SDK Language Manual

Version 2.5.10

Netmodule AG, Switzerland

April 29, 2021

Contents

5   Acknowledgements 135

1 Introduction

This manual describes the Arena scripting language. It is meant to give a complete overview of the language. This includes syntax, semantics, and standard library functions provided by the language runtime environment.

1.1 What's Arena?

Arena is a scripting language. It is closely modeled on the C programming language, but with some features removed and added to create a language more suitable to ad-hoc scripting. The following is a description of the main differences between Arena and C.

Arena does automatic memory management. This means the programmer does not have to reserve memory for strings and arrays. Additionally, variables do not have to be declared before they are used.

Arena uses dynamic typing. This means variables can be used to store arbitrary values. A variable that holds an integer at the beginning of a script may well be used to hold a string at the end of the same script. The concept extends to arrays – arrays can have elements of different types.

Arena has anonymous functions. Sometimes you may want to pass a function into another function (functions can accept other functions as their arguments), and anonymous functions provide a way of doing so without having to invent a function name. This is especially useful if you need a particular function just once and just for passing into another function.

Arena provides exception support. Exceptions can be used for handling error situations in a script. They provide out-of-band error signaling and handling.

Arena does not allow user-defined datatypes. This is a restriction common to many scripting languages. It does, however, have structure templates, which work a lot like classes in object-oriented programming languages.

Arena does not provide a way to define constants – that is, values set by the programmer that cannot change during the execution of a script. The rationale is that it is not strictly necessary to have constants provided by the language. One can simply use a global variable and write to it only once at the beginning of a script.

Apart from the functions listed above, Arena tries to emulate C as much as possible. The semantics of language construct are supposed to match C, and the standard library of functions uses the same names as the C standard library where both provide the same functionality.

1.2 Why another scripting language

There is no shortage of existing scripting languages, so why design and write another one? Two reasons, mainly.

The first reason is that many people, especially in the Unix community, know how to program in C, but having to do your own memory management all the time is a pain for small or quick projects. Arena provides a way to write "almost C" code without having to think about memory management. Dynamic typing was added because it is very convenient to have once you have already abandoned the need to declare variables before use (which you have to do in C so that the compiler can set aside memory for variables).

The second reason for writing another language is that most scripting languages of today are not really lightweight anymore. Extensive function libraries often mean that a scripting language interpreter is several megabytes in size. For fans of more minimalist approaches, several megabytes ain't it. Arena's standard library of functions is based on that of ISO C for the very reason that it is very compact and does not provide bells and whistles.

1.3 Target audience

This manual tries to describe the syntax and semantics of the Arena language, but it does not go into every detail and certainly is no guide on how to solve real problems using Arena.

It is assumed the reader already knows how to program. Most of the language constructs of Arena appear in other languages, as well, so already knowing a different programming language helps. Since Arena is modeled on C, knowing C helps a lot. For structure templates, which are not taken from C, knowledge of object-oriented programming languages such as C++ or Java should help, since structure templates are basically a low-level version of classes.

1.4 Versioning

Both the language and standard library are versioned. This manual describes version 2.2 of the language and version 2.5.10 of the standard library.

Incompatible changes to the language or library result in a change of the major version number and an implementation of the new version cannot run all scripts written for a previous version of the language. Thus, an implementation of version 2.0 of the language will not run all possible version 1.0 scripts.

Compatible changes to the language or library result in a change of the minor version

number. An implementation of such a new version must still be able to run all scripts written for a version of the language with the same major version number and a smaller minor version number. Thus, an implementation for version 1.3 of the language will still run all version 1.0, 1.1, and 1.2 scripts.

Minor version number changes for the language are only possible if some new syntax is introduced, in such a way that the new syntax would have been a syntax error in the previous version. Changes to existing syntax require a new major version.

Minor version number changes for the library are possible as long as only new library functions are introduced by the new version. Old scripts that already use the same function names for user-defined functions will still work as the user-defined functions will overwrite the library functions.

1.5 Structure of this manual

The rest of the text is divided into two main chapters. The first describes the syntax and intended semantics of the language. The second describes the standard library of functions that come with the language.

If some aspect of the behavior of the language or library is said to be "implementation-defined", this means an implementation of the language can freely choose how to behave for the described situation. However, the choice must be consistent – under the same circumstances, the same behavior must result.

If some aspect of the behavior of the language or library is said to be "undefined", this means an implementation of the language can do anything for the described situation, no matter how inconsistent. An implementation may even crash if an undefined situation arises during the execution of a script; or, as has been observed about C, an implementation may make demons fly out of your nose if you invoke undefined behavior.

1.6 License

You are free to copy, distribute, display, make derivative works of, and/or make commercial use of this manual, provided you follow these conditions:

You must keep any copyright notices and license terms intact. You are free to add your own copyright notices to parts of a derivative work that you wrote yourself.

If you make changes to the semantics of existing parts of the text, those parts must carry prominent notice that you changed them. This condition is made because this manual describes the behavior of a programming language, and changes to the text can easily change the described behavior. This could lead to the changed text describing another, slightly incompatible language.

2 Language

This section of the manual describes the syntax and semantics of the Arena scripting language.

This version of the language manual describes version 2.2 of the language.

2.1 Basic tokens

When a script is parsed by the Arena interpreter, it is first split up into tokens. These tokens are then combined to form statements and expressions. Since it is important to know what kind of tokens (for example, variable and function names) are accepted by the language, the different token types are described next.

2.1.1 Comments

Comments can be part of a script. They are ignored by the interpreter and can be used to annotate the script for human readers. There are two forms of comments: one-line comments and multi-line comments.

One-line comments start with the character "#" (hash) or the characters "//" (double forward slash). They can be placed anywhere on an input line and cause the rest of the line to be treated as a comment. The following are examples of one-line comments:

```
# this line is ignored
a = 5; // everything back here is ignored
```

Multi-line comments start with the characters "/*" (forward slash followed by asterisk) and end with the characters "*/" (asterisk followed by forward slash). Everything between those two markings is ignored. Multi-line comments can be nested – you need a matching number of "/*" and "*/" sequences to really end a comment. The following is an example of a multi-line comment:

```
/* this is lengthy explanation of what is happening,
   but you can probably figure that out yourself */
```

2.1.2 Keywords

Keyword are words reserved by the language. They are used to make up statements and expressions. They cannot be used as names for variables, functions, or templates. Keywords are case-sensitive: "do" is a language keyword, "Do" or "DO" are not.

The following is a list of all Arena keywords:

```
array    break    bool     case    catch   continue
default  do       else     extends false   float
fn       for      forced   if      include int
mixed    new      resource return  string  struct
switch   template throw    true    try     void
while
```

2.1.3 Operators

Operators are special symbols reserved by the language. They are used to combine expressions and generally represent operations performed on pieces of data. For example, the + operator denotes mathematical addition.

The following is a list of all Arena operator symbols:

```
::      ==      !=      <=      >=      <
>       ++      --      &&      ||      **
+       -       *       /       %       &
|       ^       <<      >>      !       ~
=       +=      -=      *=      /=      &=
|=      ^=      <<=     >>=
```

2.1.4 Identifiers

An identifier is a name used for a variable, a function, or a structure template. It is used in a script to refer to entities of the language by name. Identifiers are chosen by the programmer. The language actually puts some identifiers in place before a script starts (those for the standard library of functions), but those are not reserved in the same way that keywords are – you can reuse them for your own variables, functions, or structure templates if you wish.

An identifier starts with an underscore character or an upper-case or lower-case letter. A letter is one of the 26 characters in the range A-Z (no umlauts or accented letters allowed). For the rest of an identifier, the same characters are allowed, with the addition

of decimal digits. Decimal digits are characters in the range 0-9.

Keywords cannot be used as identifiers.

The following is a list of example identifiers:

```
foo
x2
my_funny_name
__something
```

2.1.5 Integer literals

An integer literal is used to represent an integer number in a script. An integer literal is made up of an optional prefix and one or more digits. An integer literal with no prefix is treated as a decimal number. Decimal digits are characters in the range 0-9. An integer literal with the prefix "0" (zero) is treated as an octal number. Octal digits are characters in the range 0-7. An integer literal with the prefix "0x" (zero x) is treated as a hexadecimal number. Hexadecimal digits are characters in the ranges 0-9, a-f, and A-F.

The following are examples of integer literals:

```
0
123
0755
0xFF
0xbeef
```

2.1.6 Float literals

A float literal is used to represent a floating point number in a script. A float literal is made up of zero or more decimal digits, followed by a period, followed by one or more decimal digits. A decimal digit is a character in the range 0-9. Optionally, an exponent can be added to the end of the literal. This is composed of the letter "e" or "E", followed by either "+" or "-", followed by one or more decimal digits. If present, the exponent is used as a base 10 exponent and multiplied with the rest of the number. As an example, "1E-2" is the same as 1 * 10^(-2) which is 0.01.

The following are examples of float literals:

```
1.0
.25
0.376568E-10
1E+30
```

2.1.7 String literals

A string literal is used to represent a string inside a script. A string literal is made up of a single or double quote character, followed by an arbitrary number of characters, followed by a matching single or double quote. If the string literal is enclosed in single quotes, it cannot contain a single quote. The same applies to string literals in double quotes; they cannot contain double quotes.

To allow the representation of characters that cannot directly appear inside a script or string, some escape sequences are permitted. An escape sequence begins with the character "\" (backslash). The following escape sequences are defined:

```
\\      a literal backslash
\b      backspace character
\e      escape character
\f      form feed character
\n      newline character
\r      carriage return character
\t      tab character
\ccc    character with octal character code ccc
\occc   character with octal character code ccc
\dccc   character with decimal character code ccc
\xcc    character with hexadecimal char code cc
```

For character code escapes, less digits than given above can be used if the character code needed is small enough. Note that if any character not listed above follows the backslash, the escape sequence results in that character. For example, the escape sequence "\q" results in the character "q".

The following are examples of string literals:

```
"Hello"
'Greetings to you!\n'
"All your base are belong to us"
'Embedded \0 zero \0 characters'
```

2.1.8 Grouping symbols

Grouping symbols are used to make up larger entities from statements and expressions or to change the order in which script code is executed. The following is a list of the grouping symbols used by the Arena language:

```
(       )       {       }           [       ]
.       ;       ,
```

2.2 Runtime type system

Types are used to provide categories for different kinds of values that a script deals with. Arena provides eight datatypes for use by the programmer. No user-defined types are possible, but a script can use structure templates to provide a sort of sub-typing for the struct datatype.

Values of some types can be converted into values of other types by use of a cast expression. More on that later in the chapter about expressions.

2.2.1 void

The void type is used in places where no meaningful value can be returned. The void type has only one value, which is written "()" (two parenthesis immediately following each other, pronounced "void" or "unit"). All Arena functions must return a value. If a function does not have a meaningful result (for example, a function that outputs a message to the user), it can return a void value instead of having to invent something else.

2.2.2 bool

The bool type is used to represent truth values. It has two values called "false" and "true". It is normally used to hold the results of boolean computations or for representing simple on-off switches.

2.2.3 int

The int type is used to hold signed integer values. The precision is at least 32 bits. This means an int can generally hold integer values between -2^{31} and $2^{31} - 1$.

Arena does not provide unsigned integers. The rationale for this is that the additional bit of precision that an unsigned type provides for large positive integer values is not

enough of a benefit to warrant extra complexity for an implementation.

2.2.4 float

The float type is used to represent signed floating point number. The precision of a float is at least that of an IEEE double precision floating point number.

Arena does not provide multiple floating point types with different precisions, like C does. Like the omission of an unsigned integer type, this was decided to keep implementation complexity down to a minimum.

