Reference

Version 3.00

m Mobile Shell, Reference, Version 3.00
Written by Lukas Knecht

www.m-shell.net

Document AB-M-REF-741

Typeset in Switzerland.

Contents

1. Introduction

m is a simple and easy to learn programming language intended for mobile phones (''Smart Phones''). **m** has been specifically designed for the limited text editing capabilities of these devices. The language thus has few special characters, and the library functions generally use short identifiers.

To obtain a flat learning curve, in particular for the novice user, and to keep editing **m** code manageable on a cell phone, the **m** language has been kept simple, while still providing a rich set of programming constructs and functions.

Likewise, the library of modules closely reflects the capabilities of smart phones. Modules have been designed with ease of use in mind, without requiring complex setup operations or even an understanding of the underlying architecture. The module library is described in the ''Library'' manual, which complements this reference manual.

To protect the phone's data, the user's purse, and the phone's integrity from malevolent scripts, permissions to use potentially dangerous functions are configurable.

2. Language

This chapter defines the **m** programming language. **m** is a procedural language supporting code reuse through a simple concept of modules.

The following sections introduce the building blocks of **m**. After each section, the **m** syntax is summarized by a formal definition in EBNF (Extended Backus Naur Form):

- Text in single quotes ″ corresponds to the actual text (terminal symbols).
- Text in bold face denotes keywords (reserved words).
- The vertical bar | separates alternatives.
- Text in brackets [] is optional.
- Text in curly braces {} can be repeated (zero times, once or many times).
- Text in parentheses () is grouped together.

m scripts are read as a series of tokens which are separated by ''separator'' characters (all characters which are not letters, digits or an underscore). White space (blank and new line) always separates two characters. The amount of white space used does not affect the meaning of a script, but white space should be added sensibly to make a script more readable by indenting lines to reflect the structure of the code.

2.1 Data Types

m supports the following data types:

- Number: numbers have a range of roughly -10^{308} to 10^{308} and have a precision of almost 17 decimal digits[1].
- String: strings are sequences of characters[2]. Strings are immutable: their length is fixed, and individual characters cannot be changed. However, there are many builtin functions (see builtin functions (Library, p. 7)) manipulating strings.
- Boolean: booleans are logical values, i.e. either `true` or `false`. For instance, the result of a comparison is of Boolean type. Booleans are also often used as flags or to denote options for functions.
- Array: arrays are collections of arbitrarily many values. Multidimensional arrays (e.g. matrices) are constructed as arrays of arrays. In **m**, arrays are dynamic in size. Elements can be appended or removed. Elements can be indexed by numbers or strings (''associative array''). See also section 2.5 (p. 11).
- Class Instance: an instance of a class (an object). Class instances are at the center of object oriented programming (OOP) in **m**. Chapter 2.11 (p. 43) explains **m**'s OOP features.
- Null: this special type denotes an uninitialized or unspecified value. The only value of this type is `null`.
- Function Reference: a reference (''pointer'') to a function. The reference can be used to specify callback functions, or to implement a simple polymorphism scheme.
- Instance Function Reference: a reference to a function of a class instance. The reference can be used to specify callback functions operating directly on class instances.
- Native Objects: are created by modules which are tied closely into the underlying operating system, e.g. by module `io` (Library, p. 36). Native objects can only be assigned and compared for identity.

[1] Internally, numbers are stored as 64 bit floating point values in IEEE format, with 52+1 bit mantissa and 11 bit exponent.

[2] Internally, each character is represented by 16 bits, thus supporting the UNICODE® basic multilingual plane. However, fonts often support only the ISO-8859-1 (Latin) character set.

2.2 Comments

Normally, all characters in an **m** script are assumed to be **m** language. Comments intended for the human reader must therefore be specially marked:

- Single line comments start with a double slash: any text from a `//` to the end of the line containing it is considered a comment.
- Multiline comments start with slash-star and end with star-slash: any text between `/*` and `*/` is considered a comment and ignored. These comments can be nested.

```
print 3*3 // this prints nine
→ 9
/* The following m code is within this comment,
   so it is ignored:
print 5/7
   This is still part of the comment.
   /* This is a nested comment ending here: */
   This is the last line of this comment. */
print 3/4
→ 0.75
```

Comment marks cannot be placed within string literals (see section 2.3 (p. 7)).

2.3 Literals

Literals are concrete values specified explicitly in the code. Except for array literals, they are fixed and cannot change during script execution. Array literals are more complex and discussed in section 2.5 (p. 11).

```
SimpleLiteral := NumberLiteral | StringLiteral | BooleanLiteral |
  FunctionLiteral | NullLiteral .
```

Number Literals

A number literal is a sequence of digits, with an optional decimal point, and an optional decimal exponent. The digits must not be separated by white space or thousands separators:

```
print 0
→ 0
print 3.1415927
→ 3.1415927
print 6.02214199e+23
→ 6.022142E+23
print 1E-3
→ 0.001
```

Integer numbers can also be written in hexadecimal notation, by prefixing them with `0x`:

```
print 0xff
→ 255
print 0x1000
→ 4096
```

```
NumberLiteral :=
  Digit {Digit} ['.' {Digit}]
    [(E' | 'e') ['-' | '+'] Digit {Digit}] |
  '0x' HexDigit {HexDigit} .
Digit :=
  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
HexDigit := Digit | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
  'a' | 'b' | 'c' | 'd' | 'e' | 'f' .
```

String Literals

A string literal is a sequence of characters between single or double quotes.

```
print 'Hello, world!'
→ Hello, world!
print "That's nice"
→ That's nice
```

In order to produce all characters, the backslash `\` serves as escape for

the following character. For instance, if the quote used to delimit the string literal occurs inside the string, it must be escaped. Likewise, the backslash itself must be escaped, as is often seen in path names:

```
print "A quote: \"To be, or not to be...\""
→ A quote: "To be, or not to be..."
print 'That\'s nice'
→ That's nice
print "c:\\system\\apps"
→ c:\system\apps
```

There are a few characters which have a special meaning when escaped:

\f	form feed (ASCII 12)
\n	new line or line feed (ASCII 10)
\r	carriage return (ASCII 13)
\t	horizontal tab (ASCII 9)
\u	hexadecimal UNICODE® (UTF-16) follows

```
print "Line1\nLine2"
→ Line1
  Line2
print "Item1\tItem2"
→ Item1    Item2
print "g\u00e9nial"
→ génial
```

The maximum length of a string literal is 256 characters.

```
StringLiteral := '"' {Char | EscapeChar | "'"} '"' |
  "'" {Char | EscapeChar | '"'} "'" .
Char := (printable ISO-8859-1 char except ', ", \)
EscapeChar := '\' ('n' | 'r' | 't' |
  'u' HexDigit HexDigit HexDigit HexDigit | (printable char)) .
```

Boolean Literals

Not surprisingly, there are just two boolean literals: `true` and `false`.

```
BooleanLiteral := false | true .
```

Function Literals

A function literal is a reference to a (already declared) function. Section 2.8 (p. 36) explains function references.

```
FunctionLiteral := '&' [ModulePrefix] Identifier .
```

Null Literal

The `null` literal denotes a ''special'' value which is different from all other values.

```
NullLiteral := null .
```

2.4 Variables

A variable is a storage location identified by a name. Values can be assigned to (stored in) the variable, and the value can later be retrieved by the same name.

Variable (and function, class and module) identifiers are sequences of ordinary latin letters, digits, and the underscore character.

- Identifiers must not start with a digit.
- Identifiers are case sensitive, i.e. lowercase and uppercase variants are different.
- Keywords (see appendix A.2 (p. 78)) cannot be used as identifiers.
- The maximum length of an identifier is 64 characters.

Examples for valid identifiers:

```
a
Z
AvogadroConstant
avogadro_constant
_4
x1
```

Examples for invalid identifiers:

```
9a // starts with a digit
end // is a keyword
This_identifier_is_too_long_to_be_accepted_as_it_is_over_64_chars
```

```
IdentifierChar := 'A' to 'Z' | 'a' to 'z' | '_' .
Identifier := IdentifierChar {IdentifierChar | Digit} .
```

There are three different kinds of variables:

- Global variables belong to a module (see section 2.9 (p. 37)) and exist as long as the process containing the module exists. Global variables can only be created within the module declaring them.
- Local variables belong to a function (see section 2.8 (p. 32)) and can only be referenced within their function. They are different from global variables with the same name, and exist as long as the function executes: they are created when the function is called, and are destroyed when the function returns. Hence, each invocation of a recursive function creates its own set of local variables. Function parameters are also local variables .
- Class fields belong to a class instance (see section 2.11 (p. 43)) and exist as long as the class instance exists. Class fields are declared when declaring the class, and created when creating a class instance. They are different from local and global variables with the same name.

The distinction between global and local variable references in the code is made by module prefixes. See section 2.9 (p. 37) for examples and an explanation.

```
ModulePrefix := [ModuleName | '.'] '.' .
Variable := [ModulePrefix] Identifier .
ModuleName := Identifier .
```

2.5 Arrays

Arrays are collections of values. The array values can be of different type, and they can themselves again be arrays. The individual array elements

are accessed by indexing with integer numbers, starting at 0 for the first element. Indexing requires putting the index value between brackets `[]`, following the array variable.

Trying to access an element with a negative or too large index throws `ExcIndexOutOfRange`.

Function `.len` (Library, p. 15) returns the number of elements in the array.

Arrays are created by array literals, or by functions in module `array` (Library, p. 20). An array literal is a comma-separated sequence of element values between brackets:

```
a=["One", "Two", "Three"];
print a[0] // first element
→ One
print a[2] // third element
→ Three
print len(a)
→ 3
print a[3] // there is no fourth element
→ ExcIndexOutOfRange thrown
```

Arrays in **m** are completely dynamic, i.e. they can grow and shrink in size. Function `.append` (Library, p. 7) appends elements to an array:

```
append(a, "Four", "Five");
print a
→ [One,Two,Three,Four,Five]
```

Associative Arrays

Array values can also be indexed by strings (''keys''), making the arrays ''associative'' and facilitating many programming tasks. Setting or getting an array element via a string key is a fast operation[3]. Normally, keys are case sensitive, but `array.new` (Library, p. 24) can also create arrays using case folded keys.

Unlike indexing with numbers, indexing with strings for nonexisting index values does not throw an exception:

[3] Internally, keys are organized into a dynamic hash table.

- Getting an element for a nonexisting key returns `null`.
- Setting an element for a nonexisting key appends the element to the array.

Arrays with string keys can still be indexed using integer values.

