Understanding MiniScript

Covers Version 1.3

Understanding MiniScript is a free webbook written and distributed by Daniel Lawson.
https://github.com/Sombrero64/MiniScript

Table of Contents

Preface

Ever scene I started programming, I wanted to create my own programming language with the features and syntax I wanted. Unfortunately, I didn't have the experience to design and program an interpreter of a complex language. However, after working on a game engine with its own basic scripting language, I decided to take what I learn so far and created **MiniScript**, a small and basic programming language with basic list and procedure support. The syntax is basic:

1. Entries (strings) are separated by spaces, which can be prevented by using quotes or apostrophes.
2. When using either quotes or apostrophes to make inputs, you must use their respective character to close the string. In other words, quotes closes quotes, and apostrophes closes apostrophes.
3. The first entry is the command's name, and the rest are inputs.
4. To get a variable's value, you write a dollar sign then the name, like **$foo**.
5. The dollar sign must be at the beginning of the entry to call a variable.
6. Comments uses the number sign or hash, and it must be at the beginning of the line.

Installing and Running

the MiniScript repository and a Lua interpreter/IDE. After downloading MiniScript, make sure you have all scripts in one folder. You don't need to download any Lua library or script, everything you need to run MiniScript is already available. When you got everything ready, run *miniscript.lua* in Lua, and if anything is working, you should see this in the console.

```
MiniScript Version 1.2
Use the run command to run a script.
Use the help command for more info on each command.
>>>
```

Type **print "Hello, world!"** to the interpreter. After submitting that command, you should see "Hello, world!" printed to the console.

```
>>> print "Hello, world!"
Hello, world!
>>>
```

If this was the case, then congrats! You made your first MiniScript program, the iconic Hello, world! Before you continue any further, create a new file in the MiniScript folder and named it whatever you want. Every time you wanted to load in a program, use the **run** command.

```
>>> run program
>>>
```

Lastly, if you needed help understand what a command does, use the **help** command. Typing just that command with no inputs lists every available command.

```
>>> help print
print ...
displays strings to the console
```

Programming in MiniScript

Variables with Math and Strings

A variable stores a string, which can be numbers and Booleans. To change the value of a variable, use the **set** command, followed with the name of your variable and the new value. To get the value of a variable, use a dollar sign followed with the name. This program below should have printed out **42** and **cat** to the console.

```
set number 42
set string cat
print $number
print $string
```

When you no longer need a variable, list, or procedure, you can use the **delete** command. If you need to remove all data, use the **reset** command.

```
delete variable foo
delete list bar
delete procedure moo
```

The **ask** command asks the user for some input and returns it as a string.

```
print "what's your name?"
ask name
print "hello" $name
```

These four commands, **add**, **subtract**, **multiply**, and **divide**, calculate their respective operator on each input and returns the result. The **remainder** command is like divide, but it returns the remainder instead of the quotient. Theses variables stores 4, 5, 24, 25, and 2 respectively.

```
add var1 2 2
subtract var2 10 5
multiply var3 6 4
divide var4 100 4
remainder var5 6 4
```

The **join** command creates a longer string from each input. This outputs “smart fella”.

```
join output "smart " "fella"
print $output
```

The **round** command rounds a number and returns the result. You can either make the round conditional or only up or down by supplying a third argument. These variables below are 4, 6, 6, and 5 respectively.

```
round var1 4.3
round var2 5.6
round var3 5.5 up
round var4 5.5 down
```

The **random** command generates a random number between two numbers. Use the round command in conjunction for random integers. The variable can store numbers between 1 to 10, like 2 or 6.

```
random number 1 10
round number $number
```

The **absolute** command returns the absolute value of a number, and the **length** command returns the size of a string (the number of characters in it). These variables below are 5.25, 3.5, and 4 respectively.

```
absolute pos 5.25
absolute neg -3.5
length len cake
```

The **sub** command returns a section of characters from a string, or a character without a fourth input. Variables *letter* and *fox* are “f” and “fox”.

```
set word "fast fox"
sub letter $word 1
sub fox $word 6 8
```

Logic and Control Flow

A Boolean or Boolean value is a string that stores either “true” or “false”, and this is used in logic calculations and control flow. The **not** command gets the opposite of the input and returns it: true returns as false, false returns as true. The value of *truth* is false.

```
set truth true
not truth $truth
```

The **any** and **all** commands checks which inputs that are true, but any returns true if at least one is true, while all returns true if all are true. Both *var1* and *var2* are true.

```
any var1 true false
all var2 true true
```

The **equal** command checks the equality of all inputs. If the inputs can be evaluated as numbers, it checks their numeral equality instead. If all inputs are equal, it returns true. Variables *var1* and *var3* are true, while *var2* and *var4* are false.

```
equal var1 a a
equal var2 ab c
equal var3 4.0 4
equal var4 6 7.3
```