2.2.5 string

The string type is used to represent an arbitrary sequence of bytes or characters. It is normally used to represent text. Note that unlike strings (character pointers, really) in C, an Arena string can contain bytes with the value 0 (zero). In C such a byte would be considered the end of the string.

2.2.6 array

The array type is used to represent a numbered collection of values. The types of the values stored in an array, called the elements of the array, are not constrained. This means each element can have a different type from the other elements. An array can have other arrays as elements.

Arrays are indexed using integers, starting at 0. This means the first element of an array has index 0, the second has index 1, the third has index 2, and so on.

2.2.7 struct

The struct (short for structure) type is used to represent a collection of values. Unlike an array, in which the elements are reference by integer indices, the elements of a struct have names. The order of elements in a struct is not significant, which is another important difference to the array type. Elements in a structure are called "fields" or sometimes "methods" (if they are of type fn, see below).

The names of structure elements are identifier tokens, but there are also library function that use normal string values as structure element names. In general, you can think of a struct as being indexed by string values.

2.2.8 fn

The fn type is used to represent functions. This type allows an Arena script to use functions like any other value. For example, functions can be used as arguments to other functions or can be returned as results from other functions. It is also possible to create so-called anonymous functions on the fly, by use of a special expression that results in an fn value.

2.2.9 resource

The resource type is used to represent operating system resources in use by a script. Examples are file handles or manually allocated memory. The resource type has automatic management that ensures that operating system resources are freed when a resource value is no longer accessible by a running script.

The contents of a resource value are opaque from the viewpoint of a running Arena script.

2.3 Scopes and namespaces

A scope is defined as the area where a given portion of source code appears a script. A namespace defines a limited area of visibility for variables, functions, and structure templates. Both concepts are related and determine what parts of a script can access other parts of the same script.

2.3.1 Top-level vs. function-level scope

The scope of a piece code is determined wholly by its position in the source code. The scope of a given piece of code cannot and does not change at runtime.

The scope active at the beginning of a script is the top-level scope. At this scope, arbitrary statements can be used, including function and structure template definitions.

When a function definition begins, the source code scope changes to function-level scope. At this scope, all statements except other function definitions and structure template definitions are allowed. This means function definitions cannot be nested.

When a function definition ends, the statements that appeared in the function-level scope become the function's body. The function body is what gets executed when a function later is called from other code. After leaving a function definition, the top-level scope is active again.

When a structure template definition begins, the scope remains top-level scope, but the following definitions up to the end of the structure template definition are considered to be part of it. Structure template definitions cannot be nested.

2.3.2 Global vs. local namespace

Namespaces are areas where variables, functions, and structure templates are stored. All the named entities of the language that are used in a script are part of a namespace. A namespace associates identifiers with the entities they name. Note that there are no separate namespaces for variables, functions, and structure templates. A given identifier can only be used for one kind of entity at a time.

Namespaces can be visible or invisible to the currently executing code. Code can only see variables, functions, and structure templates stored in a visible namespace. Entities stored in an invisible namespace are involatile until they become visible again.

There is one special namespace called the global namespace. This namespace is always visible. Variables and functions provided by the Arena standard library are stored in the global namespace. Code running at top-level scope has access to only one namespace, the global namespace.

In addition to the global namespace, there are local namespaces. A local namespace is created whenever a function is called. The code inside the function runs within a local namespace of its own. To this code, both the global namespace and the local namespace of the function are visible. The local namespace starts out empty.

The visibility rules inside a local namespace are as follows: the local namespace has priority. Only if an identifier is not found in the local namespace, the global namespace is consulted. When the namespace is written to, the write always only effects the local namespace. If a function attempts to change a variable it has obtained from the global namespace, a copy of the variable is created in the local namespace.

When a function calls another function, another local namespace is created. The previous local namespace is invisible to the code inside the called function. Only when the called function exits, that namespace becomes visible again.

When a function exits, its local namespace is destroyed. Everything that was stored in the local namespace is no longer accessible. You can assume memory that was used by the local namespace is freed at this point.

What the above boils down to is that functions have their own set of local variables and can manipulate them without affecting variables outside of the function itself.

As a side note, the struct type works just like a namespace of its own.

2.4 Statements

Statements provide a way to sequence and structure code. In other words, statements determine what gets executed and under which conditions.

The following sections include code examples that make use of expressions, which have not been described up to now. Expressions will be explained in the next chapter.

2.4.1 Basic rules for statements

Statements are executed in order that they appear in the top-level scope. Individual statements are end with a ";" (semicolon) character. Expressions can be used as statements by simply adding a semicolon at the end of the expression. For example, if "expr" is a valid expression, then the following is a valid statement:

```
expr;
```

Using an expression as a statement evaluates the expression. Evaluation of an expression results in a value in one of the types provided by the language. When an expression is used as a statement, that value is discarded.

Statements can be grouped together into one statement by using curly braces. The whole block of statements counts as one statement. When the block is executed, the statements inside it are executed in the order they are listed. For example:

```
{
   stmt1;
   stmt2;
   stmt3;
}
```

The above is a block consisting of three statements. Note that there is no semicolon at the end of the block itself. Blocks can be nested arbitrarily deep. Blocks are normally used when you want to supply a list of statements to execute in a place where only one statement is allowed by the language.

A semicolon all by itself also constitutes a valid statement that does nothing when executed. Blocks are allowed to be empty. An empty block does nothing when executed.

2.4.2 Include statement

The include statement is made up of the keyword "include" followed by a string in double quotes, followed by a semicolon as usual for ending a statement. The string is used as a filename. The contents of the file are parsed as source code as if it were present after the line with the include statement on it.

Note that the included code will be parsed at the current scope. If the current scope is inside a function, the included code cannot define functions or structure templates. Normally include statements are only used at global scope, for including files that contain libraries of functions or structure template definitions.

An implementation of Arena may search for the named include file in implementation-defined locations on the system running the script. However, it is only guaranteed that

the current working directory will be searched.

Include files can be nested arbitrarily deep. It is the responsibility of the programmer to prevent loops.

The following is an example of an include statement used to read in a file called "library.inc":

```
include "library.inc";
```

2.4.3 Control flow statements

Control flow statements influence the order in which statements are executed, or whether they are executed at all.

2.4.3.1 if statement

The if statement is used to execute code based on a condition. It consists of the keyword "if", followed by an expression in parenthesis, followed by a statement or block. The expression is called a guard expression.

When the if statement is executed, the guard expression is evaluated. If the resulting value is not of type bool, it is converted to bool (using the same rules as for cast expressions, see below). If the result is the bool value "true", the statement part of the if statement is executed. If the the result of the guard expression is "false", the statement part is not executed.

The following is an example of an if statement:

```
if (x % 2 == 0)
  print("x is even!");
```

If you need to execute multiple statements, use a block statement.

You can also give a statement to be executed when the guard expression evaluates to "false". This is done by following the first statement with the keyword "else" and another statement. An example:

```
if (x % 2 == 0)
  print("x is even!");
else
  print("Sorry, x is uneven!");
```

2.4.3.2 while loop statement

The while loop statement can be used to execute another statement or block multiple times. It consists of the keyword "while", followed by a guard expression in parenthesis, followed by a statement known as the loop body.

When a while loop is executed, the guard expression is evaluated, following the same rules as given for the guard expression of an if statement. If the result is "true", the loop body is executed. Execution of the while loop then restarts at the beginning. If the guard expression evaluates to "false", the loop body is not executed and the while loop is not restarted at the beginning.

These rules mean that a while loop only executes as long as the guard expression evaluates to "true". If the guard expression evaluates to "false" the first time it is considered, the loop body is never executed.

The code inside the while loop normally has side effects that eventually change the result of the guard expression to "false".

The following is an example of a while loop with a block statement as its loop body:

```
while (x % 2 == 0) {
  print("x was even");
  x = rand(0,999);
}
```

2.4.3.3 do loop statement

The do loop statement is a close cousin of the while loop statement; only the positions of the guard expression and loop body are exchanged. A do loop consists of the keyword "do", followed by a statement as the loop body, followed by the keyword "while" and a guard expression in parenthesis.

When a do loop is executed, the loop body gets executed first. Then the guard expression is evaluated using the same rules as given for the guard expression of an if statement. If the result is "true", the do loop is executed again. If the result is "false", execution continues after the loop.

The above rules mean that the body of a do loop is always executed at least once. It is then executed again as long as the guard expression evaluates to "true".

The following is an example of a do loop:

```
do {
  now = time();
} while (now - saved < 10);
```

2.4.3.4 for loop statement

The for loop statement offers a more versatile form of looping compared to the while and do loops detailed in the previous two sections. A for loop consists of the keyword "for", followed by three semicolon-separated expressions in parenthesis, followed by a statement that serves as the loop body. The first expression is called an initializer expression, the second a guard expression, and the third a loop expression.

When a for loop executes, the initializer expression is evaluated. This happens only once, and the result of the evaluation is discarded. Then the guard expression is evaluated using the same rules as given for the guard expression of an if statement. If the result is "true", the loop body is executed. Following the loop body, the loop expression is evaluated and its result discarded. Execution of the for loop then restarts, omitting the initializer expression. If the guard expression evaluates to "false", the loop body and loop expression are not executed and execution resumes after the for loop.

The above rules mean that a for loop executes as long as its guard expressions evaluates to "true". If it does not evaluate to "true" on the first execution of a for loop, the loop body is never executed.

Each of the three expressions in a for loop statement can be left empty. In that case the (empty) expression is replaced with the literal constant "true". This means a for loop with all three expressions left off produces an infinite loop.

For loops are often used to execute a piece of code a given number of times. For example, the following loop prints the word "hello" ten times in a row:

```
for (i = 0; i < 10; i++) {
   print("hello");
}
```

2.4.3.5 continue statement

The continue statement can be used inside of do, while, and for loops. It consists of the keyword "continue".

When a continue statement is executed inside of a loop body, the statements following the continue statement in the loop body are skipped. Processing continues as normal for the loop statement in question. Normally this means the loop's guard expression will be evaluated again.

When a continue statement is executed outside of a loop body, it has the same effect as an empty statement.

The following is a (silly) example of counting the number of odd integers between 0 and 99. A for loop is used and the increment of a counter variable is skipped by use of a continue statement if the number in question is even.

```
odd = 0;
for (i = 0; i < 100; i++) {
  if (i % 2 == 0) continue;
  ++odd;
}
print(odd, " odd numbers found");
```

2.4.3.6 break statement

The break statement can be used inside of do, while, and for loops (for the use of break in a switch statement, see the next section). It consists of the keyword "break".

When a break statement is executed inside of a loop body, the execution of the rest of the loop body is skipped. Execution then resumes with the next statement following the loop statement that contains the break statement. In effect, execution of that loop is terminated by the break statement.

When a break statement is executed outside of a loop body (or switch statement, see below), it has the same effect as an empty statement.

The following is an example use of break which exits from an infinite for loop as soon as a random number between 0 and 99 equals zero.

```
for (;;) {
  number = rand(0, 99);
  print("my number: ", number, "\n");
  if (number == 0) break;
}
```

2.4.3.7 switch statement

The switch statement is used to execute one or more of a number of statement groups depending on the value of a guard expression. It consists of the keyword "switch", followed by a guard expression in parenthesis, followed by statement groups enclosed in curly braces.

Two different kinds of statement groups are possible. There can be an arbitrary number of case groups and one default group. A case group starts with the keyword "case" followed by an expression, followed by a colon, followed by an arbitrary number of statements. If the last statement in the group is a break statement, this has a special meaning described below. The default group consists of the keyword "default" followed by a colon, followed by an arbitrary number of statements. A break statement at the

end of a default group has no special meaning relevant to the switch statement, but it still has its normal effect on an enclosing loop statement.

