<

1. The Umka Language Specification

1.1. Introduction

Umka is a statically typed embeddable scripting language. It combines the simplicity and flexibility needed for scripting with a compile-time protection against type errors.

Language grammar in this manual is specified in the EBNF notation.

1.2. Lexical elements

A program consists of keywords, identifiers, numbers, character literals, string literals, operators, punctuation and comments.

1.2.1. Keywords

Keywords have special meaning and cannot be used in any other role. Umka has the following keywords:

```
break case const continue default else for fn import
interface if in return str struct switch type var weak
```

1.2.2. Identifiers

Identifiers denote constants, types, variables and functions.

Syntax:

```
ident  = (letter | "_") {letter | "_" | digit}.
letter = "A".."Z" | "a".."z".
digit  = "0".."9".
```

Examples:

```
i
myName
printf
Point_1
```

1.2.3. Numbers

Numbers in Umka can be integer (decimal or hexadecimal) or real.

Syntax:

```
intNumber   = decNumber | hexHumber.
decNumber   = digit {digit}.
hexNumber   = "0" ("X | "x") hexDigit {hexDigit}.
realNumber  = decNumber ["." decNumber] [("E" | "e") decNumber].
hexDigit    = digit | "A".."F" | "a".."f".
```

Examples:


```
16
0xA6
0xffff
3.14
2.6e-5
7e8
```

1.2.4. Character literals

1.2.6. Operators and punctuation

Umka has a set of arithmetical, bitwise, logical, relation operators, as well as a set of punctuation characters:

```
+    -    *    /    %    &    |    ~    <<    >>
+=   -=   *=   /=   %=   &=   |=   ~=   <<=   >>=
&&   ||   !    ++   --
==   <    >    !=   <=   >=
=    :=   (    )    [    ]    {    }
^    ;    :    .
```

Implicit semicolons Umka uses semicolons as statement terminators. To reduce the number of semicolons, an implicit semicolon is automatically inserted immediately after a line's final token if that token is

- An identifier
- A number, character literal or string literal
- `str`, `break`, `continue` or `return`
- `++`, `--`, `)`, `]`, `}`or `^`

A semicolon may be omitted before a closing `)` or `}`.

1.2.7. Comments

Umka supports line comments starting with `//` and block comments starting with `/*` and ending with `*/`.

Examples:

```
// This is a line comment

/*
This is
a block
comment
*/
```

1.3. Data Types

Umka is a statically typed language. Each variable or constant has a type that is either explicitly specified in the variable declaration or inferred from the type of the right-hand side expression in a variable assignment or constant declaration. After a variable or constant is declared or assigned for the first time, it cannot change its type.

Syntax:

```
type = qualIdent | ptrType | arrayType | dynArrayType | strType |
       structType | interfaceType | fnType.
qualIdent = [ident "."] ident.
```

Any data type except the interface type can have a set of functions attached to it, which are called *methods*. A variable of that type is called the *receiver* with respect to any method which is called on it. Methods are not considered a part of the type declaration.

1.3.1. Simple types

Simple type represent single values.

Ordinal types Umka supports the following ordinal types:

- Signed integer types `int8`, `int16` , `int32`, `int`
- Unsigned integer types `uint8`, `uint16` , `uint32`, `uint`
- Boolean type `bool`
- Character type `char`

The `int` and `uint` are the recommended 64-bit integer types. Other integer types are for compatibility with an external environment.

The two possible values of the Boolean type are `true` and `false`.

Real types Umka supports the `real32` and `real` floating-point types. The `real` is the recommended 64-bit real type. The `real32` type is for compatibility with an external environment.

Pointer types A variable that stores a memory address of another variable has a pointer type. The type of that another variable is called the *base type* of the pointer type. The pointer type is specified by a `^` followed by the base type specification. If the base type is unknown, it should be specified as `void`.

Umka performs automatic memory management using reference counting. All pointers are reference-counted by default. However, in processing data structures with cyclic references (like doubly-linked lists), reference counting is unable to deallocate memory properly. In this case, one of the pointers that constitute the cycle can be declared `weak` . A weak pointer is not reference-counted and its existence does not prevent the pointed-to variable from being deallocated. A weak pointer cannot be dereferenced. Instead, it should be first converted to a conventional (or strong) pointer of the same base type. If the weak pointer does not point to a valid dynamically allocated memory block, the result of this conversion is `null`.

Syntax:

```
ptrType = ["weak"] "^" type.
```

Examples:

```
^int16
^void
weak ^Vec
```

An uninitialized pointer has the value `null`.

Function types A constant or variable that stores an entry point of a function has a function type. A function is characterized by its *signature* that consists of a set of parameter names, types and optional default values, and an optional set of returned value types.

Syntax:

```
fnType    = "fn" signature.
signature = "(" [typedIdentList ["=" expr] {"," typedIdentList ["=" expr]}] ")"
            [":" (type | "(" type {"," type} ")")].
```

Examples:

```
fn (code: int)
fn (p: int, x, y: real): (bool, real)
```

1.3.2. Structured types

Structured types represents sequences of values.

Array types An array is a numbered sequence of items of a single type. The number of items is called the *length* of the array and should be a non-negative integer constant expression.

Syntax:

```
arrayType = "[" expr "]" type.
```

Examples:

```
[3]real
[len1 + len2][10]Vec
```

Array assignment copies the contents of the array.

