LEVEL UP YOUR PROGRAMMING WITH CORE **DRAGON**

BY
AAVESH JILANI

FIRST EDITION

Contents at a Glance

PART I Dragon Language

CHAPTER 1 The History and Evolution of Dragon

1. The Creation of Dragon

Dragon was created in 2018. It took 14 months to develop the first working version. There are already many programming languages developed in the present time, but some of them are difficult to learn. The basic requirement for a developer is to make software as per client's need.

When someone uses other languages like Java, C#, and Python. He definitely requires an IDE (Integrated Development Environment) to develop a GUI based application. This saves time but you cannot do the whole working using the shortcut. Meanwhile, to define the working of GUI components and some other sort of work requires their own logics and it takes a lot of lines code in these types of languages. Sometimes it becomes difficult to identify the errors in a program because of the difficult syntax of those programming languages.

These things made me curious to make the programming easier. Every year programmers having new requirements as

per the IT world. The most difficult task is to complete the project at the given time. Every programmer decides to work on a programming language that will require less lines of codes with full of functionality. For solving these types of problems, Dragon was developed with functionality, efficiency, and reliability. The most common way of measuring the scale of a programming language is to calculate it's time and space complexity which helps people to compare it with other languages. Meanwhile, how much time and space it is taking to execute the specific code or program.

In a comparison of other languages, Dragon takes a similar time as Java to execute a program and its syntax is easy as python. OOPs, concepts are useful but quite difficult to understand for some new people. Dragon is not a pure Object-oriented programming language. It has an object and class creation part of the OOPs concept.

2. The Evolution of Dragon

The Initial version of the Dragon, printing statement was the only thing that was working. Dragon Interpreter V 1.0 was released to execute programs of language. In every version, there are some major updates. When version 1.1 was released then basic operations were able to perform. In version 1.2 few additional things were added like GUI and database operations. The current version is 1.9.7 which is a stable release. And works in Windows, Linux and Android Environment with two type Native and non-native.

CHAPTER 2 Overview of Dragon

It uses the Interpretation method instead of a compilation like Python, Perl, and Ruby. Interpreter translates the program one statement at a time. It takes less amount of time to analyze the source code. No intermediate object code generated, hence are memory efficient. Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.

Figure :- Dragon program execution

In the above figure, the processing of code in Dragon through Interpreter is defined clearly. The same process of program execution is followed in all platforms which are supported by Dragon.

1. Display output on screen

There are two keywords use to display output on console screen “show” and “showln” .The show keyword displays the

output normally. While the showln keyword reserve a full line for output and displays on a separate line.

```
show "Hello world"

showln "Hello world"
```

2. Comments in Dragon

A comment is a programmer-readable explanation or annotation in the source code of a computer program. In Dragon, there are three types of comments.

a) Single-line Comment
b) Multi-line Comment
c) Inner Comment

Single-line Comment

```
// single-line comment
```

Multi-line Comment

```
/* multi-line
    comment
 */
```

Inner Comment

```
show /*inner comment*/ "Text"
```

CHAPTER 3 Data Types, Variables, Arrays and basics

1. Data Types and Variables

Variables are used to store information to be referenced and used by programs. A *data type* is a classification that specifies which type of value a variable. Dragon is a dynamic programming language. It does not require to declare the data type. There are four data types in Dragon.

a) String
b) Integer
c) Float
d) Boolean

String

Strings are defined in double-quotes and can be multiline. Escaping Unicode characters is also supported.

```
str = "\n\tThis is
\tmultiline
\ttext
"

// \n used for new line
// \t used for tab space

z = "abcd" // string
```

If a function requires a string as an argument, but the number was passed, then the numeric value will automatically convert to string.

```
x = 90
show x

// 90 converts to "90"
```

Integer

An *integer* is a whole number (not a fraction) that can be positive, negative, or zero.

```
x = 10 // integer
```

Float

Float is a data type used to define a variable with a fractional value.

```
y = 1.61803 // float
```

Boolean

The Boolean data type is a data type that has one of two possible values (usually denoted true and false), intended to represent the two truth values of logic and Boolean algebra.

```
a = true
b = false

// boolean
```

2. Array

An *array* is a collection of a fixed number of values of a single type. Array values are initialized in square brackets.

```
arr = ["a", "b", "c"]

     /* arr is an array with string
        Values a, b and c */

showln arr[0] // output is a
```

3. Map

A map is a collection that is unordered, changeable and indexed. In Dragon, maps are written with curly brackets, and they have keys and values.

```
map = {"key1": 1, "key2": 2, "key3": 3}
```

4. Destructuring Assignment

Destructuring assignment allows us to define multiple variables for each element of an array or map. The “extract” method is used to destruct the array and maps.

For arrays, value is assigned to variable:

```
arr = ["a", "b", "c"]
extract(var1, var2, var3) = arr
show var1 // a
show var2 // b
show var3 // c
```

Which is equivalent to:

```
arr = ["a", "b", "c"]
var1 = arr[0]
var2 = arr[1]
var3 = arr[2]
```

For maps, keys and values are assigned to variable:

```
map = {"key1": 1, "test": 2}
extract(var1, var2) = map
show var1 // [key1, 1]
show var2 // [test, 2]
```

To skip value just leave argument empty:

```
extract(x, , z) = [93, 58, 90]
show x // 93
show z // 90
```

5. Include another program

To include another program in Dragon “include” keyword is used. It is also called a user-defined module.

```
include "program2.dgn"

show " world"

// output :- Hello world
```

program2.dgn code:-

```
show "Hello"
```

6. Pattern Matching

The match operator allows to match values by pattern. In this case value and type are checking. If none of case branches doesn't match, the body of case _ branch will executes.

In addition to the constant values, you can set variable name to case.

```
x = 2
show match x {
  case 1: "One"
  case 2: "Two"
  case "str": "String"
  case _: "Unknown"
}

x = "str"
match x {
  case "": {
    showln "Empty string"
  }
  case "str": {
    showln "String!"
  }
 }

//Or


func test(x) = match x {
  case a: "case a: " + a
  case b: "case b: " + b
  case c: "case c: " + c
}
a = 10
b = 20
showln test(15)  // case c: 15
showln test(20)  // case b: 20
showln test("test")  // case c: test
```

CHAPTER 4 Operators

1. Arithmetic Operators

a) Addition

Adds the values on either side of the operator.

```
show 4 + 3
 // output:- 7
```

b) Subtraction

Subtracts the value on the right from the one on the left.

```
show 4 - 3
// output:- 1
```

c) Multiplication

Multiplies the values on either side of the operator.

```
show 4 * 3

// output:- 12
```

d) Division

Divides the value on the left by the one on the right.

```
show 2 / 4

// output:- 0
```

e) Modulus

Divides and returns the value of the remainder.

```
show 3 % 4

// output:- 3
```

2. Bitwise Operators

a) Bitwise OR (|)

