MINISCRIPT 2.0 DOCUMENTATION

Written and distributed by Daniel Lawson.
https://github.com/Sombrero64/MiniScript

TABLE OF CONTENTS

PREFACE AND INTRODUCTION

Ever scene I started programming, I wanted to create my own programming language with the features and syntax I wanted. Unfortunately, around that time, I didn't have the knowledge to design and write an interpreter for any language, let alone a complex one. The only thing I can do related to designing programming languages is to create the appearance and functions of one. However, after working on a game engine with its own scripting language, I decided to take what I learned and created *MiniScript* (the name was originally *BijouScript*, however bijou meant small and elegant, and I wasn't sure that I would call MiniScript elegant.)

It's a basic programming language that features basic list and procedure support, and the possible behaviors are made up of **commands**, which each does something with its inputs and may return something. The syntax and semantics are pretty basic to understand, therefore easier and faster to implement compared to the modern programming language.

1. Entries or strings are separated by spaces. Using quotes or apostrophes prevents this, so you can use spaces in one entry. When doing this, you must use the respective character to close the string. The first entry is the name of a command, and the rest are inputs or arguments.
2. To write another command as an input, you use parentheses around that command. All entries to a command must be in the same line, including command inputs and their parentheses.
3. Variables stores a name and a string value. To call a variable, use the dollar sign followed the name like **$name**. You can name a variable with multiple spaces and use quotes to call it, but the dollar sign must be at the beginning of the entry to call a variable.
4. Comments uses the number sign or hash, and it can be either place on one line or at the end of a line. Hashes cannot be used within quotes, and there's no way to close a comment.
5. Procedures collects code between the command to **end**. Procedures cannot recognize which end it is theirs, so it's impossible to define a procedure within a procedure. Same case in the interpreter's console, because how it handles commands differently compared to the interpreter itself.
6. Whitespace (spaces and tabs) from the left and comments to the right are removed while parsing.

```
set number 0
label loop
      set number (add $number 1)
      set fizz (equal (remainder $number 3) 0)
      set buzz (equal (remainder $number 5) 0)
      set output (join (check $fizz "Fizz" "") (check $buzz "Buzz" ""))
      print (check (equal $output "") $number $output)
goto (check (not (equal $number 100)) loop)
```

After version 1.4 is completed, I decided that I wanted to “up my game” and improve the interpreter. However, to implement everything mentioned below, I need to rewrite the interpreter and every command.

- Procedures can be called as commands without the need of the **call** command. Commands are checked first, then procedures.
- Commands and procedures can return a value without the need to set the value of variable, and you can exit out procedures while calling them.
- Whitespace is removed from the left, and comments are removed from the right.
- Source code is cleaner and more organized, therefore, more manageable.

The goal of MiniScript is to improve my language design so I can create more robust and professional languages that people would actually want to use. And the goal of this documentation is to help you get fluent in the language and understand the interpreter and how to modify it to introduce new features.

INSTALLING AND RUNNING

The source code of MiniScript can be downloaded at https://github.com/Sombrero64/MiniScript, make sure you download **version 2.0** from the repository. Afterwards, make sure you have a Lua interpreter or IDE with you that runs **version 1.5.1** or later. You don't need to install any other module or library, everything included in the source code is all you need. Set the MiniScript folder as the project directory and run **miniscript.lua**.

INTERPRETER INTRODUCTION

To input a command, write the name then the inputs. If it outputs a value, it will be printed to the console.

```
>>> print "Hello, world!"
Hello, world!
>>> add 2 2
4
>>>
```

To use a command as an input for another command, write the command around parentheses.

```
>>> multiply (add 2 2) (add 6 4)
40
>>>
```

To keep notes how your program works, use commands with the number sign or hash. Anything after that would be part of the comment, and therefore be ignored while parsing.

```
>>> print dog cat
dog   cat
>>> print dog #cat
dog
```

If you need help on a command, use the **help** command with the name of the command. To list every available command, do not include an input.

```
>>> help print

== print ==
print ...
...: Strings

prints all items in ... to the console in one line

print "Hello, world!"
```

GETTING STARTED

You are going to use a text document for the programs. Create a text document in the MiniScript file and named it whatever you want. Just make sure the name doesn't include spaces. To run a program, use the **run** command with the path to the file. If you want to run the previous file, do not include an input.

```
>>> run program.txt
```

In the program, write this command and run it. As you see in the console is whatever is the string is, which is "Hello, world!"