When a switch statement is executed, its guard expression is evaluated. The resulting value is then used to decide which case group to execute. Case groups are considered in the order that they appear in the switch statement. When a case group is considered, its expression is evaluated. If the resulting value is equal (in type and value) to the value of the guard expression, the statements inside the case group are executed. If the last statement of the group is a break statement, execution of the switch ends and the next statement executed is the one following the switch statement. If there is no break at the end of the case group, the statements of the next group are also executed, without evaluating the expression of that group. This is called "fall through". This behavior continues until either a break statement at the end of a case group is encountered, a default statement group is executed, or the switch statement ends.

If a case group is considered and its value does not match the value of the switch's guard expression, the statements in the case group are not executed. The next case group is considered instead and its expression will be evaluated and checked. A default group, if present, is not included in the case statements to consider for execution.

When all case statements have been considered and no match was found, the behavior of the switch statement depends on the presence of a default group. It it is present, the statements associated with it are executed. If it is not present, the switch simply executes nothing. Note that there is no fall through out of a default group, execution of a switch always ends once the last statement of the default group has been executed.

The following example counts how many numbers between 0 and 99 are divisible by 3 or 6. It uses a switch that evaluates the remainder of a division by 6. It employs fall through since anything divisible by 6 is also divisible by 3. It uses a default group to count how many numbers were not divisible by 3 or 6.

```
three = six = none = 0;
for (i = 0; i < 100; i++) {
  switch (i % 6) {
    case 0:
      ++six;
    case 3:
      ++three;
      break;
    default:
      ++none;
  }
}
print(three, "numbers were divisible by 3\n");
```

```
print(six, "numbers were divisible by 6\n");
print(none, "number were not divisible by either\n");
```

2.4.3.8 try statement

The try statement is used to handle exceptions. It consists of the keyword "try", followed by a statement, followed by the keyword "catch", followed by an identifier in parenthesis, followed by another statement.

When a try statement is executed, the statement following the keyword "try" is executed. What gets executed next depends on whether this statement causes an exception (by use of a throw statement, see below). If the enclosed statement does not cause an exception, the next statement executed is the statement directly following the try statement; the statement in the catch part of the try statement is not executed.

If the enclosed statement does throw an exception, the value thrown is assigned to a variable with the identifier given in the catch part of the try statement. The statement given in the catch part is then executed. Execution then continues behind the try statement. The variable with the exception value remains visible to the code following the try statement. Executing the catch part of a try statement is often called "handling" the exception.

It is possible for try statements to be nested arbitrarily deep. An exception is always handled by the innermost try statement that encloses the code that caused the exception.

It is common for both statements in a try statement to actually be block statements.

The following is an example of a try statement used to encapsulate two function calls which may cause exceptions. If an exception occurs, its value is printed.

```
try {
  a = somefunc();
  b = someotherfunc();
} catch (e) {
  print("exception ", e, " occurred\n");
}
```

2.4.3.9 throw statement

The throw statement is used to cause an exception. It consists of the keyword "throw" followed by an expression.

When a throw statement is executed inside of a try statement (either directly or because it occurs inside a function called from within a try statement), the throw expression is evaluated and the resulting value becomes the exception value. Execution then continues

with the catch part of the innermost enclosing try statement.

Note that the above means a throw statement executed inside a loop body breaks out of the loop if the handling try statement is outside of the loop.

When a throw statement is executed outside of a try statement, this is considered a fatal error and execution of the whole Arena script is terminated at the point where the exception was thrown.

The following is an example of the use of a throw statement to throw an exception with the string value "oops" as the exception value:

```
throw "oops";
```

2.4.4 User-defined functions

User-defined functions provide a way to structure code into separate, named entities. Each function accepts input values, called function arguments, and computes a value called the return value of the function when called.

2.4.4.1 Function definition

A function definition declares a user-defined function to the script interpreter. It consists of the function return type, followed by an identifier naming the function, followed by a list of argument types and names in parenthesis, followed by a statement to be used as the function body. The individual argument types and names are separated by commas. The list of arguments can be left empty.

The return type can be given by using one of the keywords "void", "bool", "int", "float", "string", "array, "struct", "resource", or "fn". The intent is to specify that the function returns a value of the given type when it is called. It is a fatal error if the code of the function body does not return a value that has the return type. The special keyword "forced" can be prefixed to the return type. If it is, it is not an error if the function attempts to return a value not having the return type – instead, the language automatically casts (see cast expressions, below) the return value to the appropriate type. The special keyword "mixed" can also be used in place of a real type to indicate that the return value of the function does not always have one and the same type.

Function arguments are specified by using the optional keyword "forced", followed by a type name (same as the return type detailed above), followed by an identifier. The identifier is used to name the argument. When a function is called, the function's arguments are available to the function body as variables with names as given in the function definition. The argument type of an argument is checked when a function is called. If the "forced" keyword was used, the argument value is automatically cast to

the given type. If not, it is a fatal error to call the function with an argument value not matching the given argument type.

The type of a function argument can be left out, in which case the language behaves as if the type "mixed" had been specified.

The function body can be any statement. Most functions contain more than one statement, thus most function bodies will be block statements.

The return type, name, and argument types of a function are called the prototype of the function.

When a function definition is executed, the new function's existence is recorded in the current namespace. Since function definitions can only occur at top-level scope, this will always be the global namespace. It is not an error to define a function with the same name as an existing variable, function, or structure template. The new function definition will override any previous meaning of the same name.

The result value, or return value, of a function is determined by using a return statement, described below. A function body that does not use a return statement will automatically be made to return a void value by the language runtime system.

The following is an example of a function definition for a function named "sum" that returns an int value and excepts two int arguments named "x" and "y", respectively. The example function body returns the sum of both int values.

```
int sum(int x, int y)
{
   return x + y;
}
```

The function definition above will result in a fatal error if passed float arguments, for example. To cause the language to automatically convert both arguments to int when the function is called, the definition would have to be changed to:

```
int sum(forced int x, forced int y)
{
   return x + y;
}
```

2.4.4.2 return statement

The return statement is used to set the return value of a function and terminate the execution of a function body. It consists of the keyword "return" followed by an optional expression.

When a return statement is executed inside a function body, the return expression is evaluated and used as the return value of the function. If no return expression is present, a void value is substituted instead. Statements following the return statement in the function body are not executed. The effect of the return statement is to always end the execution of a function body.

The return value is passed back the caller of the function.

When a return statement is executed outside of a function body, it behaves like an empty statement and the return expression is not evaluated.

The following is an example of a return statement used to return the bool value "true":

```
return true;
```

2.4.5 Structure templates

A structure template is a blueprint for constructing values of type struct. Structure templates support inheritance, meaning one structure template can build upon another structure template defined earlier. Structure templates can define fields and methods that are to be created when a struct value is constructed from the template.

A structure template consists of the keyword "template", followed by an identifier to name the template, followed by field and method definitions enclosed in curly braces. Optionally, the name of the template can be followed by the keyword "extends" and an identifier naming another structure template that this template builds upon.

When a structure template is executed, the new template is stored in the current namespace and is available to code following the structure template. Since structure template can only occur at top-level scope, they are always stored in the global namespace. It is not an error if the template name is already used by an existing variable, function, or other template. The new structure template overrides any previous definition of the same name.

See the following sections for examples of structure templates. See the section "Constructor calls" in the chapter on expressions for information on how to create struct values from structure templates.

2.4.5.1 Defining structure fields

Structure fields in structure templates are used to define data fields that will appear in struct values created from the template. The definition of a structure field gives the identifier of the field. A value for the field can also be given, but this is optional.

A structure field definition without value consists of an identifier followed by a semicolon. When a struct value is constructed from the template, the resulting value will have an

element named by the identifier that contains a void value.

A structure field definition with value consists of an identifier, followed by the assignment operator ("="), followed by an expression, followed by a semicolon. When a struct value is constructed from the template, the resulting value will have an element named by the identifier that contains the result of evaluating the expression.

The following is an example of a structure template that defines two structure fields. The first field is named "i" and not given a value, the second is called "foo" and given the constant int expression 42 as a value.

```
template example
{
  i;
  foo = 42;
}
```

When a template extends another template, both may contain fields of the same name. The values given by the extending template have precedence. In the following example, a struct value constructed from template "bar" will contain a field called "i" with the int value 2.

```
template foo
{
  i = 1;
}
template bar extends foo
{
  i = 2;
}
```

2.4.5.2 Defining structure methods

A method is a function stored within a structure. This is basically the same as a struct field with type fn. The name "method" was chosen because that is how object-oriented languages name a similar construct.

A structure method definition inside a structure template is written exactly like a function definition (see above). The only difference is that the function definition occurs within the curly braces enclosing the structure template's definition.

When a struct value is constructed from the structure template, it will contain an element with the function name from the function definition. The element will contain a value of type fn that corresponds to the given function prototype and body.

The following is an example of a structure template that defines a method called "double", which is given as a function that will double its int argument.

```
template foo
{
  int double(int x)
  {
    return 2 * x;
  }
}
```

For structure templates extending other structure templates, the same rules as for structure fields apply: when both templates define a method of the same name, the definition in the extending template takes precedence. In the following example, struct values constructed from the "bar" template will contain a method called "fiddle" that quadruples its argument, whereas struct value constructed from the "foo" template will contain a method called "fiddle" that triples its argument.

```
template foo
{
  int fiddle(int x)
  {
    return 3 * x;
  }
}
template bar extends foo
{
  int fiddle(int x)
  {
    return 4 * x;
  }
}
```

Note that field and method definitions in a structure template can be intermixed in any order.

2.4.5.3 Constructor method

A constructor method is a structure method definition with a special name. A method is called the constructor method if its identifier is the same as the identifier of the structure definition it is part of.

Constructor methods play a special role when a struct value is constructed from a template, as described in the section "Constructor calls" in the chapter on expressions. Apart from that, a constructor method behaves identically to other methods defined by a structure template.

The following is an example of a structure template "foo" that contains a constructor method that will print out a message whenever it is called.

```
template foo
{
  void foo()
  {
    print("constructor method foo called!\n");
  }
}
```

2.5 Expressions

Expressions are basically descriptions on how to compute a value. Determining the value of an expression is called evaluating the expression. The result of evaluating an expression, called its value, is a value from one of the eight built-in types of the Arena scripting language.

2.5.1 Basic rules for expression nesting

Expression can be made up of other expressions by use of several operators which are detailed in the sections below. The exact meaning of compound expressions such as "2 + 3 * 5" is determined by precedence and associativity. For example, in the expression "2 + 3 * 5", the multiplication is performed before the addition. To override the order in which parts of an expression are evaluated, it is possible to put parts of an expression into parenthesis. The sub-expression thus formed must be a valid expression in itself and its value will be evaluated before the rest of the original expression. For example, to compute the addition before the multiplication in the aforementioned example, the expression would have to be changed to "(2 + 3) * 5".

The following sections list all possible types of expressions supported by the Arena scripting language. Precedence and associativity of all language operators are given near the end of the chapter.

2.5.2 Constant expressions

A constant expression consists of a literal token. There are literal tokens for the types void, bool, int, float, and string.

When a literal token expression is evaluated, the result is a value of the appropriate type. For example, the literal expression "12" evaluates to the int value 12.

The following are examples of constant expressions:

```
true
12.0
"I'm a string"
()
42
```

2.5.3 Reference expressions

A reference expression is used to refer to a variable or function. It consists of an identifier.

When a reference expression is evaluated, the result is the value of the named variable in the current namespace. If the identifier refers to a function, the result is a value of type fn. If the identifier is unknown or names a structure template, the result is a void value.

