INTRODUCTION

The Dragon is an innovative and practical general-purpose language. The supported programming paradigms are imperative, object-oriented, declarative using nested structures, functional, meta programming and natural programming. The language is portable (Windows, Linux, macOS, Android, etc.) and can be used to create Console, GUI applications. The language is designed to be simple, small, flexible and fast. The language has two versions that works on JVM and LLVM.

BASICS

MODULES

- socket
- base64
- json
- yaml
- functional
- robot
- ounit
- gragphic
- GUI
- db

COMMENTS

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

STRINGS

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

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

`show` and `showln` operators are used to output text.

USER DEFINED INPUTS

User defined inputs are used with readln function. Which is in std module. by default the input types are string.

```
select "std"
a = readln()
show a
```

DATA TYPES

Dragon data types are:

- Number - numbers (Integer,Float)
- String - strings
- Array - arrays
- Map - objects (an associative arrays)
- Object - object of class
- Function - functions
- Boolean - (true or false)

Since Dragon is dynamic programming language, which means that explicitly declare the types is not necessary.

```
 u = ["text",12] // Array
v = {"key1":1,"key2":2} // Map
w = true // boolean
x = 10 // integer
y = 1.61803 // float
z = "abcd" // string
```

If some function requires string as argument, but number was passed, then numeric value will automatically converts to string.

```
 x = 90
show x // Ok, 90 converts to "90"
```

LOOPS

while loop

```
while condition {
   body
}
```

Parentheses in condition are not necessary.

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

// or

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

do-while loop

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

Parentheses in condition are not necessary.

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

```
// or

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

for loop

```
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++
}
```

foreach loop

Iterates elements of an array or map.

```
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
}
```

FUNCTIONS DEFINITION

To define function uses the `func` keyword:

```
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
}
```

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
```

CLASSES

To define classes use the `class` keyword:

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


obj = new classname()// class object creation

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

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
```

DESTRUCTURING ASSIGNMENT

Destructuring assignment allows to define multiple variables for each element of an array or map.

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, key and value 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
```

PATTERN MATCHING

The `match` operator allows to match values by pattern.

```
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!"
  }
}
```

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`.

```
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
```

In this case is two scenarios:

- Variable is already defined. Matching to its value.
- Variable is not defined. Assign matching value to it and executes body of the `case` branch.

In the example above, the interpreter sees the first two branches as:

```
case 10:
case 20:
```

For the last branch `c` variable is not defined, so assign `c = x` and execute body of the `case c` branch.

Refinements

`case` branch may have additional comparison

```
func test(x) = match x {
  case x if x < 0: "(-∞ .. 0)"
  case x if x > 0: "(0 .. +∞)"
  case x: "0"
}

showln test(-10)  //  (-∞ .. 0)
showln test(0)  // 0
showln test(10)  //  (0 .. +∞)
```

Matching arrays

To compare elements of arrays, the following syntax is used:

- `case []:` executes if there are no elements in array
- `case [a]:` executes if value is an array. In this case, x will contain all elements
- `case [a :: b]:` executes if an array contain two or more elements
- `case [a :: b :: c :: d :: e]:` executes if an array contain five or more elements

There are two rules for the last two cases:

- If variables count matches array elements count - all variables are assigned to the value of the array.

```
match [0, 1, 2] {
  case [x :: y :: z]: // x = 0, y = 1, z = 2
}
```

- If array elements count is greater, then the rest of the array will be assigned to the last variable.

```
match [0, 1, 2, 3, 4] {
  case [x :: y :: z]: // x = 0, y = 1, z = [2, 3, 4]
}
```

An example of a recursive output array

```
func arrayRecursive(arr) = match arr {
    case [head :: tail]: "[" + head + ", " + arrayRecursive(tail) + "]"
  case []: "[]"
  case last: "[" + last + ", []]"
}

showln arrayRecursive([1, 2, 3, 4, 5, 6, 7]) // [1, [2, [3, [4, [5, [6, [7, []]]]]]]]
```

Matching array's value

To compare values of array's elements, the following syntax is used:

- `case (expr1, expr2, expr3):` executes if an array contain 3 elements and first element is equal to expr1 result, second element is equal to expr2 and third element is equal to expr3.
- `case (expr1, _):` executes if an array contain 2 elements and first element is equal to expr1 result and result of the second element is not importand.