Dynamic array types A dynamic array is a numbered sequence of items of a single type. Its length is not fixed at compile time and can change at run time due to appending or deleting of items.

Syntax:

```
dynArrayType = "[" "]" type.
```

Examples:

```
[]real
[][3]uint8
```

Dynamic arrays require initialization by assignment from a static array or from a dynamic array literal, or by explicitly calling `make()`.

Dynamic array assignment copies the pointer, but not the contents of the array.

String type A string is a sequence of ASCII characters. Its length is not fixed at compile time and can change at run time due to concatenation by using the `+` operator.

Syntax:

```
strType = "str"
```

String assignment copies the contents of the string.

Structure types A structure is a sequence of named items, called *fields*, that possibly have different types. Each field should have a unique name.

Syntax:

```
structType = "struct" "{" {typedIdentList ";"} "}".
```

Examples:

```
struct {x, y, z: int}

struct {
    q: [4]real
    normalized: bool
}
```

Structure assignment copies the contents of the structure.

Interface types An interface is a sequence of method names and signatures. Each method should have a unique name. It is allowed to specify either a method name and its signature, or a name of some other interface type whose methods are to be included to the new interface type.

Syntax:

```
interfaceType = "interface" "{" {(ident signature | qualIdent) ";"} "}".
```

Examples:

```
interface {
    print(): int
    Saveable
}
```

The interface does not implement these methods itself. Instead, any type that implements all the methods declared in the interface can be converted to that interface. Since different types can be converted to the same interface, interfaces provide polymorphic behavior in Umka.

If a pointer of type `^T` is converted to the interface, the interface stores that pointer. If a non-pointer variable of type `T` is converted to the interface, the variable is first copied to a new memory location, and the interface stores the pointer to that location. In any case, the interface can be converted back to the type `^T`. If a conversion of the interface to some type `^S` not equivalent to `^T` is attempted, the result is `null`.

Any type can be converted to the empty interface `interface{}`.

Interface assignment copies the pointer, but not the contents of the variable that has been converted to the interface.

Fiber type The `fiber` type represents a fiber context to use in multitasking.

1.3.3. Type compatibility

Two types are *equivalent* if

- They are the same type
- They are pointer types and have equivalent base types and both are either strong or weak
- They are array types and have equal length and equivalent item types
- They are dynamic array types and have equivalent item types
- They are structure or interface types and have equal number of fields, equal field names and equivalent field types
- They are function types, which are both either methods or non-methods, and have equal number of parameters, equal parameter names, equivalent parameter types, equal parameter default values (excluding the receiver parameter) and equivalent result type.

Two types are *compatible* if

- They are equivalent
- They are integer types
- They are real types
- They are pointer types and the left-hand side pointer type has the `void` base type or the right-hand side is `null`
- They are pointer types and have equivalent base types and the left-hand side pointer type is weak and the right-hand side pointer type is strong

1.3.4. Type conversions

Implicit conversions If a value `s` of type `S` is given where a value `t` of some other type `T` is expected, the `s` is implicitly converted to `t` if

- `S` and `T` are compatible
- `S` is an integer type and `T` is a real type
- `S` is `char` and `T` is `str`
- `S` is a `[]char` and `T` is `str` and the last item of `S` is the null character
- `S` is `str` and `T` is `[]char`

- `S` is an array type and `T` is a dynamic array type and the item types of `S` and `T` are equivalent
- `S` is a dynamic array type and `T` is an array type and the item types of `S` and `T` are equivalent and `len(s) <= len(t)`
- `S` is an interface type and `T` is a type that implements all the methods of `S`
- `S` is a pointer type and `T` is an interface type
- `S` is a weak pointer type and `T` is a strong pointer type and the base types of `S` and `T` are equivalent

Explicit conversions If a value `s` of type `S` is given where a value `t` of some other type `T` is expected, the `s` can be explicitly converted (cast) to `t` if

- `S` can be implicitly converted to `T`
- `S` and `T` are ordinal types
- `S` and `T` are pointer types and either `T` is `^void` or `sizeof(s^) >= sizeof(t^)` and both `S` and `T` don't contain pointers

1.4. Declarations

All types, constants, variables and functions should be declared before the first use. The only exception is that a pointer base type `T` may be declared after using the pointer type `^T` but before the end of the same `type` declaration list. No identifier may be declared twice in the same block.

Syntax:

```
decls = decl {";" decl}.
decl  = typeDecl | constDecl | varDecl | fnDecl.
```

1.4.1. Scopes

- Universe scope. All built-in identifiers implicitly belong to the universe scope. They are valid throughout the program.
- Module scope. Identifiers declared outside any function body belong to the module scope. They are valid from the point of declaration to the end of the module. If the declared identifiers are marked as exported and the module in which they are declared is imported from some other module, the identifiers are also valid in that other module, if qualified with the module name. Non-constant expressions are not allowed in declarations belonging to the module scope.
- Block scope. Identifiers declared within a function body belong to the block scope. They are valid from the point of declaration to the end of the block, including all the nested blocks, except that a variable belonging to the block scope is not valid within the nested function blocks. A declaration can be shadowed by a declaration of the same name in a nested block.