This operator is binary operator, denoted by '|'. It returns bit by bit OR of input values, i.e., if either of the bits is 1, it gives 1, else it gives 0.

```
a = 4  b = 6

showln a | b

//output:- 6
```

b) Bitwise AND (&)

This operator is binary operator, denoted by '&'. It returns bit by bit AND of input values, i.e., if both bits are 1, it gives 1, else it gives 0.

```
a = 4  b = 6

showln a & b
```

```
//output:- 4
```

c) Bitwise XOR (^)

This operator is binary operator, denoted by '^'. It returns bit by bit XOR of input values, i.e., if corresponding bits are different, it gives 1, else it gives 0.

```
a = 4  b = 6

showln a ^ b

//output:- 2
```

d) Bitwise Complement (~)

This operator is unary operator, denoted by '~'. It returns the one's compliment representation of the input value, i.e., with all bits inversed, means it makes every 0 to 1, and every 1 to 0.

```
a = 4

showln ~a

//output:- -5
```

3. Shift Operators

These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively. They can be used when we have to multiply or divide a number by two

a) Signed Right shift operator (>>)

Shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.

```
a = 15

showln a>>1

//output:- 7
```

b) Unsigned Right shift operator (>>>)

Shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0. (>>>) is unsigned-shift; it'll insert 0. (>>) is signed, and will extend the sign bit.

```
a = 15

showln a>>1

//output:- 7
```

c) Left shift operator (<<)

Shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.

```
a = 5

showln a << 1

//output:- 10
```

CHAPTER 5 Control Statements

Control statements enable us to specify the flow of program control; i.e., the order in which the instructions in a program must be executed. They make it possible to make decisions, to perform tasks repeatedly or to jump from one section of code to another.

1. If statement

The if statement evaluates the test expression following. The parenthesis are not necessary. If the test expression is evaluated to true, statements inside the body of if are executed. If the test expression is evaluated to false, statements inside the body of if are not executed. An if statement may have multiple other if statements inside. That is called ladder or nested if.

```
if a > 5
{
  showln "True"
}
```

Same as

```
if(a > 5)
{
  showln "True"
```

```
}

//both syntax are valid
```

2. Else statement

The if statement may have an optional else block. When the if statement condition false. Statements inside the body of else are executed. Statements inside the body of if are skipped from execution.

```
if(a > 5)
{
  showln "True"
}
else{
  showln "False"
}
```

3. Else-If statement

Else-if statements in C is like another if condition, it's used in a program when if statement having multiple decisions.

```
if(a > b && a > c)
{
  showln "a is bigger is size"
```

```
}

else if(b > a && b > c){
  showln "b is bigger in size"
}
else{
  showln "c is bigger in size"
}

// use of all if, else and else if
```

4. Loops and Statements

A loop is a sequence of instructions that is repeated until a certain condition is reached. An operation is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number. The parentheses are not necessary for conditions in loops.

For loop

The for loop is used to iterate the statements or a part of the program several times.

```
for initializing, condition, increment
{
   body
}

for (initializing, condition,
increment) {
   body
}
for i = 0, i < 5, i++
  show i++

// or

for (i = 0, i < 5, i++) {
  show i++
}
```

While loop

While loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed multiple times depending upon a given boolean condition. It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.

```
while condition {
```

```
    body
}


i = 0
while i < 5 {
  show i++
}

// or

i = 0
while (i < 5) {
  show i++
}
```

Do-While loop

The do while loop is a post tested loop. Using the do-while loop, we can repeat the execution of several parts of the statements. The do-while loop is mainly used in the case where we need to execute the loop at least once. The do-while loop is mostly used in menu-driven programs where the termination condition depends upon the end user.

```
do {
   body
} while condition
```

```
i = 0
do {
  show i++
} while i < 5

// or

i = 0
do {
  show i++
} while (i < 5)
```

For-each loop

It provides an alternative approach to traverse the array or Map. It is mainly used to traverse the array or Map elements. The advantage of the for-each loop is that it eliminates the possibility of bugs and makes the code more readable. It is known as the for-each loop because it traverses each element one by one.

```
for value : array {
```

```
    body
}

for key, value : map {
    body
}

for (value : array) {

}

for (key, value : map) {
    body
}
arr = [1, 2, 3, 4]
for v : arr {
  showln v
}

map = {"key1": 1, "key2": 2}
for key, value : map
  showln key + " = " value
}
```

Break Statement

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

```
//using for loop
```

```
for(i=1,i<=10,i++){
   if(i==5){
     //breaking the loop
     break
 }
 showln i
}
}
```

Continue Statement

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

```
//for loop
for(i=1,i<=10,i++){
  if(i==5){
  //using continue statement
      continue
    //it will skip the rest statement
  }
showln i
}
```

CHAPTER 5 Functions and Classes

1. Functions

A function is a program module that contains a series of statements that carry out a task. To execute a function, you invoke or call it from another function; the calling function makes a function call, which invokes the called function. Any class can contain an unlimited number of functions, and each function can be called an unlimited number of times.

To define function uses the func keyword, followed by the name.

```
func function(arg1, arg2) {
  show arg1
}
```

Shorthand Definition

There is short syntax for function body:

```
func repeat(str, count) = str * count
```

Which is equivalent to:

```
func repeat(str, count) {
  return str * count
}
```

```
/* the return keyword is used to return
back any value from function */
```

Default arguments

Function arguments can have default values.

```
func repeat(str, count = 5) = str * count
```

In this case only str argument is required.

```
repeat("*")      //  *****
repeat("+", 3)   //  +++
```

Default arguments can't be declared before required arguments.

```
func repeat(str = "*", count) = str *
count
```

Causes parsing error: `ParseError on line 1: Required argument cannot be after optional`

Inner functions

You can define function in other function.

```
func fibonacci(count) {
  func fib(n) {
    if n < 2 return n
    return fib(n-2) + fib(n-1)
```

```
  }
  return fib(count)
}
show fibonacci(10)   //   55
```

2. Classes

A class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).

To define classes use the class keyword, followed by the name.

```
class classname{
func function(arg1, arg2) {
  show arg1
} }

obj = new classname()
//class object creation, Calling the class
//new keyword used for making class object

obj.function(arg1,arg2)
// function calling with class object
```

Constructor

Constructor name should be same as class name and will be created with func keyword:

```
class cons{
func cons(val){
   showln val
 } }

obj = new cons(val)
/* class object creation with constructor
arguments */
```

PART II Modules

In Dragon, Module is a library that is used as per the requirement. It is a bunch of predefined functions. Those are required to use the functionality of Dragon. Every Module contains different functions. Some Module names are std, math, http, etc.

CHAPTER 1 std Module

The std module contains common functions. The std word stands for “standard”. Which also means the collection of standard functions. The basic requirement of the program needs these functions.

Constants

ARGS : *string* = command-line arguments

Example:-

```
select "std"

showln ARGS[0] //This will print the first
command line argument value.
```

Functions

1) arrayCombine

It is used for combining to arrays into one. It creates a map of the values of both arrays. Two parameters are required in this function. In which the first one is key and second values, both will be arrays.