FizzBuzz classical problem can be solved using Pattern Matching:

```
for i = 1, i <= 100, i++ {
  showln match [i % 3 == 0, i % 5 == 0] {
    case (true, false): "Fizz"
    case (false, true): "Buzz"
    case (true, true): "FizzBuzz"
    case _: i
  }
}
```

USING MODULES IN A PROGRAM

for using the modules in program the select keyword is used along with the module name in double quotes.

```
select "std" //std is a module in Dragon

echo("Hello World") //a function of std module for printing output on screen
```

INCLUDING A PROGRAM

A program in Dragon also can be used as an user defined module. The include keyword is used along with the program name in double quotes for including the program data.

```
include "program2.dgn" //Including the program2.dgn file code as an user defined module

addition(20,30) //a user defined function inside program2.dgn file
```

---

std

This module contains common functions

Constants

`ARGS` : *string* = `command-line arguments`

```
select "std"

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

Functions

1) `arrayCombine(keys, values)`
- creates map by combining two arrays. This function require two arguments and both should be arrays.

Example

```
 a = ["val1","val2"]
b = [20,30]

arrayCombine(a,b)  //This  will  create  a  map  {val1=20,
val2=30}
```

2) `arrayKeyExists(key, map)` - checks existing key in map. returns 1 if exists exist, 0 if not exist

Example

```
 ab = {"val1":10,"val2":20}

arrayKeyExists("val2",ab) //return 1, val2 exist in map ab

arrayKeyExists("val3",ab) //return 0, val3 not exist in map
ab
```

`arrayKeys(map)` - returns array of map keys

`arrayValues(map)` - returns array of map values

`charAt(input, index)` - returns char code in position `index` of string `input`

`echo(arg...)` - prints values to console, separate them by space and puts newline at the end. Takes variable number of arguments

Example

```
echo(1, "abc") // prints "1 abc" to console
echo(1, 2, 3, 4, 5, "a", "b") // prints "1 2 3 4 5 a b"
```

`indexOf(input, what, index = 0)` - finds first occurrence of `what` in string `input`, starting at position `index`

`join(array, delimiter = "", prefix = "", suffix = "")` - join array to string with `delimiter`, `prefix` and `suffix`

`lastIndexOf(input, what, index = 0)` - finds last occurrence of `what` in string `input`, starting at position `index`

`length(x)` - returns length of string, array/map size or number of function arguments

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

Example

```
newarray(4) // [0, 0, 0, 0]
newarray(2, 3) // [[0, 0, 0], [0, 0, 0]]
```

`parseInt(str, radix)` - parses string into integer in the radix

`parseLong(str, radix)` - parses string into long in the radix

`rand(from = 0, to = ..)` - 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 from `from` to `to`

`range(from = 0, to, step = 1)` - 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`

Example

```
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
```

`readln(x)` - reads a line from console

`replace(str, target, replacement)` - replaces all occurrences of string `target` with string `replacement`

`replaceAll(str, regex, replacement)` - replaces all occurrences of regular expression `regex` with string `replacement`

`replaceFirst(str, regex, replacement)` - replaces first occurrence of regular expression `regex` with string `replacement`

`sleep(time)` - causes current thread to sleep for `time` milliseconds

`sort(array, comparator = ..)` - sorts array by natural order or by `comparator` function

`split(str, regex, limit = 0)` - splits string `str` with regular expression `regex` into array. `limit` parameter affects the length of resulting array

Example

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

`sprintf(format, args...)` - formats string by arguments

`substring(str, startIndex, endIndex = ..)` - returns string from `startIndex` to `endIndex` or to end of string if `endIndex` is not set

Example

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

`sync(callback)` - calls an asynchronous function synchronously

Example

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

`thread(func, args...)` - creates new thread with parameters if passed

Example

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

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

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

`time()` - returns current time in milliseconds from 01.01.1970

`toChar(code)` - converts char code to string

Example