Feel free to change the string to anything else before continuing.

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

```
>>> run program.txt
Hello, world!
>>>
```

PROGRAMMING IN MINISCRIPT

INTRODUCTION TO VARIABLES

A variable is a data type (an entry to program data) that remembers a name and a string value, which may be considered a number or Boolean. To set the value of a variable, use the **set** command with the variable's name and a string value. To get a value of a variable, write a dollar sign at the beginning of an entry, followed with the name of a variable. If no such variable exists, it returns an empty string.

```
set foo 42
print $foo
```

When you no longer need to use a variable, you can remove its entry to save space. Use the **delete** command with the data type ("variable", "list", "procedure", and "label") and name. You can also remove all program data by setting the first input to "all".

```
set foo 42
print $foo
delete variable foo
print $foo
```

VARIABLES WITH MATH

Four commands: **add**, **subtract**, **multiply**, and **divide**, calculate on each input (starting on the first one, each are converted to numbers) with their respective operator then returns the result.

```
set a (add 2 2)
set b (subtract 10 5)
set c (multiply 6 4)
set d (divide 100 4)
```

The **remainder** command functions similarly to divide, but it returns remainders instead of quotients. Use this with equal to check if a number is a multiple of another number. Mess around with the values of x and y in this example below and see what you get.

```
set x 15
set y 5
print (equal (remainder $x $y) 0)
```

ADDITIONAL COMMANDS

The **power** (^) command returns the result of a number to the power of another number.

```
print (power 4 3)
```

The **round** command rounds a number to the nearest integer depending on its value.
To round only up or down, write “up” or “down” as a second input.

```
print (round 6.5) (round 6.2) (round 6.7)
print (round 8.5 down) (round 8.5 up)
```

The **random** command generates a random number between two numbers.
Use it with round for random integers.

```
print (random 1 10)
print (round (random 1 10) down)
```

ADVANCED CALUTATING

The calculate command is used to calculate one or two numbers with a specific function and returns the result. There are over 9 supported functions.

Function	Description
absolute	Absolute value of X
square	Square root of X
sin	Sine of X
cos	Cosine of X
tan	Tangent of X
asin	Arc sin of X
acos	Arc cosine of X
atan	Arc tangent of X
log	Logarithm of X (in Y)

```
print (calculate absolute -6.5)
print (calculate square 4)

print (calculate sin 5)
print (calculate cos 5)
print (calculate tan 5)

print (calculate log 5)
print (calculate log 5 3)
```

VARIABLES WITH STRINGS

The **length** command returns the length (amount of characters) of a string.

```
print (length "abc")
print (length 42.3)
print (length true)
```

The **lower** and **upper** commands both changes the capitalization of letters in a string to either lowercase or uppercase respectively.

```
set foo "This IS a TEST of this Function"
print (lower $foo)
print (upper $foo)
```

The **sub** command returns a section of characters in a string between two numbers. Without a third input, it rather gets the character at the second input.

```
set foo "fast fox"
print (sub $foo 6 8)
print (sub $foo 3)
```

VARIABLES WITH LOGIC

The Boolean value of a string (**true**/**false**) is used in logic calculations and control flow. There are 6 commands that uses these values. The **not** command returns the opposite of a Boolean value. True to false, and vice versa.

```
print (not false)
print (not true)
```

The **any** and **all** commands checks the value of each Boolean input and returns true if the amount required matched the requested amount. Command any is true when atleast one is true, and all is true if all are true.

```
print (all false false) (all true false) (all true true)
print (any false false) (any true false) (any true true)
```

The **equal** command returns true if all inputs are equal, and the **order** command returns true if all inputs are correctly ordered from greatest to smallest, and all inputs are not equal. Before checking the values, it first checks if all inputs can be converted into numbers. If so, then evaluate all inputs as numbers instead as strings.

```
print (equal "abc" "abc") (equal "ABC" "abc")
print (equal 5.0 5) (equal 6.5 3.2) (equal 4 4 a)

print (order 7.5 3.2 1) (order 8 2 4)
print (order a b) (order A a)
```

The **check** command checks the first input's Boolean value then return either the second (returned if **true**) or third input (returned if **false**) depending on its value.

```
set x "on"
set y "off"

print (check true $x $y)
print (check false $x $y)
```

GOTO AND LABELS

The **goto** function tells the interpreter to set the line number to a number value or the value of a label.

```
print "hello"
goto 1
```

To define a label, use the **label** command followed with the name. Afterwards, use the label's name as a goto input. Its value is set to the line number it is defined at, but you can change the value for a specific line, which can be useful for labels for later lines.