The following are examples of reference expressions:

```
a
foo
some_long_identifier
```

2.5.3.1 Static reference expressions

A static reference expression is used to refer to elements of a structure template. It consists of an identifier, followed by the operator symbol "::" (double colon), followed by another identifier.

The first identifier is a template name that is looked for in the current namespace. If it does not denote an existing structure template, a fatal error is generated. Otherwise, a separate namespace is created. The field and method definitions of the structure template are then executed inside the new namespace. The second identifier is then used like a normal reference expression inside the new namespace. The new namespace is destroyed after obtaining the value of the static reference, which is the value of the whole static reference expression.

The following are examples of static references:

```
foo::bar
some_template::some_field
```

2.5.3.2 Indexing of elements

Indexing is used to refer to elements of array and struct values. Indices can be placed directly after reference expressions, static reference expressions, and all kinds of function and method calls.

An array index consists of one or more expressions, each enclosed in square brackets. When an array index is evaluated, the indexed expression and the expression(s) used as the index are evaluated. If the result value of the indexed expression is not an array, a void value is returned. Otherwise, the result of the index expression is cast to an integer (see below for type casting rules) and used as an index into the array. If the resulting integer index is valid for the array in question, the element stored at that index is the result of the indexed expression. Otherwise, the result is a void value.

The following is an example expression that assumes "a" is the name of an array variable and references the third element of the array:

```
a[2]
```

As a special case, negative indexing is allowed. A negative index is taken to be an offset from the end of the array. This way, the index -1 accesses the last element of an array. -2 accesses the element immediately preceding the last element, and so on. If a negative index reaches beyond the beginning of an array, the result is a void value.

Struct values contain values indexed by identifiers. A reference to a struct field consists of the operator symbol "." (period) followed by an identifier.

When a struct index is evaluated, the preceding expression is evaluated. If the result is not a struct value, the result is a void value. Otherwise, the index identifier is used as an element name for the struct value. If the struct has an element of that name, the value stored under that name is the result of the indexing expression. If the struct value does not have an element with the given name, a void value is used as the result.

The following is an example of an expression that uses "a" as the name of a struct variable and indexes a field "name" off the variable's value:

```
a.name
```

Array and struct indices can be freely mixed. Multiple array and struct indices can

follow each other. Evaluation proceeds from left to right. The following are examples of expressions with multiple indices:

```
a[2].foo[3][7].value
str.data[100]
a[0][1][2]
foo.bar.foobar
a[1].bar.foo[2]
```

The last example above would be evaluated as follows: first the variable reference "a" would be evaluated. If the resulting value is an array, the second element of the array is accessed. If the result is a struct, the field named "bar" is accessed. If the result is again a struct, the field named "foo" is accessed. If this results in an array value, the third element of that array is accessed and used as the value for the whole expression. If any value produced along the way does not have the expected type (array or struct, depending on the kind of indexing used), the result of the whole expression is a void value.

2.5.4 Cast expressions

Cast expressions are used to convert values from one type to another. A cast expression consists of an opening parenthesis, followed by a type name, followed by closing parenthesis, followed by an expression. No whitespace is allowed between parenthesis and type name.

The result of a cast expression is obtained by first computing the value of the inner expression and then converting it to the type named in the cast expression. If the value produced by the inner expression already has the right type, it is directly used as the result of the cast expression. Otherwise, the type conversion rules given in the following sections are applied.

This is an example of a cast expression casting the integer constant "1" to float:

```
(float) 1
```

2.5.4.1 Conversion to void

Since the void type has only one value, all values of all other types are converted to that one value.

2.5.4.2 Conversion to bool

Converting a void value to bool results in the bool value "false".

Converting an int value to bool results in the bool value "false" if the int value is 0 (zero). Otherwise, the result is the bool value "true".

Converting a float value to bool results in the bool value "false" if the float value is 0.0 (zero). Otherwise, the result is the bool value "true".

Converting a string value to bool results in the bool value "false" if the string is empty (that is, contains no characters). Otherwise, the result is the bool value "true".

Converting an array value to bool results in the bool value "false" if the array is empty (that is, contains no elements). Otherwise, the result is the bool value "true".

Converting a struct value to bool results in the bool value "false" if the struct is empty (that is, contains no fields or methods). Otherwise, the result is the bool value "true".

Converting an fn value to bool results in the bool value "true".

Converting a resource value to bool results in the bool value "true".

2.5.4.3 Conversion to int

Converting a void value to int results in the int value 0 (zero).

Converting a bool value to int results in the int value 0 (zero) if the bool value is "false". If the bool value is "true", the resulting int value is 1 (one).

Converting a float value to int results in an int value that corresponds to the integral part of the float value. If the integral part of the float value cannot be represented as an int, the resulting value is undefined.

Converting a string value to int attempts to interpret the string as an integer literal. Only an initial part of the string consisting solely of digits is considered for conversion.

Converting an array value to int results in an int value that gives the number of elements in the array.

Converting a struct value to int results in an int value that gives the number of elements in the struct.

Converting an fn value to int results in the int value 1 (one).

Converting a resource value to int results in the int value 1 (one).

2.5.4.4 Conversion to float

Converting a void value to float results in the float value 0.0 (zero).

Converting a bool value to float results in the float value 0.0 (zero) if the bool value is "false". If the bool value is "true", the resulting float value is 1.0 (one).

Converting an int value to float results in a float value with the same integral value as the original int value and no fractional part.

Converting a string value to float attempts to interpret the string as an float literal. Only an initial part of the string consisting solely of character that can occur in a float literal is considered for conversion.

Converting an array value to float results in an float value that gives the number of elements in the array.

Converting a struct value to float results in an float value that gives the number of elements in the struct.

Converting an fn value to float results in the float value 1.0 (one).

Converting a resource value to float results in the float value 1.0 (one).

2.5.4.5 Conversion to string

Converting a void value to string results in an empty string value.

Converting a bool value to string results in an empty string value if the bool value is "false" or in a string value containing the single character "1" (digit one) if the bool value is "true".

Converting an int value to string results in a string value containing the integer literal for the original int value.

Converting a float value to string results in a string value containing the float literal for the original float value.

Converting an array value to string results in a string value containing the word "Array".

Converting a struct value to string results in a string value containing the word "Struct"

Converting an fn value to string results in a string value containing the word "Function".

Converting a resource value to string results in a string value containing the word "Resource".

2.5.4.6 Conversion to array

Converting a non-array value to an array results in a one-element array that contains the original value at index 0 (zero).

2.5.4.7 Conversion to struct

Converting a non-struct value to a struct results in a struct with a single field named "value" that contains the original value.

2.5.4.8 Conversion to fn

Attempting to convert a non-fn value to fn is a fatal error.

2.5.4.9 Conversion to resource

Attempting to convert a non-resource value to resource is a fatal error.

2.5.5 Assignment expressions

An assignment expression is used to assign a value to a variable. It consists of an identifier, followed by the assignment operator "=" (equals sign), followed by an expression.

Evaluation of an assignment expression evaluates the inner expression and stores the result in the current namespace, in the form of a variable with the name given by the identifier in the assignment expression. Any previous meaning of the same identifier is lost. The assignment expression itself has the same result value as the inner expression. The following is an example expression that assigns the float value "12.5" to a variable named "val":

```
val = 12.5
```

Note that if an exception is thrown while evaluating the right side of an assignment, the assignment does not take place and the variable retains its previous value.

Since an assignment expression has the assigned value as its own value, and assignment associates to the right, it is possible to assign a value to multiple variables with an expression like this:

```
a = b = 0
```

2.5.5.1 Indexing in assignments

Array and struct indices can be used in an assignment expression just like they can be used in combination with reference expressions. For example, the following expression will assign the bool value "true" to the fifth element of an array stored in the variable "map":

```
map[4] = true
```

There is a difference to using indices in references, though. The above example will enforce "map" to be a variable of type array. If it is not an array before the assignment, an empty array will be created on the fly, the fifth element be set to "true", and the resulting array will be assigned to the variable "map". In the same way, when struct indexing is used on something that is not a struct, an empty struct value will be created

on the fly and substituted for the original non-struct value.

If a negative array index is used in an assignment that does not fall into the bounds of the array, the effect is to assign to the first element of the array.

Consider the following example:

```
a.foo.data[3] = 12
```

No matter what the value of the variable "a" is before the assignment, the following will be true after the assignment expression was evaluated: "a" will be a struct with at least the field "foo". The field "foo" will itself be a struct with a least the field "data". The field "data" will itself contain an array with at least four elements, the one at index 3 containing the int value 12. Values that already had the correct type for the assignment are not disturbed: for example, if the "data" field above already existed as an array of ten elements, it would still be an array of ten elements after the assignment; just the element at index 3 would have been overwritten with an int value of 12.

If both an index and the outer assignment have side effects on the same structure or array, the side effects of the index expression are discarded after the value of the index has been computed. In the following example, the value of "s.sp" is not changed after evaluation of the whole assignment expression:

```
s.stack[s.sp++] = 42
```

2.5.5.2 Combining assignments and operators

Instead of the plain assignment operator, the following operators can also be used:

```
+=        -=        *=        /=      &=        |=
^=        <<=       >>=
```

These are all composed of a normal operator symbol of the Arena language and the assignment operator symbol. The meaning of a special assignment is best explained by an example. Consider this expression using a special assignment operator:

```
a += 2
```

This expression behaves exactly the same as another, longer expression:

```
a = a + 2
```

In effect, using a special assignment operator is exactly the same as first referencing the target of the assignment, combining the result with the operator and inner expression given, and assigning the result to the target of the assignment.

2.5.6 Function calls

Function calls are used to call library functions or user-defined functions. A function call consists of an identifier naming the function, followed by a comma-separated list of expressions (the function call arguments) in parenthesis. The argument list is allowed to be empty.

When a function call expression is evaluated, the existence of the function is checked. If the identifier name is not found in the current namespace or does not refer to a function or fn variable, a fatal error is generated. If the function is found, the number of argument expressions is checked against the number of arguments given in the function's prototype. It is a fatal error to pass less arguments than present in the prototype. It is allowed to pass more arguments, extra arguments will be made available to a function's body as described below.

When it has been determined that a function call is valid as described above, the argument expressions are evaluated. Argument expressions are evaluated from left to right. The types of the resulting values are checked against the function's prototype, as described in the section about function definition statements (above, in the chapter about statements).

If the argument type check succeeds, a new local namespace is created. The values of the function's arguments are then added to the new namespace as if they were local variables assigned inside the function's body. For example, consider a function with the following prototype:

```
int mult(int x, int y)
```

When this function is called with the arguments 42 and 12, the local namespace of the function will contain an int variable named "x" with initial value 42 and another int variable named "y" with initial value 12.

In addition to the named arguments, the local int variable "argc" is defined and is assigned the number of arguments actually passed to the function. The variable "argv" is also defined and contains an array filled with copies of all function arguments. The function's body can use these two variables to gain access to extra parameters given in a call of the function, beyond those named in the function's prototype.

When these preparations are complete, the function's body is executed inside its own local namespace. If the function body executes a return statement, the value used in the statement becomes the result of the function call expression. If the function does not

explicitly return a value, a void value is automatically generated. The local namespace of the function is then destroyed, which frees all local variables, including the values of the function arguments.

The following are examples of function call expressions:

```
printf("Hello World!\n");
array_merge(a, b ,c);
versions()
my_func(12, "foo", 42);
```

The above rules mean that function arguments are passed to the function as copies. For example, consider the function call:

```
foo(a, b)
```

When this function call is evaluates, the variables "a" and "b" are referenced and copies of their current values are passed to the body of the function "foo". No matter what the function does with its argument values, the values of the variables "a" and "b" as stored in the namespace outside of the function's body are not changed.