The **order** command checks if the inputs are ordered correctly from greatest to smallest. Only *var1* is true, the other two are false.

```
order var1 67 32 12
order var2 82 35 42
order var3 74 32 32
```

The **goto** command tells the interpreter to set the line number to the command's input.

```
print hi
goto 1
```

The **goif** command functions almost identical to goto, but it changes line numbers depending on the Boolean input's value. If the first input is false without a third input, it does nothing.

```
set number 0
add number $number 1
print $number
equal loop $number 10
goif $loop 1 3
```

Labels stores a number, which can be used goto and goif. To define a label, use the **label** command. Its number is defaulted to its line number, but with a second input, you can set its number to something else.

```
label loop
print hello
goto loop
```

The **check** command returns two values depending on the Boolean value. If the Boolean value is true, return the third input, or the fourth input if false. When the Boolean value is false without a fourth input, the output variable remains unchanged. Variable var1 is set to “off”, and var2 is set to “on”.

```
set true "on"
set false "off"
check var1 false $true $false
check var2 true $true $false
```

The **parse** command parses a string as code then attempts to run it. This can be useful if you want to run generated code.

```
parse "print hello"
```

The **if** command parses and run either two strings depending on the Boolean value. If the first input is false without a third input, it does nothing. This outputs “quick fox” and “lazy dog”.

```
set line1 "print 'quick fox'"
set line2 "print 'lazy dog'"
if false $line1 $line2
if true $line1 $line2
```

Lists and Procedures

A list stores unlimited amount of items, which stores a string value and an index. To define a list and set the items in a list, use the **list** command.

```
list fruits "apples" "bananas" "oranges"
```

The **insert**, **delete**, and **replace** commands are used to manage the items in a list.

- insert creates a new item at a specified index or at the end of the list
- remove deletes an item with its index
- replace changes the value of an item with its index

```
list fruits "apples" "bananas" "oranges"
insert fruits grapes
delete fruits 2
replace fruits 1 pears
```

To get items in a list, you can either use **item** or **find**. The item command returns the value of an item by its index, and the find command returns the index of the first item that equals to the target value. Both commands return nothing if it cannot find an item. Theses variables are “apples” and 2 respectively.

```
list fruits "apples" "bananas" "oranges"
item apples fruits 1
find bananas fruits "bananas"
```

The size command returns the number of items in a list. Variable output is 3.

```
list fruits "apples" "bananas" "oranges"
size output fruits
```

A procedure stores code that can be used multiple times without the need to constantly use gotos. The **define** command creates a procedure with the lines between it and the **end** keyword as its code. Use the **call** command with the procedure's name to run its contents.

```
define hello
print "hello"
end

call hello
```

You can provide inputs to your procedures. Define a list and write its name as a second input to the define command.

```
list commandInputs
define command commandInputs
item input1 commandInputs 1
print $input1
end

call command 42
```

To provide outputs for your procedures, you can use the input list's first item as a name of a variable you want to output it.

```
list commandInputs
define command commandInputs
item input1 commandInputs 1
set $input1 pie
end

call command output
print $output
```

Editing MiniScript

At this point, this includes the overview of MiniScript. This chapter covers how to edit the MiniScript Lua files to introduce new features and commands.

- miniscript: main script, some interpreter code and all commands are stored here
  - index: line number
  - program: contents of the running program
  - data: program data
  - commands: interpreter commands
  - functions: functions.lua module
  - strings: strings.lua module
- functions: some additional functions
- strings: some additional strings

Interpreter Functions

Name	Description
parse	Parses a string and returns the name and inputs.
call	Finds then runs a command by using its name and inputs.
combine	Combines all inputs by using a local function.
functions.round	Rounds a number either up or down depending on its value.
functions.randomFloat	Generates a random float.
functions.checkFile	Checks if a file exists on the computer.
functions.readFile	Reads the contents of a file and converts it into a table with strings.
functions.printTable	Prints each item from a table to the console.
functions.firstOfTable	Returns the first item of a table and its table without it.

Creating Commands

All the commands are stored the **commands** variable. The “design” of a command follows these principles.

- additional inputs are separated with ~, applying that command will still work without the inputs on afterwards.
- endless or infinite inputs are applied with …
- option inputs are applied with /, applying that this input is limited to some strings.

```
-- example input1 input2 ~ input3a/input3b ...
["example"] = function(args)

end
```

Here’s a complete example of a command, which returns the square root of a number.

```
-- square return number
commands["square"] = function(args)
	local output, number = args[1], tonumber(args[2])
	if output ~= nil and number ~= nil then
		local squared = math.sqrt(number)
		data.variables[output] = squared
	end
end
```