Declaration export syntax:

```
exportMark = ["*"].
```

1.4.2. Type declarations

Syntax:

```
typeDecl     = "type" (typeDeclItem | "(" {typeDeclItem ";"} ")").
typeDeclItem = ident exportMark "=" type.
```

Examples:

```
type Vec = struct {x, y, z: int}
type (
    i8 = int8
    u8 = uint8
)
```

Built-in types `void`

```
int8 int16 int32 int
uint8 uint16 uint32 uint
bool
char
real32 real
fiber
```

1.4.3. Constant declarations

A constant declaration generally requires an explicit constant value. However, if a constant declaration is preceded by another integer constant declaration within the same declaration list, its explicit value can be omitted. In this case, the value will be set to the preceding constant value incremented by 1.

Syntax:

```
constDecl     = "const" (constDeclItem | "(" {constDeclItem ";"} ")").
constDeclItem = ident exportMark ["=" expr].
```

Examples:

```
const a = 3
const b* = 2.38
const (
    c = sin(b) / 5
    d = "Hello" + " World"
)
const (
    a = 5
    b       // 6
    c       // 7
```

```
    d = 19
    e       // 20
)
```

Built-in constants `true false null`

1.4.4. Variable declarations

Syntax:

```
varDecl = fullVarDecl | shortVarDecl.
```

Full variable declarations Full variable declarations require explicit type specification and have an optional list of initializer expressions. If the initializer expressions are omitted, the variables are initialized with zeros.

Syntax:

```
fullVarDecl    = "var" (varDeclItem | "(" {varDeclItem ";"} ")").
varDeclItem    = typedIdentList ["=" exprList].
typedIdentList = identList ":" type.
identList      = ident exportMark {"," ident exportMark}.
```

Examples:

```
var e1, e2: int = 2, 3
var f: String = d + "!"
var (
    g: Arr
    h: []real
)
```

Short variable declarations Short variable declarations are always combined with the assignment statements. They don't require explicit type specifications, since they infer the variable types from the types of the right-hand side expressions.

Syntax:

```
shortVarDecl        = declAssignmentStmt.
singleDeclAssgnStmt = ident ":=" expr.
listDeclAssgnStmt   = identList ":=" exprList.
declAssignmentStmt  = singleDeclAssgnStmt | listDeclAssgnStmt.
```

Examples:

```
a := 5
s, val := "Hello", sin(0.1 * a)
```

1.4.5. Function and method declarations

A function or method declaration can be either a complete definition that includes the function block, or a *prototype* declaration if the function block is omitted. A prototype should be resolved somewhere below in the same module by duplicating the declaration, now with the function block. If a prototype is not resolved, it is considered an external C/C++ function. If such a function has not been registered via the Umka API, it is searched in the shared library (Umka implementation file) `mod.umi`, where `mod` is the current module name. If the library does not exist or contains no such function, an error is triggered.

Function and method declarations are only allowed in the module scope. In the block scope, functions should be declared as constants or variables of a function type. Methods cannot be declared in the block scope.

The method receiver type should be a pointer to any declared type, except interface types. Methods should be declared in the same module as the method receiver type name.

Syntax:

```
fnDecl       = "fn" [rcvSignature] ident exportMark signature (fnBlock | fnPrototype).
rcvSignature = "(" ident ":" type ")".
fnBlock      = block.
fnPrototype  = .
```

Examples:

```
fn tan(x: real): real {return sin(x) / cos(x)}

fn getValue(): (int, bool) {
    return 42, true
}

fn (a: ^Arr) print(): int {
    printf("Arr: %s\n", repr(a^))
    return 0
}
```

Built-in functions Built-in functions don't necessarily adhere to the general rules for functions.

Input/output functions

```
fn printf(format: str, a1: T1, a2: T2...): int
fn fprintf(f: ^std.File, format: str, a1: T1, a2: T2...): int
fn sprintf(buf, format: str, a1: T1, a2: T2...): int
```

Write `a1`, `a2`… to the console, or to the file `f`, or to the string `buf`, according to the `format` string. `T1`, `T2`… should be ordinal, or real, or string types. Additional run-time type checking is performed to match the `format` string. Return the number of bytes written.

```
fn scanf(format: str, a1: ^T1, a2: ^T2...): int
fn fscanf(f: ^std.File, format: str, a1: ^T1, a2: ^T2...): int
fn sscanf(buf, format: str, a1: ^T1, a2: ^T2...): int
```

Read `a1`, `a2`… from the console, or from the file `f`, or from the string `buf`, according to the `format` string. `T1`, `T2`… should be ordinal, or real, or string types. Additional run-time type checking is performed to match the `format` string. Return the number of values read.

Mathematical functions

```
fn round(x: real): int          // Rounding to the nearest integer
fn trunc(x: real): int          // Rounding towards zero
fn fabs (x: real): real         // Absolute value
fn sqrt (x: real): real         // Square root
fn sin  (x: real): real         // Sine
fn cos  (x: real): real         // Cosine
fn atan (x: real): real         // Arctangent
fn atan2(y, x: real): real      // Arctangent of y / x
fn exp  (x: real): real         // Exponent
fn log  (x: real): real         // Natural logarithm
```

Memory management functions `fn new(T): ^T`

Allocates memory for a variable of type `T` , initializes it with zeros and returns a pointer to it.

`fn make([]T, length: int): []T`

Constructs a dynamic array of `length` items of type `T`.