```
toChar(48) // "0"
```

`toHexString(number)` - converts number into hex string

`toLowerCase(str)` - converts all symbols to lower case `toUpperCase(str)` - converts all symbols to upper case

`trim(str)` - removes any leading and trailing whitespaces in string

`try(unsafeFunction, catchFunction = func(type, message) = -1)` - suppress any error in `unsafeFunction` and returns the result of the `catchFunction` if any error occurs

Example

```
try(func() = "success") // success
try(func() = try + 2) // -1
try(func() = try(), func(type, message) = sprintf("Error handled:\ntype: %s\nmessage: %s", type, message))
```

types

Contains functions for type checking and conversion

Constants

`OBJECT` : *number* = `0`

`NUMBER` : *number* = `1`

`STRING` : *number* = `2`

`ARRAY` : *number* = `3`

`MAP` : *number* = `4`

`FUNCTION` : *number* = `5`

Functions

1) `byte(value)` - converts value to byte

returns the output value.

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

showln byte(10) //output:- 10
```

2) `double(value)` - converts value to double

returns the output value.

```
showln double(1912412) //output:- 1912412.0
```

3) `float(value)` - converts value to float

returns the output value.

```
showln float(10) //output:- 10.0
```

`int(value)` - converts value to int

`long(value)` - converts value to long

`number(value)` - converts value to number if possible

Example

```
show typeof(number("2.3")) // 1 (NUMBER)
```

`short(value)` - converts value to short

`string(value)` - converts value to string

Example

```
show typeof(string(1)) // 2 (STRING)
```

`typeof(value)` - returns the type of value

Example

```
show typeof(1) // 1 (NUMBER)
show typeof("text") // 2 (STRING)
show typeof([]) // 3 (ARRAY)
```

math

Contains math functions and constants

Constants

`E` : *number* = `2.718281828459045`

`PI` : *number* = `3.141592653589793`

Functions

`abs(x)` - absolute value of `x`

`acos(x)` - arc cosine

`asin(x)` - arc sine

`atan(x)` - arc tangent

`atan2(y, x)` - returns angle θ whose tangent is the ratio of two numbers

`cbrt(x)` - cube root

`ceil(x)` - returns the ceiling of `x`

Example

```
ceil(6.4) // 7
```

`copySign(magnitude, sign)`

`cos(x)` - trigonometric cosine

`cosh(x)` - hyperbolic cosine

`exp(x)` - e^x

`expm1(x)` - e^x-1

`floor(x)` - returns floor of `x`

Example

```
floor(3.8) // 3
```

`getExponent(x)`

`hypot(x, y)`

`IEEEremainder(x, y)`

`log(x)`

`log1p(x)`

`log10(x)`

`max(x, y)`

`min(x, y)`

`nextAfter(x, y)`

`nextUp(x)`

`pow(x, y)`

`rint(x)`

`round(x)`

`signum(x)`

`sin(x)`

`sinh(x)`

`sqrt(x)`

`tan(x)`

`tanh(x)`

`toDegrees(x)`

`toRadians(x)`

`ulp(x)`

date

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`

Functions

`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.

Returns DateValue.

`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.

`formatDate(date, format = default)` - formats date by given format and returns string

Example

```
d = date(2016, 4, 8)
showln    formatDate(d,    newFormat("yyyy/MM/dd"))    //
"2016/05/08"
```

`parseDate(dateString, format = default)` - parses date from string by given pattern. Returns DateValue

Example

```
showln parseDate("2016/05/08", newFormat("yyyy/MM/dd"))
```

`toTimestamp(date)` - returns timestamp in milliseconds

Types

`DateValue`

`DateFormatValue`

files

Contains functions for working with files

Constants

`FILES COMPARATOR` : *function* = `func(f1, f2) = compare(f1, f2)`

function which compares two file descriptors

Functions

`canExecute(f)` - checks execute permission of the descriptor `f`

`canRead(f)` - checks read permission of the descriptor `f`

`canWrite(f)` - checks write permission of the descriptor `f`

`copy(src, dst)` - copies file src to dst location

`delete(f)` - removes file or directory. Returns 1 if delete was successfull, 0 otherwise

`exists(f)` - checks file or directory existing. Returns 1 if exists, 0 otherwise

`fclose(f)` - closes file

`fileSize(f)` - returns file size in bytes

`flush(f)` - flushes write buffer into file

`fopen(path, mode = "r")` - opens file файл with `path` in given `mode`:

- "" - 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.

