

# umscript

A scripting language for custom applications.

umscript is a scripting language which has been built with a similar syntax than C. It is however rather simplified. There are no arrays. Variables are not typed. A script is compiled into a in memory tree for execution which can be called ("Evaluated for its value, the return value") with a environment object.

## Syntax Reference

### 1. UMDiscreteValues & UMTerm & UMLEnvironment

---

#### 1.1 UMDiscreteValue

A UMDiscreteValue is a object which has a specific value. This can be one of the followings

- NULL            A null value (UMDiscreteNull)
- BOOL            A boolean which is either YES or NO
- INT             a integer (int)
- LONGLONG      a long long integer (long long)
- DOUBLE        a fractional value (double)
- STRING        a string Value (NSString)
- DATA          an arbitrary data object (NSData)

UMDiscreteNull is returned by a function not returning a value or by a undefined variable etc.

---

#### 1.2 UMTerm

A UMTerm is an object which is either a UMDiscreteValue or a calculated object (such as 1+1) which needs to be evaluated to be converted to a UMDiscreteValue. Every function call is a UMTerm. A whole programm is a UMTerm returning a UMDiscreteValue as a return value.

There are the following direct term types:

|                     |  |
|---------------------|--|
| UMTermType_discrete | A direct UMDiscreteValue   |
| UMTermType_field    | A field value. What „Field“ means depends on the application.<br>It can be a database field for example. |
| UMTermType_variable | A variable is a placeholder for a UMDiscreteValue in memory which<br>is addressed by its variable name.  |
| UMTermType_function | A function call  |

|                       |   |
|-----------------------|---|
| UMTermType_identifier | A identifier such as a jump label.  |
| UMTermType_nullterm   | A null UMTerm (placeholder for something which is not there)                  |
| UMTermType_token      | A internal token fed from the parser before its identified as any other type. |

---

## 1.3 UMLEnvironment

An UMLEnvironment is an object holding all the variables, the custom functions and the callbacks for reading/writing fields. It is usually subclassed by the application which uses the umscript library. Its a holder of all the variables.

## 2. Built in functions

---

### 2.1 Addition (UMFunction\_add)

Syntax:  $\{value1\} + \{value2\}$

this takes two or more values and adds them together.

If the values are both strings, this is a concatenation of strings.

If the values are both data, this is a concatenation of data.

The return type is the type of the first element.

Example:

1 + 2.1 will return 3

2.1 + 1 will return 3.1

"1" + 2.1 will return "12.1"

---

### 2.2 Subtraction (UMFunction\_sub)

Syntax:  $\{value1\} - \{value2\}$

Subtraction is analog to addition except the numbers are subtracted.

---

### 2.3 Multiplication (UMFunction\_mul)

Syntax:  $\{value1\} * \{value2\}$

Multiplication is analog to addition except the numbers are multiplied.  
Multiplication of a string with an integer n will concatenate the string n times.

So watch out: "3" \* 3 returns 333 and not 9  
use (int)"3" \* 3 instead

---

## 2.4 Division (UMFunction\_div)

Syntax:  $\{value1\} / \{value2\}$

division is analog to multiplication except the numbers are divided.

---

## 2.5 Bitwise AND (UMFunction\_bit\_and)

Syntax:  $\{value1\} \& \{value2\}$

---

## 2.6 Bitwise OR (UMFunction\_bit\_or)

Syntax:  $\{value1\} | \{value2\}$

---

## 2.7 Bitwise XOR (UMFunction\_bit\_xor)

Syntax:  $\{value1\} \wedge \{value2\}$

---

## 2.8 Bitwise Leftshift (UMFunction\_bit\_shiftleft)

Syntax:  $\{value1\} \ll \{value2\}$

---

## 2.9 Bitwise Rightshift (UMFunction\_bit\_rightshift)

Syntax:  $\{value1\} \gg \{value2\}$

---

## 2.10 Logical NOT (UMFunction\_not)

Syntax: **!{value}**

---

## 2.11 Logical AND (*UMFunction\_and*)

Syntax: **{value1} && {value2}**

---

## 2.12 Logical OR (*UMFunction\_and*)

Syntax: **{value1} || {value2}**

---

## 2.13 Logical XOR (*UMFunction\_and*)

Syntax: **{value1} ^^ {value2}**

---

## 2.14 Assignment (*UMFunction\_assign*)

Syntax: **{variable\_or\_field} = {value}**

---

## 2.15 Variable

Syntax: **\$name**

---

## 2.16 Greater Than (*UMFunction\_greaterthan*)

Syntax: **{value1} > {value2}**

Returns YES if {value1} is greater than but not equal to {value2} and NO otherwise

