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 & UMEnvironment

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. It can be a database field for example.
UMTermType_variable	A variable is a placeholder for a UMDiscreteValue in memory which 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 UMEnvironment

An UMEnvironment 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.

2. Built in functions

2.1 Addition (UMFunction_add)

Syntax: *{value1}* **+** *{value2}*

this takes two 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"

Current Limitation:

1+2 will be parsed as 1 and +2 (positive value of 2) and is thus not an addition.
write as 1 + 2 instead (adding spaces between + and the numbers)
This will likely be changed in the future to follow standard behaviour.

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.

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*} **^** {*value2*}

2.8 Bitwise Leftshift (UMFunction_bit_shiftleft)

Syntax: {*value1*} **<<** {*value2*}

2.9 Bitwise Rightshift (UMFunction_bit_rightshift)

Syntax: {*value1*} **>>** {*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 Field

Syntax: ***%name***

2.17 Greater Than (*UMFunction_greaterthan)*

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

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

2.18 Greater Than or equal to (*UMFunction_greatertorequal)*

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

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

2.19 Less Than (*UMFunction_lessthan)*

Syntax: {value1} **<** *{value2}*

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

2.20 Less Than or equal to (*UMFunction_lessorequal)*

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

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

2.21 Equal (*UMFunction_equal)*

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

Returns YES if value1 is equal to value2.

2.22 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.

2.23 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.

2.24 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.

2.25 Block (UMFunction_block)

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

2.26 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.

2.27 for

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

This is equivalent to:

```
{initialisation};

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

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

2.28 return

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

defines the return value of a function

2.29 preincrease

Syntax: **++***variable*

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

2.30 postincrease

Syntax: *variable***++**

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

2.31 predecrease

Syntax: **--***variable*

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

2.32 postdecrease

Syntax: *variable***--**

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

2.33 goto

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

2.34 label

Syntax {labelname} **:**

2.35 Modulo

Syntax: *{var1}* **%** *{var2}*

2.36 not equal

Syntax: *{var1}* **!=** *{var2}*

2.37 value conversion to integer

Syntax: **int(***{var1}***)**

2.38 value conversion to string

Syntax: **string(***{var1}***)**

2.39 value conversion to double

Syntax: **double(***{var1}***)**

2.40 value conversion to boolean

Syntax: **bool(***{var1}***)**

2.41 value conversion to longlong

Syntax: **longlong(***{var1}***)**

2.42 Substring

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

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.

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.

3.3 Fields

Fields are placeholders for values provided by the application.
Field names are starting with a percent sign. When a field value is read, the environment gets a callback to provide the value. When a field value i written, the environment is called to set the value.

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.