Gibberish (programming language)

Gibberish is a Turing-complete (proof will come soon in the form of a BF interpreter written in Gibberish), stack-based programming language created by User:Javawizard. It's designed to be confusingly comprehensive. To that end, the instruction set is quite large, but every instruction is a letter, and each letter can mean something different depending on where it appears.

Contents [hide]

Syntax

With some exceptions, Gibberish is a character-based language. This means that every character in a Gibberish program (with the exception of inline strings, which use sets of square brackets) does a particular action. Gibberish is also a stack-based program, and, to an extent, reflective.

There are three instruction sets that can be used in a gibberish program. The commands `e`, `f`, `g`, and `x` are present in all of them with exactly the same meaning, and allow for switching between instruction sets. The command `[` is also present in all instruction sets, as are `0 - 9`.

A Gibberish program has a single stack that it can store data on. This is the only form of memory available to a Gibberish program. The stack can hold numbers and strings.

When a gibberish program is run, the interpreter starts from the beginning of the file and runs each character in sequence, starting with the first character. An instruction set is not specified when a gibberish program starts, so the first few instructions must switch the instruction set or use only commands, such as `[`, that are present on all instruction sets.

Spaces, tabs, and newlines are ignored when reading a gibberish program. All other invalid characters cause an error to occur.

Instructions

Below is a set of tables describing the specific characters that are available for use in a Gibberish program.

All Instruction Sets

These commands are present on all instruction sets, and can be used in a Gibberish program before an instruction set is selected.

Character	Name	Description
[	Inline String	Starts a special string sequence. Everything until the next "]" character is read, and the text between the "[" and the "]" is pushed onto the stack as a string. However, matching pairs of "[" and "]" are paired together, so that `[hello[]bye]` would push "hello[]bye" onto the stack.To embed an actual "]" character within a string, you could use `[test]3946eamagtec[text]c`, which would result in "test]text" at the top of the stack.
0 - 9	Inline Literal Number	Pushes the literal digit X (where X is the character, which is a number from 0 to 9) as a number onto the stack.
e	First Instruction Set	Switches to the first instruction set.
f	Second Instruction Set	Switches to the second instruction set.
g	Third Instruction Set	Switches to the third instruction set.
x	*n*th Instruction Set	Pops a number off of the stack and switches to the instruction set indicated by that number. The number should be 0, 1, 2, or 3. 0 causes there to be no active instruction set (which is the same state that the program is in when it first starts). 1, 2, and 3 switch to the first, second, and third instruction sets, respectively. For example, `1x` is the same as `e`.
j	Get Instruction Set	Pushes a number representing the current instruction set onto the stack. If this is the first, second, or third instruction set, then the number 1, 2, or 3, respectively, is pushed onto the stack.
z	Nop	Does nothing.

First Instruction Set

These commands are present in the first instruction set, which can be accessed with the `e` command.