In array literals, preceding an element value with a key and a colon adds the corresponding key:

```
h=["Joe":150, "Jack":165, "William":180, "Averell":195];
print h["Jack"]
→ 165
print h["Lucky Luke"] // element does not exist
→ null
h["Lucky Luke"]=185; // element is appended
print h
→ [150,165,180,195,185]
print h[2]
→ 180
```

See also: `.append` (Library, p. 7), `.keys` (Library, p. 14), module `array` (Library, p. 20).

```
Literal := SimpleLiteral | ArrayLiteral .
ArrayKey := Expression .
ArrayValue := Expression .
ArrayElement := [ArrayKey ':'] ArrayValue .
ArrayLiteral := '[' [ ArrayElement {',' ArrayElement} ] ']' .
Designator := Variable { Selector } [ InstanceFunctionReference ] .
Selector := '[' Expression ']' | InstanceSelector .
```

2.6 Expressions

Generally speaking, expressions define (arithmetic, bitwise, comparison, or logical) operations on (variable, literal, or function result) operands.

Operands

In **m**, there are four types of operands:

- Designators: the operand is the value of a variable or array element, e.g. `count`, `list[i]`, or a class instance field.
- Function Calls: the operand is the result of a function call, e.g. `io.read(f, 10)`, `math.sin(x)`. Functions are explained in sections 2.8 (p. 32) and 2.11 (p. 47).
- Literals: the operand is a literal, i.e. an explicit value, e.g. `42`, `"Hello"`.
- Expression: the operand is an expression in parentheses, e.g. `(7.2*x)`, `(not exists[key])`.

Operation Precedence

Each operation has a precedence defining the order in which operations are executed: as a general rule, arithmetic and bitwise operations are executed before comparisons, and comparisons are executed before boolean operations. Within each group, multiplicative operations have higher precedence than additive ones. Operations of equal precedence are executed from left to right. The order of execution can be changed by grouping subexpressions into parentheses.

```
t=3; s=7; m="aha";
print t + 5*s - 2/4 // multiplicative before additive
→ 37.5
print s&4 > t|4 // bitwise before comparison
→ false
print 13>s or m>"b" // comparison before boolean ops
→ true
print (20+t)*(s-24) // parentheses change the order
→ -391
```

Arithmetic Operators

The arithmetic operators are (P is the precedence):

Op	P	Description
`x+y`	4	Addition.
`x-y`	4	Subtraction.
`x*y`	5	Multiplication.
`x/y`	5	Division.
`x%y`	5	Integer remainder: `x - y*trunc(x/y)`; if `y=0`, throws `ExcDivideByZero`.
`-x`	6	Change sign of `x`.

```
print 22 / 7
→ 3.142857149
print 97 % 11
→ 9
print 97 % -11
→ 9
print -97 % 11
→ -9
```

Except for `%`, these operations never throw an exception if an invalid operation is attempted or overflow or underflow occurs. Instead, the result becomes (negative or positive) infinity, or zero:

```
x=1e200;
print x*x
→ Inf
print -2/0
→ -Inf
print 1/x/x
→ 0
```

Bitwise Operators

Bitwise operators work on integer numbers, treating them like signed binary numbers of 32 bits. Such operations are typically used to represent sets of binary states (e.g. flags) in a single value, or for hardware related operations.

The bitwise operators are (P is the precedence):

Op	P	Description
`x|y`	4	Bitwise or.
`x^y`	4	Bitwise exclusive or.
`x&y`	5	Bitwise and.
`x shl y`	5	Bitwise shift left.
`x shr y`	5	Bitwise shift right.
`~x`	6	Bitwise not.

```
print 1|2|4|8
→ 15
print 10&(2|4)
→ 2
print 14^11
→ 5
print ~(14&11) & (14|11) // ~(a&b) & (a|b) = a^b
→ 5
print 13 shl 4 // 13*16
→ 208
print 341 shr 2 // trunc(341 / 4)
→ 85
print ~0 // set all bits in signed integer
→ -1
print -5 & ~19
→ -24
```

Concatenation Operator

The concatenation operator concatenates two strings or a string with the string representation of another value (P is the precedence):

Op	P	Description
`x + y`	4	Concatenation: `x` followed by `y`.

Note that if neither of the two operands is a string, the two operands are assumed to be numbers and added.

```
print "One" + "Two"
→ OneTwo
print "x=" + 3/4
→ x=0.75
```

Comparison Operators

Comparing two operands always produces a boolean value. Testing for equality and inequality works for all pairs of operands. Operands of different types (e.g. a number and a string) are never equal.

Only numbers and strings can be ordered, i.e. compared for less or greater than. Strings are ordered by their UNICODE® character values, which orders uppercase before lowercase, and does not produce a general lexical ordering. Use `.collate` (Library, p. 9) to lexically compare strings.

Trying to order operands other than numbers or strings throws `ExcNotComparable`.

Two arrays are only equal if they are the same array. `.equal` (Library, p. 9) compares two arrays element by element.

`null` is only equal to itself.

The comparison operators are (P is the precedence):

Op	P	Description
`x = y`	3	`true` if `x` is equal to `y`.
`x # y`	3	`true` if `x` is not equal to `y`.
`x <> y`	3	The same as x # y.
`x < y`	3	`true` if `x` is less than `y`.
`x <= y`	3	`true` if `x` is less than or equal to `y`.
`x > y`	3	`true` if `x` is greater than `y`.
`x >= y`	3	`true` if `x` is greater than or equal to `y`.

```
print 7>5
→ true
print "o" + "ne" = "one"
→ true
print "two" < "three"
→ false
print "Two" < "three" // no lexical ordering
→ true
print 14 = "a"
→ false
print 13 # "b"
→ true
print 13 < "14"
→ ExcNotComparable thrown
```

Class Instance Tests

The class instance test checks whether an expression is an instance of a given class. In other words, `x is C` tests whether an assignment

```
v:C=x
```

succeeds. Section 2.11 (p. 43) explains classes and class hierarchies.

The class instance test operator is (P is the precedence):

Op	P	Description
`x is C`	3	`true` if `x` is an instance of class C, or is `null`.

```
// declare a simple class hierarchy
class C end
class D is C end

// numbers and strings are never class instances
print 7 is C,"hello" is D
→ false false
// test actual instances
x=C()
print x is C, x is D
→ true false
x=D()
print x is C, x is D
→ true true
// null is an instance of all classes
x=null
print x is C, x is D
→ true true
```

Boolean Operators

The boolean operators are (P is the precedence):

Op	P	Description
`x or y`	1	Logical or: `true` if either `x` or `y` is `true`, `false` if both `x` and `y` are `false`.
`x and y`	2	Logical and: `true` if both `x` and `y` is `true`, `false` if either `x` or `y` are `false`.
`not x`	6	Logical not: `true` if `x` is `false`, `false` if `x` is `true`.

```
print false or false, false or true, true or false,
  true or true
→ false true true true
print false and false, false and true, true and false,
  true and true
→ false false false true
print not false, not true
→ true false
```

The second operand is only evaluated if the first operand doesn't already

determine the result. This is often useful when doing combined checks, as it avoids evaluation of invalid expressions:

```
ok=m#0 and 17%m = 3 // deadly 17%0 is never evaluated
```

```
Expression := Predicate {or Predicate} .
Predicate := Comparison {and Comparison} .
Comparison :=
  Sum [('=' | '<>' | '#' | '<' | '>' | '<=' | '>=') Sum] |
  is ClassIdentifier] .
Sum := Product {('+' | '-' | '|' | '^') Product} .
Product := Factor {('*' | '/' | '%' | '&' | shl | shr) Factor} .
Factor := ['-' | '~' | not ]
  (Designator | FunctionCall | IndirectFunctionCall | InstanceCreation |
   Literal | '(' Expression ')' ) .
```

See 2.11 (p. 43) for an explanation of class identifiers and instance creation.

2.7 Statements

Statements are the smallest unit of execution in **m**. Statements change values of variables, call functions and control the flow of execution. Most of the time, several statements are executed in a sequence, one after the other.

A sequence of statements is called a *statement list*. Within a statement list, statements must be separated by a semicolon. This is the only place where **m** requires a semicolon. In particular, there is no need to put a semicolon at the end of each statement[4]. However, ending or preceding each statement with a semicolon is not an error, it just produces empty statements which are ignored during execution.

For instance, the following two code fragments are completely equivalent:

[4] This minimalistic approach was chosen to reduce the number of control characters required for a valid **m** script.

```
use math;
function f(x);
  return x*x*math.exp(x/40);
end;
for x=0 to 10 by 0.1 do;
  y=f(x);
  print x,y;
end;
```

```
use math
function f(x)
  return x*x*math.exp(x/40)
end
for x=0 to 10 by 0.1 do
  y=f(x); // this is the only required semicolon
  print x,y
end
```

```
Statement := |
  Assignment | DeclaringAssignment | ConstAssignment |
  Increment | Expression |
  IfStatement | WhileStatement | DoStatement | ForStatement |
  BreakStatement | ReturnStatement | ThrowStatement |
  TryStatement | PrintStatement .
StatementList = Statement { ';' Statement } .
```

2.7.1 Assignments

This statement type assigns the value of an expression to a variable or array element. It also declares the variable if it didn't occur in the preceding code yet. A variable can be reassigned as often as required, generally also with values of different types (although this is not considered good programming practice, and there is an exception: variables declared to hold instances of a given class can only be assigned such instances, or `null`).

```
x = 28*3;
x = ["a", "b", "c"];
x[1] = "b2";
x[2] = null;
x["new"] = "d";
print x
→ [a,b,null,d]
```

When assigning an array, the array is not copied: the expression and the variable or array element it is assigned to will denote the same array:

```
ma = ["Ma", "Dalton"];
joe = ma; // joe and ma refer to the same array
joe[0] = "Joe"; // this also modifies ma
print ma
→ [Joe,Dalton]
```

`array.copy` (Library, p. 20) copies an array element by element.

If a variable is being declared, i.e. didn't occur in the preceding code, it can be marked as constant by prefixing the assignment with `const`. Array elements cannot be marked constant: a constant array can be modified after it has been assigned to another variable.

```
const C = 2.997e8;
C = 4; // illegal
const A = [1, 2, 3];
A[1] = 7; // illegal
b = A;
b[1] = 7; // perfectly legal, also modifies A[1]
print A
→ [1,7,3]
```

```
Assignment := Designator '=' Expression .
DeclaringAssignment := VariableDeclaration '=' Expression .
ConstAssignment := const VariableDeclaration '=' Expression .
VariableDeclaration := Identifier OptionalType .
OptionalType := [ ':' ClassIdentifier ] .
```

Declaring assignments declare a variable to hold only instances of a given class. See 2.11 (p. 45) for examples and details.