2.5.6.1 Passing arguments "by reference"

As detailed in the last section, function arguments are normally passed into function bodies as copies. Even if the argument expressions are variable reference, a function body cannot manipulate the variables themselves.

However, there is a special syntax for passing argument expressions to a function that makes it possible for the function's body to influence the value of variables that are used as arguments. It consists of placing an ampersand before variable reference expressions or indexed variable reference expressions that are used as function arguments. This is called passing "by reference", though Arena does not exactly use references for this construct (the method that Arena uses is called "copy-retract" or "copy-in copy-out").

When a function call using this syntax is evaluated, the normal function call semantics as described in the last section are in effect. However, when the function's body finishes executing, the language tries to update the values of all arguments that were passed "by reference". This is best explained by an example. Consider the following function body:

```
void swap(mixed a, mixed b)
{
   c = a;
   a = b;
```

```
    b = c;
}
```

For example, this function might be called like this:

```
swap(&x, &y);
```

During the function call, the values of the variables "x" and "y" are available inside the function's body as local variables "a" and "b" (copy-in). When the function's code has been executed, the language checks whether the local variable "a" is still defined. If yes, its value is copied into the variable "x" outside the function. The same happens for local variable "b" and "y" outside the function (copy-out). The order given here is for explanatory purposes. The language takes care that the copy-out actions happen atomically with regard to each other – from the script's point of view, all copy-out actions look as if they happen at exactly the same time. For example, the above example function might be called like this:

```
swap(&i, &a[i])
```

In this case, the array index used for the update of the second variable will always be the same one that was used for the actual argument value passed into the function, even if the function changes its first argument.

If the same variable is passed into a function twice or more using "by reference" passing more than once, the value of the variable after the function call is implementation-defined.

Note that passing "by reference" only works for arguments named in the called function's prototype. It does not work for arguments accessed via the special "argv" array.

2.5.7 Basic rules for structure templates

Structure templates are used to construct values of the struct datatype. This process is called creating an instance of the template. Another use of a template is to use a static reference, which means accessing something inside the structure template without actually creating an instance.

In both cases, the language needs to create concrete versions of the abstract definitions given in the template. This happens as follows: a new local namespace is created. Inside this namespace, the definitions given in the template are executed. Field definitions with values are executed like assignment expressions. Field definitions without values are executed like assignment expressions that assign a void value. Method definitions are executed as normal. The result is a local namespace that contains all fields and methods from the template with their default values.

If a template extends another template, the process above is used recursively, depth-first. This means the chain of templates extending each other is searched until a template that does not extend another is found. The definitions from that template are evaluated first, followed by those in the template that extends the first one, and so on until the definitions from the template that started the process are evaluated. This means definitions in a template can override all fields and methods from another template that it extends.

If the process was used to create a struct value, the completed local namespace is then used to populate the new struct value. If the process was used for evaluating a static reference, the referenced member is copied and the namespace discarded.

2.5.8 Constructor calls

Constructor call expressions are used to create struct values from structure templates. A constructor call consists of the keyword "new", followed by an identifier naming a template, followed by a comma-separated list of argument expressions enclosed in parenthesis. The argument list is allowed to be empty.

When a constructor call expression is evaluated, the identifier is used to look for a structure template definition in the local and global namespace. It is a fatal error if none is found. If the template is found, the initial values of a new struct value are computed as described under "Basic rules", above.

If a constructor method is defined in the template, it is called using the argument expressions given as arguments in the constructor call expression. If the template itself does not define a constructor method but a template it extends does, the constructor of the parent template is called instead. Consider this example:

```
template foo
{
  void foo()
  {
    print("this is foo\n");
  }
}
template bar extends foo
{
  i = 12;
}
```

When a constructor call is evaluated for template "bar", the constructor method defined in the "foo" template will be called. Note that it is legal for there to be no constructor method to call at all.

Normal argument type checks take place for constructor methods. Using an incorrect

number of arguments or arguments of unsuitable types results in a fatal error. Values returned from a constructor (by use of a return statement) are discarded.

During execution of the constructor method, a special local variable "this" is defined. It contains a copy of the struct value that is being constructed. It behaves like a function argument passed "by reference", meaning the constructor method's body can use it to access and change elements in the struct value that is the result of the whole constructor call expression.

Note that the argument expressions given in the constructor call expression are only evaluated when a constructor method is actually called. If no constructor method is defined, the argument expressions are not evaluated.

At the end of the evaluation of a constructor call expression, an additional element called "__template" is added to the new struct value. It contains a string value with the name of the template used to create the struct value.

An example. The following structure template contains a constructor method that will set an field called "i" to the value of the first argument used in the constructor call expression:

```
template foo
{
  void foo(int x)
  {
    this.i = x;
  }
}
```

The above example can be used in a constructor call expression like this:

```
new foo(12)
```

The result is a value of type struct. This value will have three elements: a field called "i" with the int value 12, a method called "foo", and a field called "__template" that contains the string value "foo".

2.5.9 Method calls

A method call works like a normal function call, but refers to a function defined by a structure template or contained in a struct value.

The conventions for argument evaluation, type checks and namespaces are the same as for function calls, described above.

2.5.9.1 Static method calls

A static method calls is used to call a function defined in a structure template. It consists of an identifier naming a template, followed by the characters "::" (double colon), followed by another identifier naming the method, followed by an argument list of expressions in parenthesis.

It is a fatal error if the template named by the first identifier is not defined in the current namespace. It is also a fatal error if the named template does not contain, either directly or via inheritance from an extended template, a method with the name given by the second identifier.

The following are examples of static method calls:

```
foo::bar(1, 2, 3)
input::check("foo", false)
login::logout()
```

2.5.9.2 Dynamic method calls

A dynamic method call is used to call a method contained in a struct value. It consists of appending a single period, followed by an identifier and an argument list of expressions in parenthesis, to some other expression that results in a struct value.

If a method call is appended to a non-struct value or the named method does not exist in the struct value, a fatal error is generated.

If the method exists and the arguments are compatible with its prototype, the method's body is called as described for normal functions. A special local variable called "this" is also defined and contains a copy of the struct that contains the called method. This variable can be used to access fields and methods stored in the same struct value. Changes to the variable "this" will be copied into the real struct variable (if any) when the method body is finished executing.

The following are examples of dynamic method calls (the last is a method call applied to the result of a previous constructor call):

```
foo.bar()
registry[512].files.destroy(2)
new foo().something("foo", 42)
```

2.5.10 Operators

Operators work a lot like functions, but instead of names and argument lists they consist of an operator symbol applied to one or more other expressions. Which other expressions are combined by the operator depends on the kind of operator, as described next.

A prefix operator expression affects a single inner expression and consists of the operator symbol prefixed to another expression.

An infix operator expression affects two inner expressions and consists of the operator symbol written between the two other expressions.

A postfix operator expression affects a single inner expression and consists of the operator symbol suffixed to another expression.

Operators work on different types of expressions. All operators automatically cast the values of their argument expression to a type appropriate to the operator, as described below for different kinds of operators.

Not all operators evaluate all of their argument expressions. The rules for evaluation are also described below.

2.5.10.1 Math operators

Math operators are used to represent arithmetic operations. They work with values of types int and float.

A math operator always evaluates all its argument expressions. If at least one of the argument expressions results in a float value, both values are cast to float before use. Otherwise both values are cast to int.

There is only a single math prefix operator. It uses the operator symbol "-" (minus sign) and denotes negation of the value of the argument expression.

The following table lists the infix math operators and their respective meanings.

```
+     addition
-     subtraction
*     multiplication
/     division
%     remainder
**    exponentiation
```

If the result of a math operator expression falls outside of the domain of the type of its arguments (after casting), the result is an undefined value of the same type as the argument values.

The following are examples of math operator expressions:

```
-12
1 + 2
1.2 * 5
2 ** 10
```

2.5.10.2 Boolean operators

Boolean operators are used to represent logic computations on truth values. When a boolean operator computes the value of one of its argument expressions, the result is always cast to bool.

The prefix operator "!" (exclamation mark) denotes logical negation. It always computes the value of its argument expression.

The infix operator "||" (double vertical bar) denotes logical disjunction ("or"). It always evaluates its first, left argument expression. If the result is the value "true", the result of the whole expression is also "true" and the second argument expression is not evaluated. Otherwise, the second argument expression is evaluated and its bool value is the result of the whole expression.

The infix operator "&&" (double ampersand) denotes logical conjunction ("and"). It always evaluates its first, left argument expression. If the result is the value "false", the result of the whole expression is also "false" and the second argument expression is not evaluated. Otherwise, the second argument expression is evaluated and its bool value is the result of the whole expression.

The following are examples of boolean operator expressions:

```
!failed
x && y
(x || y) && !z
```

2.5.10.3 Equality operators

Equality operators are used to compare values for equality. The two equality operators always evaluate both their argument values. No casting of the resulting values takes place.

If both arguments to an equality operator are of type array, struct, or resource, the result of the equality operator expression is implementation-defined.

The operator "==" (double equals sign) denotes an equality test. The value of the whole expression is "true" if both argument values are of the same type and represent the same value of that type. Otherwise the value of the whole expression is "false".

For values of type fn, two values are considered equal if and only if they refer to the

same function body.

The operator "!=" (exclamation mark followed by equals sign) denotes an inequality test. The value of the whole expression is "true" if the argument values are of different types or do not represent the same value if they are of the same type. Otherwise the value of the whole expression is "false".

The following are examples of equality operator expressions:

```
1 != 2
x == "foo"
divisor != 0.0
```

2.5.10.4 Order operators

Order operators are used to compare the ordering of two values with respect to each other. An order operator always evaluates both of its argument expressions. If only one of the values is a literal constant, the other value is cast to the same type. Otherwise, the second value is cast to the type of the first value (the first value is the one produced by the argument expression on the left of the operator symbol).

Possible result values of an order operator expression are "true" and "false", depending on whether the ordering the expression checks for is present for the argument values.

Ordering of void values is always "false" by convention since there is only one value in the datatype.

Ordering of bool values is such that the value "false" is smaller than "true", but not equal.

Ordering of int values is the same as for whole numbers in mathematics.

Ordering of float values is the same as for rational numbers in mathematics.

Ordering of string values is such that the bytes forming the string are compared from left to right, interpreting them as numbers in the range 0-255. The comparison stops as soon as one of the bytes is smaller or larger than the other one. The string with larger byte is considered to be larger than the other. If both bytes are the same, the comparison moves on to the next byte in both strings. If this process reaches the end of exactly one of the strings, that string is considered to be the smaller of the two. If the process reaches the end of both strings at the same time, the strings are considered equal.

Ordering of array, struct, fn, and resource values is implementation-defined.

The following table lists all order operators and the condition that they check for.

```
<    left value smaller than right value
```

```
>     left value larger than right value
<=    left value smaller or equal to right value
>=    left value larger or equal to right value
```

The following are examples of order operator expressions:

```
a < b
x >= 10
epsilon < 0.01
```

2.5.10.5 Bitwise operators

Bitwise operators are used to manipulate bits in int values. A bitwise operator always evaluates all of its argument expressions and casts their values to int.

The prefix operator "~" (tilde) denotes bitwise negation of its argument value.

The prefix operator "++" (double plus sign) returns the value of its argument expression increased by one. If the argument is a reference expression or indexed reference expression, the increased value is also stored in the namespace in the same place that the original value was obtained from.

The prefix operator "--" (double minus sign) returns the value of its argument expression decreased by one. If the argument is a reference expression or indexed reference expression, the decreased value is also stored in the namespace in the same place that the original value was obtained from.

The infix operator "|" (vertical bar) computes the bitwise "or" of its argument values. This means bits set in either of the argument values will be set in the result value.