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

```
label skip 5
goto skip

print "foo"
print "bar"
```

CONTROL FLOW WITH GOTOS

Using gotos and labels with the check command, you can make conditional statements, which does either two things depending on the value. Mess around the value of statement to see what you get.

```
label if 8
label else 12
label continue 16

set statement true
goto (check $statement if else)

# if true
print "is true"
goto continue

# if false
print "is false"
goto continue

# end
print "finish"
```

Looping is possible too for repeating code for numbers or items in a list.

```
set number 0
set break false
label loop
      set number (add $number 1)
      set break (not (equal $number 10))
      print $number
goto (check $break loop)
```

LISTS

A list is a data type that stores multiple items and remembers each item's value and index. There are 7 commands that are used to manage and receive items in a list. To set the items in a list, use the **list** item with the name and all items.

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

To add a new item in a list, use the **insert** command with the list's name and the item's value. By default, it's placed at the end of the list, but you can specify its index to any place in the list.

```
insert fruits "grapes"
insert fruits "pears" 2
```

To remove an item from a list, use the **remove** command with the name and index.

```
remove fruits 2
```

To set the value of an item from a list, use the **replace** command with the name, index, and new value.

```
replace fruits 1 "pineapples"
```

To get an item from a list, you can either use the item or find command. The **item** command returns the value of an item at a specific index, and the **find** command finds the first item that equals to a target value and either returns its index or nothing depending on if it's found.

```
print (item fruits 1)
print (find fruits "bananas")
```

To get the number of items in a list, use the **size** command with the name.

```
print (size fruits)
```

PROCEDURES

A **procedure** is a data type that stores code, and the name of itself and the input list. Procedures can be used to minimize and organize code. To define a procedure, use the define command with name. Any code between that command and the end keyword. Afterwards, use the procedure's name as a command to the procedure's contents.

```
define procedure
      print "Hello, world!"
end

procedure
```

To break out of a procedure, use the exit command. Supplying an additional input will make the procedure call output a value.

```
define foo
      print "test 1"
      exit
      print "test 2"
end

define bar
      exit (add 2 2)
end

foo
print (bar)
```

Procedures also has inputs, supply a second input, which is the name of a list. When you call a procedure, the items in that list will be set to the procedure's inputs.

```
define root rootInputs
      set number (item rootInputs 1)
      exit (multiply $number $number)
end

print (root 4)
print (root (root 3) (root 2))
```

It's important to know that the line number is temporarily set back to one while calling a procedure. Keep this to mind while using gotos and labels in procedures.

```
define print10s
      set number 0
      set break false
      label loop
            set number (add $number 1)
            set break (not (equal $number 10))
            print $number
      goto (check $break loop)
end

print10s
```

CONDITIONAL PROCEDURES

You can also make conditional statements with procedures and the check command. Mess around the value of statement to see what you get.

```
define if
      print "is true"
end

define else
      print "is false"
end

set statement true
set name (check $statement if else)
$name
```

MINISCRIPT COMMANDS

Here's a complete list of every command available in MiniScript, use the **help** command for more information.

#	Name	Description	Inputs
1	set	Sets the value of NAME to VALUE.	NAME – Variable Name VALUE – Variable Value
2	delete	Deletes NAME of TYPE. If TYPE is "all", includes all program data.	TYPE – Data Type NAME – Name
3	list	Sets the items in NAME to …	NAME – List Name … – Items
4	insert	Adds a new item of value of VALUE at INDEX (or at the end of a list) at LIST,	LIST – List Name VALUE – Item Value INDEX – Item Index
5	remove	Removes an item from LIST at INDEX.	LIST – List Name INDEX – Item Index
6	replace	Sets the value of an item at INDEX from LIST to VALUE.	LIST – List Name INDEX – Item Index VALUE – Item Value
7	item	Returns the value of an item at INDEX form LIST.	LIST – List Name INDEX – Item Index
8	find	Finds the first item in LIST that equals to VALUE and returns the item's index.	LIST – List Name VALUE – Target Value
9	size	Returns the number of items in LIST.	LIST – List Name
10	add	Adds each number in … and returns the result, starts with the first item.	… – Numbers
11	subtract	Subtracts each number in … and returns the result, starts with the first item.	… – Numbers
12	multiply	Multiplies each number in … and returns the result, starts with the first item.	… – Numbers
13	divide	Divides each number in … and returns the result, starts with the first item.	… – Numbers
14	remainder	Divides each number in … and returns the remainder, starts with the first item.	… – Numbers
15	join	Joins each string in … and returns the result, starts with the first item.	… – Strings