Character	Name	Description
u	Duplicate	Duplicates the top item on the stack. This essentially makes a copy of the top item on the stack and pushes the copy onto the stack.
a	Add	Pops the top two items from the stack, adds them together, and pushes the result onto the stack.
s	Subtract	Subtracts the top item on the stack from the next item down, popping both items from the stack. The result is pushed onto the stack.
m	Multiply	Pops the top two items from the stack, multiplies them with each other, and pushes the result onto the stack.
d	Divide	Pops the top item from the stack, then divides the next item by it. Both items are popped from the stack. The result is pushed onto the stack.
t	To String	Pops the top item on the stack, which should be a number, and converts it to a string. This string is the pushed onto the stack.
i	To Number	Pops the top item off the stack, which should be a string, and parses it into a number. If it is not a valid number, then it is pushed back onto the stack as a string, so that the stack remains unmodified. If it is a valid number, a number representing it is pushed onto the stack.
c	Concatenate	Pops the top item from the stack, then pops the next item off of the stack. The first item is then concatenated onto the end of the second item, and the result is pushed onto the stack. Both items must be strings.
o	Output	Pops the top item off of the stack and prints it to stdout. If the item is a number, this command acts as if `t` had been run first. A newline is printed to stdout after printing the item.
q	Inline Output	Same as `o`, but doesn't print a newline after printing the item.
n	Read Character	Reads a single character from stdin and pushes it onto the stack as a number.
l	Read Line	Reads everything from stdin until the next newline and pushes it onto the stack as a string.
h	Substring	Pops the top three items from the stack, then computes a string consisting of the characters in the third item (which is a string), from the position indicated by the second item, inclusive, to the first item, exclusive. This string is then pushed onto the stack. Indexes are 0-based, and it is an error for an index to be out of range. For example, `[gibberish]37ho` would print "beri".
y	String Length	Pops the top item from the stack, which is a string, and pushes the length of that string onto the stack as a number.
v	Discard	Pops the top item from the stack and discards it.
p	Copy	Pops the top item from the stack, which should be a number. Then, copies the *n*th item (where *n* is the number just popped) on the stack and pushes the copy onto the stack. *n* is 0-based, so `0p` is the same as `u`.
k	Move	Pops the top item from the stack, which should be a number. Then, moves the *n*th item (where *n* is the number just popped) on the stack to the top of the stack, shifting all items higher up from it downward. *n* is 0-based, so `0k` is effectively a no-op.
r	Stack Size	Pushes a number representing the current size of the stack onto the stack. The number represents the size of the stack *before* the number is pushed on, so `123ro` outputs "3", not "4".

Second Instruction Set

These commands are present in the second instruction set, which can be accessed with the `f` command.

Character	Name	Description
u	Greater Than	Checks to see if the second item on the stack is greater than the top item. Both items are popped from the stack. If the second item is greater, the number 1 is pushed onto the stack. Otherwise, the number 0 is pushed onto the stack.
d	Less Than	Checks to see if the second item on the stack is less than the top item. Both items are popped from the stack. If the second item is less than the first, the number 1 is pushed onto the stack. Otherwise, the number 0 is pushed onto the stack.
s	Skip	Pops the first item off of the stack and skips that many instructions. `0r` does nothing. For example, `1111111333r3333qqqqqqq` would print "1111333". The number of instructions to skip must not be negative
t	Skip Two	Same as `fs`, but multiplies the number of instructions to skip by two before skipping. This allows blocks to be easily embedded; for example, to create a statement that runs a block of code if the top number on the stack is 1 or another block of code if the top number on the stack is 0 (essentially an if/else), you could do `eufnt[...]ct[...]c`, where the first ... runs if the top item on the stack is 1, and the second ... runs if the top item on the stack is 0.
p	Stack Insert	Pops the first item off the stack, which should be a number. Then pops the second item off of the stack, and inserts it into the stack at the position indicated by the first item. The item at that position is shifted toward the top of the stack, and everything else is shifted along with it. For example, `[Ping][Pong]0fpeqq` prints PongPing.
a	And	Pops the first two items off of the stack. If they are both 1, the number 1 is pushed onto the stack. Otherwise, then number 0 is pushed onto the stack.
o	Or	Pops the first two items off of the stack. If either item is 1, the number 1 is pushed onto the stack. Otherwise, the number 0 is pushed onto the stack.
n	Not	Pops the first item from the stack. If it is the number 1, pushes 0 onto the stack. Otherwise, the number 1 is pushed onto the stack.
c	Execute	Pops the first item from the stack, which should be a string, and runs it as Gibberish code. This is the easiest way to create block statements. A block "if" statement that executes an arbitrarily-long piece of code if the first item on the stack is 1 can be done with "fn2ms[...]c", where ... is the code to execute.
w	While	Pops the first item off of the stack and checks to see if it is 1. If it is, pops the next item, which should be a string, off of the stack and runs it. The next item is then popped and checked to see if it is 1. If it is, the same thing happens again.
q	Equal	Pops the first two items off of the stack. If they are equal, pushes 1 onto the stack. Otherwise, pushes 0 onto the stack.
l	Left Shift	Pops the first two numbers off of the stack, shifts the second one left by the number of bits specified by the first one, and pushes the result onto the stack.
r	Right Shift	Pops the first two numbers off of the stack, shifts the second one right by the number of bits specified by the first one, and pushes the result onto the stack.