2.7.2 Increment

This statement type increments or decrements a numeric variable by a numeric expression (+=, −=), or simply by one (++, −). These statements are just shorthand notations for full assignments[5]:

Increment	Equivalent Assignment
`x += expr`	`x = x + expr`
`x -= expr`	`x = x - expr`
`x++`	`x = x + 1`
`x-`	`x = x - 1`

```
s=7;
s+=13;
s--;
print s
→ 19
```

```
Increment := Designator
  ('+=' Expression | '-=' Expression | '++' | '--') .
```

2.7.3 If Statement

An `if` statement executes some code depending on the value of boolean expressions (e.g. comparisons). Its simplest form executes statements (the `print` in the example) if a condition (`a > 13`) evaluates to `true`:

```
a=15;
if a > 13 then
  print a + " is greater than 13"
end
→ 15 is greater than 13
```

An optional `else` block may contain statements which are executed if the condition evaluates to `false`:

[5] They are not completely equivalent: in `s[f(x)]+=3`, `f(x)` is evaluated only once, whereas in `s[f(x)]=s[f(x)]+3`, `f(x)` is evaluated twice.

```
a=9;
if a > 13 then
  print a + " is greater than 13"
else
  print a + " is less than 13"
end
→ 9 is less than 13
```

To test for more than just two alternatives, an arbitrary number of `elsif` blocks can be added. These must occur after the `if/then` and before the (optional) `else` block:

```
a=13;
if a > 14 then
  print a + " is greater than 14"
elsif a < 13 then
  print a + " is less than 13"
elsif a = 13 then
  print a + " is equal to 13"
else
  print a + " must be 14"
end
→ 13 is equal to 13
```

If any of the conditions evaluates to `true`, the remaining conditions are not evaluated.

Throws `ExcNotBoolean` if any of the evaluated conditions is not boolean.

```
IfStatement := if Expression then StatementList
  {elsif Expression then StatementList}
  [else StatementList]
  end .
```

2.7.4 While Statement

The `while` statement repeats some code as long as a condition evaluates to `true`. The condition is tested *before* each repetition.

```
a=[430, 241, 187, 53, -1, 17];
s=0; i=0;
while i<len(a) and a[i]>=0 do
  s += a[i]; i++
end;
print i, s
→ 4 911
```

Throws `ExcNotBoolean` if the condition is not boolean.

```
WhileStatement := while Expression do StatementList end .
```

2.7.5 Do-Until Statement

The `do` statement repeats some code until a condition evaluates to `true`. The condition is tested *after* each repetition.

```
x=2; y=x;
do
  y0=y; y=(y+x/y)/2
until y>=y0;
print y, y*y
→ 1.4142135624 2
```

Throws `ExcNotBoolean` if the condition is not boolean.

```
DoStatement := do StatementList until Expression .
```

2.7.6 For Statement

The `for` statement lets an index variable iterate through a range of numbers or through the elements of an array, and executes some code for each value. The index variable must be a simple variable, either local in the current function or global in the current module. It cannot be an array element or a variable in another module.

- The `for` loop iterating through a range of numbers looks as follows:

```
for index=StartExpr to EndExpr [by IncrExpr] do
  statements
end
```

The range is defined by `StartExpr` and `EndExpr`, and an optional `IncrExpr` defining the amount by which the variable is incremented after each iteration. `IncrExpr` defaults to `1`.

All three expressions are evaluated only once, before the loop is entered.

The loop exits if `index > EndExpr` (if `IncrExpr > 0`), or if `index < EndExpr` (if `IncrExpr <= 0`).

```
for x=5 to 6 by 0.25 do
  print x*x
end
→ 25
   27.5625
   30.25
   33.0625
   36
```

A `for` loop over a range is equivalent to the following `while` loop:

```
index=StartExpr; e=EndExpr; d=IncrExpr;
while d>0 and index <= e or d<=0 and index >= e do
  statements;
  index += d
end
```

Care must be taken when using `for` loops with fractional numbers: rounding errors may lead to surprising results:

```
for i=5 to 6 by 0.2 do
  print i
end
→ 5
   5.2
   5.4
   5.6
   5.8
print i-6
→ 8.881784E-16
```

- The `for` loop iterating through the elements of an array looks as follows:

```
for index in ArrayExpr do
  statements
end
```

The array is defined by `ArrayExpr`. `index` iterates through all elements of the array, starting at index 0 and ending with the last element (index `len(ArrayExpr)-1`).

```
a=[430, 241, 187, 53, -1, 17]; s=0;
for x in a do
  s += x
end;
print s
→ 927
```

A `for` loop over an array is equivalent to the following `while` loop:

```
a=ArrayExpr; i=0;
while i<len(a) do
  index = a[i];
  statements;
  i++
end
```

```
ForStatement := for IndexVariable
   ( = Expression to Expression [by Expression] | in Expression )
  do StatementList end .
IndexVariable := Identifier .
```

2.7.7 Case Statement

The `case` statement executes a sequence of statements depending on the value of an expression matching the tag or tags of this sequence. It looks as follows:

```
case Expression
  in TagExpr1:
  statements1
  in TagExpr2a, TagExpr2b:
  statements2
else
  statements3
end
```

This `case` statement is equivalent to the following `if` statement:

```
x=Expression;
if x=TagExpr1 then
  statements1
elsif x=TagExpr2a or x=TagExpr2b then
  statements2
else
  statements3
end
```

Expression is evaluated only once.

The tags (*TagExpr1*, *TagExpr2a*,...) are evaluated when they are tested. Once a matching tag has been found, the remaining tags are not evaluated.

Equality of expression and tag is tested using the = operator (see section 2.6 (p. 17)). Arrays are thus not compared elementwise, and string comparison is case sensitive.

The following example prints a different message for different values of `i`:

```
for i=1 to 10 do
  case i
    in 1:
    print i,"is somewhat prime"
    in 2, 3, 5, 7:
    print i,"is prime"
  else
    print i,"is not prime"
  end
end
→ 1 is somewhat prime
   2 is prime
   3 is prime
   4 is not prime
   5 is prime
   6 is not prime
   7 is prime
   8 is not prime
   9 is not prime
   10 is not prime
```

```
CaseStatement := case Expression
  { in TagList ":" StatementList }
  [ else StatementList ] end .
TagList := Expression { "," Expression } .
```

2.7.8 Break Statement

The `break` statement exits from the loop (`while`, `do-until`, `for`) containing it, and continues execution after the end of the loop.

```
x=-3;
while true do
  if x<0 then break end;
  y=x; x=x/2+1/x;
  if x>=y then break end
end;
print x, x*x
→ -3 9
```

`break` always exits the innermost loop containing it. Breaking out of an outer loop is not possible.

```
BreakStatement := break .
```

2.7.9 Return Statement

The `return` statement returns the value of an expression as a function result. Outside a function, it ends execution of the module's body; the return value is discarded.

See section 2.8 (p. 32) for examples.

```
ReturnStatement := return Expression .
```

2.7.10 print Statement

The `print` statement provides a simple way of producing output. It writes a line with zero, one or several expressions to the console. The expressions are separated by single spaces. `print` without expressions just outputs a new line.

```
print "odd:",3/7
→ odd: 0.4285714286
print
→
```

Expressions are formatted depending on their type:

- A *Number* is printed as string of length 12 or less (and rounded, if necessary). If the value cannot be represented within 12 characters, scientific representation is chosen.

```
print 13.5
→ 13.5
print 3e11
→ 300000000000
print -3e11; // -300000000000 has 13 characters
→ -3E+11
```

- A *String* is printed as is.

```
print "Hello,", 'world!'
→ Hello, world!
```

- A *Boolean* is printed as `"true"` or `"false"`:

```
print 1 < 3
→ true
```

- An *Array* is printed elementwise, up to a length of 128. Elements which are themselves arrays are printed as `[...<len>]`.

```
a=[];
for i=1 to 10 do append(a, i) end;
print a
→ [1,2,3,4,5,6,7,8,9,10]
a[0]=a;
print a
→ [[...<10>],2,3,4,5,6,7,8,9,10]
for i=11 to 100 do append(a, i) end;
print a
→ [[...<100>],2,3,4,5,6,7,8,9,10,11,12,13,14,
   15,16,17,18,19,20,21,22,23,24,25,26,27,28,
   29,30,31,32,33,34,35,36,37,38,39,...<100>]
```

- A *Function Reference* is printed as `&module.function`.

```
f=&io.read;
print f
→ &io.read
```

- The *Null* value is printed as `null`.
- A *Native Object* is printed as `type@address`. `type` defines the object type, `address` is the location of the underlying native object in memory.

```
f=io.create("sample.xml");
print f
→ stream@41255c
```

For finer control over output formatting, see `.str` (Library, p. 17) and module `io` (Library, p. 36).

```
PrintStatement := 'print' [ Expression { ',' Expression } ] .
```

2.8 Functions

Functions are a way to write repeatedly occuring computations only once, but use them wherever needed. They also help in structuring longer scripts into smaller, easily understandable units. By putting often occuring functions into separate modules (see section 2.9 (p. 37)), function libraries can be created.

Functions normally have a set of *parameters* as input and return a single *function result* as output. Since the function result can be an array, an arbitrary number of values can be returned.

The function is left by returning a value with a `return` statement. If the function is left by reaching its end, `null` is returned.

The following example declares a function `sqrt` computing the square root of a number `x` greater than or equal to 1, then calls it with parameters `x=2` and `x=9`:

```
function sqrt(x)
  y=x;
  do
    y0=y; y=(y+x/y)/2
  until y>=y0;
  return y
end // of function sqrt

print sqrt(2), sqrt(9)
→ 1.4142135624 3
```

This is quite a simple function with a single parameter `x` and a simple function result (the value of `y`).

Multiple parameters are separated by commas. The following function `find` finds the index of the first element in an array `a` with a value equal to `x` (there is a standard function for this: `array.index` (Library, p. 22)).

```
function find(a, x)
  i=0;
  while i<len(a) and a[i]#x do
    i++
  end;
  return i
end

print find([9, 11, 13], 11)
→ 1
print find([9, 11, 13], 8)
→ 3
```

A function can be recursive, i.e. can call itself: the following function `clone` returns a full copy of its parameter `t`. It uses `array.copy` (Library, p. 20) to copy all the elements, and `.isarray` (Library, p. 12) to test whether `t` is an array.

```
function clone(t)
  if isarray(t) then
    c=array.copy(t);
    // recursively clone the elements
    for i=0 to len(t)-1 do c[i] = clone(t[i]) end;
    return c
  else
    return t
  end
end
```

If a function returns an array, the call can be followed by expressions in brackets accessing certain elements:

```
a=['one':1, 'two':2, 'three':3];
print keys(a)[2]
→ three
```

Function parameters are like local variables; they can optionally be declared to hold an instance of a given class. Likewise, the function can be declared to return an instance. See 2.11 (p. 45) for examples and details.

