge. A Funge-98 interpreter need not implement any fingerprints whatsoever. Funge-98 "standard" libraries are no more than "officially sanctioned" extensions, catalogued in the Registry by their fingerprints. Since extensions are forever accumulating, see the registry for a list of "standard" fingerprints.

## Appendix

### Instruction Quick Reference

/ modifiers:

* 98 Funge-98 only, not in Befunge-93.
* 2D Minimum 2 dimensions (not in Unefunge).
* 3D Minimum 3 dimensions (not in Unefunge or Befunge).
* c Concurrent Funge. Check y to see if these instructions are implemented.
* f Filesystem Funge. Check y to see if these instructions are implemented.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Decimal** | **ASCII** | **Instruction** | **Stack Before** | **Stack After** | **Other Effects** |
| *(bottom ... top)* | |
| 32 | space | Space |  |  | not normally executed |
| 33 | ! | Logical Not | b | NOT b |  |
| 34 | " | Toggle Stringmode |  |  | stringmode <- NOT stringmode |
| 35 | # | Trampoline |  |  | pos <- pos + delta |
| 36 | $ | Pop | n |  |  |
| 37 | % | Remainder | a b | a REM b |  |
| 38 | & | Input Integer |  | a | a = readint() |
| 39 | ' | Fetch Character/98 |  | c | pos <- pos + delta |
| 40 | ( | Load Semantics/98 | en..e1 n | f 1 | overloads A-Z |
| 41 | ) | Unload Semantics/98 | en..e1 n |  | unloads last A-Z |
| 42 | \* | Multiply | a b | a \* b |  |
| 43 | + | Add | a b | a + b |  |
| 44 | , | Output Character | c |  | writechar(c) |
| 45 | - | Subtract | a b | a - b |  |
| 46 | . | Output Integer | a |  | writeint(a) |
| 47 | / | Divide | a b | a / b |  |
| 48 | 0 | Push Zero |  | 0 |  |
| 49 | 1 | Push One |  | 1 |  |
| 50 | 2 | Push Two |  | 2 |  |
| 51 | 3 | Push Three |  | 3 |  |
| 52 | 4 | Push Four |  | 4 |  |
| 53 | 5 | Push Five |  | 5 |  |
| 54 | 6 | Push Six |  | 6 |  |
| 55 | 7 | Push Seven |  | 7 |  |
| 56 | 8 | Push Eight |  | 8 |  |
| 57 | 9 | Push Niner |  | 9 |  |
| 58 | : | Duplicate | v | v v |  |
| 59 | ; | Jump Over/98 |  |  | nothing executed until next semicolon |
| 60 | < | Go West |  |  | delta <- (-1,0) |
| 61 | = | Execute/98/f | STR | r | r = system-execute(STR) |
| 62 | > | Go East |  |  | delta <- (1,0) |
| 63 | ? | Go Away |  |  | delta <- (1,0)?(-1,0)?(0,1)?(0,-1) |
| 64 | @ | Stop |  |  | halt IP |
| 65-90 | A-Z |  |  |  | Fingerprint-Defined/98 |
| 91 | [ | Turn Left/98/2D |  |  | delta <- rot(-90, delta) |
| 92 | \ | Swap | a b | b a |  |
| 93 | ] | Turn Right/98/2D |  |  | delta <- rot(90, delta) |
| 94 | ^ | Go North/2D |  |  | delta <- (0,-1) |
| 95 | \_ | East-West If | b |  | delta <- if (b) (-1,0) else (1,0) |
| 96 | ` | Greater Than | a b | a > b | either 1 or 0 |
| 97 | a | Push Ten/98 |  | 10 |  |
| 98 | b | Push Eleven/98 |  | 11 |  |
| 99 | c | Push Twelve/98 |  | 12 |  |
| 100 | d | Push Thirteen/98 |  | 13 |  |
| 101 | e | Push Fourteen/98 |  | 14 |  |
| 102 | f | Push Fifteen/98 |  | 15 |  |
| 103 | g | Get | Va | v | v = fetch-funge-space(offset+Va) |
| 104 | h | Go High/98/3D |  |  | delta <- (0,0,-1) |
| 105 | i | Input File/98/f | Va f STR | Va Vb | inputs file |
| 106 | j | Jump Forward/98 | s |  | pos <- pos + delta \* s |
| 107 | k | Iterate/98 | n |  | execute next instruction now, n times |
| 108 | l | Go Low/98/3D |  |  | delta <- (0,0,1) |
| 109 | m | High-Low If/98/3D | b |  | delta <- if (b) (0,0,-1) else (0,0,1) |
| 110 | n | Clear Stack/98 | en..e1 |  |  |
| 111 | o | Output File/98/f | Va Vb f STR |  | outputs file |
| 112 | p | Put | v Va |  | store-funge-space(offset+Va,v) |
| 113 | q | Quit/98 | r |  | immediate exit, returncode = r |
| 114 | r | Reflect/98 |  |  | delta <- delta \* -1 |
| 115 | s | Store Character/98 | c |  | store-funge-space(position+delta,v) |
| 116 | t | Split/98/c |  |  | Split IP |
| 117 | u | Stack Under Stack/98 | n | (en..e1) |  |
| 118 | v | Go South/2D |  |  | delta <- (0,1) |
| 119 | w | Compare/98/2D | a b |  | if (a>b) ']' elsif (a<b) '[' else 'z' |
| 120 | x | Absolute Delta/98 | Va |  | delta <- Va |
| 121 | y | Get SysInfo/98 | c | en(..e1) |  |
| 122 | z | No Operation/98 |  |  |  |
| 123 | { | Begin Block/98 | en..e1 n | (en..e1) | offset <- pos + delta, etc |
| 124 | | | North-South If/2D | b |  | delta <- if (b) (0,-1) else (0,1) |
| 125 | } | End Block/98 | en..e1 n | (en..e1) | offset <- SOSS Va, etc |
| 126 | ~ | Input Character |  | c | c = readchar() |