16	round	Rounds NUMBER to the nearest integer depending on its value, or only up or down (FORCE).	NUMBER – Number FORCE – Rounding Force (up/down)
17	random	Returns a random number between MIN and MAX.	MIN – Minimum MAX – Maximum
18	power	Returns NUMBER to the power of REPEATER	NUMBER – Number 1 REPEATER – Number 2
19	calculate	Calculates X and Y with OPERATOR. Supported: absolute, square, sin, cos, tan, asin, acos, atan, and log.	OPERATOR – Math Function X – Number 1 Y – Number 2
20	length	Returns the length (amount of characters) in STRING.	STRING – String
21	sub	Returns a section of STRING between START to END, or the character at START.	STRING – String START – Start / Index END – End
22	lower	Returns STRING lowercased.	STRING – String
23	upper	Returns STRING uppercased.	STRING – String
24	not	Returns the opposite of BOOLEAN.	BOOLEAN – Boolean Value
25	all	Returns true if all Booleans in … is true.	… – Booleans
26	any	Returns true if at least one Boolean in … is true.	… – Booleans
27	equal	Returns true if all in … are equal. If all inputs can be converted into numbers, evaluate as numbers instead as strings.	… – Values
28	order	Returns true if all in … are correctly ordered from greatest to smallest and are not equal. If all inputs can be converted into numbers, evaluate as numbers instead as strings.	… – Values
29	check	Returns either IF or ELSE depending on the Boolean value of BOOLEAN. IF when true, or ELSE when false.	BOOLEAN – Boolean Value IF – Value when True ELSE – Value when False
30	print	Prints … to the console.	… – Values
31	ask	Asks the user for some input and returns it.	
32	goto	Tells the interpreter to set its line number to LINE (which is either a number or label).	LINE – Number / Label Name

33	label	Defines a goto label named NAME. By default, its value is the current line number, but it can be specified with LINE.	NAME – Label Name LINE – Label Value
34	define	Defines a procedure named NAME, which stores code between the command's instance to the end keyword. Sets the items of INPUTS when called for procedure inputs.	NAME – Procedure Name INPUTS – Input List Name
35	exit	Stops procedure calling and reports VALUE.	VALUE – Procedure Output
36	run	Finds the file located at PATH and gets the contents, then runs it as code. Without PATH, use the previous file path.	PATH – File Path
37	help	Returns information on a command. Without COMMAND, display all available commands.	COMMAND – Command Name
38	about	Displays information on MiniScript.	

EDITING MINISCRIPT

MINISCRIPT OVERVIEW

- modules: modules
  - data: data module, containing the code to every command.
  - parser: parser
  - functions: additional functions
  - help: content for the **help** command
- miniscript: interpreter
- LICENSE: MiniScript license

DATA OVERVIEW

- content – module variable
  - data – program data
    - commands
      - name = function
    - variables
      - name = value
    - lists
      - name = items
    - procedures
      - name = {function, inputs name}
    - labels
      - name = value
  - program
    - contents – program contents
    - index – line number
    - report – procedure call output
    - break – procedure call breaking
  - interpreter
    - filepath – file path for the **run** command