---

## 2.17 Greater Than or equal to (*UMFunction\_greatertorequal*)

Syntax: **{value1} >= {value2}**

Returns YES if {value1} is greater than or equal to {value2} and NO otherwise

---

## 2.18 Less Than (*UMFunction\_lessthan*)

Syntax: `{value1} < {value2}`

Returns YES if `{value1}` is less than but not equal to `{value2}` and NO otherwise

---

## 2.19 Less Than or equal to (*UMFunction\_lessequal*)

Syntax: `{value1} <= {value2}`

Returns YES if `{value1}` is less than or equal to `{value2}` and NO otherwise

---

## 2.20 Equal (*UMFunction\_equal*)

Syntax: `{value1} == {value2}`

Returns YES if value1 is equal to value2.

If both sides are strings, then they are string compared case sensitive

---

## 2.21 if , if else (*UMFunction\_if*)

Syntax: `if( {condition} ) { {block} }`  
`if( {condition} ) { {block} } else { {block} }`

The if statement does only evaluate the block if the condition is true. The else block is evaluated otherwise if present. Note that in comparison to traditional C language, there is no syntax with an if and a single statement. In other words the `{ }` brackets are not optional here.

---

## 2.22 while (*UMFunction\_while*)

Syntax: `while( {condition} ) { {block} }`

The block is executed as long as the condition is true.

Inside the block a **continue** statement will jump out right to the next execution loop and a **break** will jump out of the loop completely.

---

## 2.23 do while (*UMFunction\_dowhile*)

Syntax: `do { {block} } while ({condition})`

The block is executed once and then as long as the condition is true.

Inside the block a **continue** statement will jump out right to the next execution loop and a **break** will jump out of the loop completely.

---

## 2.24 Block (UMFunction\_block)

Syntax:        **{ statements1; statement;2 etc etc }**

---

## 2.25 Switch / case / default

Syntax:        **switch(condition) { switchblock }**

The condition is evaluated and the continuation of execution inside the switchblock is started at the case label with the value of the result.

Example

```
switch(var)
{
    case 1:
    case 2:
        $b = 3;
    case 3:
        $b = 6;
        break;
    case 4:
        $b = 7;
        break;
    default:
        $b = 9;
}
```

if var is equal to 1, the execution starts after "case 1:" and stops at the "break". This means \$b will be 6. The same is true for var=2 or var=3. For var = 4, the value of \$b will be 7. For any other value it will be 0. "break" jumps out of the switch block.

---

## 2.26 for

Syntax:        **for( {initialisation}; {looptest}; {increase} ) { block }**

This is equivalent to:

```
{initialisation};
```

```
while( {looptest} )  
{  
    {block};  
    {increase};  
}
```

**break** and **continue** are analogous to while.

---

## 2.27 return

Syntax:        **return;**  
                 **return value;**

defines the return value of a function

---

## 2.28 preincrease

Syntax:        **++variable**

the variable is increase by 1.  
the term is resulting as the increased value

---

## 2.29 postincrease

Syntax:        **variable++**

the return value is the value of the variable.  
The variable is increased afterwards.

---

## 2.30 predecrease

Syntax:        **--variable**

the variable is increase by 1.  
the term is resulting as the increased value

---

## 2.31 postdecrease

Syntax:        **variable--**

the return value is the value of the variable.  
The variable is increased afterwards.

---

## 2.32 goto

Syntax:       **goto** *{labelname}*  
Execution continues at the named label.