### Concurrent Funge-98

Befunge-93 does not allow for multithreaded execution. However, Concurrent Funge-98, a superset of Funge-98, defines a list of any number of concurrently running instruction pointers called the *IP list*. In Concurrent Funge, IP's are sometimes called *threads* and each has its own location, delta, and stack.

You can also think of a Concurrent Funge-98 interpreter as having an internal and imaginary *clock* which produces *ticks* sequentially. Each tick, the IP list is processed: instructions encountered by each IP are dealt with in the sequence the IPs appear on the list, and each IP then moves as specified by its delta.

The list is always processed repetitively, sequentially, and in the same direction, and when IP's are deleted they fall out and the next one which would normally execute anyway actually executes.

Creating additional IP's is done with the t "Split" instruction, available in Concurrent Funge-98 only. It causes the current IP to be duplicated, and this duplicate is added to the IP list such that it is executed for the first time *before* the parent IP is next executed.

When a child IP is borne unto Funge-Space thus, its location, storage offset, and stack are all copied verbatim from the parent IP's. The child IP's delta is reversed (a la r) from its parent's, though.

The @ "Stop" instruction kills the current IP. If the current IP is the only active IP in the IP list, the program ends. If it was not the last in the IP list, there would then be a gap in the list: in this case the top part of the list "shifts down" and the next thread in the list is executed.

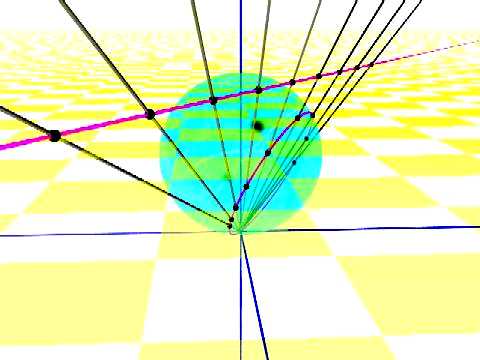
### Lahey-Space

Lahey-Space is a mathematical model of the space used in Funge-98.

The requirements for a line in Lahey-space are the following: Starting from the origin, no matter what direction you head, you eventually reach the origin. If you go the other way you reach the origin from the other direction.

We surmise that if you head out on a Lahey-line, you would eventually return along the same Lahey-line, heading in the same direction, but coming from the other direction. This would mean you would eventually repeat your path, and (in Funge terms) have consistant wrapping through unlimited space.

Now, imagine a sphere of radius one centered one unit above the origin. Imagine a plane placed two units above the origin, i.e. resting on top of the sphere.



One thing you might notice is that when you draw a straight line from the origin to any point on the plane, you intersect the sphere exactly twice, once at the origin, once in a unique location. In this manner, you can map the plane uniquely onto the sphere.

Now, each normal line from a point A on the plane to a point B on the plane can be transformed into a Lahey-line, which in our model is represented as an arc of the sphere containing both point A' on the sphere and point B' on the sphere. This arc can then be extended into a full circle, going 'around' the sphere to wrap.

### Other Topologies

As mentioned, Funge is a family of programming languages, and Befunge has many relatives and descendants. This document only covers Cartesian Funges. Other Funges, such as Honefunges (hex-net topology) and Kleinefunges (Klein-bottles) are certainly possible.

However, Advanced Funges may not find either torodial space *or* Lahey-space sufficient for complex topologies, so this spec provides a language for defining wrapping behaviour in a mathematical way.

We can define an a *wrapping function* W() along the lines of:

W(x,y) = (x<0 -> x:=79, x>79 -> x:=0, y<0 -> y:=24, y>24 -> y:=0)

for Befunge-93. Complex topologies can define their own wrapping functions. If these functions are strictly and clearly specified in the documentation of the Advanced Funge in question, it will save a lot of confusion to users, and is highly recommended.

Copyunder (u)1998 [Cat's-Eye Technologies](http://www.cats-eye.com/). See <http://www.cats-eye.com/license/> for license information.