The infix operator "&" (ampersand) computes the bitwise "and" of its argument values. This means only bits set in both the argument values will be set in the result value.

The infix operator "^" (caret) computes the bitwise "exclusive or" of its argument values. This means only bits set in exactly one of the argument values will be set in the result value.

The postfix operator "++" (double plus sign) returns the value of its argument expression. In addition, if the argument expression is a reference or indexed reference expression, the value stored in the namespace is increased by one. The previous value is returned as result of the whole expression.

The postfix operator "--" (double minus sign) returns the value of its argument expression. In addition, if the argument expression is a reference or indexed reference expression, the value stored in the namespace is decreased by one. The previous value is returned as result of the whole expression.

The following are examples of bitwise operator expressions:

```
i++
flags & 0x40
x ^ y
--refcount
```

2.5.10.6 Operator precedence

If multiple operators occur in one expression, the order in which they are evaluated depends on the relative precedence of the two operators. Operators with higher precedence are evaluated first.

If the same operator occurs multiple times in an expression, the order of evaluation depends on the associativity of the operator. If the operator is left-associative, it is evaluated so that applications proceed from left to right. For a right-associative operator, applications proceed from right to left.

To change the order of evaluation or to use more than one instance of a non-associative operator in a single expression, the programmer can enclose subexpressions in parenthesis. Expressions inside parenthesis are evaluated first, independent of any operators outside the parenthesis.

The following table lists all operator symbols. Operators listed at the top have lower precedence than those listed below them. Operators listed on the same line have the same precedence. Associativity is given on the same line as the operator symbols it applies to.

```
Associativity    Operators
right            = += -= *= /= |= &= ^= <<= >>=
none             ?
right            ||
right            &&
right            !
none             == != < <= > >=
left             & | ^
left             + - (infix)
left             * / %
right            **
left             << >>
left             ~ - (postfix)
left             ++ --
```

Casts have higher precedence than any operator and associate to the right.

2.5.11 Conditional expression

A conditional expression is the expression equivalent to an if-else statement. It consists of an expression, followed by a "?" (question mark) character, followed by another expression, followed by a ":" (colon) character, followed by a third expression.

When a conditional expression is evaluated, the value of the first argument expression is evaluated and its result value is cast to bool. If the result is "true", the value of the second expression is evaluated and used as the value of the whole expression. The third expression is not evaluated. If the value of the first expression is "false", the third expression is evaluated and its value used as the value of the whole expression. The second expression is not evaluated in that case.

The following are examples of conditional expressions:

```
x % 2 == 0 ? "even" : "odd"
x ? false : true
```

2.5.12 Source file and line expressions

Source file and line expressions are used to refer to the script they appear in. They are mostly useful for printing error messages annotated with script source code locations.

The expression "__FILE__" is evaluated to a string value that contains the name of the script file that the expression appears in.

The expression "__LINE__" is evaluated to an int value that gives the line number that the expressions appears on, relative to the script file that it appears in.

2.5.13 Anonymous functions

An anonymous function is a function that does not have a name. Such a function cannot be defined by use of a function definition statement since that mandates an identifier to be used as the function's name. Instead, an anonymous function can be constructed by an expression that evaluates to an fn value.

Anonymous functions are useful as arguments to library functions that expect another function as one of their arguments. If the argument function is needed only once, an anonymous function saves the programmer from having to invent a name for the function.

An anonymous function expression consists of a backslash character, followed by an argument list definition in parenthesis, followed by a function body in curly braces. The argument list definition and function body have the same syntax as those found in regular function definitions (see above in the section about statements), minus the return type definition.

When an anonymous function expression is evaluated, the result is an fn value which contains a function with the prototype and function body given in the anonymous function expression. Note that the return type is not given in the expression; it is assumed to be "mixed".

The following are examples of anonymous function expressions:

```
\ (x) { return 2 * x; }
\ (float x) { return sin(x) + 1.0; }
\ (forced int x, forced int y) { return x + y; }
```

Anonymous functions are sometimes called "lambda functions". The choice of the backslash character for leading an anonymous function expression is influenced by this (since it is the ASCII character most similar to a Greek lambda) and was in fact stolen from the functional programming language Haskell.

3 Library

The following sections describe the functions provided by the Arena standard library. Functions are grouped into sections of functions which work alike or on the same datatypes.

Most of the library was inspired by the ANSI C standard library, with some influences from Haskell and PHP.

Each function is prefixed by its prototype, which is followed by a short explanation of what the function does and what its return values are. Note that no standard library function modifies any of its arguments, so it makes no difference whether arguments are passed "by reference" or normally.

This version of the library manual describes version 2.5 of the library.

3.1 Runtime system

The runtime system library functions provide ways to deal with aspects of the runtime system of the language. For example, since the types of variables are dynamic, there are functions for checking the type of values.

This part of the library also contains pre-defined variables that contain information about the precision and possible values of the float and int types.

3.1.1 FLT_RADIX

```
FLT_RADIX
```

The int variable FLT_RADIX is automatically set by the library to contain the radix used for the representation of float values. This is normally 2, meaning binary representation.

3.1.2 FLT_DIG

```
FLT_DIG
```

The int variable FLT_DIG is automatically set by the library to contain the precision of float values, measured in decimal digits.

3.1.3 FLT_MANT_DIG

```
FLT_MANT_DIG
```

The int variable FLT_MANT_DIG is automatically set by the library to contain the number of digits, base FLT_RADIX, that form the mantissa of a float value.

3.1.4 FLT_MAX_EXP

```
FLT_MAX_EXP
```

The int variable FLT_MAX_EXP is automatically set by the library to contain the largest positive integer exponent to which FLT_RADIX can be raised and remain representable as a float value.

3.1.5 FLT_MIN_EXP

```
FLT_MIN_EXP
```

The int variable FLT_MIN_EXP is automatically set by the library to contain the smallest negative integer exponent to which FLT_RADIX can be raised and remain representable as a float value.

3.1.6 FLT_EPSILON

```
FLT_EPSILON
```

The float variable FLT_EPSILON is automatically set by the library to contain the smallest float value that can be added to 1.0 so that the result is a float value different from 1.0.

3.1.7 FLT_MAX

```
FLT_MAX
```

The float variable FLT_MAX is automatically set by the library to contain the largest number that can be represented by a float value.

3.1.8 **FLT_MIN**

```
FLT_MIN
```

The float variable FLT_MIN is automatically set by the library to contain the smallest number that can be represented by a float value.

3.1.9 **INT_MAX**

```
INT_MAX
```

The int variable INT_MAX is automatically set by the library to contain the maximum value that an int variable can hold.

3.1.10 **INT_MIN**

```
INT_MIN
```

The int variable INT_MIN is automatically set by the library to contain the minimum value that an int variable can hold.

3.1.11 **type_of**

```
string type_of(mixed x)
```

The type_of function returns a string containing the type of the argument value "x". It can return "void", "bool", "int", "float", "string", "array", "struct", "fn", or "resource".

3.1.12 **tmpl_of**

```
mixed tmpl_of(mixed x)
```

The tmpl_of function returns the name of the template that the value "x" was created from, if "x" is a struct and has been constructed from a structure template. If not, the tmpl_of function returns void.

3.1.13 **is_void**

```
bool is_void(mixed x, ...)
```

The is_void function returns true if all of its arguments are values of type void. It returns false otherwise.

3.1.14 is_bool

```
bool is_bool(mixed x, ...)
```

The is_bool function returns true if all of its arguments are values of type bool. It returns false otherwise.

3.1.15 is_int

```
bool is_int(mixed x, ...)
```

The is_int function returns true if all of its arguments are values of type int. It returns false otherwise.

3.1.16 is_float

```
bool is_float(mixed x, ...)
```

The is_float function returns true if all of its arguments are values of type float. It returns false otherwise.

3.1.17 is_string

```
bool is_string(mixed x, ...)
```

The is_string function returns true if all of its arguments are values of type string. It returns false otherwise.

3.1.18 is_array

```
bool is_array(mixed x, ...)
```

The is_array function returns true if all of its arguments are values of type array. It returns false otherwise.

3.1.19 is_struct

```
bool is_struct(mixed x, ...)
```

The is_struct function returns true if all of its arguments are values of type struct. It returns false otherwise.

3.1.20 is_fn

```
bool is_fn(mixed x, ...)
```

The is_fn function returns true if all of its arguments are values of type fn. It returns false otherwise.

3.1.21 is_resource

```
bool is_resource(mixed x, ...)
```

The is_resource function returns true if all of its arguments are values of type resource. It returns false otherwise.

3.1.22 is_a

```
bool is_a(mixed x, string type)
```

The is_a function returns true if the argument "x" has type "type", using the same type names as given for the type_of function. Additionally, the "type" argument can also be the name of a structure template. In this case the is_a function checks whether "x", which must be struct value, was constructed from the named template or a template directly or indirectly extending the named template. The is_a function returns false if the value "x" does not have the indicated type.

Note that the meaning of is_a for structure templates is only guaranteed to work reliably as long as all needed templates are still visible and have not been overwritten since the struct value in question had been constructed.

3.1.23 is_function

```
bool is_function(string name)
```

The is_function function returns true if the argument "name" refers to a function defined in the local or global namespace. It returns false otherwise.

3.1.24 is_var

```
bool is_var(string name)
```

The is_var function returns true if the argument "name" refers to a variable defined in the local or global namespace. It returns false otherwise.

3.1.25 is_tmpl

```
bool is_tmpl(string name)
```

The is_tmpl function returns true if the argument "name" refers to a structure template defined in the local or global namespace. It returns false otherwise.

3.1.26 is_local

```
bool is_local(string name)
```

The is_local function returns true if the argument "name" refers to an entity defined in the current local namespace. It returns false otherwise.

3.1.27 is_global

```
bool is_global(string name)
```

The is_global function returns true if the argument "name" refers to an entity defined in the global namespace. It returns false otherwise.

3.1.28 cast_to

```
mixed cast_to(mixed x, string type)
```

The cast_to function returns a cast of the value "x" to the datatype given as "type", using the same names as returned by the type_of function. It is a fatal error to pass an unknown type.

3.1.29 set

```
bool set(string name, mixed val)
```

The set function sets the variable named by the argument "name" to the value given by the argument "val". It returns true if the operation is successful or false if the name is not a valid variable name.

3.1.30 get

```
mixed get(string name)
```

The get function returns the value of the variable named by the "name" argument. If the name is not a valid variable name or no variable of that name is defined, the get function returns void.

3.1.31 get_static

```
mixed get_static(string tmpl, string name)
```

The get_static function returns a value from a structure template, using "tmpl" as the template name and "name" as the name of the struct element to return. It returns void if the named structure template does not exists or does not have a member of the given name.

3.1.32 unset

```
void unset(forced string name, ...)
```

The unset function removes entities from the current local namespace, using all its arguments as entity names. Note that removing names from the local namespace will make entities from the global namespace visible again if they were obscured by the local names. Global entities can only be removed by a call to unset from top-level scope.

Note that at top-level scope, it is possible to unset standard library variables and functions.

3.1.33 global

```
void global(forced string name, ...)
```

The global function uses all its arguments as variable names. It copies the named variables from the current local namespace to the global namespace.

3.1.34 assert

```
void assert(forced bool x, ...)
```

The assert function casts all its argument values to type bool and then exits the running script with an "assertion failure" message if at least one of the expressions yields a false value. If all input values are true, the assert function does nothing.

This function is mainly used for debugging and input sanity checking.

3.1.35 versions

```
struct versions()
```

The versions function accepts no arguments and returns a struct containing version numbers for the language and standard library used by the current language implementation. At least the following fields are defined:

```
v_language_major     Language major version
v_language_minor     Language minor version
v_library_major      Library major version
v_library_minor      Library minor version
```

Additional fields are allowed to be present and are implementation-defined. Future versions of the standard library are guaranteed to always define a versions functions with the behavior defined above.