`fn append(a: []T, x: (^T | []T)): []T`

Constructs a copy of the dynamic array `a` and appends `x` to it. The `x` can be either a new item of the same type as the item type `T` of `a`, or another dynamic array of the same item type `T`.

`fn delete(a: []T, index: int): []T`

Constructs a copy of the dynamic array `a` and deletes the item at position `index` from it.

`fn slice(a: ([]T | str), startIndex [, endIndex]: int): ([]T | str)`

Constructs a copy of the part of the dynamic array or string `a` starting at `startIndex` and ending before `endIndex`. If `endIndex` is omitted, it is treated as equal to `len(a)`. If `endIndex` is negative, `len(a)` is implicitly added to it.

`fn len(a: ([...]T | []T | str)): int`

Returns the length of `a`, where `a` can be an array, a dynamic array or a string.

`fn sizeof(a: T): int`

Returns the size of `a` in bytes.

`fn sizeofself(a: interface{...}): int`

Returns the size in bytes of the variable that has been converted to the interface `a`.

`fn selfhasptr(a: interface{...}): bool`

Checks whether the type of the variable that has been converted to the interface`a` is a pointer, string, dynamic array, interfaces or fibers, or has one of these types as its item type or field type.

Multitasking functions

```
type FiberFunc = fn(parent: ^fiber, anyParam: ^T)
fn fiberspawn(childFunc: FiberFunc, anyParam: ^T): ^fiber
```

Creates a new fiber and assigns `childFunc` as the fiber function. `anyParam` is a pointer to any data buffer that will be passed to `childFunc` .

`fn fibercall(child: ^fiber)`

Resumes the execution of the `child` fiber.

`fn fiberalive(child: ^fiber)`

Checks whether the `child` fiber function has not yet returned.

Miscellaneous functions `fn repr(a: T): str`

Returns the string representation of `a`.

`fn error(msg: str)`

Triggers a run-time error with the message `msg`.

1.5. Expressions

1.5.1. Primary expressions

A primary expression is either an identifier optionally qualified with a module name, or a built-in function call.

Syntax:

```
primary     = qualIdent | builtinCall.
qualIdent   = [ident "."] ident.
builtinCall = qualIdent "(" [expr {"," expr}] ")".
```

Examples:

```
i
std.File
atan2(y0, x0)
printf("a = %f\n", a)
```

1.5.2. Type casts

A type cast explicitly converts the type of an expression to another type. It consists of the target type followed by the expression in parentheses.

Syntax:

```
typeCast = type "(" expr ")".
```

Examples:

```
int('b')
^[3]real(iface)
```

1.5.3. Composite literals

A composite literal constructs a value for an array, dynamic array, structure or function and creates a new value each time it is evaluated. It consists of the type followed by the brace-bound list of item values (for arrays or dynamic arrays), list of field values with optional field names (for structures) or the function body (for functions).

The number of array item values or structure field values and the names of structure fields should match the literal type. The types of array item values, dynamic array item values or structure field values should be compatible to those of the literal type.

Syntax:

```
compositeLiteral = arrayLiteral | dynArrayLiteral | structLiteral | fnLiteral.
arrayLiteral     = "{" [expr {"," expr}] "}".
dynArrayLiteral  = arrayLiteral.
structLiteral    = "{" [[ident ":"] expr {"," [ident ":"] expr}] "}".
fnLiteral        = fnBlock.
```

Examples:

```
[3]real{2.3, -4.1 / 2, b}
[]interface{}{7.2, "Hello", [2]int{3, 5}}
Vec{x: 2, y: 8}
fn (x: int) {return 2 * x}
```

1.5.4. Designators and selectors

A number of postfix *selectors* can be applied to a primary expression, a type cast or a composite literal to get a *designator*.

Syntax:

```
designator = (primary | typeCast | compositeLiteral) selectors.
selectors  = {derefSelector | indexSelector | fieldSelector | callSelector}.
```

Dereferencing selector The dereferencing selector `^` accesses the value pointed to by a pointer. It can be applied to a pointer with a non-`void` base type.

Syntax:

```
derefSelector = "^".
```

Examples:

```
p^
^int(a)^
```

Index selector The index selector `[...]` accesses an array item. It can be applied to an array, a dynamic array, a string or a pointer to one of these types. The index should be an integer expression. Item indexing starts from 0. If the index is out of bounds, an error is triggered.

Syntax:

```
indexSelector = "[" expr "]".
```

Examples:

```
b[5]
longVector[2 * i + 3 * j]
```

Field or method selector The field or method selector `.` accesses a field or method. For accessing a field, it can be applied to a structure that has this field or to a pointer to such a structure. For accessing a method, it can be applied to any declared type `T` or `^T` such that `^T` is compatible with the type that implements this method. If neither field nor method with the specified name is found, an error is triggered.

Syntax:

```
fieldSelector = "." ident.
```

Examples:

```
v.x
data.print
```

Call selector The call selector `(...)` calls a function and accesses its returned value. It can be applied to a value of a function type, including methods. All actual parameters are passed by value. If the number and the types of the actual parameters are not compatible with the function signature, an error is triggered.

Syntax:

```
callSelector = actualParams.
actualParams = "(" [expr {"," expr}] ")".
```

Examples:

```
f(x + 1, y + 1)
stop()
```

1.5.5. Operators

Operators combine expressions into other expressions.