Syntax:-

```
arrayCombine(key,value)
```

Example:-

```
select "std"

a = [1,2,3,4] // first array
b = [6,7,8,9] // second array

c = arrayCombine(a,b) //first is key
and second is value
showln c
```

```
//output :- {3=8, 4=9, 1=6, 2=7}
```

2) arrayKeyExists

The arrayKeyExists is used for finding a specific key in the map. It will give output in the form of 1 or 0. Which means exists or not exists. There are two parameters required in this function. The first one is key and the second is a map.

Syntax:-

```
arrayKeyExists(key, map)
```

Example:-

```
select "std"

ar = {"a":3,"b":4} //map

showln   arrayKeyExists("c",ar)    //
first one is key and second is map

showln arrayKeyExists("a",ar)

//output:- 0
//output:- 1
```

```
// the "c"  key was not find that's why output 0, but the "a" key was there then it becomes 1.
```

3) arrayKeys

The arrayKeys method is used to make an array of keys in the map. There is one parameter requires in this function. Which will be a map.

Syntax:-

```
arrayKeys(map)
```

Example:-

```
select "std"

ar = {"a":3,"b":4} //map

showln arrayKeys(ar) // map given as parameter

//output:- [a, b]
```

4) arrayValues

This method makes an array of values in the map. While the arrayKeys was only for Keys this one is for Values. One parameter is required in the method. That is a map.

Syntax:-

```
arrayValues(map)
```

Example:-

```
select "std"

ar = {"a":3,"b":4} // map

showln arrayValues(ar) // map given as
parameter

//output:- [3, 4]
```

5) charAt

It returns the character of the specific location of the given input. There are two parameters requires in this

method. The first one is input and the second is the index number.

Syntax:-

```
charAt(input, index)
```

Example:-

```
select "std"

a = "Hello World" //string

showln charAt(a,6) //first parameter input, second index number

//output:- 87

//87 is the ASCII value of “W” which is on 6th index position of String “a”
```

6) echo

It is used for printing the output. But different from the “show” and “showln”. All the inputs are given as parameters.

Syntax:-

```
echo(arg...)
```

Example:-

```
select "std"

echo("Hello World") // output:- Hello World

echo("Abc",123) //output:- Abc 123
```

7) indexOf

It finds the first occurrence of character or word from the starting index of input. It requires three parameters. Those are input, what and index.

Syntax:-

```
indexOf(input, what, index = 0)
```

Example:-

```
select "std"

s = "hello"
indexOf(s,"l")   /*return   2,   the
position of first "l" */

indexOf(s,"l",3)   /*return   3,   the
position of "l" from search postion 3
*/
```

8) join

This function joins the array to string with delimiter, suffix and prefix. The suffix and prefix are not necessary parameters.

Syntax:-

```
join(array, delimiter, prefix,
suffix)
```

Example:-

```
select "std"


s = ["h","e","l","l","o"]
join(s,"") //return hello

join(s,"","abc","xyz")
//return abchelloxyz
```

9) lastIndexOf

This function finds the last occurrence of given string three inputs are used for this. Input which string, value which has to be find is and index position from where to start the search. The index parameter is optional.

Syntax:-

```
lastIndexOf(input, what, index)
```

Example:-

```
select "std"


s = "hello"
lastIndexOf(s,"l")  /*return  3,  the
position of last "l" */

lastIndexOf(s,"l",2) /*return 2, the
position  of  "l"  from  back  search
postion 2 */
```

10) length

This function returns length of string, array/map size or number of function arguments.

Syntax:-

```
length(x)
```

Example:-

```
select "std"

s = "hello"
length(s) //return 5, length of string

arr = ["a","b"]
length(arr) //return 2, size of array
```

11) newarray

This function creates array with size. newarray(x) - creates 1D array, newarray(x,y) - creates 2D array

Syntax:-

```
newarray(size...)
```

Example:-

```
select "std"

a = newarray(4) // [0, 0, 0, 0]

b = newarray(2, 3)
// [[0, 0, 0], [0, 0, 0]]
```

12) parseInt

This function parses string into integer in the radix. Minimum one parameters require. String. Maximum two parameters accepted. Second is radix.

Syntax:-

```
parseInt(str, radix)
```

Example:-

```
select "std"

parseInt("10") //return 10, As Integer

parseInt("101",2)
//return 5, As Integer
```

13) parseLong

This function parses string into long in the radix
Minimum one parameters require string.
Maximum two parameters accepted. Second is radix.

Syntax:-

```
parseLong(str, radix)
```

Example:-

```
select "std"

parseLong("10000")
//return 10000, As Long Integer

parseLong("10101",2)
//return 21, As Long Integer
```

14) rand

This function returns pseudo-random number. rand() - returns float number from 0 to 1 rand(max) - returns random number from 0 to max rand(from, to) - return random number between from and to

Syntax:-

```
rand(from, to)
```

Example:-

```
select "std"
```

```
rand() /* return 0.14329640493102536,
anything random between 0 and 1 */

rand(400) /* return 364, anything
random between 0 and 400 number */
rand(1000,2000) /* return 1163,
anything random between 1000 and 2000
*/
```

15) range

This function creates lazy array by number range.
range(to) - creates range from 0 to to (exclusive) with step 1.
range(from, to) - creates range from from to to (exclusive) with step 1.

range(from, to, step) - creates range from from to to (exclusive) with step step.

Syntax:-

```
range(from, to, step)
```

Example:-

```
select "std"

show range(3) // [0, 1, 2]
r = range(-5, 0)
// [-5, -4, -3, -2, -1]
show r[0] // -5
show r[2] // -3
for x : range(20, 9, -5) {
  showln x
} // 20 15 10
```

16) readln

This function reads a line from console. It is used for user input values.

Syntax:-

```
readln()
```

Example:-

```
select "std"

a = readln()
//program ask for user input

showln a //prints the value of a
```

17) replace

This function replaces all occurrences of string target with string replacement.

Syntax:-

```
replace(str, target, replacement)
```

Example:-

```
select "std"

s = "hello"
replace(s,"l","p")
/*   return   heppo,   replace   all
occurrences of l replaced with p */
```

18) replaceAll

This function replaces all occurrences of regular expression regex with string replacement.

Syntax:-

```
replaceAll(str, regex, replacement)
```

Example:-

```
select "std"

s = "hello world"
replaceAll(s,"hello","bye")
/* return bye world, replace all
occurences of word hello with bye */
```

19) replaceFirst

This function replaces first occurrence of regular expression regex with string replacement.

Syntax:-

```
replaceFirst(str, regex, replacement)
```

Example:-

```
select "std"

s = "hello"
replaceFirst(s,"l","p")
/* return heplo, replace first
occurrence of l replaced with p */
```

20) sleep

This function causes current thread to sleep for given time milliseconds.