FUNCTIONS OVERVIEW

Name	Description
functions, find	Finds the first item TABLE that equals to TARGET and returns it's index/key, or **nil** if not found.
functions, contains	A manageable variation of the find function, returns a Boolean depending on if TALBE contains TARGET.
functions, nilCheck	Returns true if all arguments are not nil.
functions, round	Rounds NUMBER either up or down depending on the value.
functions, randomFloat	Returns a random float number between MIN and MAX.
functions, checkFile	Attempts to find FILE and returns true if such file exists.
functions, readFile	Returns the contents of FILE as a table, returns an empty table if not found.
functions, printTable	Prints the contents of TABLE.
data, getDataType	Finds an entry in TYPENAME as NAME. Defines if as DEFAULT if not found.
data, getList	A manageable variation for lists.
data, getProcedure	A manageable variation for procedures.
data, comparableAsNumbers	Checks if all items in INPUTS can be converted as numbers.
data, combine	Combine all items in INPUTS with the output of CODE and returns the combination. Starts with the first one, and the arguments of CODE should two.
data, gate	Checks all items in INPUTS with the output of CODE and returns the Boolean result. Starts with the first one, and the arguments of CODE should two.
data, runCommand	Modify the contents of a table of tokens to create a command structure and runs it.
data, runContents	Runs the contents of a program and repeats doing so until the line number is greater than the length of the contents.
parser, cleanUpString	Removes whitespace (tabs and spaces) from the left and comments (number sign / hash) from the right from STRING.
parser, tokenizer	Generates tokens out of STRING.
parser, generate	Use the items in TOKENS to generate a table.

PARSER OVERVIEW

For every line in a program, the parser is responsible for taking the string and generating a table for the interpreter to use. In the table, it consists of the command's entries, which is treated as names, inputs, and subcommands.

1. Clean up the string from whitespace and comments
    a. Test every character until it does not detect a tab or space, then save the remaining string.
    b. Test every character until it detects a number sign, then returns the result.
2. Generate tokens out of the string.
    a. If the character is either a quote or apostrophes, and it's not in a string:
        i. Remember that it was a string, and what character it was.
    b. If the character equals to the remembered character, and it's in a string:
        i. Remember that it's no longer a string.
    c. If the character is parentheses, and it's not in a string:
        i. Make the character a unique token.
        ii. Make a new token entry.
    d. If the character is a space, and it's not in a string:
        i. Make a new token entry.
    e. If these conditions aren't satisfied:
        i. Join the last token with the character.
3. Generate a table out of the tokens.

ADDING COMMANDS

The commands are stored in content.data.commands, and you can easily create and use a command. First, design a command with the name, inputs, and various more information.

- For inputs that are optional, write a tilde (~) before them.
- For inputs that are limited to options, write / on each possible input.
- For unlimited inputs, write … at the end.

Afterwards, define a function inside the commands table with the name. You can do anything with it, but remember that inputs store the command's inputs, and all outputs are converted into strings.

```
-- double NUMBER
["double"] = function(inputs)
	local number = tonumber(inputs[1]) or 1
	return number * number
end
```

Use the interpreter or write a program to test your command. If you get no output or receive a Lua error, debug your command. The functions.printTable function can be helpful.

DOCUMENTING COMMANDS

To make sure your command can be used with the help command, you should document information on your command on help.lua. Define a new item consisting of the following information:

1. Command structure.
2. Input information.
3. Command information.
4. Command example.

```
}, ["double"] = {
	"double NUMBER",
	{["NUMBER"] = "Number"},
	[[multiplies NUMBER by NUMBER]],
	[[print (double 4)]]
}
```

Test this by using the **help** command, followed with the command's name.

```
help double

== double ==
double NUMBER
NUMBER: Number

multiplies NUMBER by NUMBER

print (double 4)
```

ADDING DATA TYPES

To add a data type, add a new item in content.data.

```
-- program data
["data"] = {
	["commands"] = {}, -- [name] = function(args)
	["variables"] = {}, -- [name] = value
	["lists"] = {}, -- [name] = {...}
	["procedures"] = {}, -- [name] = {code, inputs name}
	["labels"] = {}, -- [name] = value

	["switches"] = {} -- [name] = value
},
```

To support the data type with the **delete** command, add a singular variation of the type's name to types.

```
-- delete TYPE ~ NAME
["delete"] = function(inputs)
	local typeEntry, name = inputs[1], inputs[2]
	local types = {"variable", "list", "procedure", "label", "switch"}
	if typeEntry ~= nil then
		if typeEntry == "all" then
			for _, value in ipairs(types) do
```

Lastly, if your data type can be used without defining it first, like lists and procedures, define a new function in content that uses the **getDataType** function.

```
-- get and return switch, define it if not found
function content.getSwitch(name)
	return getDataType("switches", name, {false})
end
```