Syntax:

```
expr          = logicalTerm {"||" logicalTerm}.
logicalTerm   = relation {"&&" relation}.
relation      = relationTerm [("==" | "!=" | "<" | "<=" | ">" | ">=") relationTerm].
relationTerm  = term {("+" | "-" | "|" | "~") term}.
term          = factor {("*" | "/" | "%" | "<<" | ">>" | "&") factor}.
factor        = designator | intNumber | realNumber | charLiteral | stringLiteral |
                  ("+" | "-" | "!" | "~") factor | "&" designator | "(" expr ")".
```

Examples:

```
2 + 2
-f(x) * (a[0] + a[1])
"Hello, " + person.name + '\n'
p != null && p[i] > 0
&Vec{2, 5}
```

Unary operators	+	Unary plus	Integers, reals
	-	Unary minus	Integers, reals
	~	Bitwise "not"	Integers
	!	Logical "not"	Booleans
	&	Address	Addressable designators

Binary operators			
	+	Sum	Integers, reals, strings
	-	Difference	Integers, reals
	*	Product	Integers, reals
	/	Quotient	Integers, reals
	%	Remainder	Integers
	&	Bitwise "and"	Integers
	|	Bitwise "or"	Integers
	~	Bitwise "xor"	Integers
	<<	Left shift	Integers
	>>	Right shift	Integers
	&&	Logical "and"	Booleans
	||	Logical "or"	Booleans
	==	"Equal"	Ordinals, reals, pointers, strings
	!=	"Not equal"	Ordinals, reals, pointers, strings
	>	"Greater"	Ordinals, reals, strings
	<	"Less"	Ordinals, reals, strings
	>=	"Greater or equal"	Ordinals, reals, strings
	<=	"Less or equal"	Ordinals, reals, strings

The / operator performs an integer division (with the remainder discarded) if both operands are of integer types, otherwise it performs a real division.

The && and || operators don't evaluate the second operand if the first operand is sufficient to evaluate the result.

Operand type conversions Operand types are implicitly converted in two steps:

- The right operand type is converted to the left operand type
- The left operand type is converted to the right operand type

Operator precedence

```
*   /   %   <<   >>   &          Highest
+   -   |   ~
==  !=  <   <=   >   >=
&&
||                               Lowest
```

1.6. Statements

Statements perform some actions but don't evaluate to any value.

Syntax:

```
stmt       = decl | block | simpleStmt |
             ifStmt | switchStmt | forStmt | breakStmt | continueStmt | returnStmt.
simpleStmt = assignmentStmt | shortAssignmentStmt | incDecStmt | callStmt.
```

A declaration in a block scope is also considered a statement.

1.6.1. Block

A block is a statement list enclosed in braces. A block scope is associated with any block.

Syntax:

```
block    = "{" StmtList "}".
stmtList = stmt {";" stmt}.
```

1.6.2. Assignment

An assignment evaluates the right-hand expression (or expression list) and assigns it to the left-hand side addressable designator (or designator list).

Full assignment Syntax:

```
assignmentStmt  = singleAssgnStmt | listAssgnStmt.
singleAssgnStmt = designator "=" expr.
listAssgnStmt   = designatorList "=" exprList.
```

```
designatorList  = designator {"," designator}.
exprList        = expr {"," expr}.
```

Examples:

```
b = 2
person.msg = "Hello, " + person.name + '\n'
x[i], x[i + 1] = x[i + 1], x[i]
```

Short assignment A short assignment combines one of the operators `+`, `-`, `*`, `/`, `%`, `&`, `|`, `~` with assignment according to the following rule: `a op= b` is equivalent to `a = a op b`.

Syntax:

```
shortAssignmentStmt = designator
                      ("+=" | "-=" | "*=" | "/=" | "%=" | "&=" | "|=" | "~=") expr.
```

Examples:

```
b += 2
mask[i] <<= 3
```

1.6.3. Increment and decrement

Increments or decrements an addressable integer designator by 1.

Syntax:

```
incDecStmt = designator ("++" | "--").
```

Examples:

```
b++
numSteps[i]--
```

1.6.4. Function or method call

Calls a function or method. If the returned value type is non-`void`, the value is discarded.

Syntax:

```
callStmt = designator.
```

Examples:

```
setValue(2, 4)
v[i].print()
```

1.6.5. The `if` statement

Executes a block if a given Boolean expression evaluates to `true`. Optionally, executes another block if the expression evaluates to `false`. An optional short variable declaration may precede the Boolean expression. Its scope encloses the Boolean expression and the two blocks.

Syntax:

```
ifStmt = "if" [shortVarDecl ";"] expr block ["else" (ifStmt | block)].
```

Examples:

```
if a > max {max = a}

if x, ok := getValue(); ok {
    printf("Got " + repr(x) + "\n")
} else {
    printf("Error\n")
}
```

1.6.6. The `switch` statement

Executes one of several statement lists depending on the value of the given ordinal expression. If the expression is equal to any of the constant expressions attached by a `case` label to a statement list, this statement list is executed and the control is transferred past the end of the `switch` statement. If no statement list is selected, the optional `default` statement list is executed. An optional short variable declaration may precede the ordinal expression. Its scope encloses the expression and all the statement lists.