  Returns a file descriptor for using in other functions.

Example

```
 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"
```

`getParent(f)` - returns parent path of the given descriptor `f`

`isDirectory(f)` - checks if descriptor `f` is directory

`isFile(f)` - checks if descriptor `f` is file

`isHidden(f)` - checks if descriptor `f` is hidden

`lastModified(f)` - returns last modification time

`listFiles(f)` - returns array with filenames in given directory.

f - directory descriptor

Example

```
 f1 = fopen("E:/examples", "") // opens directory examples
for getting information
list = listFiles(f1) // gets array with filenames in
directory
```

`mkdir(f)` - creates the directory. Returns 1 if operation was successfull, 0 otherwise

`mkdirs(f)` - creates the directories. Returns 1 if operation was successfull, 0 otherwise

`readAllBytes(f)` - reads all bytes from file. Returns array with bytes

Example

```
 f1 = fopen("file.bin", "rb")
array = readAllBytes(f1)
```

`readBoolean(f)` - reads boolean (1 byte). Returns 0 if byte was 0, 1 otherwise

`readByte(f)` - reads one byte

`readBytes(f, array, offset = 0, length = length(array))` - reads `length` bytes of file `f` to `array` starting from `offset`. Returns number of readed bytes

Example

```
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
```

`readChar(f)` - reads one char (2 bytes). Returns number char's code

`readDouble(f)` - reads 8 bytes double number

`readFloat(f)` - reads 4 bytes float number

`readInt(f)` - reads 4 bytes integer number

`readLine(f)` - reads line from file opened in text mode

`readLong(f)` - reads 8 bytes long number

`readShort(f)` - reads 2 bytes short number

`readText(f)` - reads all file's content as string

`readUTF(f)` - reads string in binary mode

`rename(from, to)` - renames (or moves) file

Example

```
f1 = fopen("C:/file1", "i")
f2 = fopen("E:/file2", "i")
rename(f1, f2)
fclose(f1)
fclose(f2)
```

`setLastModified(f, time)` - sets last modified time

`setReadOnly(f)` - marks descriptor read only

`setExecutable(f, executable, ownerOnly = true)` - sets execute permission

`setReadable(f, readable, ownerOnly = true)` - sets read permission

`setWritable(f, writable, ownerOnly = true)` - sets write permission

`writeBoolean(f, v)` - writes boolean (0 or 1) to file

`writeByte(f, v)` - writes one byte to file

`writeBytes(f, array, offset = 0, length = length(array))` - writes `length` bytes to file `f` from byte `array` starting from `offset`

`writeChar(f, v)` - writes one char (2 bytes) to file. `v` can be number - writes number, or string - writes code of first symbol

`writeDouble(f, v)` - writes 8 bytes double number to file

`writeFloat(f, v)` - writes 4 bytes float number to file

`writeInt(f, v)` - writes 4 bytes integer number to file

`writeLine(f, v)` - writes string to file in text mode **adds line break at the end of the string**

`writeLong(f, v)` - writes 8 bytes long number to file

`writeShort(f, v)` - writes 2 bytes short number to file `writeText(f, v)` - writes string to file in text mode. Unlike `writeLine` does not add line break

`writeUTF(f, v)` - writes string to file in binary mode

http

Contains network functions

Functions

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

`download(url)` - downloads content by url as bytes array

Example

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

`urlencode(str)` - converts string to URL-format

socket

Constants

`EVENT_CONNECT` : *string* = `connect`

`EVENT_CONNECTING` : *string* = `connecting`

`EVENT_CONNECT_ERROR` : *string* = `connect_error`

`EVENT_CONNECT_TIMEOUT` : *string* = `connect_timeout`

`EVENT_DISCONNECT` : *string* = `disconnect`

`EVENT_ERROR` : *string* = `error`

`EVENT_MESSAGE` : *string* = `message`

`EVENT_PING` : *string* = `ping`

`EVENT_PONG` : *string* = `pong`

`EVENT_RECONNECT` : *string* = `reconnect`

`EVENT_RECONNECTING` : *string* = `reconnecting`

`EVENT_RECONNECT_ATTEMPT` : *string* = `reconnect_attempt`

`EVENT_RECONNECT_ERROR` : *string* = `reconnect_error`

`EVENT_RECONNECT_FAILED` : *string* = `reconnect_failed`

Functions

`newSocket(url, options = {})` - creates new SocketValue

options (map with keys):