Syntax:-

```
sleep(time)
```

Example:-

```
select "std"

sleep(1000)
//wait for 1000 milliseconds

showln "Hello, World"
/* print Hello, World after waiting
1000 milliseconds */
```

21) sort

This function sorts array by natural order.

Syntax:-

```
sort(array)
```

Example:-

```
select "std"
```

```
a = [1,6,3,8]
sort(a) //return [1,3,8,6]
```

22) split

This function splits string str with regular expression regex into array. limit parameter affects the length of resulting array.

Syntax:-

```
split(str, regex, limit)
```

Example:-

```
select "std"

split("a5b5c5d5e", "5")
// ["a", "b", "c", "d", "e"]


split("a5b5c5d5e", "5", 3)
// ["a", "b", "c5d5e"]
```

23) sprintf

This function formats string by arguments. Similar like C programming's printf function.

Syntax:-

```
sprintf(format, args...)
```

Example:-

```
select "std"

sprintf("Value of %s is %d","a",20)
//return "Value of a is 20"
```

24) substring

This function returns string from startIndex to endIndex or to end of string if endIndex is not set.

Syntax:-

```
substring(str, startIndex, endIndex)
```

Example:-

```
select "std"

substring("abcde", 1) // bcde
substring("abcde", 2, 4) // cd
```

25) sync

This function calls an asynchronous function synchronously.

Syntax:-

```
sync(callback)
```

Example:-

```
select "std"
select "http"

result = sync(func(ret) {
   http(url, func(t) = ret(t))
})
```

26) thread

This function creates new thread with parameters if passed.

Syntax:-

```
thread(func, args...)
```

Example:-

```
select "std"
```

```
thread(func() {
  show "New Thread"
})

thread(::newthread, 10)
thread("newthread", 20)

func newthread(x) {
  show "New Thread. x = " + x
}
```

27) time

This function returns current time in milliseconds from 01.01.1970.

Syntax:-

```
time()
```

Example:-

```
select "std"

time() //return 1593268596562
```

28) toChar

This function converts char code to string.

Syntax:-

```
toChar(code)
```

Example:-

```
select "std"

toChar(65) //return "A"
```

29) toHexString

This function converts number into hex string.

Syntax:-

```
toHexString(number)
```

Example:-

```
select "std"

toHexString(12) //return "c"
```

30) toLowerCase

This function converts all characters to lower case.

Syntax:-

```
toLowerCase(str)
```

Example:-

```
select "std"

a = "HELLO"
toLowerCase(a) //return "hello"
```

31) toUpperCase

This function converts all characters to upper case.

Syntax:-

```
toUpperCase(str)
```

Example:-

```
select "std"

a = "heLLo"
toUpperCase(a) //return "HELLO"
```

32) trim

This function removes any leading and trailing whitespaces in string.

Syntax:-

```
trim(str)
```

Example:-

```
select "std"

a = " Dragon "
trim(a) //return "Dragon"
```

33) try

This function suppress any error in unsafeFunction and returns the result of the catchFunction if any error occurs.

Syntax:-

```
try(unsafeFunction, catchFunction = func(type, message) = -1)
```

Example:-

```
select "std"

try(func() = "success") // success
try(func() = try + 2) // -1

try(func() = try(), func(type,
message) = sprintf("Error
handled:\ntype: %s\nmessage: %s",
type, message))
```

CHAPTER 2 types Module

The types module contains data type conversion functions. Those functions also used for type casting. The types module also used for checking the data type of a variable.

Constants

OBJECT : *number* = 0

NUMBER : *number* = 1

STRING : *number* = 2

ARRAY : *number* = 3

MAP : *number* = 4

FUNCTION : *number* = 5

Functions

1) byte

This function converts value to byte.

Syntax:-

```
byte(value)
```

Example:-

```
select "types"
//selecting types module

byte(10) //return 10
```

2) double

This function converts value to double.

Syntax:-

```
double(value)
```

Example:-

```
select "types"
//selecting types module

double(1912412) //return 1912412.0
```

3) float

This function converts value to float.

Syntax:-

```
float(value)
```

Example:-

```
select "types"
//selecting types module

float(10) //return 10.0
```

4) int

This function converts value to int.

Syntax:-

```
int(value)
```

Example:-

```
select "types"
//selecting types module

int("100") //return 100
```

5) long

This function converts value to long.

Syntax:-

```
long(value)
```

Example:-

```
select "types"
//selecting types module

long("100000") //return 100000
```

6) number

This function converts value to number (Integer, double or long) if possible.

Syntax:-

```
number(value)
```

Example:-

```
select "types"
//selecting types module

number("2.3") //return 2.3
```

7) short

This function converts value to short.

Syntax:-

```
short(value)
```

Example:-

```
select "types"
//selecting types module

short("102") //return 102
```

8) string

This function converts value to string.

Syntax:-

```
string(value)
```

Example:-

```
select "types"
//selecting types module

string(1) //return "1"
```

9) typeof

This function returns the type of value.

Syntax:-

```
typeof(value)
```

Example:-

```
select "types"
//selecting types module

typeof(1) //return 1 (NUMBER)
typeof("types") //return 2 (STRING)
typeof([]) //return 3 (ARRAY)
```

CHAPTER 3 math Module

The math module contains mathematical operational functions. Those functions are useful while making some calculations program and similar things.

Constants

E : *number* = 2.718281828459045

PI : *number* = 3.141592653589793

Functions

1) abs

This function returns the absolute number of given value.

Syntax:-

```
abs(value)
```

Example:-

```
select "math"

showln abs(1.2900)
//return 1.29
```

2) acos

This function returns the arcosine value.

Syntax:-

```
acos(value)
```

Example:-

```
select "math"

showln acos(1/2)
//return 1.5707963267948966
//non radian value
```

3) asin

This function is used to find the arcsine of given value.

Syntax:-

```
asin(value)
```

Example:-

```
select "math"

showln asin(2/2)
//return 1.5707963267948966
```

4) atan

This function is used to find the arc tangent of given value.

Syntax:-

```
atan(value)
```

Example:-

```
select "math"

showln atan(1)
//return 0.7853981633974483
```

5) atan2

This function returns angle, whose tangent is the ratio of two numbers.

Syntax:-

```
atan2(value)
```

Example:-

```
select "math"

showln atan2(1,2)
//return 0.4636476090008061
```

6) cbrt

This function returns the cube rate of given value.

Syntax:-

```
cbrt(value)
```

Example:-

```
select "math"

showln cbrt(64)
//return 4.0
```

7) ceil

This function returns the smallest integer that is greater than or equal to the value.

Syntax:-

```
ceil(value)
```

Example:-

```
select "math"

showln ceil(2.5)
//return 3.0
```

8) copySign

This function returns float value consisting of magnitude from parameter x and the sign from parameter y.

Syntax:-

```
copySign(magnitude, sign)
```

Example:-

```
select "math"

showln copySign(4,-2)
//return -4.0
```

9) cos

It is trigonometric cosine. This function returns the cos mathematical value of given value.