Syntax:

```
switchStmt = "switch" [shortVarDecl ";"] expr "{" {case} [default] "}".
case       = "case" expr {"," expr} ":" stmtList.
default    = "default" ":" stmtList.
```

Examples:

```
switch a {
    case 1, 3, 5, 7: printf(repr(a) + " is odd\n")
    case 2, 4, 6, 8: printf(repr(a) + " is even\n")
    default:         printf("I don't know")
}
```

1.6.7. The `for` statement

Executes a block repeatedly.

Syntax:

```
forStmt = "for" (forHeader | forInHeader) block.
```

The general `for` statement Executes a block while a given Boolean expression evaluates to `true`. The expression is re-evaluated before each iteration. An optional simple statement may follow the Boolean expression. If present, it is executed after executing the block and before re-evaluating the Boolean expression. An optional short variable declaration may precede the Boolean

expression. Its scope encloses the Boolean expression, the simple statement and the block.

Syntax:

```
forHeader = [shortVarDecl ";"] expr [";" simpleStmt].
```

Examples:

```
for true {
    printf("Infinite loop\n")
}

for a > 1 {
    process(&a)
}

for k := 1; k <= 128; k *= 2 {
    printf(repr(k) + '\n')
}
```

The `for...in` statement Iterates through all items of an array, a dynamic array or a string. Before each iteration, the index and the value of the next item are evaluated and assigned to the corresponding variables declared via an implicit short variable declaration in the statement header. The index variable may be omitted.

Syntax:

```
forInHeader = [ident ","] ident "in" expr.
```

Examples:

```
for x in a {
    sum += x
}

for index, item in data {
    if item > maxItem {
        maxItem = item
        maxIndex = index
    }
}
```

1.6.8. The `break` statement

Terminates the execution of the innermost enclosing `for` statement and transfers the control past the end of the `for` statement.

Syntax:

```
breakStmt = "break".
```

Examples:

```
for x in a {
    if fabs(x) > 1e12 {break}
    sum += x
}
```

1.6.9. The `continue` statement

Terminates the execution of the current iteration of the innermost enclosing `for` statement and transfers the control to the point immediately before the end of the `for` statement block.

Syntax:

```
continueStmt = "continue".
```

Examples:

```
for x in a {
    if x < 0 {continue}
    sum += x
}
```

1.6.10. The `return` statement

Terminates the execution of the innermost enclosing function and transfers the control to the calling function. For functions with non-`void` return value types, the `return` keyword should be followed by an expression or expression list whose types are compatible with the return value types declared in the function signature.

For functions with non-`void` return value types, the function block should have at least one `return` statement.

Syntax:

```
returnStmt = "return" [exprList].
```

Examples:

```
fn doNothing() {
    return
}

fn sqr(x: real): real {
    return x * x
}

fn p(a: int): (int, bool) {
    return 2 * a, true
}
```

1.7. Modules

An Umka program consists of one or more source files, or modules. The main module should have the `main()` function with no parameters and no return values, which is the entry point of the program.

Modules can be imported from other modules. In this case, all the identifiers declared as exported in the module scope of the imported module become valid in the importing module, if qualified with the imported module name.

Syntax:

```
program    = module.
module     = [import ";"] decls.
import     = "import" (importItem | "(" {importItem ";"} ")").
importItem = stringLiteral.
```

Examples:

```
import "../import/std.um"
fn main() {
    std.println("Hello, World!")
}
```

1.8. Standard library

The standard library is contained in the `std.um` module.

1.8.1. Input/output

Types `type File* = ^struct {}`

File handle.

Constants
```
const (
    seekBegin* = 0
    seekCur*   = 1
    seekEnd*   = 2
)
```

Codes defining the offset origins in `fseek()`: the beginning of file, the current position, the end of file.

Functions `fn fopen*(name: str, mode: str): File`

Opens the file specified by the `name` in the given `mode` (identical to C): `"r"` to read from a text file, `"rb"` to read from a binary file, `"w"` to write to a text file, `"wb"` to write to a binary file, etc. Returns the file handle.

`fn fclose*(f: File): int`

Closes the file `f`. Returns 0 if successful.

`fn fread*(f: File, buf: interface{}): int`

Reads the `buf` variable from the file `f`. `buf` can be of any type that doesn't contain pointers, strings, dynamic arrays, interfaces or fibers, except for `^[]uint8`. Returns 1 if successful.

`fn fwrite*(f: File, buf: interface{}): int`

Writes the `buf` variable to the file `f`. `buf` can be of any type that doesn't contain pointers, strings, dynamic arrays, interfaces or fibers, except for `^[]uint8`. Returns 1 if successful.

`fn fseek*(f: File, offset, origin: int): int`

Sets the file pointer in the file `f` to the given `offset` from the `origin` , which is either `seekBegin`, or `seekCur`, or `seekEnd`. Returns 0 if successful.

`fn ftell*(f: File): int`

Returns the file pointer position.

`fn remove*(name: str): int`

Removes the file specified by the `name`. Returns 0 if successful.

`fn feof*(f: File): bool`

Returns the end-of-file indicator.

```
fn println*(s: str): int
fn fprintln*(f: File, s: str): int
```

Write the string `s` followed by a newline character to the console or to the file `f`. Return the number of bytes written.

`fn getchar*(): char`

Returns a character read from the console.