- forceNew (boolean)
- multiplex (boolean)
- reconnection (boolean)
- rememberUpgrade (boolean)
- secure (boolean)
- timestampRequests (boolean)
- upgrade (boolean)
- policyPort (integer)
- port (integer)
- reconnectionAttempts (integer)
- reconnectionDelay (timestamp - long)
- reconnectionDelayMax (timestamp - long)

- timeout (timestamp - long) - set -1 to disable
- randomizationFactor (double)
- host (string)
- hostname (string)
- path (string)
- query (string)
- timestampParam (string)
- transports (array of strings)

Types

`SocketValue`

Functions

`close()` - disconnects the socket

`connect()` - connects the socket

`connected()` - returns connected status (1 - connected, 0 - no)

`disconnect()` - disconnects the socket

`emit(event, data)` - emits an event

`hasListeners(event)` - returns true if there is listeners for specified event

`id()` - returns socket id

`off(event = ..)` - removes specified event handler, or removes all if no arguments were passed

`on(event, listener)` - adds event listener

`once(event, listener)` - adds one time event listener

`open()` - connects the socket

`send(data)` - send messages

base64

Contains base64 encoding and decoding functions

Constants

`BASE64_URL_SAFE` : *number* = `8`

Url safe encoding output

Functions

`base64decode(data, type = 0)` - decodes base64-encoded byte array or string into byte array

`base64encode(data, type = 0)` - encodes byte array or string into base64-encoded byte array

`base64encodeToString(data, type = 0)` - encodes byte array or string into base64-encoded string

json

Contains functions for working with the json format

Functions

`jsondecode(data)` - converts data to json string

Example

```
select "json"

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

`jsonencode(jsonString)` - converts string to data

Example

```
select "json"

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

yaml

Contains functions for working with the yaml format

Functions

`yamldecode(data)` - converts data to yaml string

`yamlencode(yamlString)` - converts yaml string to data

functional

Contains functions for operating data in functional style

Constants

`IDENTITY` : *function* = `func(x) = x`

function which returns passed argument

Functions

`chain(data, functions...)`

`combine(functions...)` - combines functions

Example

```
f = combine(::f1, ::f2, ::f3)
// same as
f = func(f1, f2, f3) = f3(f2(f1))
```

`dropwhile(data, predicate)` - skips elements while predicate function returns true

`filter(data, predicate)` - filters array or object.

`predicate` is a function which takes one argument for arrays or two arguments for objects

Example

```
nums = [1,2,3,4,5]
show filter(nums, func(x) = x % 2 == 0) // [2, 4]
```

`flatmap(array, mapper)` - converts each element of an array to other array

Example

```
nums = [1,2,3,4]
show flatmap(nums, func(x) {
  arr = newarray(x)
  for i = 0, i < x, i++
    arr[i] = x
  return arr
}) // [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
```

`foreach(data, consumer)` - invokes function `consumer` for each element of array or map `data`

If `data` - array, then in the consumer function, one parameter is needed, if the object is two (the key and the value).

Example

```
foreach([1, 2, 3], func(v) { show v })
foreach({"key": 1, "key2": "text"}, func(key, value) {
  show key + ": " + value
})
```

`map(data, mapper...)` - converts elements of array or map. If `data` is array - `mapper` converts his elements, if `data` is object - you need to pass `keyMapper` - converts keys and `valueMapper` - converts values

Example

```
nums = [3,4,5]
show map(nums, func(x) = x * x) // [9, 16, 25]
```

`reduce(data, identity, accumulator)` - converts elements of an array or a map to one value, e.g. sum of elements or concatenation string. `accumulator` takes one argument for array and two arguments for object (key and value).