Optional Parameters

Optional parameters are parameters with a default value: if the parameter is omitted, the default value is assumed. The expression to compute the default value can be any expression which is valid in the global context (i.e. it cannot use a preceding function parameter). It is evaluated when the function is called, not when the function is declared.

When calling a function, the number of actual parameters must not be less than the number of mandatory parameters in the declaration of the function, and not be greater than the total number of declared parameters.

The following rewrites function `find` by adding an optional parameter `start` indicating the position to start searching at. The default value of `start` is `0`, so calling `find` with only two parameters produces exactly the same result as before:

```
function find(a, x, start=0)
  while start<len(a) and a[start]#x do
    start++
  end;
  return start
end

print find([9, 11, 13], 11)
→ 1
print find([9, 11, 13], 11, 2) // start=2
→ 3
```

Functions with optional parameters can have options for simplified syntax in interactive use (see section 3.1 (p. 55)). Options are simply single character names for optional parameters.

```
function grow(years,interest=2) /i:interest
  a=1;
  while years>0 do
    a+=a*interest/100; years--
  end;
  return a
end

grow(10,5)
→ 1.21899442
grow/i=5 10 // works only in interactive shells
→ 1.6288946268
```

Forward Declaration

Functions must be declared before they can be used. This means that if two functions call each other, at least one must be declared with `forward` and implemented later. In the following example, either `f` or `g` must be forward declared, since function `f` calls function `g` and vice versa:

```
function g(x, a=3.2) forward // g is made known

function f(x)
  if x<3 then
    return g(x*x) // g is called
  else
    return x+2
  end
end

function g(x, a=3.2) // here g is declared
  return f(x+a)
end
```

The default values of the optional parameters of the forward declared function are only used to mark optional parameters, their values are ignored. The default values are taken from the function implementation. The number of mandatory and optional parameters in forward declaration and implementation must match.

Function References

Function references allow to change the function called in an expression during the execution of a script: when a function reference is assigned to a variable or a parameter, the function can be called via the variable. The reference of a function is obtained by prefixing it with an ampersand character `&`:

```
f=&lower; // f now references the lower function
print f("Hello") // a call to lower
→ hello
f=&upper; // f now references the upper function
print f("Hello") // a call to upper
→ HELLO
```

Function references are often used to pass a function as a parameter to another function: the function `integ` approximates the integral of `f` from `a` to `b`:

```
function integ(f, a, b, n=100)
  s=(f(a)+f(b))/2; h=(b-a)/n;
  for i=1 to n-1 do
    s+=f(a+i*h)
  end;
  return s*h
end

function inv(x) return 1/x end
print integ(&inv, 1, 2)
→ 0.6931534305
print integ(&math.sin, 0, math.pi/2)
→ 0.9999794382
print integ(&math.sin, 0, math.pi/2, 10000)
→ 0.9999999979
```

```
FunctionDeclaration := function Identifier FunctionBody .
FunctionBody := '(' [ParameterList] ')' OptionalType
   ( forward | {FunctionOption} StatementList end ) .
ParameterList := (MandatoryParameter {',' MandatoryParameter} |
  OptionalParameter) {',' OptionalParameter} .
MandatoryParameter := VariableDeclaration .
OptionalParameter := VariableDeclaration '=' Expression .
FunctionOption := '/' OptionName ':' ParameterName .
OptionName := IdentifierChar | Digit .
ParameterName := Identifier .

ActualParameterList := Expression {',' Expression} .
FunctionCall :=
  ([ModulePrefix] Identifier '(' [ActualParameterList] ')'
  { Selector } [ InstanceFunctionReference ] .
IndirectFunctionCall :=
  Designator '(' [ActualParameterList] ')'
  { Selector } [ InstanceFunctionReference ] .
```

2.9 Modules

A module in **m** is a script (a text file) which can be loaded by other scripts, giving access to the functions and variables declared in the module.

Modules serve two purposes:

- They help in structuring complex scripts and make them easier to understand and maintain.
- They offer a way of extending the functionality of **m** by adding new functions which can then be used by all scripts or interactive **m** sessions. Entire libraries of often needed functions can be created that way. The standard library of **m** described in the next chapter is organized into modules.

To load a module, a `use` clause is required:

```
use ModuleName1, ModuleName2, ...
```

This loads the modules `ModuleName1`, `ModuleName2` and so on, and initializes them, i.e. executes their main code . Each module is only initialized once per process, even if it is loaded several times by different modules.

Module names are not case sensitive, since they are related to file names on the underlying operating system.

```
use System // load module "system"
print System.appdir; // this will work
print system.appdir // this is the same
```

An alias name can be used in addition to the module name to denote the module, e.g. to abbreviate a long module name. Alias names are local to the module containing the use clause. Like module names, they are not case sensitive:

```
use ModuleName as AliasName
```

As an example, consider the following module `accounts` maintaining a list of accounts and allowing transfers between them:

```
S=[]; // initialization of the module
function get(nr)
  x=accounts.S[nr];
  // all accounts start at zero
  if x=null then x=0 end;
  return x
end
function xfer(f, t, x)
  ..S[f]=get(f)-x;
  ..S[t]=get(t)+x
end
```

Within the functions, the global variable `S` must be prefixed by the module name (`accounts.S`), or by the double dot prefix indicating the current module (`..S`).

The module can then be used as follows:

```
// load the module and name it 'acc'.
use accounts as acc
// transfer money out of the blue to the bank
acc.xfer('blue', 'bank', 100000);
print acc.get('bank')
→ 100000
// transfer money from the bank to the Daltons
acc.xfer('bank', 'Daltons', 10000);
print acc.get('bank')
→ 90000
// show all accounts
print acc.S
→ [-100000, 90000, 10000]
```

Module Prefixes

Global variables and functions must normally be prefixed by the name of the module defining them (or the corresponding alias), and a dot. The prefix for the main script and the builtin functions and variables is just a dot, without a name.

Within a module, global variables and functions of the same module can be prefixed by a *double dot* `..`: in the code for module `accounts` above, `..S` denotes the same variable as `accounts.S`.

The prefixing is not always required when the variable or function is referenced in the module containing it. Furthermore, functions declared in the main script or builtin standard functions only need a prefix if a function with the same name exists in the current module. The following table summarizes how variables and functions *without module prefix* are interpreted:

	Variable `x`	Function `f`
main module	global `.x`	`.f`
function in main module	local `x`	`.f`
module `M`	global `M.x`	`M.f` if it exists, `.f` otherwise
function in module `M`	local `x`	`M.f` if it exists, `.f` otherwise

Module Version

Each module has a version, which is a number in the form major.minor; the minor component by convention has a 1/100th granularity.

The module version is a special variable `version`, which can only be modified in the module itself by assigning a number literal to it. If no number has been assigned, the version is `0.0`. The version of an uncorrectly loaded optional module (see below) is `null`.

Source of module `client`:

```
version=1.23
```

```
use client
// require at least version 1.20 of client module
if client.version>=1.20 then
  ...
end
```

The version of the builtin module is always the version of the **m** application. See also `.version` (Library, p. 19).

Optional Modules

Not all devices support all modules, or a module may simply not be installed on a device. To cope with these cases in the code, a module can be loaded in a `use` clause with the `try` prefix:

```
use try ModuleName
```

Loading a module with the `try` prefix has the following effects:

- If the module and all the modules it uses are correctly loaded, the result is almost the same as without `try`. However, a reference to an undeclared function or variable of the module will not be detected until the code reaches the corresponding statement and throws `ErrNotAvailable`. This allows to run code even if some functions or variables of a module do not exist.
- If the module `ModuleName` itself or one of the modules it uses is not found or cannot be loaded, no error is marked. However, all refer-

ences to its variables and functions will result in `ErrNotAvailable` being thrown; only the module's `version` variable is accessible and will return `null`.

```
use try nirvana
nirvana.f(1, 2)
→ ErrNotAvailable thrown
print nirvana.val
→ ErrNotAvailable thrown
// nirvana.version is null: the module cannot be used
print nirvana.version
→ null

use try math
// there is no sinh function in module math
print math.sinh(1.2)
→ ErrNotAvailable thrown
// math.version is not null: the module can be used
print math.version
→ 1.08
```

```
ModuleImportList := use ModuleImport { ',' ModuleImport } .
ModuleImport := [try] ModuleName [as AliasName] .
AliasName := Identifier .
```

2.10 Exceptions

An exception is the result of an attempt to perform an invalid operation. By default, exceptions result in a popup window showing the exception message text.

An exception thrown by **m** will always have the following format:

```
ExceptionFormat := tag ':' message
```

The tag is always an (english) identifier, and independent of the language chosen when installing **m**. The message however depends on the language. See section A.1 (p. 73) for a list of **m** exception tags.

Catching Exceptions

Exceptions can also be handled (''catched'') in **m** itself:

```
try
  // code potentially throwing exceptions
catch exc by
  // code handling the exception exc
end
```

The result of such a `try` block is the following:

- If the code between `try` and `catch` does not throw any exception, the code between `catch` and `end` will never be executed.
- If the code between `try` and `catch` does throw an exception, the exception will be assigned to the variable denoted by `catch` and the following code will be executed.

In the following example, `a[1]` tries to access an non-existing element. **m** throws an `ExcIndexOutOfRange`, which is catched and simply printed:

```
try
  a=[12];
  print a[1]
catch e by
  print "Got", e
end
→ Got ExcIndexOutOfRange: Array index is out of range
```

Try blocks can be nested to any depth (as long as the required memory is available).

```
TryStatement := try StatementList
  catch ExceptionVariable by StatementList end .
ExceptionVariable := Identifier .
```

Throwing Exceptions

Exceptions can also be thrown explicitly in the code:

```
throw expression
```

This will evaluate `expression` and use it as exception message. In the following example, an exception with the message ''state.dat does not exist'' will be thrown if this file does not exist.

```
if not files.exists("state.dat") then
  throw "state.dat does not exist"
end
```

```
ThrowStatement := throw Expression .
```

2.11 Object Oriented Programming

Starting with version 3.00, **m** offers the following OOP features:

1. Classes with fields and (virtual) functions.
2. Single inheritance to build class hierarchies.
3. Creation of class instances (objects) and initialization with constructor functions.
4. Polymorphism through function overriding.
5. Instance function references providing a callback mechanism for listeners or event handlers (''delegates'').