1.8.2. Conversions

Functions

```
fn atoi*(s: str): int                   // String to integer
fn atof*(s: str): real                  // String to real
fn itoa*(x: int): str                   // Integer to string
fn ftoa*(x: real, decimals: int): str   // Real to string with `decimals` decimal places
```

1.8.3. Math

Constants `const pi* = 3.14159265358979323846`
`const randMax* = 0x7FFFFFFF`

Functions `fn srand*(seed: int)`

Initializes the pseudo-random number generator with `seed`.

`fn rand*(): int`

Returns an integer pseudo-random number between 0 and `randMax` inclusive.

`fn frand*(): real`

Returns a real pseudo-random number between 0 and 1 inclusive.

1.8.4. Timer

Functions `fn time*(): int`

Returns the number of seconds since 00:00, January 1, 1970 UTC.

`fn clock*(): real`

Returns the number of seconds since the start of the program.

1.8.5. Command line and environment

`fn argc*(): int`

Returns the number of command line parameters.

`fn argv*(i: int): str`

Returns the `i`-th command line parameter, where `i` should be between 0 and `argc() - 1` inclusive.

`fn getenv*(name: str): str`

Returns the environment variable with the specified `name`.

1.9. Embedding API

The Umka interpreter is a shared library that provides the API for embedding into a C/C++ host application.

1.9.1. API types

```
typedef union
{
    int64_t intVal;
    uint64_t uintVal;
    int64_t ptrVal;
    double realVal;
} UmkaStackSlot;
```

Umka fiber stack slot.

```
typedef void (*UmkaExternFunc)(UmkaStackSlot *params, UmkaStackSlot *result);
```

Umka external function pointer. When an external C/C++ function is called from Umka, its parameters are stored in `params` in right-to-left order.

```
enum
{
    UMKA_MSG_LEN = 512
};

typedef struct
{
    char fileName[UMKA_MSG_LEN];
    int line, pos;
    char msg[UMKA_MSG_LEN];
} UmkaError;
```

Umka error description structure.

1.9.2. API functions

```
void UMKA_API *umkaAlloc(void);
```

Allocates memory for the interpreter and returns the interpreter instance handle.

```
bool UMKA_API umkaInit(void *umka, const char *fileName, const char *sourceString,
                       int storageSize, int stackSize, int argc, char **argv);
```

Initializes the interpreter instance. Here, `umka` is the interpreter instance handle, `fileName` is the Umka source file name, `sourceString` is an optional string buffer that contains the program source, `storageSize` is the size, in bytes, of the static storage used for storing string literals and constant composite literals, `stackSize` is the fiber stack size, in slots, `argc` and `argv` represent the standard C/C++ command-line parameter data. If `sourceString` is not `NULL`, the program source is read from this string rather than from a file. A fictitious `fileName` should nevertheless be specified. Returns `true` if the source has been successfully loaded.

```
bool UMKA_API umkaCompile(void *umka);
```

Compiles the Umka program into bytecode. Here, `umka` is the interpreter instance handle. Returns `true` if the compilation is successful and no compile-time errors are detected.

```
bool UMKA_API umkaRun(void *umka);
```

Runs the Umka program previously compiled to bytecode, i. e., calls its `main()` function. Here, `umka` is the interpreter instance handle. Returns `true` if the program execution finishes successfully and no run-time errors are detected.

```
bool UMKA_API umkaCall(void *umka, int entryOffset,
                       int numParamSlots, UmkaStackSlot *params, UmkaStackSlot *result);
```

Calls the specific Umka function. Here, `umka` is the interpreter instance handle, `entryPoint` is the function entry point offset previously obtained by calling `umkaGetFunc()`, `numParamSlots` is the number of Umka fiber stack slots occupied by the actual parameters passed to the function (equal to the number of parameters if no structured parameters are passed by value), `params` is the array of stack slots occupied by the actual parameters, `result` is the pointer to the stack slot to be occupied by the returned value. Returns `true` if the Umka function returns successfully and no run-time errors are detected.

```
void UMKA_API umkaFree(void *umka);
```

Deallocates memory allocated for the interpreter. Here, `umka` is the interpreter instance handle.

```
void UMKA_API umkaGetError(void *umka, UmkaError *err);
```

Gets the last compile-time or run-time error. Here, `umka` is the interpreter instance handle, `err` is the pointer to the error description structure to be filled.

```
void UMKA_API umkaAsm(void *umka, char *buf, int size);
```

Generates the Umka assembly listing for the Umka program previously compiled to bytecode. Here, `umka` is the interpreter instance handle, `buf` is the pointer to the string buffer to be filled, `size` is the buffer size.

```
void UMKA_API umkaAddModule(void *umka, const char *fileName, const char *sourceString);
```

Adds an Umka module contained in the `sourceString`. A fictitious `fileName` should be specified.

```
void UMKA_API umkaAddFunc(void *umka, const char *name, UmkaExternFunc entry);
```

Adds a C/C++ function to the list of external functions that can be called from Umka. Here, `umka` is the interpreter instance handle, `name` is the function name, `entry` is the function pointer.

```
int UMKA_API umkaGetFunc(void *umka, const char *moduleName, const char *funcName);
```

Gets an Umka function that can be called from C/C++ using `umkaCall()`. Here, `umka` is the interpreter instance handle, `moduleName` is the Umka module name, `funcName` is the Umka function name. Returns the function entry point offset.