Syntax:-

```
cos(value)
```

Example:-

```
select "math"

showln cos(10)
//return -0.8390715290764524
```

10) cosh

It is hyperbolic cosine. This function returns the cos mathematical value of given value.

Syntax:-

```
cosh(value)
```

Example:-

```
select "math"

showln cosh(10)
//return 11013.232920103324
```

11) exp

This function returns the exponential of given value.

Syntax:-

```
exp(value)
```

Example:-

```
select "math"

showln exp(2)
//return 7.38905609893065
```

12) expm1

This function returns the exponential of given value and also does -1 from that.

Syntax:-

```
expm1(value)
```

Example:-

```
select "math"

showln expm1(2)
//return 6.38905609893065
```

13) floor

This function returns the floor value of given parameter.

Syntax:-

```
expm1(value)
```

Example:-

```
select "math"

showln expm1(2)
//return 6.38905609893065
```

14) getExponent

This function returns the unbiased exponent used in the representation of a double or float.

Syntax:-

```
getExponent(value)
```

Example:-

```
select "math"

showln getExponent(345.5)
//return 8
```

15) hypot

This function returns sqrt($x^2 + y^2$) without intermediate overflow or underflow.

Syntax:-

```
hypot(x,y)
```

Example:-

```
select "math"

showln hypot(3,4)
//return 5.0
```

16) IEEEremainder

This function computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.The remainder value is mathematically equal to f1 – f2 x n, where n is the mathematical integer closest to the exact mathematical value of the quotient f1/f2, and if two mathematical integers are equally close to f1/f2, then n is the integer that is even.

Syntax:-

```
IEEEremainder(x,y)
```

Example:-

```
select "math"

showln remainder(31.34,2.2)
//return 0.5399999999999974
```

17) log

This function returns the natural logarithm (base e) of a double value as a parameter.

Syntax:-

```
log(value)
```

Example:-

```
select "math"

showln log(145.256)
//return 4.978497702968366
```

18) log1p

This function returns the natural logarithm of the sum of the arguments and 1.For the small values, the result of log1p(a) is much closer to the true result of ln(1 + a) than the floating-point evaluation of log(1.0 + a).

Syntax:-

```
log1p(value)
```

Example:-

```
select "math"

showln log1p(23.45)
//return 3.196630215920881
```

19) log10

This function returns the base 10 logarithmic value of given double value as a parameter.

Syntax:-

```
log10(value)
```

Example:-

```
select "math"

showln log10(1000)
//return 3.0
```

20) max

This function returns maximum of two numbers.

Syntax:-

```
max(x,y)
```

Example:-

```
select "math"

showln max(34,62)
//return 62
```

21) min

This function returns minimum of two numbers.

Syntax:-

```
min(x,y)
```

Example:-

```
select "math"

showln min(34,62)
//return 34
```

22) nextAfter

This function returns the floating-point number adjacent to the first argument in the direction of the second argument. If both arguments are equal then the second argument is returned.

Syntax:-

```
nextAfter(x,y)
```

Example:-

```
select "math"

showln nextAfter(98.2,56.4)
//return 98.19999999999999
```

23) nextUp

This function returns the floating-point value adjacent to the parameter.

Syntax:-

```
nextUp(x)
```

Example:-

```
select "math"

showln nextUp(69.19)
//return 69.19000000000001
```

24) pow

This function is used to calculate a number raise to the power of some other number.

Syntax:-

```
pow(x,y)
```

Example:-

```
select "math"

showln pow(30,2)
//return 900.0
```

25) rint

This function is used to round of the floating-point argument to an integer value (in floating-point format).

Syntax:-

```
rint(value)
```

Example:-

```
select "math"

showln rint(12.7)
//return 13.0
```

26) round

This function returns the closest long to the argument. The result is rounded to an integer by adding 1/2, taking the floor of the result after adding 1/2, and casting the result to type long.

Syntax:-

```
round(value)
```

Example:-

```
select "math"

showln round(4567.9874)
//return 4568
```

27) signum

This function returns the Sign function of a value passed to it as argument.

Syntax:-

```
signum(value)
```

Example:-

```
select "math"

showln signum(20)
//return 1.0

Showln signum(-20)
//return -1.0
```

28) sin

This function returns the trigonometric sine of an angle in between 0.0 and pi.

Syntax:-

```
sin(value)
```

Example:-

```
select "math"

showln sin(45)
//return 0.4999999999999994
```

29) sinh

This function returns the hyperbolic sine of a double value passed to it as argument.

Syntax:-

```
sinh(value)
```

Example:-

```
select "math"

showln sinh(3.5)
//return 16.542627287634996
```

30) sqrt

This function returns the square root of a value of type double passed to it as argument.

Syntax:-

```
sqrt(value)
```

Example:-

```
select "math"

showln sqrt(30)
//return 5.477225575051661
```

31) tan

This function returns the trigonometric tangent of an angle.

Syntax:-

```
tan(value)
```

Example:-

```
select "math"

showln tan(toRadians(45))
//return 0.9999999999999999
```

32) tanh

This function returns the hyperbolic tangent of a double value passed to it as argument.

Syntax:-

```
tanh(value)
```

Example:-

```
select "math"

showln tanh(3.5)
//return 0.9981778976111987
```

33) toDegrees

This function is used to convert an angle measured in radians to an approximately equivalent angle measured in degrees.

Syntax:-

```
toDegrees(value)
```

Example:-

```
select "math"

showln toDegrees(PI)
//return 180.0
```

34) toRadians

This function is used to convert an angle measured in degrees to an approximately equivalent angle measured in radians.

Syntax:-

```
toRadians(value)
```

Example:-

```
select "math"

showln toRadians(180.0)
//return 3.141592653589793
```

35) ulp

This function returns the size of an ulp of the argument. An ulp stands for unit of least precision. It calculates the distance between the given double or float value and the double or float value next larger in magnitude.

Syntax:-

```
ulp(value)
```

Example:-

```
select "math"

showln ulp(34.543)
//return 7.105427357601002E-15
```

36) eval

This function evaluates the expression and returns the calculated result.

Syntax:-

```
eval(value)
```

Example:-

```
select "math"

showln eval("1 + 2 * 3")
//return 7
```

CHAPTER 4 date Module

The date module contains functions for working with date and time.

Constants

STYLE_FULL : *number* = 0

STYLE_LONG : *number* = 1

STYLE_MEDIUM : *number* = 2

STYLE_SHORT : *number* = 3

Types

DateValue

DateFormatValue

Functions

1) newDate

newDate(...) - newDate() - returns current date.

newDate(timestamp) - returns date by given timestamp.

newDate(dateString) - parses and returns date by given string.

newDate(pattern, dateString) - parses and returns date by given string in pattern format.

newDate(year, month, day) - returns date by year, month and day.

newDate(year, month, day, hour, minute) - returns date by year, month, day, hour and minute.

newDate(year, month, day, hour, minute, second) - returns date by year, month, day, hour, minute and second.