Classes and Class Instances

A class is a declaration of related variables (fields) and functions (''methods'') operating on them. A class only declares a type or pattern; the data is created by creating class instances.

Classes can be based on existing classes, inheriting all their fields and functions. Inherited functions can be overwritten to change the behaviour or functionality offered by the class.

A class is declared by the keyword `class` followed by the class name, its fields and its functions, followed by the keyword `end`:

```
class Sum
  s
  function add(x)
    s+=x
  end
  function res()
    return s
  end
end
```

This declares a class `Sum` with a field `s`, and two functions: one to add a value `x` to `s`, the other to return the sum of all added values.

A class always belongs to the module declaring it (which can be the builtin module or main script). A class is hence uniquely identified by the module declaring it and its name within the module: if class `Sum` is declared in module `Aggreg`, it must be referenced as `Aggreg.Sum` (or with the corresponding alias of `Aggreg`) in other modules.

Classes must be declared before they can be used. This means that if two classes reference each other, at least one must be declared with `forward` and defined later. In the following example, either `C` or `D` must be forward declared, since class `C` references class `D` and vice versa:

```
class C forward // make C known, without any details

class D
  x: C // C can be used, but C.y is not yet visible
end

class C // define C
  y
  function f(d: D)
    return y*d.x.y
  end
end
```

```
ClassIdentifier := [ModulePrefix] Identifier .
ClassDeclaration := class Identifier
  ( forward | [ is ClassIdentifier ] ClassBody end ) .
ClassBody := { VariableDeclaration | FunctionDeclaration [';'] }  .
```

Variable Declarations, revisited

A variable can be declared to always reference an instance of a given class (or to be `null`). This allows to directly access the fields and functions of the instance. For instance, to declare a variable `x` referencing an instance of `Sum`, follow the first assignment (i.e. the ''declaration'') of `x` by a colon and the class it references:

```
x:Sum=null
```

A variable cannot be redeclared, or declared lazily: the first assignment occuring in the source must declare its type, or it remains of undeclared type (like an ordinary **m** variable).

Whenever an expression is assigned to a variable of declared class, the value being assigned is checked. If it is not an instance of the declared class and not `null`, `ExcNotSuchInstance` is thrown:

```
x:Sum=null
a=23*7
x=a // a holds a number, not a Sum instance
→ ExcNotSuchInstance thrown
```

Function Declarations, revisited

Function parameters are like local variables, and can be declared to hold instances of a given class. For example, a function to multiply an instance `s` of `Sum` by a factor `f` and returning the resulting instance can be declared as follows:

```
function multiply(s: Sum, f): Sum
  ...
end
```

As with assignments to variables of declared class, the expressions assigned to the parameters are checked when calling the function, and the return value is checked when returning a value from the function:

```
multiply("no sum", 3)
→ ExcNotSuchInstance thrown

function getsum(): Sum
  return "also no sum"
end
y=getsum()
→ ExcNotSuchInstance thrown
```

Class Fields

A useful class normally contains fields, i.e. variables which each class instance holds. Fields are declared by simply listing their name in the class body, optionally separated by semicolons. A field must be declared before it can be referenced in a function. When an instance is created, all its fields are initialized to `null`.

Class fields are accessed as follows:

- To access a class field of an instance variable with declared type, append a dot and the field name:

```
s:Sum=...
s.s=0
```

- To access a class field of an expression without declared type, it must be ''casted'' to the desired type before accessing any of its fields. `ExcNotSuchInstance` is thrown if the expression is not an instance of the desired type.

```
sums=[...]
print sums[3].(Sum)s // accessing s requires a cast
```

- Within a class function, class fields are directly accessible, as shown in functions `add()` and `res()` of class `Sum`: `s` can be used like any other variable.

```
InstanceSelector := '.' [ '(' ClassIdentifier ')' ]
  (FieldIdentifier | FunctionIdentifier '(' [ActualParameterList] ')') .
```

Class Functions

Most classes also contain functions. Class functions operate on an instance, i.e. its fields. For instance, the function `add` in class `Sum` adds a value to the field `s` of the instance.

Within a function of class `C`, the instance is accessible via the (predeclared) parameter-like variable `this:C`. Explicitly mentioning `this` to access an instance field may be required if there is a parameter of the same name:

```
class C
  x
  function setx(x)
    this.x=x // assign parameter x to field x
  end
end
```

The rules for calling class functions are the same as those on accessing class fields:

- To call a class function on an instance variable with declared type, append a dot and the function name:

  ```
  s:Sum=...
  s.add(3) // call add on s
  ```

- To call a class function on an expression without declared type, it must be ''casted'' to the desired type before calling any of its functions. `ExcNotSuchInstance` is thrown if the expression is not an instance of the desired type.

  ```
  sums=[...]
  sums[3].(Sum)add(4) // requires a cast to Sum
  ```

- Within a class function, another function of the class can be called directly, without first denoting the instance.

```
class Sum
  ...
  function addtwice(x)
    // same as this.add(x); this.add(x)
    add(x); add(x)
  end
end
```

There are two ways to define a class function: either directly within the class body, or by declaring it as `forward` and then defining it outside the class, by prefixing the function name by the class identifier. The latter may be required when classes and their functions are referencing each other.

```
class C forward // make C known, without any details

class D
  x: C // C can be used, but C.y is not yet visible
  function mult(a) forward
end

class C // define C
  y
end

function D.mult(a) // C is defined, now define D.mult
  return x.y*a
end
```

The next section shows how class functions can be overwritten in subclasses.

```
ClassFunctionDeclaration := function ClassName '.' Identifier
  FunctionBody .
```

Inheritance, Sub- and Superclasses

One of the key properties of classes is their extensibility: new classes can be declared based on existing classes, inheriting their fields and functions. By overriding functions, the behaviour of class instances can also be extended or modified.

To define a new class extending an existing class, append `is` and the existing class name after the new class identifier:

```
class Avg is Sum
  n // element counter to calculate the average
  function add(x) // overrides Sum.add()
    s+=x; n++
  end
  function res() // overrides Sum.res()
    return s/n
  end
  function count()
    return n
  end
end
```

This establishes the following simple class hierarchy:

- `Avg` is a subclass of `Sum` (each class can have many subclasses).
- `Sum` is the superclass of `Avg` (each class except `.Instance` (p. 51) has exactly one superclass).

Since `Avg` extends `Sum`, it inherits its field `s` and its functions `add()` and `res()`. The functions are overwritten to implement averaging behaviour, and the extended class also gets a new function `count()` returning the element count.

The functions of the superclass are always accessible by the builtin parameter `super`. It references the current instance like `this`, but seen an instance of the superclass when determining which function to call. Hence, the overriding functions in `Avg` could also be written as:

```
  function add(x) // overrides Sum.add()
    super.add(x); n++
  end
  function res() // overrides Sum.res()
    return super.res()/n
  end
```

By declaring class functions as `forward` without implementing them, *abstract classes* can be declared. For instance, an interface-like abstract

class `Aggregator` could be the base class of all aggregating classes `Sum` and `Avg`:

```
class Aggregator
  function add(x) forward
  function res() forward
end

class Sum is Aggregator
  ...
```

Instance Creation and Constructors

Defining a class also defines a function of the same name, its creator function. Calling this function has the following effects:

1. A new instance of the class is created.
2. All fields of the class (and its superclasses) are set to `null`.
3. The class constructor function `init()` of the new instance is called, with the parameters that were passed to the creator function.
4. The new instance is returned.

```
// create a new Sum instance and assign it to x
x:Sum=Sum()
print x
→ .Sum(s=null)
```

Defining (overriding) the `init()` function of `Sum`, its field `s` can be properly initialized to zero:

```
class Sum
  s
  function init()
    s=0
  end
  function add(x)
  ...
end
```

```
x:Sum=Sum()
print x
→ .Sum(s=0)
```

The `init()` function can take arbitrary parameters, and they can be different in number and type for each superclass:

```
class Person
  name
  height

  function init(name="unknown",height=180)
    this.name=name; this.height=height
  end
end
print Person()
→ .Person(name=unknown,height=180)
print Person("Lucky Luke")
→ .Person(name=Lucky Luke,height=180)
print Person("Joe",155)
→ .Person(name=Joe,height=155)
```

Note that there are no destructor functions in **m**. Class instances which are no longer needed are automatically deleted by the garbage collector, without explicit cleanup.

```
InstanceCreation := ClassIdentifier '(' [ActualParameterList] ')' .
```

The .Instance Base Class

There is a single builtin class `.Instance` which is the implicit base class of all classes. It is declared as an empty class with empty constructor function:

```
class Instance
  function init() end
end
```

The following two declarations are therefore equivalent

```
class Sum
  ...
end
class Sum is .Instance
  ...
end
```

Even though it generally makes little sense, it is perfectly valid to create an instance of `.Instance`:

```
x:.Instance=.Instance()
```

Instance Function References

An instance function reference is like a function reference (see section 2.8 (p. 36)), but always operates on a given instance defined when obtaining the reference.

Instance function references are most useful to implement callbacks in an object oriented environment, for instance event listeners. Sometimes they are also called ''delegates'' or ''delegate functions'', since the function reference acts like a delegate of the instance passed to another instance.

Consider the following function passing values in array `a` to a consumer function `c`:

```
function consume(a, c)
  for v in a do
    c(v)
  end
end
function out(n)
  print n
end
consume([7,-8,9], &out) // ordinary function reference
→ 7
   -8
   9
s:Sum=Sum()
consume([7,-8,9], s.&add) // instance function reference
print s, s.res()
→ .Sum(s=8), 8
```

The second call to `consume()` calls `s.add(v)` on each call of `c(v)`.

```
InstanceFunctionReference :=
  '.' [ '(' ClassIdentifier ')' ] '&' Identifier .
```

2.12 Source Structure

After introducing all elements of the **m** language, the complete structure of an **m** source can be defined:

```
MSource := { ModuleImport | FunctionDeclaration | ClassDeclaration |
             ClassFunctionDeclaration | StatementList } .
```

The `StatementList`s (there can be several) are the ''main code'' of the script which is executed directly. In a module, this corresponds to the module initialization code which is executed the first time the module occurs in a `use` clause.

3. Interactive Shells

m cannot only execute complete scripts, it can also be used interactively, as a shell. When working in shell mode, there are a few differences to normal **m** scripts:

- **m** statements are executed interactively: **m** code can be entered and is executed immediately. Global variables and functions are preserved between executions.
- The syntax allows some simplifications (see section 3.1 (p. 55)).
- Each time a shell is created, it loads and executes `autoexec.m` before prompting the user. The script is first searched among the ordinary scripts in `system.docdir` (Library, p. 50). If it is not found, the default script in `system.appdir` (Library, p. 49) is executed.