Third Instruction Set

These commands are present in the third instruction set, which can be accessed with the `g` command.

Character	Name	Description
q	Quit	Stops execution of the currently-running Gibberish program.
w	Recall While	Same as `w` in the second instruction set, but this one stores the string to execute separately so that it doesn't have to be duplicated back onto the stack each time the loop runs. This also pops the string before it pops the number to check. Essentially, only the number (which indicates whether to keep looping or not) is popped each time through the loop, as opposed to `fw` which pops the code to run each time, too.
n	Is Number	Pops the first item off of the stack. If it's a number, pushes 1 onto the stack. Otherwise, pushes 0 onto the stack.
s	Is String	Pops the first item off of the stack. If it's a string, pushes 1 onto the stack. Otherwise, pushes 0 onto the stack.
a	Bitwise And	Pops the first two numbers off of the stack, computes the bitwise **and** of the numbers, and pushes the new number onto the stack. The results are undefined if the numbers are not whole integers, or if the numbers are negative.
o	Bitwise Or	Pops the first two numbers off of the stack, computes the bitwise **or** of the numbers, and pushes the new number onto the stack. The results are undefined if the numbers are not whole integers, or if the numbers are negative.
i	Integer	Converts the number on the stack to an integer by rounding it down toward negative infinity.
m	Modulus	Same as `ed`, but instead of dividing numbers, this computes the modulus of the numbers. For positive numbers, this is the same as the remainder.
t	To Char	Pops the first item off of the stack, which should be a number, and pushes a new string onto the stack consisting of exactly one character, which is the ASCII character represented by that number. For example, `188emagteo` prints "A".
c	Char At	Pops the first two items off of the stack, and gets the *n*th character from the second one (which should be a string), where *n* is the first one (except that *n* is 0-based), and pushes a number representing that character's value in ASCII onto the stack. For example, `[ABC]1gceo` prints "66", since `1gc` selects the second character in ABC, and B is ASCII 66.
r	Replace Char	Pops the first three items off of the stack, replaces the *n*th character in the third item (which is a string), where *n* is the second item (but *n* is 0-based), with the first item, which should be a string containing only a single character. The resulting string is then pushed onto the stack.
p	Inverted Copy	Same as `ep`, but the address to copy from is specified relative to the bottom of the stack.
k	Inverted Move	Same as `ek`, but the address to move from is specified relative to the bottom of the stack.
b	Swap	Swaps the top item on the stack with the item below it.
d	Swap 2	Swaps the top item on the stack with the one below the one below it.
h	Swap 3	Swaps the top item on the stack with the one below the one below the one below it. In other words, 2 items are left intact between the swapped items.

Examples

Hello, world!

```
[Hello, world!]eo
```

Number List

Asks the user to enter a number, then prints out the numbers from 1 to that number, separated by spaces.

```
[Type a number.]eoli1a1g1[euq[ ]q1au2pfqn]w[]eo
```

Quine

```
[eu91a9m1augteqgbeq2agteqo]eu91a9m1augteqgbeq2agteqo
```

Computational Class

By the following conversion, Gibberish can simulate the running of Underload programs. Since Underload is Turing-complete, so is Gibberish.

```
~ gb
: eu
! ev
* ec
( [
) ]
a e91a9m1agtbec91a9m3agtec
^ fc
S eq
```

Interpreter

A partially-complete interpreter, written in Java, is available here. The interpreter only implements a few instructions at present. If anyone wants to finish it up, feel free to contact Javawizard (see link at the top of the page).

A full interpreter written in Python is here. It should be a full implementation, but it is not an official one and it doesn't have Javawizard's blessing. It runs the examples fine though, and it implements every command.

See also

- Gibberish/JavaScript, another language named Gibberish.
- TextGarbage, Similar concept

Categories: Languages | Turing complete | Stack-based