Example:-

```
select "date"

a = newDate()

showln a
```

```
//return, 2020-06-04 03:14:11.435

showln newDate(2019,11,07)

//return, 2019-11-07 00:00:00.000

showln newDate(2019,11,29,11,30)

//return, 2019-11-29 11:30:00.000
```

2) newFormat

newFormat(...) - newFormat() - returns default date format.

newFormat(pattern) - returns date format by given pattern.

newFormat(type) - returns format: 0 - default, 1 - date, 2 - time, 3 - date and time.

newFormat(pattern, locale) - returns date format by given pattern and locale.

newFormat(type, style) - returns format: 0 - default, 1 - date, 2 - time, 3 - date and time. style: 0 - full, 1 - long, 2 - medium, 3 - short.

Returns DateFormatValue.

Example:-

```
select "date"

a = newFormat("dd/mm/yyyy")
d = newDate()
showln formatDate(d,a)
//returns, 04/06/2020
```

3) formatDate

This function formats date by given format and returns string.

Syntax:-

```
formatDate(date, format)
```

Example:-

```
select "date"

a = newFormat("dd/mm/yyyy")
d = newDate(2019,11,07)
showln formatDate(d,a)
//returns, 07/1/2019
```

4) parseDate

This function parses date from string by given pattern. Returns DateValue.

Syntax:-

```
parseDate(dateString, format)
```

Example:-

```
select "date"

showln parseDate("2019/10/09",
newFormat("yyyy/MM/dd"))
//returns, 2019/10/09
```

5) toTimestamp

This function returns timestamp of given date in milliseconds.

Syntax:-

```
toTimestamp(date)
```

Example:-

```
select "date"

d = newDate()

showln toTimestamp(d)

//return, 1593857554769
```

CHAPTER 5 files Module

The files module contains functions for working with files.

Constants

FILES_COMPARATOR : *function* = func(f1, f2) = compare(f1, f2)

function which compares two file descriptors

Functions

1) fopen

This function opens file with path in given mode.

File Modes:-

- "" - opens file or directory for getting info;
- "r" - opens file for read in text mode;
- "rb" - opens file for read in binary mode;
- "w" - opens file for write in text mode;
- "w+" - opens file for append in text mode;
- "wb" - opens file for write in binary mode;
- "wb+" - opens file for append in binary mode

Syntax:-

```
fopen(path, mode)
```

Example:-

```
select "files"

f1 = fopen("text.txt")
/* opens file text.txt for read in
text mode */
f2 = fopen("E:/1.dat", "rbwb")
/* opens file 1.dat on drive E for
binary read and write */
```

2) fclose

This function closes file.

Syntax:-

```
fclose(file)
```

Example:-

```
select "files"

f1 = fopen("text.txt")
fclose(f1)
//file closed
```

3) fileSize

This function returns file size in bytes.

Syntax:-

```
fileSize(file)
```

Example:-

```
select "files"

f1 = fopen("text.txt")
showln fileSize(f1)
//return 11
```

4) exists

This function checks file or directory existing. Returns 1 if exists, 0 otherwise.

Syntax:-

```
exists(file)
```

Example:-

```
select "files"

f1 = fopen("text.txt")
showln exists(f1)
//return 1 (exists)
```

5) canExecute

This function checks execute permission of file. Returns 1 if true, 0 otherwise.

Syntax:-

```
canExecute(file)
```

Example:-

```
select "files"

f1 = fopen("text.txt")
showln canExecute(f1)
//return 0 (No permission)
```

6) canRead

This function checks reads permission of file. Returns 1 if true, 0 otherwise.

Syntax:-

```
canRead(file)
```

Example:-

```
select "files"

f1 = fopen("text.txt")
showln canRead(f1)
//return 1 (permission true)
```

7) canWrite

This function checks reads permission of file. Returns 1 if true, 0 otherwise.

Syntax:-

```
canWrite(file)
```

Example:-

```
select "files"

f1 = fopen("text.txt")
showln canWrite(f1)
//return 1 (permission true)
```

8) copy

This function copies file from source to destination. Returns 1 if success, -1 otherwise.

Syntax:-

```
copy(source,destination)
```

Example:-

```
select "files"

showln copy("text.txt","new.txt")
//return 1 (copy success)
```

9) delete

This function removes file or directory. Returns 1 if delete was successful, 0 otherwise.

Syntax:-

```
delete(file/directory)
```

Example:-

```
select "files"

showln delete("new.txt")
//return 1 (delete success)
```

10) getParent

This function returns parent path of the given file.

Syntax:-

```
getParent(file)
```

Example:-

```
select "files"

f = fopen("text.txt")

showln getParent(f)
//return My files (Folder name)
```

11) isDirectory

This function checks if given value is a directory. Returns 1 if true, 0 otherwise.

Syntax:-

```
isDirectory(value)
```

Example:-

```
select "files"

showln isDirectory("text.txt")
//return 0 (not directory)
```

12) isFile

This function checks if given value is a file. Returns 1 if true, 0 otherwise.

Syntax:-

```
isFile(value)
```

Example:-

```
select "files"

showln isFile("text.txt")
//return 1 (true it's a file)
```

13) isHidden

This function checks if file or directory is hidden. Returns 1 if hidden, 0 otherwise.

Syntax:-

```
isHidden(value)
```

Example:-

```
select "files"

showln isHidden("text.txt")
//return 0 (Not hidden)
```

14) lastModified

This function returns last modification time. Returns time in milliseconds.

Syntax:-

```
lastModified(file)
```

Example:-

```
select "files"

f = fopen("text.txt")
showln lastModified(f)
//return 1593930215000
```

15) listFiles

This function returns array with filenames in given directory.

Syntax:-

```
listFiles(file)
```

Example:-

```
select "files"

f = fopen("/home/ubuntu/my","")
/* opens directory examples for
getting information */

showln listFiles(f)
/* gets array with filenames in
directory */
```

16) mkdir

This function creates the directory. Returns 1 if operation was successful, 0 otherwise.

Syntax:-

```
mkdir(value)
```

Example:-

```
select "files"

showln mkdir("newfolder")
//return 1 (Directory created!)
```

17) mkdirs

This function creates the multiple directories. Returns 1 if operation was successful, 0 otherwise.

Syntax:-

```
mkdirs(value)
```

Example:-

```
select "files"

showln mkdirs("newfolder/test/exp")
/* return 1 (Multiple Directory
created!) */
```

18) rename

This function renames (or moves) file. Return 1 if success, 0 otherwise.

Syntax:-

```
rename(value)
```

Example:-

```
select "files"

showln rename("text.txt","ntext.txt")
/* return 1 (Moved or renamed)*/
```

19) setReadOnly

This function marks file directory read only. Returns 1 if success, 0 otherwise.

Syntax:-

```
setReadOnly(file)
```

Example:-

```
select "files"

f = fopen("text.txt")
showln setReadOnly(f)
// return 1 (Success)
```

20) setExecutable

This function sets execute permission. Returns 1 if success, 0 otherwise.