Example

```
nums = [1,2,3,4,5]
show reduce(nums, 0, func(x, y) = x + x) // 15
```

`sortby(array, function)` - sorts elements of an array or an object by `function` result

Example

```
data = [
  {"k1": 2, "k2": "x"},
  {"k1": 7, "k2": "d"},
  {"k1": 4, "k2": "z"},
  {"k1": 5, "k2": "p"},
]
show sortby(data, func(v) = v.k1) // [{k1=2, k2=x}, {k1=4,
k2=z}, {k1=5, k2=p}, {k1=7, k2=d}]
show sortby(data, func(v) = v.k2) // [{k1=7, k2=d}, {k1=5,
k2=p}, {k1=2, k2=x}, {k1=4, k2=z}]
```

`stream(data)` - creates stream from data and returns StreamValue

StreamValue functions:

- `filter(func)` - filters elements
- `map(func)` - converts each element
- `flatMap(func)` - converts each element to array
- `sortBy(func)` - sorts elements by comparator function
- `takeWhile(func)` - takes elements while predicate function returns true
- `dropWhile(func)` - skips elements while predicate function returns true
- `skip(count)` - skips count elements
- `limit(count)` - limits elements size
- `custom(func)` - performs custom operation
- `reduce(func)` - converts elements to one value
- `forEach(func)` - executes function for each element
- `toArray()` - returns array of elements
- `count()` - returns count of elements

`takewhile(data, predicate)` - takes elements while predicate function returns true

robot

Contains functions for working with clipboard, processes, automation

Constants

`BUTTON1` : *number* = `16`

left mouse button code

`BUTTON2` : *number* = `8`

middle mouse button code

`BUTTON3` : *number* = `4`

right mouse button code

`VK_DOWN` : *number* = `40`

key down code

`VK_ESCAPE` : *number* = `27`

Escape key code

`VK_FIRE` : *number* = `10`

Enter key code

`VK_LEFT` : *number* = `37`

key left code

`VK_RIGHT` : *number* = `39`

key right code

Functions

`click(buttons)` - performs click with given mouse buttons

Example

```
click(BUTTON3) // right mouse button click
```

`delay(ms)` - delay by given milliseconds

`shell_exec(args...)` - executes the system commands with parameters

`execProcess(args...)` - executes the process with parameters

Example

```
execProcess("mkdir", "Test")
execProcess("mkdir Test")
execProcess(["mkdir", "Test"])
```

`execProcessAndWait(args...)` - same as `execProcess`, but waits until process completes, returns it's exit code

`fromClipboard()` - gets text from clipboard `keyPress(key)` - performs pressing key

`keyRelease(key)` - performs releasing key

`mouseMove(x, y)` - moves mouse pointer to given point

`mousePress(buttons)` - performs pressing the given mouse button

`mouseRelease(buttons)` - performs releasing the given mouse button

`mouseWheel(value)` - performs scrolling (< 0 - up, > 0 - down)

`setAutoDelay(ms)` - sets delay after each automation event

`toClipboard(text)` - adds text to clipboards

`typeText(text)` - performs typing text by pressing keys for each character

ounit

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

Functions

`assertEquals(expected, actual)` - checks that two values are equal

`assertFalse(actual)` - checks that value is false (equals 0)

`assertNotEquals(expected, actual)` - checks that two values are not equal

`assertSameType(expected, actual)` - checks that types of two values are equal

`assertTrue(actual)` - checks that value is true (not equals 0)

`runTests()` - executes tests and returns information about it's results

Example

```
select "ounit"

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

func testTypes() {
  assertSameType(0, 0.0)
}

func testFail() {
  assertTrue(false)
}

showln runTests()

/*
testTypes [passed]
Elapsed: 0,0189 sec

testAdditionOnNumbers [passed]
Elapsed: 0,0008 sec

testFail [FAILED]
Expected true, but found false.
Elapsed: 0,0001 sec

Tests run: 3, Failures: 1, Time elapsed: 0,0198 sec
*/
```