3.2 Math functions

The math functions mostly work on float values and perform mathematical computations.

3.2.1 exp

```
float exp(forced float x)
```

The exp function computes the exponential of its input value.

3.2.2 log

```
float log(forced float x)
```

The log function computes the natural logarithm of its input value.

3.2.3 log10

```
float log10(forced float x)
```

The log10 function computes the base 10 logarithm of its input value.

3.2.4 sqrt

```
float sqrt(forced float x)
```

The sqrt function computes the square root of its input value.

3.2.5 ceil

```
float ceil(forced float x)
```

The ceil function computes the smallest non-fractional number that is larger than or equal to the input value.

3.2.6 floor

```
float floor(forced float x)
```

The floor function computes the largest non-fractional number that is smaller than or equal to the input value.

3.2.7 fabs

```
float fabs(forced float x)
```

The fabs function computes the absolute value of its input value.

3.2.8 sin

```
float sin(forced float x)
```

The sin function computes the sine of its input value.

3.2.9 cos

```
float cos(forced float x)
```

The cos function computes the cosine of its input value.

3.2.10 tan

```
float tan(forced float x)
```

The tan function computes the tangent of its input value.

3.2.11 asin

```
float asin(forced float x)
```

The asin function computes the arc-sine of its input value.

3.2.12 acos

```
float acos(forced float x)
```

The acos function computes the arc-cosine of its input value.

3.2.13 atan

```
float atan(forced float x)
```

The atan function computes the arc-tangent of its input value.

3.2.14 sinh

```
float sinh(forced float x)
```

The sinh function computes the hyperbolic sine of its input value.

3.2.15 cosh

```
float cosh(forced float x)
```

The cosh function computes the hyperbolic cosine of its input value.

3.2.16 tanh

```
float tanh(forced float x)
```

The tanh function computes the hyperbolic tangent of its input value.

3.2.17 abs

```
int abs(forced int x)
```

The abs function computes the absolute value of its input value.

3.3 Printing functions

The printing functions can be used to provide human-readable output from a script.

3.3.1 print

```
void print(mixed x, ...)
```

The print function casts all of its arguments to string and outputs the resulting strings end-to-end, without intervening whitespace.

3.3.2 dump

```
void dump(mixed x, ...)
```

The dump function outputs a dump of all its argument values. For each value, the type and a string representation of the actual value are printed. The exact formatting of the

output is implementation-defined.

This function is mainly intended for debugging purposes.

3.3.3 sprintf

```
string sprintf(string fmt, ...)
```

The sprintf function takes a format string fmt and additional arguments that are formatted according to the format string. The format string consists of normal characters and conversion specifiers. A conversion specifier starts with the character "%" (percent sign), followed by optional conversion flag characters, followed by an optional field width, followed by an optional precision, followed by a type specifier. For each conversion specifier, one additional argument should be passed into the sprintf function. Missing values are filled up with void values. Each conversion specifier consumes one argument, which is then formatted according to the specifier. The return value of sprintf is a string with all non-specifier characters from the format string copied over and all conversion specifiers replaced by the result of formatting their corresponding argument.

The following conversions flag characters are defined:

```
0       zero-pad the value
-       left-adjust the value
        (space) put a blank before positive numbers
+       always but a sign before a number
```

The field width specifies how many characters the conversion result will use as a minimum. If the field width is larger than needed, the default is to right-adjust the value using space characters. If the field width is smaller than needed, the result will still be as wide as necessary for the printed value.

The precision, if present, needs to be given as a "." (period) character followed by an optional decimal digit string. For integer conversions, the precision specifies the minimum number of digits to print. For float conversions, the precision specifies how many digits to print after the decimal dot. For string conversions, the precision specifies the maximum number of characters to print from the string. If the decimal digit string is missing, the precision is taken to be 0 (zero).

The type specifier, which must be present, defines which type of argument value is expected by the conversion. If the actual sprintf argument does not have the expected type, a cast to that type is generated before use. The following type specifiers are defined:

```
d      int, print in decimal
i      int, print in decimal
f      float
o      int, print in octal
s      string
x      int, print in lower-case hexadecimal
X      int, print in upper-case hexadecimal
```

It is possible to include a percent character in the output by using the special conversion specifier "%%" (double percent sign).

The result of passing an invalid or malformed conversion specifier into sprintf is undefined.

To give an example, the following sprintf invocation assigns the string "00012.30 is here" to the variable "x":

```
x = sprintf("%08.2f is here", 12.3);
```

3.3.4 printf

```
void printf(string fmt, ...)
```

The printf function works like the sprintf function, but instead of returning a formatted string, it directly outputs the formatted string.

3.4 String functions

The string functions provide ways to manipulate strings and check some properties of strings.

3.4.1 strlen

```
int strlen(forced string x)
```

The strlen function returns the number of bytes contained in its argument string.

3.4.2 strcat

```
string strcat(mixed x, ...)
```

The strcat function converts all its arguments to string and returns a string containing the concatenation of the resulting string values.

3.4.3 **strchr**

```
mixed strchr(forced string hay, forced string needle)
```

The strchr function searches for the leftmost occurrence of the first character of the "needle" argument in the "hay" argument. Character positions are counted from 0 (zero). If the character is not found, void is returned.

3.4.4 **strrchr**

```
mixed strrchr(forced string hay, forced string needle)
```

The strrchr function searches for the rightmost occurrence of the first character of the "needle" argument in the "hay" argument. Character positions are counted from 0 (zero). If the character is not found, void is returned.

3.4.5 **strstr**

```
mixed strstr(forced string hay, forced string needle)
```

The strstr function searches for the leftmost occurrence of the string "needle" inside the string "hay". Character positions are counted from 0 (zero). If the substring is not found, void is returned.

3.4.6 **strspn**

```
int strspn(forced string hay, forced string set)
```

The strspn function returns the number of leading characters of the string "hay" that are contained in the set of characters defined by the string "set".

3.4.7 **strcspn**

```
int strcspn(forced string hay, forced string set)
```

The strspn function returns the number of leading characters of the string "hay" that are not contained in the set of characters defined by the string "set".

3.4.8 strpbrk

```
mixed strpbrk(forced string hay, forced string set)
```

The strpbrk function returns the position of the first character in the string "hay" that is contained in the set of characters defined by the string "set". Character positions are counted from 0 (zero). If no matching character is found, void is returned.

3.4.9 strcoll

```
int strcoll(forced string a, forced string b)
```

The strcoll function compares the strings "a" and "b" according to the current locale (language) settings of the operating system. It returns a negative int value if "a" is found to be smaller than "b". It returns 0 (zero) if "a" and "b" are found to be equal. It returns a positive int value if "a" is found to be larger than "b".

3.4.10 tolower

```
string tolower(forced string x)
```

The tolower function returns a copy of its input string with all upper-case letters converted to lower case.

3.4.11 toupper

```
string toupper(forced string x)
```

The toupper function returns a copy of its input string with all lower-case letters converted to upper case.

3.4.12 isalnum

```
bool isalnum(forced string x)
```

The isalnum function returns true if its input string contains only alphanumeric charac-ters. This means only letters and decimal digits are allowed. It returns false otherwise.

3.4.13 isalpha

```
bool isalpha(forced string x)
```

The isalpha function returns true if its input string contains only letters of the alphabet. It returns false otherwise.

3.4.14 iscntrl

```
bool iscntrl(forced string x)
```

The iscntrl function returns true if its input string contains only control characters; that is characters with an ASCII value below 32. It returns false otherwise.

3.4.15 isdigit

```
bool isdigit(forced string x)
```

The isdigit function returns true if its input string contains only decimal digits. It returns false otherwise.

3.4.16 isgraph

```
bool isgraph(forced string x)
```

The isgraph function return true if its input string contains only graphical characters; that is no control characters and no spaces. It returns false otherwise.

3.4.17 islower

```
bool islower(forced string x)
```

The islower function returns true if its input string contains only lower-case letters. It return false otherwise.

3.4.18 isprint

```
bool isprint(forced string x)
```

The isprint function returns true if its input string contains only printable characters; that is no control characters. It returns false otherwise.

3.4.19 ispunct

```
bool ispunct(forced string x)
```

The ispunct function returns true if its input string contains only punctuation characters; that is no control characters, no spaces, and no alphanumeric characters. It returns false otherwise.

3.4.20 isspace

```
bool isspace(forced string x)
```

The isspace function returns true if its input string contains only whitespace characters. Whitespace characters are space, form-feed, newline, carriage return, horizontal tab, and vertical tab. The isspace function returns false if other characters are found in the input string.

3.4.21 isupper

```
bool isupper(forced string x)
```

The isupper function returns true if its input string contains only upper-case letters. It returns false otherwise.

3.4.22 isxdigit

```
bool isxdigit(forced string x)
```

The isxdigit function returns true if its input string contains only hexadecimal digits. It returns false otherwise.

3.4.23 substr

```
string substr(forced string x, int pos)
```

The substr function, when called with two arguments, returns a substring of the string "x" starting at character position "pos". Character positions are numbered starting from 0 (zero). If the position exceeds the number of characters in the input string, an empty string is returned.

```
string substr(forced string x, int pos, int max)
```

When called with three arguments, the "max" argument is used as a maximum length for the returned substring. Excess characters from the original string are not copied.

3.4.24 left

```
string left(forced string x, int max)
```

The left function returns the leftmost "max" characters from the string "x", or less if the string "x" does not have enough characters.

3.4.25 right

```
string right(forced string x, int max)
```

The right function returns the rightmost "max" characters from the string "x", or less if the string "x" does not have enough characters.

3.4.26 ord

```
mixed ord(forced string x)
```

The ord function returns the ASCII code of the first character of the string "x" as an int value. If "x" is an empty string, void is returned. If the first character of "x" is not a 7-bit ASCII character, the result is implementation-defined; however, the result must be compatible with the chr function.

3.4.27 chr

```
string chr(int x)
```

The chr function returns a one-character string containing the character with the ASCII value "x". If the input value does not describe a 7-bit ASCII character, the result is implementation-defined; however, the result must be compatible with the ord function.

3.4.28 explode

```
array explode(forced string input)
```

The explode function converts the input string to an array, so that the resulting array has one element for each character of the original string. Each element will contain one original character in the form of a single-character string.

3.4.29 implode

```
string implode(array input)
```

The implode function converts the input array into a string. Each element of the input array is converted to a string and the concatenation of all these is the return value of the implode function.

The implode function can be used to reverse the effects of the explode function.

3.4.30 ltrim

```
string ltrim(forced string input)
```

The ltrim function returns a copy of the input string with all leading whitespace characters removed.

3.4.31 rtrim

```
string rtrim(forced string input)
```

The rtrim function returns a copy of the input string with all trailing whitespace characters removed.

3.4.32 trim

```
string trim(forced string input)
```

The trim function returns a copy of the input string with all leading and trailing whitespace characters removed.

3.5 Array functions

The array functions are used to construct and manipulate array values.

3.5.1 mkarray

```
array mkarray(mixed x, ...)
```

The mkarray function creates an array that contains all the argument values. It is allowed to call mkarray with no arguments at all, in which case the returned array value is an empty array.

3.5.2 qsort

```
array qsort(array x)
```

The qsort function returns a sorted copy of the input array. Element values are compared as if the "smaller or equal" operator were used. The resulting order is implementation-defined if the array contains values of different types.

3.5.3 is_sorted

```
bool is_sorted(array x)
```

The is_sorted function returns true if the input array is sorted, according to the same condition that the qsort function uses to sort an array. The result is false if the input array is not sorted.