---

## 2.33 label

Syntax        *{labelname}* :

---

## 2.34 Modulo

Syntax:       *{var1}* % *{var2}*

---

## 2.35 not equal

Syntax:       *{var1}* != *{var2}*

---

## 2.36 value conversion to integer

Syntax:       **(int)** *{var1}*

---

## 2.37 value conversion to string

Syntax:       **(string)** *{var1}*

---

## 2.38 value conversion to double

Syntax:       **(double)***{var1}*



---

## 2.39 value conversion to boolean

Syntax:        **(bool)** {var1}

---

## 2.40 value conversion to longlong

Syntax:        **(long long)** {var1}

---

## 2.41 Substring

Syntax:        **substr**({value}, {startpos}, {length})  
                 **substr**({value}, {startpos} )

returns a string of length "length" or shorter which starts at the startpos position of the original string. The first position of a string is position 0. If length is omitted, the whole remaining is returned.

---

## 2.41 Stringcompare

Syntax:        **stringcompare**({string1}, {string2})  
                 **stringcompare**({string1}, {string2}, {int})

compares the two strings string1 and string2.  
it returns -1 if strings are in ascending order  
returns 0 if strings are equal  
returns 1 if strings are in decending order

if the third parameter is passed with a value other than 0, then the comparison is case insensitive.

---

## 2.42 Datetime

Syntax:        **datetime**({format}, {timezone}, {locale})  
                 **datetime**({format}, {timezone})  
                 **datetime**({format}, {timezone})  
                 **datetime**()

returns the date in the specified format, timezone and locale

Defaults are:

|          |                          |
|----------|--------------------------|
| format   | yyyy-MM-dd HH:mm:ss.SSSS |
| timezone | UTC                      |
| locale   | en_US                    |

The format string uses the format patterns from the Unicode Technical Standard #35. The version of the standard supported can slightly vary depending of the operating system  
For MacOS X 10.9 the following applies:

[http://www.unicode.org/reports/tr35/tr35-31/tr35-dates.html#Date\\_Format\\_Patterns](http://www.unicode.org/reports/tr35/tr35-31/tr35-dates.html#Date_Format_Patterns)

---

## 2.43 Hash

Syntax:       **hash**(*{date or string}*, *{options}*)  
              **hash**(*{date or string}* )

returns a hashvalue

By default the output is a hexstring of a SHA1 hash.

As input you can specify a string or a data object.

options is a string with space separated options. The following values are accepted in options field

|        |                           |
|--------|---------------------------|
| SHA1   | calculate a SHA1 hash     |
| SHA224 | calculate a SHA224 hash   |
| SHA256 | calculate a SHA256 hash   |
| SHA384 | calculate a SHA384 hash   |
| SHA512 | calculate a SHA512 hash   |
| string | return as hexstring value |
| data   | return as data object     |

---

## 2.44 Namedlists

Syntax:       **namedlist\_add**(*{list-name}*, *{value}*)  
              **namedlist\_remove**(*{list-name}*, *{value}*)  
              {bool} = **namedlist\_contains**(*{list-name}*, *{value}*)

named lists are a universal construct of general purpose lists which you can use for blacklists or whitelists. You can add entries to the list using **namedlist\_add**, remove the entries using **namedlist\_remove** or verify if they are in the list already by using **namedlist\_contains**.

---

## 2.44 Regular Expressions

Syntax:       **regex**(*{pattern}*, *{value}*)

returns true if the patter matches the value.

---

## 2.45 Country Of

Syntax:       **country\_of**(*{msisdn}*)

returns the country name of the msisdn indicated. Does not expect + in front of msisdn.

## 3. Constants, Variables and Fields

---

### 3.1 Constants

Constants are embedded discrete values.

|       |                             |
|-------|-----------------------------|
| "abc" | is a discrete string        |
| 123   | is an integer               |
| 123.0 | is a double                 |
| 123LL | is a long long              |
| YES   | is a boolean of value true  |
| NO    | is a boolean of value false |

Strings can have escape characters in them such as `\n \t` or `\0x1D` etc.

---

### 3.2 Variables

Variables are placeholders in memory for a discrete value.

Variable names are starting with a dollar sign. They are stored in the environment and keep their value as long as the environment is kept. They are currently all global and are held by the `UMEnvironment` object.

---

### 3.3 Fields

Fields are placeholders for values provided by the application.

they can be set with `setfield(name,value)` or read with `getfield(name)`. This will call the application specific `UMEnvironment` object which then can do whatever makes sense with this specific application

---

### 3.4 Comments

Comments are in C style such as

`// a single line comment`

`/* a multil  
line comment */`

---

### 3.5 Preprocessor

There is no preprocessor available.