graphic

Contains functions for working with graphics

Constants

`VK_DOWN` : *number* = `40`

`VK_ESCAPE` : *number* = `27`

`VK_FIRE` : *number* = `10`

`VK_LEFT` : *number* = `37`

`VK_RIGHT` : *number* = `39`

`VK_UP` : *number* = `38`

Functions

`clip()`

`color()`

`drawstring()`

`foval()`

`frect()`

`keypressed()`

`line()`

`mousehover()`

`oval()`

`prompt()`

`rect()`

`repaint()`

`window()`

GUI

Contains functions for working with GUI

Constants

`BorderLayout` : *map* =

```
{
  AFTER_LINE_ENDS=After,
  LINE_END=After,
  LINE_START=Before,
  BEFORE_LINE_BEGINS=Before,
  CENTER=Center,
  EAST=East,
  BEFORE_FIRST_LINE=First,
  PAGE_START=First,
  AFTER_LAST_LINE=Last,
  PAGE_END=Last,
  NORTH=North,
  SOUTH=South,
  WEST=West
}
```

`BoxLayout` : *map* = `{X_AXIS=0, Y_AXIS=1, LINE_AXIS=2, PAGE_AXIS=3}`

`DISPOSE_ON_CLOSE` : *number* = `2`

`DO_NOTHING_ON_CLOSE` : *number* = `0`

`EXIT_ON_CLOSE` : *number* = `3`

`HIDE_ON_CLOSE` : *number* = `1`

`Align` : *map* =

```
{
  BOTTOM=3,
  CENTER=0,
  EAST=3,
  HORIZONTAL=0,
  LEADING=10,
  LEFT=2,
  NEXT=12,
  NORTH=1,
  NORTH_EAST=2,
  NORTH_WEST=8,
  PREVIOUS=13,
  RIGHT=4,
  SOUTH=5,
  SOUTH_EAST=4,
  SOUTH_WEST=6,
  TOP=1,
  TRAILING=11,
  VERTICAL=1,
  WEST=7
}
```

Functions

1) `newWindow(title)` - creates a new window

```
select "GUI" //selecting GUI module

win = newWindow("My Window") //Object created of newWindow.
Window title is My Window
//this object will be used for defining the properties.
```

Properties of newWindow

a) setSize(width,height)

```
win.setSize(400,400)
//size of window. 2 parameters require and both integer.
width and height are in pixels.
```

b) setLocation(x axis,y axis)

```
win.setLocation(100,100)
//location of window on screen. 2 parameters require and
both integer. x axis and y axis location in pixels
```