Syntax:-

```
setExecutable(file,true or false)
```

Example:-

```
select "files"

f = fopen("text.txt")
showln setExecutable(f,true)
// return 1 (Success)
```

21) setReadable

This function sets read permission. Returns 1 if success, 0 otherwise.

Syntax:-

```
setReadable(file,true or false)
```

Example:-

```
select "files"

f = fopen("text.txt")
showln setReadable(f,true)
// return 1 (Success)
```

22) setWritable

This function sets write permission. Returns 1 if success, 0 Otherwise.

Syntax:-

```
setWritable(file,true or false)
```

Example:-

```
select "files"

f = fopen("text.txt")
showln setWritable(f,true)
// return 1 (Success)
```

23) setLastModified

This function sets last modified time. Returns 1 if success, 0 otherwise.

Syntax:-

```
setLastModified(file,time)
```

Example:-

```
select "files"

f = fopen("text.txt")
showln setLastModified(f, 100000000)
//time in milliseconds.
// return 1 (Success)
```

24) flush

This function flushes write buffer into file.

Syntax:-

```
flush(file)
```

Example:-

```
select "files"

f = fopen("text.txt","rw")
writeText(f,"Hello World")
// Writing text in file.
flush(f)
```

25) readAllBytes

This function reads all bytes from file. Returns array with bytes.

Syntax:-

```
readAllBytes(file)
```

Example:-

```
select "files"

f = fopen("file.bin","rb")
array = readAllBytes(f)
//return bytes array
```

26) readBoolean

This function reads boolean (1 byte). Returns 0 if byte was 0, 1 otherwise.

Syntax:-

```
readBoolean(file)
```

Example:-

```
select "files"

f = fopen("file.bin","rb")
val = readBoolean(f)
```

27) readByte

This function reads one byte.

Syntax:-

```
readByte(file)
```

Example:-

```
select "files"

f = fopen("file.bin","rb")
val = readByte(f)
```

28) readBytes

This function reads length bytes of file to array starting from offset. Returns number of read bytes.

Syntax:-

```
readBytes(file,array,offset,length)
```

Example:-

```
select "files"

select "std"

f1 = fopen("file.bin", "rb")

array = newarray(2048)
```

```
readedCount = readBytes(f1, array)

 // reads 2048 bytes

readedCount = readBytes(f1, array, 10)
/* reads 2048 bytes starting from 11
byte */

readedCount = readBytes(f1, array, 20,
10)

/* reads 10 bytes, starting from 21
byte */
```

29) readChar

This function reads one char (2 bytes). Returns number char's code.

Syntax:-

```
readChar(file)
```

Example:-

```
select "files"

f = fopen("text.txt","r")
val = readChar(f)
//return 65 (Char code of A)
```

30) readDouble

This function reads 8 bytes double number.

Syntax:-

```
readDouble(file)
```

Example:-

```
select "files"

f = fopen("file.bin","rb")
val = readDouble(f)
```

31) readFloat

This function reads 4 bytes float number.

Syntax:-

```
readFloat(file)
```

Example:-

```
select "files"

f = fopen("file.bin","rb")
val = readFloat(f)
```

32) readInt

This function reads 4 bytes integer number.

Syntax:-

```
readInt(file)
```

Example:-

```
select "files"

f = fopen("file.bin","rb")
val = readInt(f)
```

33) readLine

This function reads line from file opened in text mode.

Syntax:-

```
readLine(file)
```

Example:-

```
select "files"

f = fopen("text.txt","r")
val = readLine(f)
```

34) readLong

This function reads 8 bytes long number.

Syntax:-

```
readLong(file)
```

Example:-

```
select "files"

f = fopen("text.txt","r")
val = readLong(f)
```

35) readShort

This function reads 2 bytes short number.

Syntax:-

```
readShort(file)
```

Example:-

```
select "files"

f = fopen("text.txt","r")
val = readShort(f)
```

36) readText

This function reads all file's content as string.

Syntax:-

```
readText(file)
```

Example:-

```
select "files"

f = fopen("text.txt","r")
showln readText(f)
//return Hello World (file data)
```

37) readUTF

This function reads string in binary mode.

Syntax:-

```
readUTF(file)
```

Example:-

```
select "files"

f = fopen("text.txt","r")
val = readUTF(f)
```

38) writeBoolean

This function writes boolean (0 or 1) to file.

Syntax:-

```
writeBoolean(file, true or false)
```

Example:-

```
select "files"

f = fopen("text.txt","w")
writeBoolean(f,true)
flush(f)
fclose(f)
```

39) writeByte

This function writes one byte to file.

Syntax:-

```
writeByte(file,value)
```

Example:-

```
select "files"

f = fopen("text.txt","w")
writeByte(f,0)
flush(f)
fclose(f)
```

40) writeBytes

This function writes length bytes to file from byte array starting from offset.

Syntax:-

```
writeBytes(f, array, offset, length)
```

Example:-

```
select "files"

f = fopen("text.txt","w")

array =
["Dragon","Programming","Language"]

writeBytes(f1, array)

writeBytes(f1, array, 10)

writeBytes(f1, array, 20, 10)

flush(f)
fclose(f)
```

41) writeChar

This function writes one char (2 bytes) to file. Value can be number - writes number, or string - writes code of first symbol.

Syntax:-

```
writeChar(file,value)
```

Example:-

```
select "files"

f = fopen("text.txt","w")
writeChar(f,"A")
flush(f)
fclose(f)
```

42) writeDouble

This function writes 8 bytes double number to file.

Syntax:-

```
writeDouble(file,value)
```

Example:-

```
select "files"

f = fopen("text.txt","wb")
writeDouble(f,34632523.3464)
flush(f)
fclose(f)
```

43) writeFloat

This function writes 4 bytes float number to file.

Syntax:-

```
writeFloat(file,value)
```

Example:-

```
select "files"

f = fopen("text.txt","wb")
writeFloat(f,242.34)
flush(f)
fclose(f)
```

44) writeInt

This function writes 4 bytes integer number to file.

Syntax:-

```
writeInt(file,value)
```

Example:-

```
select "files"

f = fopen("text.txt","wb")
writeInt(f,12525)
flush(f)
fclose(f)
```

45) writeLine

This function writes string to file in text mode adds line break at the end of the string.

Syntax:-

```
writeLine(file,value)
```

Example:-

```
select "files"

f = fopen("text.txt","w")
writeLine(f,"Hello World")
flush(f)
fclose(f)
```

46) writeLong

This function writes 8 bytes long number to file.

Syntax:-

```
writeLine(file,value)
```

Example:-

```
select "files"

f = fopen("text.txt","wb")
writeLine(f,2362623636334)
flush(f)
fclose(f)
```

47) writeShort

This function writes 2 bytes short number to file.

Syntax:-

```
writeShort(file,value)
```

Example:-

```
select "files"

f = fopen("text.txt","wb")
writeShort(f,23)
flush(f)
fclose(f)
```

48) writeText

This function writes string to file in text mode. Unlike writeLine does not add line break.

