Reverse Programmer Notation (RProgN)

Developed by William Lemon

# Abstract

The idea of this language was to try to write a stack based programming language using the syntax of Reverse Polish Notation. “*a b +”.* Although languages already exist that do this, eg. PostScript, None do it so that any **RPN** would implicitly work. Essentially, this was designed to Turing Complete Reverse Polish Notation.

Initially written in Lua, RProgN is an interpreted esoteric language. It contains two memory blocks, the **Reg <Stack>** and the **Mem <Associative Array>**.

The syntax itself is split up into whitespace separated ‘functions’, which are run from left to right. In RProgN, constants do not exist, instead, each number and string counts as a default ‘function’, which can be overwritten.

Strings are the only things that do not follow this rule, instead, when the interpreter is scanning across a word and finds either a “ or a ‘, the word is instead changed to a constructor for a string. See [Strings](#_Strings).

# Strings

In RProgN, Hello\_World’’ is a valid constructor for a string which places, (Without quotes) “Hello\_World” onto the stack. Spaces can exist in string constructors when between a pair of quotes, such that ‘Hello World’ is the constructor for “Hello World” (Without the quotes). Escapes, thus, do not exist in RProgN, and instead the programmer is expected to use the literal versons of these characters enclosed in strings. To represent a ‘, one could do ‘Hello ‘”’”’World’”’”, This is a closing apostrophe, followed by an opening quotation mark, which then can freely use the apostrophe, than the closing quotation mark. Then, another apostrophe can be used to finish writing “World”. Because a whitespace character was not used between the two different quote Types, the word never ended, and thus, it is automatically appended. More simply, one should use “Hello ‘World’”. New lines, null characters and that in the like do not terminate a string, only the same mark used to start the string can, as such, new lines and that in the like can be represented via a quote, followed by a literal new line, than another quote.

# Flow of Control

Flow of Control in RProgN is handled through specific functions which push the read pointer to various parts of the script. As per default, the functions “if”, “while”, “for”, “function”, and “else” (yes, else is a function) will push the pointer ahead of the matching ‘end’ or ‘else’. When encountered, the functions themselves determine the correct information, such as the case for ‘for’ statements, and the ‘(i=a; i<=b; i+=c)’ values for the for loop, and will either continue reading if it should, or shunt to the matching else/end if it should do that instead.

## ‘if’ statements

The If statement simply peeks the top of the stack, checks if it’s a truthy value, and will either continue reading the script, or shunt to the matching else/end otherwise.

## ‘while’ statements

While statements work exactly like the if statement, except when they encounter their matching end, they return to the while statement, and check the top of the stack again, and will repeat this until the top of the stack is falsey when encountering the while.

## ‘for’ statements

When a for statement is encounted, the top 3 values of the stack are popped.

In order from top to bottom, the values are ‘a’, ‘b’, and ‘c’.  
An iterator is created, starting at value a, until and including value b, it will increment by c each loop, which may be negative or 0. In each iteration, the value of the iterator is pushed to the top of the stack.

## ‘else’ statement

The else statement, or more accurately, function, will regardless of contents, jump to the matching end. As the else statement is usable in all flow of control functions, it’s possible to use it to define an, ‘After this is finished, do this’ type statement. Yet it’s hardly useful for anything other than if statements.

# Functions

Functions are the main data type of RProgN, each constant value is actually defined as a function that pushes what it represents to the reg. ‘1’ is internally defined (in the Lua representation) as ‘reg.push(1)’. As such, it’s possible to redefine the nature of (almost) everything in RProgN, the only exception being String Constructors, which require special behaviour to exist. This includes flow of control functions.

## Default Functions

As per default, RProgN should contain the following functions where a,b,c... are values from the top of the stack down, implicitly popped.

* ‘-‘ : pushes b - a
* ‘\*’: pushes b \* a
* ‘/’: pushes b / a
* ‘//’: pushes the integral division of b / a
* ‘%’: Returns the modulas of a % b
* ‘^’: pushes b to the power of a
* ‘]’: clones the top of the stack
* ‘[‘: pops the top value of the stack
* ‘\’: pushes a, then b, so that a and b switch places.
* ‘==’: pushes true if a = b, false otherwise
* ‘=’: See above
* ‘>=’, ‘>’, ‘<=’, ‘<’, ‘!=’: Comparative functions, such that a ‘’ b
* ‘getraw’: pushes the value associated with a to the stack. Used normally get functions instead of calling them
* ‘Q’: pushes the raw inputted string of the current function, or code itself to the stack.
* ‘asoc’: Associates value b to the string a, such that writing a will be interpreted as a call to b. If a is a stack, then b will be a reference to c on the stack a.
* ‘recall’: Similar to getraw, but from the global mem table. Depreciated.
* ‘char’: Pushes the ASCII character associated with a
* ‘byte’: Pushes the ASCII value of a
* ‘max’: pushes the larger of a and b
* ‘min’: pushes the smaller of a and b
* ‘p’: outputs a
* ‘rand’: pushes a random float between a and b
* ‘randomseed’: seed the random generator with a
* ‘time’: pushes the current os time in Unix Time
* ‘len’: pushes the size of a, the such that if a is a string, then the length of that string, or if a is a stack, the size of that stack
* ‘floor’: pushes the floor of a
* ‘sub’: pushes the substring of c between a and b
* ‘do’: Interpret a as RProgN
* ‘stack’: pushes a new stack (Which also doubles as an associative array)
* ‘push’: push a to b
* ‘pop’: pop from a to the stack
* ‘peek’: push the top value of a
* ‘mem’: push the mem associative array
* ‘reg’: push the reg stack, to the reg stack...
* ‘get’: push the value of a from array b
* ‘set’: set the value of b of array c to a
* ‘truthy’: push true if the value is truthy, false otherwise.
* ‘debug’: Output debug information.
* ‘local’: push the ‘local’ array to the stack. Useful with asoc, as local is also accessible from the current namespace.
* ‘find’: push the index in which a is first found in b
* ‘split’: push a stack containing all individual substrings of b that match a
* ‘tostack’: push a stack containing each individual substring of a to the stack.
* ‘inverse’: push an upside down version of a
* ‘invert’: flip a upside down

## Defining Functions

Defining a function is simply done with a ‘function’ ‘end’ pair, such that.

function

-- CODE GOES HERE --

end

The above snippet would push a function containing the inner code to the top of the stack. To name it, and as such, use it, one would require associating it to a name, as below.

function

-- CODE GOES HERE --

end 'HelloWorld' asoc