`message(text = "")` - creates new Message window

`newOpenFile()` - creates new Open File window

`newSaveFile()` - creates new Save File window

`borderLayout(hgap = 0, vgap = 0)` - creates BorderLayout

`boxLayout(panel, axis = BoxLayout.PAGE_AXIS)` - creates BoxLayout

`cardLayout(hgap = 0, vgap = 0)` - creates CardLayout

`flowLayout(align = FlowLayout.CENTER, hgap = 5, vgap = 5)` - creates FlowLayout

`gridLayout(rows = 1, cols = 0, hgap = 0, vgap = 0)` - creates GridLayout

`noLayout()` - creates null layout

`newButton(text = "")` - creates new button

`newText(text = "", align = SwingConstants.LEADING)` - creates new label

`newPanel(layoutManager = ...)` - creates new panel with optional layout manager

`newTextBox(text = "")` - creates new text box

`newPassBox(text = "")` - creates new password box

`newTextarea()` - creates new Text Area

`newMenuBar()` - creates new Menu Bar

`newMenu()` - creates new Menu

`newMenuItem(text = "")` - creates new Menu Item

`newSelectBox()` - creates new Select Box `newScroll()` - creates new Scroll Box

db

Constants

`CLOSE_ALL_RESULTS` : *number* = `3`

`CLOSE_CURRENT_RESULT` : *number* = `1`

`CLOSE_CURSORS_AT_COMMIT` : *number* = `2`

`CONCUR_READ_ONLY` : *number* = `1007`

`CONCUR_UPDATABLE` : *number* = `1008`

`EXECUTE_FAILED` : *number* = `-3`

`FETCH_FORWARD` : *number* = `1000`

`FETCH_REVERSE` : *number* = `1001`

`FETCH_UNKNOWN` : *number* = `1002`

`HOLD_CURSORS_OVER_COMMIT` : *number* = `1`

`KEEP_CURRENT_RESULT` : *number* = `2`

`NO_GENERATED_KEYS` : *number* = `2`

`RETURN_GENERATED_KEYS` : *number* = `1`

`SUCCESS_NO_INFO` : *number* = `-2`

`TRANSACTION_NONE` : *number* = `0`

`TRANSACTION_READ_COMMITTED` : *number* = `2`

`TRANSACTION_READ_UNCOMMITTED` : *number* = `1`

`TRANSACTION_REPEATABLE_READ` : *number* = `4`

`TRANSACTION_SERIALIZABLE` : *number* = `8`

`TYPE_FORWARD_ONLY` : *number* = `1003`

`TYPE_SCROLL_INSENSITIVE` : *number* = `1004`

`TYPE_SCROLL_SENSITIVE` : *number* = `1005`

Functions

`getConnection(...)` - `getConnection(connectionUrl)`

`getConnection(connectionUrl, driverClassName)`

`getConnection(connectionUrl, user, password)`

`getConnection(connectionUrl, user, password, driverClassName)`

Creates connection and returns ConnectionValue.

`mysql(connectionUrl)` - creates mysql connection

`sqlite(connectionUrl)` - creates sqlite connection

Types

`ConnectionValue`

Functions

`clearWarnings()`

`close()`

`commit()`

`createStatement()`

`getAutoCommit()`

`getCatalog()`

`getHoldability()`

`getNetworkTimeout()`

`getSchema()`

`getTransactionIsolation()`

`getUpdateCount()`

`isClosed()`

`isReadOnly()`

`prepareStatement()`

`rollback()`

`setHoldability()`

`setTransactionIsolation()`

`ResultSetValue`

Functions

`absolute()`

`afterLast()`

`beforeFirst()`

`cancelRowUpdates()`

`clearWarnings()`

`close()`

`deleteRow()`

`findColumn()`

`first()`

`getArray()`

`getBigDecimal()`

`getBoolean()`

```
getByte()

getBytes()

getConcurrency()

getCursorName()

getDate()

getDouble()

getFetchDirection()

getFetchSize()

getFloat()

getHoldability()

getInt()

getLong()

getNString()

getRow()

getRowId()

getShort()

getStatement()

getString()

getTime()

getTimestamp()

getType()

getURL()

insertRow()

isAfterLast()
```

`isBeforeFirst()`

`isClosed()`

`isFirst()`

`isLast()`

`last()`

`moveToCurrentRow()`

`moveToInsertRow()`

`next()`

`previous()`

`refreshRow()`

`relative()`

`rowDeleted()`

`rowInserted()`

`rowUpdated()`

`setFetchDirection()`

`setFetchSize()`

`updateBigDecimal()`

`updateBoolean()`

`updateByte()`

`updateBytes()`

`updateDate()`

`updateDouble()`

`updateFloat()`

`updateInt()`

`updateLong()`

`updateNString()`

`updateNull()`

`updateRow()`

`updateShort()`

`updateString()`

`updateTime()`

`updateTimestamp()`

`wasNull()`

`StatementValue`

Functions

`addBatch()`

`cancel()`

`clearBatch()`

`clearParameters()`

`clearWarnings()`

`close()`

`closeOnCompletion()`

`execute()`

`executeBatch()`

`executeLargeBatch()`

`executeLargeUpdate()`

`executeQuery()`

`executeUpdate()`

`getFetchDirection()`

`getFetchSize()`

`getGeneratedKeys()`

`getMaxFieldSize()`

`getMaxRows()`

`getMoreResults()`

`getQueryTimeout()`

`getResultSet()`

`getResultSetConcurrency()`

`getResultSetHoldability()`

`getResultSetType()`

`getUpdateCount()`

`isCloseOnCompletion()`

`isClosed()`

`isPoolable()`

`setBigDecimal()`

`setBoolean()`

`setByte()`

`setBytes()`

`setCursorName()`

`setDate()`

`setDouble()`

`setEscapeProcessing()`

`setFetchDirection()`

`setFetchSize()`

`setFloat()`

`setInt()`

`setLargeMaxRows()`

`setLong()`

`setMaxFieldSize()`

`setMaxRows()`

`setNString()`

`setNull()`

`setPoolable()`

`setQueryTimeout()`

`setShort()`

`setString()`

`setTime()`

`setTimestamp()`

`setURL()`