3.1 Simplified Syntax for Interactive Use

Since input capabilities of cellphones are poor, interactive shells support a simplified syntax for function calls, and automatic output of computed expressions:

- A single `Expression` will be executed as `'print' Expression`, unless it is `null`:

```
m>0.85*23.10
→ 19.635
m>use math as m
m>m.sin(m.pi/4)
→ 0.7071067812
```

- A `SimpleFunctionCall` calls a function with only string or number literal parameters, and options defined for the function.

Unquoted words (sequences not containing white space) on the command line which are not keywords (see appendix A.2 (p. 78)) and are not starting with a digit or separator are interpreted as string parameters.

Numbers are interpreted as numeric parameters.

Options for optional parameters (see section 2.8 (p. 32)) can be specified anywhere with a preceding slash. If an equal sign follows, the following word or number is assigned to the corresponding parameter. If no equal sign follows, `true` is assigned to the corresponding parameter.

Commas to separate the parameters are not permitted.

Again, the function result is printed if it is not null:

```
m>date  // maps to date()
→ 2005-02-07 11:03:07
m>dir c:\*.m/r  // maps to dir('c:\\*.m', true)
→ C:\system\apps\mShell\autoexec.m
   C:\documents\mShell\Jukebox.m
```

Simple function calls can only be used to call functions with parameters which are string or number literals.

```
SimpleFunctionCall :=
  [ModulePrefix] Identifier {SimpleParam | SimpleOption} .
SimpleParam :=
  SimpleChar {SimpleChar} | StringLiteral | NumberLiteral .
SimpleOption := '/' (IdentifierChar | Digit) ['=' SimpleParam] .
SimpleChar :=
  (printable ISO-8859-1 char except white space and '/') .
```

3.2 Shell Builtin Functions

`autoexec.m` declares a number of function for interactive use. Most are just wrappers around existing functions, to avoid typing longer names. With these functions, files on the phone can be easily manipulated:

```
// list all JPG files on the current drive
dir \*.jpg/r/l
→ -- 05-08-09  2966 \documents\mShell\GraphTest.jpg
   -- 11:37:02 17909 \Nokia\Images\FE_img\FEscr(0).jpg
   ...
// copy the JPG files in \documents\mShell to drive e:
cp \documents\mShell\*.jpg e:
→ 1
// search for the mShell properties file
dir \*.prp/r
→ \System\Apps\mShell\mShell.prp
// show its contents
type \system\apps\mShell\mShell.prp
→ mfont=LatinPlain12
   outsize=20000
   keep=busy
```

If a customized `autoexec.m` in `system.docdir` is created without incorporating the original script, these function are no longer available.

.cp

- `function cp(src, dst, recursive=false)→ Number`

  `/r:recursive`

Copies a file, files matching a pattern, or an entire directory tree. Wrapper for `files.copy` (Library, p. 29).

.del

- `function del(pattern, recursive=false)→ Number`

  `/r:recursive`

Deletes a file, files matching a pattern, also in complete directory tree. Wrapper for `files.delete` (Library, p. 29).

.dir

- `function dir(pattern="*", recursive=false, long=false, hidden=false, modified=0)→ null`
  `/h:hidden`
  `/l:long`
  `/m:modified`
  `/r:recursive`

List files matching `pattern` on standard output. If `pattern` is a directory, lists all files in it. Options are the following:

- With `/h` (`hidden=true`), also lists hidden files and directories.
- With `/l` (`long=true`), lists files and directories in a long format, including readonly and hidden attributes and modification date (format `YY-MM-DD` or `hh:mm:ss`).
- With `/m=secs` (`modified=secs`), lists only files which were modified within the last `secs` seconds.

.edit

- `function edit(name)→ null`

Loads a file into the builtin editor and shows it. Wrapper for `files.edit` (Library, p. 30).

.exit

- `function exit()→ null`

Exit this shell. This is equivalent to closing it. This function is only available if module `proc` is available.

.md

- `function md(path, all=false)→ Number`
  `/a:all`

Creates a directory or directories. Wrapper for `files.mkdir` (Library, p. 31).

.mv

- `function mv(src, dst, recursive=false)→ Number`
  `/r:recursive`

Moves a file, files matching a pattern, or an entire directory tree. Wrapper for `files.move` (Library, p. 31).

.rd

- `function rd(path, recursive=false)→ Number`
  `/r:recursive`

Removes a directory or an entire directory tree. Wrapper for `files.rmdir` (Library, p. 33).

.ren

- `function ren(old, new)→ Number`

Renames a single file. Wrapper for `files.rename` (Library, p. 32).

.run

- `function run(script, show=false)→ null`
  `/s:show`

Run another **m** script. If `show=true`, the script's console is shown. This function is only available if module `proc` is available.

.send

- `function send(name, subject=null)→ null`

Interactively sends a file over a channel chosen by the user. Wrapper for `files.send` (Library, p. 34).

.type

- `function type(file, utf16=false, tail=false)→ null`
  `/u:utf16`
  `/t:tail`

Writes the contents of `file` to standard output.

If `utf16=true`, assumes the file to be UTF-16 little endian encoded. Otherwise, raw encoding is assumed.

If `tail=true`, only outputs the last 300 bytes. If `tail=n` where `n` is a number, outputs the last `n` bytes.

4. Producing Standalone Applications

Standalone applications for Symbian OS can easily be produced from applications written in **m**, resulting in standard `.sis` files. These can be installed on other devices, without requiring previous installation of **m**.

Since the `.sis` creation process depends on several resources and supports all platforms for which **m** is available, it is currently implemented as a web application, available at www.m-shell.net/Makemsis.aspx.

4.1 Input Files

The input to the conversion consists of several files:

- One **.mex File** of the complete **m** code of your application (`.mex` files are created from within the **m** application). This is the only mandatory file.

  If your application requires other `.mex` files which will be spawned as subprocesses from the main application, they must go into a ''Document Directory Zip File''.

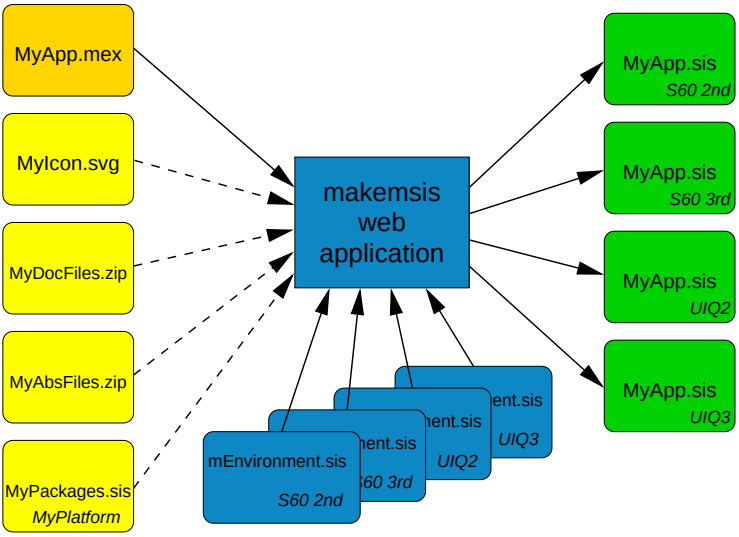
- An optional **Icon File** for the icon of your application. This should be a `.svg` (Scalable Vector Graphics) file, stored using the Tiny SVG profile[1]. Bitmap icons (PNG, GIF, JPEG) can also be submitted, and the web application will attempt to vectorize them, but do not expect brillant results in this case.

  If no icon file is supplied, a default icon will be used.

[1] Tools to create SVG icons include Adobe Illustrator or Inkscape.

- Optional **Embedded SIS Files**. These are `.sis` files contain components your application depends on and may or may not be already installed, for instance your own native modules.
- Optional **Document Directory Zip Files**. These are `.zip` files whose contents will be extracted into the document directory during installation. The (`system.docdir`) constant contains the path to this directory, and it is also the current directory when the application starts.
- Optional **Absolute Path Zip Files**. These are `.zip` files whose contents will be extracted into the root of the installation drive, allowing to put files into arbitrary locations not related to the application directory.

The files you submit will be combined into a single `.sis` file.



4.2 Settings

The following settings configure the characteristics of the generated application and the `.sis` file:

- **Platform**: the platform for which you want the `.sis` file. This is the only mandatory setting.
- **UID**: the UID (UID3) of your application, uniquely identifying it in the Symbian Universe. The UID is a 32-bit number and must be specified as eight hexadecimal digits (e.g. `e7a1f1c0`). If you don't specify a UID, the conversion process will pick one based on the name of your `.mex` file, so the same file name will also produce the same UID each time you create the `.sis` file.

  For any application you plan to distribute, you should obtain an UID from Symbian. Picking one randomly may cause a conflict of your application with other applications. Note also that the valid UID range depends on the platform and signing type.
- **Application Name**: a short name for your application, shown to the user. It must not contain quotes or unprintable characters, and defaults to the `.mex` file name.
- **Caption**: a caption for your application, shown to the user. It must not contain quotes or unprintable characters, and defaults to the Application Name.
- **Description**: a description for your application, presented to the user during installation. It must not contain quotes or unprintable characters, and defaults to the Caption.
- **Version**: the version number of your application, presented to the user during installation. It must be in the format `x.yy`, and defaults to `1.00`.
- **Include m Environment**: whether the produced `.sis` file should embed the **m** runtime environment. You should probably only uncheck this box:

Either during testing, as the resulting `.sis` will be much smaller and the environment is already present on a device where **m** is installed.

Or if you are requesting `Extended` or `Certified` capabilities and are going to sign the resulting `.sis` yourself.

- **Vendor**: the name of the vendor of your application (probably your name), presented to the user during installation if you will properly sign your application. It must not contain quotes or unprintable characters, and defaults to ''m User''.
  This setting is ignored for 2nd edition platforms.

- **Don't sign sis file**: normally, the produced `.sis` file will be self signed and ready to install. Check this if you are requesting `Extended` or `Certified` capabilities and are going to sign the resulting `.sis` yourself.
  This setting is ignored for 2nd edition platforms.

- **Capability Set**: the requested set of capabilities (see also 6.1 (p. 67)). The required capabilities depend on the **m** functions your application is using. If they require `Extended` or `Certified` capabilities, you should not include the (self signed) **m** environment, and must sign the `.sis` file yourself.
  This setting is ignored for 2nd edition platforms.

5. SMS Control

If the **m** Supervisor & Viewer application is installed, the **m** application can be controlled via SMS commands. Commands must be prefixed by the *smskey* configured in the properties (see section A.3 (p. 78)).