1.10. Appendix: Language grammar

```
program             = module.
module              = [import ";"] decls.
import              = "import" (importItem | "(" {importItem ";"} ")").
importItem          = stringLiteral.
decls               = decl {";" decl}.
decl                = typeDecl | constDecl | varDecl | fnDecl.
```

```
typeDecl            = "type" (typeDeclItem | "(" {typeDeclItem ";"} ")").
typeDeclItem        = ident exportMark "=" type.
constDecl           = "const" (constDeclItem | "(" {constDeclItem ";"} ")").
constDeclItem       = ident exportMark "=" expr.
varDecl             = fullVarDecl | shortVarDecl.
fullVarDecl         = "var" (varDeclItem | "(" {varDeclItem ";"} ")").
varDeclItem         = typedIdentList "=" exprList.
shortVarDecl        = declAssignmentStmt.
fnDecl              = "fn" [rcvSignature] ident exportMark signature [block].
rcvSignature        = "(" ident ":" type ")".
signature           = "(" [typedIdentList ["=" expr] {"," typedIdentList ["=" expr]}] ")"
                      [":" (type | "(" type {"," type} ")")].
exportMark          = ["*"].
identList           = ident exportMark {"," ident exportMark}.
typedIdentList      = identList ":" type.
type                = qualIdent | ptrType | arrayType | dynArrayType | strType |
                      structType | interfaceType | fnType.
ptrType             = ["weak"] "^" type.
arrayType           = "[" expr "]" type.
dynArrayType        = "[" "]" type.
strType             = "str".
structType          = "struct" "{" {typedIdentList ";"} "}".
interfaceType       = "interface" "{" {(ident signature | qualIdent) ";"} "}".
fnType              = "fn" signature.
block               = "{" StmtList "}".
fnBlock             = block.
fnPrototype         = .
stmtList            = Stmt {";" Stmt}.
stmt                = decl | block | simpleStmt | ifStmt | switchStmt | forStmt |
                      breakStmt | continueStmt | returnStmt.
simpleStmt          = assignmentStmt | shortAssignmentStmt | incDecStmt | callStmt.
singleAssgnStmt     = designator "=" expr.
listAssgnStmt       = designatorList "=" exprList.
assignmentStmt      = singleAssgnStmt | listAssgnStmt.
shortAssignmentStmt = designator
                      ("+=" | "-=" | "*=" | "/=" | "%=" | "&=" | "|=" | "~=") expr.
singleDeclAssgnStmt = ident ":=" expr.
listDeclAssgnStmt   = identList ":=" exprList.
declAssignmentStmt  = singleDeclAssgnStmt | listDeclAssgnStmt.
incDecStmt          = designator ("++" | "--").
callStmt            = designator.
ifStmt              = "if" [shortVarDecl ";"] expr block ["else" (ifStmt | block)].
switchStmt          = "switch" [shortVarDecl ";"] expr "{" {case} [default] "}".
case                = "case" expr {"," expr} ":" stmtList.
default             = "default" ":" stmtList.
forStmt             = "for" (forHeader | forInHeader) block.
```

```
forHeader           = [shortVarDecl ";"] expr [";" simpleStmt].
forInHeader         = [ident ","] ident "in" expr.
breakStmt           = "break".
continueStmt        = "continue".
returnStmt          = "return" [exprList].
exprList            = expr {"," expr}.
expr                = logicalTerm {"||" logicalTerm}.
logicalTerm         = relation {"&&" relation}.
relation            = relationTerm
                      [("==" | "!=" | "<" | "<=" | ">" | ">=") relationTerm].
relationTerm        = term {("+" | "-" | "|" | "~") term}.
term                = factor {("*" | "/" | "%" | "<<" | ">>" | "&") factor}.
factor              = designator | intNumber | realNumber | charLiteral | stringLiteral |
                      ("+" | "-" | "!" | "~") factor | "&" designator | "(" expr ")".
designatorList      = designator {"," designator}.
designator          = (primary | typeCast | compositeLiteral) selectors.
primary             = qualIdent | builtinCall.
qualIdent           = [ident "."] ident.
builtinCall         = qualIdent "(" [expr {"," expr}] ")".
selectors           = {derefSelector | indexSelector | fieldSelector | callSelector}.
derefSelector       = "^".
indexSelector       = "[" expr "]".
fieldSelector       = "." ident.
callSelector        = actualParams.
actualParams        = "(" [expr {"," expr}] ")".
compositeLiteral    = arrayLiteral | dynArrayLiteral | structLiteral | fnLiteral.
arrayLiteral        = "{" [expr {"," expr}] "}".
dynArrayLiteral     = arrayLiteral.
structLiteral       = "{" [[ident ":"] expr {"," [ident ":"] expr}] "}".
fnLiteral           = fnBlock.
typeCast            = type "(" expr ")".
ident               = (letter | "_") {letter | "_" | digit}.
intNumber           = decNumber | hexHumber.
decNumber           = digit {digit}.
hexNumber           = "0" ("X | "x") hexDigit {hexDigit}.
realNumber          = decNumber ["." decNumber] [("E" | "e") decNumber].
charLiteral         = "'" (char | escSeq) "'".
stringLiteral       = """ {char | escSeq} """.
escSeq              = "\" ("0" | "a" | "b" | "e" | "f" | "n" | "r" | "t" | "v" |
                           "x" hexNumber).
letter              = "A".."Z" | "a".."z".
digit               = "0".."9".
hexDigit            = digit | "A".."F" | "a".."f".
char                = "\x00".."\xFF".
```