The Kipple Language Specification

Kipple is a minimalistic, Turing-complete, stack-based programming language where data is stored in 26 stacks named *a*-*z*. The values on the stacks are 32-bit signed integers. The language consists of four operators and one control structure. Each operator takes one or two operands, and an operand can either be a non-negative integer (0-2147483647) or a stack identifier (*a*-*z*). Stack identifiers are case insensitive. Note that although an integer operand has to be zero or positive, the stacks can hold negative numbers as well (which can be accomplished with the - operator). All the stacks start out empty, except stack *i* which will contain the programs input (see Input and Output).

The operators

- **Push: > or <**
  Syntax: *Operand*>*StackIndentifier* or *StackIndentifier*<*Operand*
  The Push operator takes the operand to the left and pushes it onto the specified stack. E.g. `12>a` will push the value *12* onto stack *a*. `a>b` will pop the topmost value from stack *a* and push it onto stack *b*. Popping an empty stack always returns *0*. `a<b` is equivalent with `b>a`.
- **Add: +**
  Syntax: *StackIndentifier*+*Operand*
  The Add operator pushes the sum of the topmost item on the stack and the operand onto the stack. If the operand is a stack, then the value is popped from it. E.g. if the topmost value of stack *a* is *1*, then `a+2` will push *3* onto it. If *a* is empty, then `a+2` will push *2* onto it. If the topmost values of stack *a*and *b* are *1* and *2*, then `a+b` will pop the value *2* from stack *b* and push *3* onto stack *a*.
- **Subtract: -**
  Syntax: *StackIndentifier*-*Operand*
  The Subtract operator works exactly like the Add operator, except that it subtracts instead of adding.
- **Clear: ?**
  Syntax: *StackIndentifier***?**
  The Clear operator empties the stack if it's topmost item is *0*.

The interpreter will ignore anything that isn't next to an operator, so the following program would work: `a+2 this will be ignored c<i`. However, the proper way to add comments is by using the # character. Anything between a # and an End-of-line character is removed before execution. ASCII character #10 is defined as End-of-line in Kipple.

Operands may be shared by two operators. E.g. `a>b c>b c?` may be written as `a>b<c?`.

The program `1>a<2 a+a` will result in *a* containing the values *[1 4]* and **not** *[1 3]*. Likewise for the **-** operator.

The control structure

There is only one control structure in Kipple: the loop.
Syntax: **(***StackIndentifier code* **)**. As long as the specified stack is not empty, the *code* within the matching parentheses will be repeated. Loops may contain other loops. Example: `(a a>b)` will move all the values of stack *a* onto stack *b* (though the order will be reversed). A functionally identical, but more elegant way to do this is `(a>b)`.

Input and output

Before a Kipple program is executed, all input is pushed onto stack *i*. That means it's impossible to get input during program execution. Input is read as a string of bytes.

When a Kipple program ends the contents of stack *o* are written to output. Output is written as bytes, even though the stack contains 32-bit integers. The following program will write it's input to output (i.e. cat): `(i>o)`

To make output of numbers more convenient, the special stack @ has been added to the language. When a program tries to push a value onto stack @, the ASCII values of each digit is pushed onto it instead. E.g. while the program `100>o` will output "d" (d is ASCII #100), the program `100>@ (@>o)` will output "100".

Strings

There are no such things as strings in Kipple. However, the official interpreter has a preprocessor which translates strings (anything between double quotes) in the code to Kipple statements. *Note that this is a feature of the interpreter, not the language!*

Example: `o<"abc"` will be translated into `o<97 o<98 o<99`. `"abc">o` however, will be translated into `99>o 98>o 97>o`. In other words, the order in which the characters are pushed onto the stack is depending on which push operator is used. Escape characters are currently not supported. Using the preprocessor, Hello World can be written as simple as this: `"Hello World!">o`