The available SMS commands are:

- *smskey* `run` *script args*: starts the **m** application if it is not already running, then starts the script *script* with the arguments *args*. Use function `proc.args` to get the arguments from within the script. If the script is already running, this command is ignored.
- *smskey* `shutdown`: stops all scripts and exits the **m** application. If **m** is not running, this command is ignored.
- *smskey* `start`: starts the **m** application. If **m** is already running, this command is ignored.
- *smskey* `status`: **m** status inquiry, replies with an SMS describing the status of the **m** application and some GSM information. If **m** is running, the reply will look like:

  ```
  m status: running, mem=mem,
    net=mcc,mnc, loc=lac,cid, sig=signal
  ```

  If **m** is not running, the reply will look like:

  ```
  m status: NOT running (category reason),
    net=mcc,mnc, loc=lac,cid, sig=signal
  ```

  The meaning of the fields is the following:

`mem`	bytes of memory used by **m**
`category`	**m** exit category (if panicked)
`reason`	**m** exit reason (if panicked)
`mcc`	GSM mobile country code
`mnc`	GSM mobile network code
`lac`	GSM location area code
`cid`	GSM cell id
`signal`	GSM signal strength

- *`smskey`* `status` *`phone`*: like `status` above, but the response is sent to phone number *`phone`*. *`phone`* must not contain white space.
- *`smskey`* `stop` *`script`*: stops execution of script *`script`*. If *`script`* is not running, this command is ignored.

The following examples require the `smsctrl` property to be enabled, and `smskey` to be set to `mshell`:

1. SMS to start the **m** application:

```
mshell start
```

2. SMS to start the `Supervisor` script, passing it `0769988776` as an argument:

```
mshell run Supervisor 0769988776
```

3. SMS to check the status of the **m** application:

```
mshell status
→ m status: NOT running (E32USER-CBase 71),
   net=228,115, loc=1616,17689, sig=3
```

m is not running because it crashed with a E32USER-CBase 71 panic. The phone is somewhere near cell 17689 in area 1616 of the Swisscom GSM network.

6. m and Symbian Platform Security

With the 3rd edition of its OS, Symbian OS has introduced *platform security*, with mandatory signing of applications and libraries. Platform security constrains runtime environments like **m**, in particular those like **m** which permit software development directly on the device.

6.1 Capabilities

Platform security is implemented by granting applications, libraries and processes created from them *capabilities*. Although somewhat similar to the user permissions described in the previous section, capabilities are completely independent of user permissions. Successfully executing an **m** function requires both: the corresponding user permissions granted by **m** and the capabilities granted by the OS:

- If a function is not permitted by the **m** user, it throws `ExcNotPermitted`.
- If a function is not permitted by platform security, it throws `ErrPermissionDenied`.

Regarding **m**, capabilities currently can be split into four classes:

- The *basic* capabilities, always available in **m**. These are the only capabilities granted when **m** has been ''self signed'', i.e. signed with a certificate generated by the **m** developers (or by yourself, if you have downloaded the corresponding tools). Basic capabilities must be granted by the user when installing the **m** application. Note that the default setting on some Symbian 3rd Edition devices disallows installing self signed applications; the setting can usually

be changed from the program manager application. A few devices completely prevent installation of self signed applications.

A special rule applies to the ''Location'' capability required to access network and cell information: in **m**, it is considered an extended capability, even though it is a basic one on many devices. The self signed **m** package therefore does not include it to remain installable on as many devices as possible.

- The *extended* capabilities, always available on Symbian 2nd Edition phones, and when **m** has been open signed (see next section), or by the Symbian Signed process.
- The *certified* capabilities, always available on Symbian 2nd Edition phones, and when **m** has been signed with a developer certificate (see section 6.3 (p. 69)), or by the Symbian Signed process.
- The *approved* capabilities which are only granted by the platform producer or the phone manufacturer. **m** on Symbian 3rd Edition phones currently does not support any of the functions requiring approved capabilities.

The **View→About** dialog and the `system.caps` (Library, p. 50) constant indicate the capabilities granted to the **m** process:

`system.caps`	Granted capabilities
`basic`	Only basic.
`extended`	Basic and extended.
`certified`	Basic, extended and certified.
`all`	Basic, extended, certified and approved. Currently only available on Symbian 2nd Edition phones.

As the platform security framework is not stable yet, it can be changed anytime by Symbian. Please refer to the official symbian documentation for up to date information. For a list of capabilities, see e.g. forum.nokia.com/main/platforms/s60/capability_descriptions.html.

6.2 Open Signing Online

If you want to use **m** with extended capabilities on a single Symbian 3rd Edition phone, the easiest way is to have it signed online. The online

signed install package has the following restrictions:

- It is bound to a single IMEI, i.e. a specific device.
- Its validity is currently limited to 36 months.

To online sign **m**, follow these steps:

1. Get the IMEI of the device you want to install **m** on. Within **m**, you can obtain it from `gsm.imei` (Library, p. 197). Or dial `*#06#` on the phone.
2. Locate the unsigned online signable version of the **m** installation packages for your device, e.g. `mEnvironment-S60-3rd-OS.sis` and `mShell-S60-3rd-OS.sis`.
3. Go to Symbian Signed, enter your IMEI, the `mEnvironment` installation package to upload, your e-mail address, and select all capabilities. The online signable version of **m** requires at least the following 13 capabilities: `LocalServices`, `Location`, `NetworkServices`, `PowerMgmt`, `ProtServ`, `ReadDeviceData`, `ReadUserData`, `SurroundingsDD`, `SwEvent`, `TrustedUI`, `UserEnvironment`, `WriteDeviceData`, `WriteUserData`.

   Follow the online instructions. Within a few seconds, you should be able to download the signed installable package.
4. Repeat the previous step for the `mShell` installation package.
5. Remove any self signed or DevCert signed version of **m** from the device. The online signed version has different UIDs (`0xe7e0cab8` and `0xe7e0cab7`, from the UID test range), and cannot replace production UID versions or coexist with them.

6.3 Open Signing with a DevCert

If you want to use **m** with *certified* capabilities on Symbian 3rd Edition phones, the only way is currently to obtain an ACS publisher ID certificate from a certificate issuer and your own developer certificate (''DevCert'') from Symbian Signed. This allows you to sign the `.sis` files containing

the binaries with certified capabilities. A developer certificate has the following restrictions:

- It is bound to a set of specific IMEIs (currently up to 1000), i.e. specific devices, specified when obtaining the certificate. This means that the packages are not installable on other devices.
- Its validity is currently limited to 36 months.

6.3.1 Obtaining a DevCert

Step by step, a DevCert can be obtained as follows:

First, register yourself with an ACS publisher ID, and install the required software:

1. Go to www.trustcenter.de/order/publisherid/dev and order your ACS publisher ID certificate. The certificate is not free, it has to be regularly renewed, and obtaining it requires verification of your (our your company's) identity by the issuer.
2. Go to www.symbiansigned.com and register yourself, with your certified ID.
3. Download the `DevCertRequest` application from Symbian Signed, and install it on a PC running Windows.

Then, for each DevCert, execute the following steps:

1. Make sure you know the IMEI of the phones you want to create the certificate for.
2. Run the `DevCertRequest` application. The result will be two files, a certificate request file (`.csr` suffix) and a private key file (`.key` suffix), possibly encripted by a password you have chosen.

   The application will tell you quite precisely what it needs. When asked for the application capabilities, select them all. The DevCert version of **m** requires at least the following 17 capabilities: `CommDD`, `DiskAdmin`, `LocalServices`, `Location`, `MultimediaDD`, `NetworkControl`, `NetworkServices`,

PowerMgmt, ProtServ, ReadDeviceData, ReadUserData, SurroundingsDD, SwEvent, TrustedUI, UserEnvironment, WriteDeviceData, WriteUserData.

3. Log in to www.symbiansigned.com, select ''My Symbian Signed'' and request a developer certificate. Upload the certificate request generated before. After a few seconds, the actual certificate should be ready for download from ''My DevCerts''. Download the file, giving it a `.cer` or `.cert` suffix.

6.3.2 Signing m with the DevCert

To sign **m** with a developer certificate, you need the following:

1. The unsigned DevCert version of the **m** installation packages for your device, e.g. `mEnvironment-S60-3rd-DC.sis` and `mShell-S60-3rd-DC.sis`.
2. The `signsis.exe` application. As this 1.2 MB application cannot be distributed separately, you must download and install the entire Symbian 3rd Edition C++ SDK for (e.g. from developer.nokia.com) to get it.
3. Your developer certificate file, e.g. `MyDevCert.cer`.
4. Your private key file to autenticate the developer certificate, e.g. `MyDevCert.key`, with password.

Then sign the unsigned installation packages by running the following commands from the Windows command prompt[1]:

```
signsis -s mEnvironment-S60-3rd-DC.sis
  mEnvironment-S60-3rd-MyDC.sis
  MyDevCert.cer MyDevCert.key  password
signsis -s mShell-S60-3rd-DC.sis mShell-S60-3rd-MyDC.sis
  MyDevCert.cer MyDevCert.key  password
```

`mEnvironment-S60-3rd-MyDC.sis` and `mShell-S60-3rd-MyDC.sis` should now be installable on the phones specified in the developer certificate.

[1] `signsis` will run without problems under Linux using `wine`.

A. Appendix

A.1 Exception Tags

This section lists the exceptions tags with their english error message.

Environment Exceptions

Environment exceptions are usually thrown by the underlying operation system, e.g. when trying to access a file which does not exist.

- `ErrAbort`: Operation aborted.
- `ErrAccessDenied`: Access denied.
- `ErrAlreadyExists`: File already exists.
- `ErrArgument`: Invalid function argument.
- `ErrBadHandle`: Object handle is bad.
- `ErrBadName`: Name is bad.
- `ErrCancel`: Operation canceled.
- `ErrCommsBreak`: Break in communications occured.
- `ErrCommsFrame:`: Serial framing error.
- `ErrCommsLineFail:`: Serial line failed.
- `ErrCommsOverrun:`: Serial overrun error.
- `ErrCommsParity:`: Serial parity error.
- `ErrCorrupt`: File or database corrupted.
- `ErrCouldNotConnect:`: Could not connect.
- `ErrCouldNotDisconnect:`: Could not disconnect.
- `ErrDied`: Thread or process died.