Syntax:-

```
writeText(file,value)
```

Example:-

```
select "files"

f = fopen("text.txt","w")
writeText(f,"This is an Example!")
flush(f)
fclose(f)
```

49) writeUTF

This function writes string to file in binary mode.

Syntax:-

```
writeUTF(file,value)
```

Example:-

```
select "files"

f = fopen("text.txt","wb")
writeUTF(f,"Test")
flush(f)
fclose(f)
```

CHAPTER 6 http Module

The http module Contains network functions.

Functions

1) http

This function performs request with given method, object params and connection options to url, response will be send to function callback.

Syntax:-

```
http(url) - performs GET-request to
url.

http(url, method) - performs request
with method (GET, POST, PUT, DELETE,
PATCH, OPTIONS) to url.

http(url, callback) - performs GET-
request to url, response will be send
to function callback.

http(url, method, params) - performs
request with given method and object
params to url.

http(url, method, callback) -
performs request with given method to
url, response will be send to
function callback.
```

http(url, method, params, callback) - performs request with given method and object params to url, response will be send to function callback.

http(url, method, params, options, callback) - performs request with given method, object params and connection options to url, response will be send to function callback.

Connection options is a object (map):

- header - sets http-header (string or array).
- encoded - is params object already urlencoded.
- content_type - sets Content-Type.
- extended_result - marks that response should be extended and should contains:
  - text - server response text
  - message - server response message
  - code - server response code
  - headers - response http-header
  - content_length - Content-Length
  - content_type - Content-Type

Example:-

```
select "http"

http("http://jsonplaceholder.typicod
e.com/users", "POST", {"name":
"Dragon", "versionCode": 10},
func(v) {
  showln "Added: " + v
})
```

2) download

This function downloads content by url as bytes array.

Syntax:-

```
download(url)
```

Example:-

```
select "http"
select "files"
bytes = download("http://url")
f = fopen("file", "wb")
writeBytes(f, bytes)
flush(f)
fclose(f)
```

3) urlencode

This function converts string to URL-format.

Syntax:-

```
urlencode(string)
```

Example:-

```
select "http"
select "std"

url = urlencode("www.google.com")
http(url,::echo)
//chaining echo function to print
response.
```

CHAPTER 7 base64 Module

The base64 module Contains base64 encoding and decoding functions.

Functions

1) base64encode

This function encodes byte array or string into base64-encoded byte array.

Syntax:-

```
base64encode(data)
```

Example:-

```
select "base64"

showln base64encode("Hello")

/* return [83, 71, 86, 115, 98, 71,
56, 61] */
```

2) base64decode

This function decodes base64-encoded byte array or string into byte array.

Syntax:-

```
base64decode(data)
```

Example:-

```
select "base64"

select "std"

arr = base64decode([83, 71, 86, 115,
98, 71, 56, 61])

for val : arr{

show toChar(val)

}

//return Hello
```

3) base64encodeToString

This function encodes byte array or string into base64-encoded string.

Syntax:-

```
base64encodeToString(data)
```

Example:-

```
select "base64"

showln base64encodeToString("Dragon")
//return RHJhZ29u
```

CHAPTER 8 json Module

The json module Contains functions for working with the json format.

Functions

1) jsonencode

This Function converts string to data.

Syntax:-

```
jsonencode(jsonString)
```

Example:-

```
select "json"

data = {
  "key1": 1,
  "key2": [1, 2, 3],
  "key3": "text"
}
show jsonencode(data)
/*
{"key1":1,"key2":[1,2,3],"key3":"text
"} */
```

2) jsondecode

This function converts data to json string.

Syntax:-

```
jsondecode(data)
```

Example:-

```
select "json"

show
jsondecode("{\"key1\":1,\"key2\":[1,2
,3],\"key3\":\"text\"}")
/* return {key2=[1, 2, 3], key3=text,
key1=1} */
```

CHAPTER 9 yaml Module

The yaml module Contains functions for working with the yaml format.

Functions

1) yamlencode

This function converts yaml string to data.

Syntax:-

```
yamlencode(yamlString)
```

Example:-

```
select "yaml"

showln
yamlencode({"Name":"Aavesh","Position
":"Creator"})
/* return {Name: Aavesh,Position:
Creator} */
```

2) yamldecode

This function converts data to yaml string.

Syntax:-

```
yamldecode(data)
```

Example:-

```
select "yaml"

showln    yamldecode("Name:    Aavesh
\nPosition: Creator ")
/* return {Name=Aavesh,
Position=Creator} */
```

CHAPTER 10 ounit Module

The ounit module Contains functions for testing. Invokes all functions with prefix test and checks expected and actual values, counts execution time.

Functions

1) runTests

This function executes tests and returns information about its results.

Syntax:-

```
runTests()
```

Example:-

```
select "ounit"

showln runTests()
```

2) assertEquals

This function checks that two values are equal.

Syntax:-

```
assertEquals(expected, actual)
```

Example:-

```
select "ounit"

func testAdditionOnNumbers() {
  assertEquals(6, 0 + 1 + 2 + 3)
}

showln runTests()
/* return,
testAdditionOnNumbers [passed]
Elapsed: 0.0000 sec

Tests run: 1, Failures: 0, Time
elapsed: 0.0000 sec
*/
```

3) assertFalse

This function checks that value is false (equals 0).

Syntax:-

```
assertFalse(actual)
```

Example:-

```
select "ounit"

func testFail() {
  assertFalse(true)
}
showln runTests()
/* return,
testFail [FAILED]
Error: Expected false, but found true.
Elapsed: 0.0001 sec

Tests run: 1, Failures: 1, Time
elapsed: 0.0001 sec
*/
```

4) assertNotEquals

This function checks that two values are not equal.

Syntax:-

```
assertNotEquals(expected, actual)
```

Example:-

```
select "ounit"

func testNotEqual() {
    assertNotEquals(5,2+3)
 }
showln runTests()
/* return,

  testNotEqual [FAILED]
  Error: Values are equals: 5
  Elapsed: 0.0001 sec

  Tests run: 1, Failures: 1, Time
  elapsed: 0.0001 sec
  */
```

5) assertSameType

This function checks that types of two values are equal.

Syntax:-

```
assertSameType(expected, actual)
```

Example:-

```
select "ounit"

func testType() {
  assertSameType(5,12)
}

showln runTests()

/* return,

testType [passed]
Elapsed: 0.0000 sec

Tests run: 1, Failures: 0, Time
elapsed: 0.0000 sec
*/
```

6) assertTrue

This function checks that value is true (not equals 0).

Syntax:-

```
assertTrue(actual)
```

Example:-

```
select "ounit"

func testTr() {
  assertTrue(true)
}
showln runTests()

/* return,
testTr [passed]
Elapsed: 0.0000 sec

Tests run: 1, Failures: 0, Time
elapsed: 0.0000 sec
*/
```