- `ErrDirFull`: Directory is full.
- `ErrDisconnected:`: Link is disconnected.
- `ErrDiskFull`: Disk is full.
- `ErrDivideByZero`: Integer division by zero.
- `ErrEof`: Eof reached.
- `ErrExtensionNotSupported`: Extension function is not supported.
- `ErrGeneral`: General problem.
- `ErrHardwareNotAvailable`: Hardware is not available or not enabled.
- `ErrInUse`: File or device is in use.
- `ErrLocked`: Object locked.
- `ErrNoMemory`: Out of memory. *This exception cannot be catched.*
- `ErrNotFound`: File or item not found.
- `ErrNotReady`: Device is not ready.
- `ErrNotSupported`: Operation not supported.
- `ErrOverflow`: Numeric overflow.
- `ErrPathNotFound`: Path not found.
- `ErrPermissionDenied`: Permission denied by platform security.
- `ErrServerTerminated`: Server has terminated.
- `ErrServerBusy`: Server is busy.
- `ErrSessionClosed`: Server session has been closed.
- `ErrTimedOut:`: Operation timed out.
- `ErrTooBig:`: Value or array too big.
- `ErrTotalLossOfPrecision`: Total loss of precision.
- `ErrUnderflow`: Numeric underflow.
- `ErrWrite`: Write failed.
- `ExcNotPermitted`: Operation not permitted by user.

Programming Exceptions

Programming exceptions are thrown by **m**, and usually caused by an error in your code or an unexpected user input.

- `ExcArrayNotNumber`: Operand is an array, not a number.
- `ExcBooleanNotNumber`: Operand is a boolean, not a number.
- `ExcForwardFunction`: Function is only forward declared.
- `ExcFunctionNotNumber`: Operand is a function, not a number.
- `ExcIndexOutOfRange`: Array index is out of range.
- `ExcInterrupted`: Interrupted function call.
- `ExcInvalidIndexType`: Array index is neither number nor string.
- `ExcInvalidNumber`: Wrong number format.
- `ExcInvalidUTF8`: Invalid UTF-8 character read.
- `ExcModuleBusy`: Module is busy with other function.
- `ExcNativeNotNumber`: Operand is native object, not a number.
- `ExcNoSuchClass`: Class not loaded.
- `ExcNoSuchKey`: No array element for key.
- `ExcNotArray`: Operand is not an array.
- `ExcNotAvailable`: Function or variable is unavailable.
- `ExcNotBoolean`: Operand is not a boolean.
- `ExcNotComparable`: Can only order two numbers or two strings.
- `ExcNotFunction`: Operand is not a function reference.
- `ExcNotNative`: Operand is not a native object.
- `ExcNotNumber`: Operand is not a number.
- `ExcNotString`: Operand is not a string.
- `ExcNotSuchInstance`: Not such instance.
- `ExcNullNotNumber`: Operand is null, not a number.
- `ExcNullNotInstance`: Operand is null, not an instance.
- `ExcStringNotNumber`: Operand is a string, not a number.

- `ExcStringPosOutOfRange`: String position is out of range.
- `ExcTooManyGlobals`: Too many global variables, split into modules.
- `ExcUnknownField`: Unknown field referenced by native function.
- `ExcUnknownModule`: Unknown module referenced by native function.
- `ExcValueOutOfRange`: Value or parameter is outside valid range.
- `ExcWrongNative`: Operand has wrong native object type.
- `ExcWrongParamCount`: Too many or too few function parameters.

Internal Error Exceptions

Internal error exceptions are thrown by **m** when it detects an internal inconsistency. These exceptions cannot be catched, and are most likely caused by a bug in **m** or in a native module.

- `ErrDisabledFunction`: Internal error: interpreting disabled function.
- `ErrDuplicateModule`: Internal error: duplicate module.
- `ErrDuplicateNative`: Internal error: duplicate native function.
- `ErrEndOfCode`: Internal error: falling through end of code.
- `ErrInvalidDll`: Internal error: DLL did not return module.
- `ErrInvalidFrame`: Internal error: invalid stack frame contents.
- `ErrInvalidFunctionIndex`: Loader error: invalid function index.
- `ErrInvalidInstruction`: Internal error: invalid instruction.
- `ErrInvalidModuleIndex`: Loader error: invalid module index.
- `ErrInvalidStack`: Internal error: invalid stack.
- `ErrInvalidVariableIndex`: Loader error: invalid variable index.
- `ErrMissingDll`: Internal error: module DLL is missing.
- `ErrNativeFunction`: Internal error: interpreting native function.
- `ErrNoCode`: Internal error: interpreting without code.

- `ErrNoNativeFunction`: Internal error: no native function to add option to.
- `ErrNotInstance`: Internal error: using non-instance as instance
- `ErrRTVersionMismatch`: Internal error: runtime version mismatch. *Get an up to date version of the runtime or native module.*
- `ErrStringExtension`: Internal error: string extension.

A.2 Reserved words

In the **m** language, keywords, like identifiers, are case sensitive. The following keywords are reserved and cannot be used as identifiers:

and	const	forward	or	true
as	do	function	return	try
break	else	if	shl	until
by	elsif	in	shr	use
case	end	is	then	while
catch	false	not	throw	
class	for	null	to	

A.3 Properties (.prp) File

Global behaviour of the **m** application is configured in the **m** properties. Selecting **View→Properties** opens a dialog to edit the properties.

The properties are stored in an ASCII text file `system.appdir+"mShell.prp"` containing key-value pairs. Each pair is on a single line, the key and the value separated by an equal (=) character.

The following keys are recognized by **m**:

- `autogo=path1,path2,...`
  A comma separated list of scripts or executables to run when starting **m**. In conjunction with `onboot`, these are run when the phone is switched on.

- `bgcolor=black|white|red|green|blue|yellow|cyan|magenta|#rrggbb`
  The background color of console and editor. `#rrggbb` is a HTML-like hexadecimal notation, e.g. `#ff00ff` for magenta.

- `encodings=bom|utf-8|utf-16le|utf-16be|8-bit`
  The encoding to use for **m** source files and files loaded into and saved from the **m** editor. This setting does *not* change the behaviour of the I/O streams of module `io` (Library, p. 36).
  If set to `bom`, files read are expected to carry an initial Byte Order Mark (BOM, character `0xfeff`) determining their encoding; files

without BOM are treated as sequences of 8-bit characters. In this mode, files are saved in UTF-8 with initial BOM.
If set to `utf-8`, files are read and saved in UTF-8. No BOM is expected or written.
If set to `utf-16le`, files are read and saved in UTF-16 Little Endian. No BOM is expected or written.
If set to `utf-16be`, files are read and saved in UTF-16 Big Endian. No BOM is expected or written.
If set to `8-bit`, files are read and saved considering only the lower eight bits of all characters. No BOM is expected or written.

- `fgcolor=black|white|red|green|blue|yellow|cyan|magenta|#rrggbb`
  The foreground (text) color of console and editor.

- `keep=true|yes|y|1 | false|no|n|0 | busy`
  If set to `true`, `yes`, `y` or `1`, the **m** application cannot be exited automatically by the system, e.g. if it is running low on memory, or if **m** is to be removed because it is updated by a new installation.
  If set to `busy`, exiting is prevented if there are processes running or waiting for input.
  For all other values, **m** behaves like any other ''well behaving'' application, i.e. it can be exited at any time if the operating system requests it.

- `mfont=typeface,points,bold,italic`
  The font to use in the **m** console and editor. A leading star (`*`) on the typeface is ignored. `points` (integer), `bold` (boolean) and `italic` (boolean) are optional. See also `ui.mfont` (Library, p. 92).

- `onboot=true|yes|y|1 | false|no|n|0 | once | restart`
  If set to `true`, `yes`, `y` or `1`, the **m** application will be started automatically when the phone is booted up, i.e. switched on.
  If set to `once`, **m** is only started at the next bootup, as the entry is automatically set to `n` afterwards. This is the recommended setting for disaster prevention during script testing.
  If set to `restart`, **m** is started automatically when the phone is booted up, and restarted each time about 20 seconds after it exits (orderly or because of a crash).

This feature requires the **m** Supervisor & Viewer application to be installed.

- `outsize=charcount`
  The maximum number of characters in the console output, before truncating at the beginning. Truncation happens in chunks of about 500 characters. Set to 0 for an unlimited output size. Handling large output output sizes slows **m** down.

- `perms=permissions`
  The permission bits, defining the permissions granted to **m** scripts. See section A.4 (p. 82).

- `smsctrl=true|yes|y|1 | false|no|n|0`
  If set to `true`, `yes`, `y` or `1`, the **m** application can be controlled via SMS commands, even if it is not running. See chapter 5 (p. 65).
  This feature requires the **m** Supervisor & Viewer application to be installed.

- `smskey=keyword`
  Any SMS containing *`keyword`* as the first characters (ignoring case) is considered a command and sent to the **m** application.

- `smsnr=suffix`
  The last digits of the sender phone number which can control the **m** application via SMS. If empty, anybody knowing `smskey` can control **m**.

- *`_filename`*`=pos`
  The last position of the cursor when editing file *filename*.

All other keys are silently ignored. This can be used to disable entries by just putting e.g. a hash mark in front of them.

A sample properties file might look as follows:

```
autogo=c:\documents\mShell\TrackMe.m,e:\PhoneMonitor.mex
keep=busy
mfont=Monospace,14,false,false
onboot=once
fgcolor=#008000
bgcolor=white
outsize=10000
encoding=utf-8
perms=159
smsctrl=yes
smsnr=4561234
smskey=mshell
_C:\documents\mShell\SmsService.m=133
```

A.4 User Permissions

Permission for certain operations can be granted and denied by the user. Any operation with insufficient permissions will throw `ExcNotPermitted`. Selecting **View→Permissions** opens a dialog to edit the permissions.

The individual permissions are:

Name	Bit	Meaning
`ReadDoc`	1	Read access to files in `system.docdir` and its subdirectories.
`WriteDoc`	2	Write access to files in `system.docdir` and its subdirectories.
`ReadApp`	4	Read access to other application's data.
`WriteApp`	8	Write access to other application's data.
`FreeComm`	16	Access to free communication resources (receiving messages, Bluetooth).
`ReadAll`	32	Read access to all files.
`WriteAll`	64	Write access to all files. **Granting write access to all files also allows modifying the permissions.**
`CostComm`	128	Access to chargeable communication resources (sending messages, TCP/IP).

Thus, if a function requires `Read(file)`, then

- If `file` denotes a file or directory in `system.docdir` or one of its subdirectories, the `ReadDoc` permission must be granted for the function to succeed.
- If `file` denotes a file or directory outside `system.docdir` or one of its subdirectories, the `ReadAll` permission must be granted for the function to succeed.

Likewise, if a function requires `Write(file)`, the `WriteDoc` or `WriteAll` permissions must be granted.

Index