3.5.4 array_unset

```
array array_unset(array x, int index)
```

The array_unset returns a copy of the input array "x", with the element at position "index" replaced by a void value. The number of elements in the array or their indices

do not change.

3.5.5 **array_compact**

```
array array_compact(array x)
```

The array_compact function returns a copy of the input array with all void values removed. The result is an array with the minimum size needed to hold all non-void values from the input array.

3.5.6 **array_search**

```
mixed array_search(array hay, mixed needle)
```

The array_search function searches the input array "hay" for an element matching the value "needle", using the same rules as the "==" operator. It returns the integer index of the matching element or void if no matching element is found.

3.5.7 **array_merge**

```
mixed array_merge(mixed x, ...)
```

The array_merge function casts all its arguments to array and then merges all these into a single large array value that is returned. The merged array first contains all the elements from the first argument array in their original order, followed by the elements from the second array, and so on until all arguments are consumed.

3.5.8 **array_reverse**

```
array array_reverse(array x)
```

The array_reverse function returns a copy of the input array "x" with the order of the elements reversed.

3.6 List functions

The list functions provide a way to work with arrays that makes them similar to the list datatypes found in functional languages such as Standard ML or Haskell.

3.6.1 nil

```
array nil()
```

The nil function returns an empty array.

3.6.2 cons

```
array cons(mixed head, array tail)
```

The cons function prepends the element "head" to the array "tail" and returns the resulting array.

3.6.3 length

```
int length(array list)
```

The length function returns the number of elements that are present in the input array.

3.6.4 null

```
bool null(array list)
```

The null function returns true if the input array is empty. It returns false otherwise.

3.6.5 elem

```
bool elem(array list, mixed search)
```

The elem function returns true if the input array "list" contains an element that equals the "search" argument, using the same rules as the equality (==) operator. The elem function returns false if no matching element is found.

3.6.6 head

```
mixed head(array list)
```

The head function returns the first element of the array "list". If the input array is empty, the head function returns a void value.

3.6.7 tail

```
array tail(array list)
```

The tail function returns a copy of the input array with the first element removed.

3.6.8 last

```
mixed last(array list)
```

The last function returns the last element of the input array, or a void value if the input array is empty.

3.6.9 init

```
array init(array list)
```

The init function returns a copy of the input array with the last element removed.

3.6.10 take

```
array take(array list, int count)
```

The take function returns an array that contains the first "count" elements from the input array "list". Less elements are returned if the input array is not large enough.

3.6.11 drop

```
array drop(array list, int count)
```

The drop function returns a copy of the input array "list" with the first "count" elements removed. If the input array contains less than "count" elements, an empty array is returned.

3.6.12 intersperse

```
array intersperse(array list, mixed elem)
```

The intersperse function inserts a copy of the value "elem" between all the elements of the input array "list". If the input array is empty or contains exactly one element, no new elements are inserted.

3.6.13 replicate

```
array replicate(mixed elem, int count)
```

The replicate function returns an array of "count" elements, each containing the value "elem". If the given count is zero or negative, an empty array is returned.

3.7 Structure functions

The structure functions are used to construct, inspect, and manipulate struct values.

3.7.1 mkstruct

```
struct mkstruct(mixed key, mixed val, ...)
```

The mkstruct function constructs a struct value from its argument values. The first argument is cast to string and used as a field name. The value for that field is taken from the second argument. The same principle is used for the remaining arguments. If the number of arguments is odd, a void value is used as value for the last field.

3.7.2 struct_get

```
mixed struct_get(struct x, string field)
```

The struct_get function returns the value of the field named "field" in the struct value "x". It returns void if the field name does not exist in the struct value.

3.7.3 struct_set

```
struct struct_set(struct x, string field, mixed val)
```

The struct_set function adds a field called "field" to the input struct "x". The new field value is given by the "val" argument. The modified struct value is returned.

3.7.4 struct_unset

```
struct struct_unset(struct x, string field)
```

The struct_unset function returns a copy of the input struct value "x", with the field named "field" removed from the struct. It is not an error if no such field exists in the input struct.

3.7.5 struct_fields

```
array struct_fields(struct x)
```

The struct_fields function returns an array of strings that contains the names of all fields in the input struct that are not of type fn.

3.7.6 struct_methods

```
array struct_methods(struct x)
```

The struct_methods function returns an array of strings that contains the names of all fields in the input struct that are of type fn.

3.7.7 is_field

```
bool is_field(struct x, string name)
```

The is_field function returns true if the field "name" exists in the input struct "x" and its value is not of type fn. It returns false otherwise.

3.7.8 is_method

```
bool is_method(struct x, string name)
```

The is_method function returns true if the field "name exists in the input struct "x" and its value is of type fn. It returns false otherwise.

3.7.9 struct_merge

```
struct struct_merge(mixed x, ...)
```

The struct_merge function casts all its arguments to struct and then merges the resulting struct values. This is done so that values of the same name in later arguments overwrite the fields from earlier arguments (arguments being considered from left to right, as usual). This function will merge both fields and methods.

3.8 Functions on functions

This group of functions provides ways to deal with values of type fn.

3.8.1 is_builtin

```
bool is_builtin(fn f)
```

The is_builtin function returns true if the given function has been defined by the language implementation itself, for example as part of the standard library. It returns false otherwise.

This function can be used to check whether a standard library function has been overwritten by a user-defined function of the same name.

3.8.2 is_userdef

```
bool is_userdef(fn f)
```

The is_userdef function returns true if the given function has been defined by the running script or any of its included files. It returns false otherwise.

3.8.3 call

```
mixed call(fn f, ...)
```

The call function executes a function call. It behaves as if the function "f" had been called with the rest of the arguments as its arguments. It returns the return value of "f".

3.8.4 call_array

```
mixed call_array(fn f, array args)
```

The call_array function executes a function call. It behaves as if the function "f" had been called with the arguments given in the array "args". It returns the return value of "f".

3.8.5 call_method

```
mixed call_method(fn f, struct s, ...)
```

The call_method function executes the function "f" with an additional local variable called "this" that contains a copy of the struct value "s". The remaining arguments are used as arguments to "f". The return value is the return value of "f".

Note that in contrast to a normal method call, any modifications to "this" inside of the function body of "f" are not copied back to the struct "s".

3.8.6 call_method_array

```
mixed call_method_array(fn f, struct s, array args)
```

The call_method_array function executes the function "f" with an additional local variable called "this" that contains a copy of the struct value "s". The array argument "args" is used as arguments to "f". The return value is the return value of "f".

Note that in contrast to a normal method call, any modifications to "this" inside of the function body of "f" are not copied back to the struct "s".

3.8.7 prototype

```
struct prototype(fn f)
```

The prototype function returns a struct describing the prototype of the input function "f". The result struct contains an element "ret" that describes the return value of "f", and an element "args" that describes the expected arguments of "f".

Types are described by using a two-element struct containing the fields "type" and "force". The "type" field contains a type name string as returned by the type_of library function. The "force" field contains bool true if the "forced" keyword was used in the definition of the return value or argument in question. It contains false otherwise.

The "ret" element of the return struct of the prototype function directly contains a type description struct as detailed above. The "args" element contains an array of such descriptions, with the array element at index 0 (zero) corresponding to the first argument of the described fn value.

3.8.8 map

```
array map(fn f, array x, ...)
```

The map function applies the function "f" to each element of the array "x" with the rest of the arguments being passed on to "f". The return values of "f" are collected into a new array which is returned from the map function.

3.8.9 filter

```
array filter(fn f, array x, ...)
```

The filter function applies the function "f" to each element of the array "x" with the rest of the arguments being passed on to "f". The return value of "f" is then cast to bool. If the result is true, the element of "x" in question is copied to the output array.

3.8.10 foldl

```
mixed foldl(fn f, mixed init, array x, ...)
```

The foldl function applies the function "f" to the array "x" – with the rest of the arguments following "x" being passed on to "f" – so that the array is reduced to a single value. The value "init" is appended to the left of the array and the values of the array are processed from left to right, so that "f" is applied as follows, where "x0" means the first element of the input array, "x1" the second element, and "xn" the last element of the input array; "args" denotes the arguments following "x" in the call to foldl:

```
f(f(f(f(init, x0, args), x1, args), ... ), xn, args)
```

If the input array is empty, the argument value "init" is returned. Note that the construction of foldl requires that the function "f" accepts values from the input array as its second argument and its own return value as its first argument. The value "init" must also be acceptable as a first argument to "f".

3.8.11 foldr

```
mixed foldr(fn f, array x, mixed init, ...)
```

The foldr function applies the function "f" to the array "x" – with the rest of the arguments following "init" being passed on to "f" – so that the array is reduced to a single value.

The value "init" is appended to the right of the array and the values of the array are processed from right to left, so that "f" is applied as follows, where "x0" means the first element of the input array, "x1" the second element, and "xn" the last element of the input array; "args" denotes the arguments following "init" in the call to foldr:

```
f(x0, f(x1, f(..., f(xn, init, args), args), args), args)
```

If the input array is empty, the argument value "init" is returned. Note that the construction of foldr requires that the function "f" accepts values from the input array as its first argument and its own return value as its second argument. The value "init" must also be acceptable as a second argument to "f".

3.8.12 take_while

```
array take_while(fn f, array input, ...)
```

The take_while function applies the function "f" to the elements of the array "input" with the rest of the arguments being passed on to "f". The return value is cast to bool. Elements are copied to the output array as long as the result of "f" is true. Copying stops on the first element of "input" that makes "f" return false.

3.8.13 drop_while

```
array drop_while(fn f, array input, ...)
```

The drop_while function applies the function "f" to the elements of the array "input" with the rest of the arguments being passed on to "f". The return value is cast to bool. Elements are skipped as long as the return value of "f" is true. Starting from the first element that makes "f" return false elements are copied to the output array.

3.9 Random number functions

The following functions provide a simple pseudo-random number generator.

3.9.1 RAND_MAX

```
RAND_MAX
```

The int variable RAND_MAX is automatically set by the library to contain the highest integer number that the random number generator can produce.

3.9.2 rand

```
int rand(int min, int max)
```

The rand function generates a random number between the lower bound "min" and the upper bound "max", inclusive. The effect of negative bounds is implementation-defined.

If the lower bound exceeds RAND_MAX, the lower bound is returned. If the upper bound exceeds RAND_MAX, it is clipped to RAND_MAX before use.

If the random number generator was not seeded prior to the first call to rand, a random seed based on the current date and time is automatically generated.

3.9.3 srand

```
void srand(int seed)
```

The srand function seeds the pseudo-random generator with the seed value "seed".

3.10 Environment functions

The environment functions provide a limited way for a script to deal with its execution environment.

3.10.1 argc

```
argc
```

The int variable argc is defined by the library to contain the number of command line arguments passed to the script. The name of the script itself counts as an argument, so this value is always at least 1 (one).

3.10.2 argv

```
argv
```

The array variable argv is defined by the library to be an array of strings containing all the command line arguments of the script. The first element contains the script name.

3.10.3 exit

```
void exit(int status)
```

The exit function aborts execution of the script and reports the integer return value "status" to the operating system. The exit function does not return.

3.10.4 getenv

```
mixed getenv(string name)
```

The getenv function searches the execution environment for an environment variable with the given name. If such an environment variable exists, its value is returned in the form of a string value. If not, void is returned.

3.11 File I/O functions

The following functions provide a way to perform I/O operations on files. Files must be opened before they can be used. Opening results in a file handle that is encapsulated in a resource value.

Each file handle has four basic properties: buffering, file position, error indicator, and end-of-file indicator. Each of these can be queried and set by the functions described below.

When the resource value of a file handle becomes unset or the containing namespace is destroyed, the file handle is closed. This writes all data that may still be buffered in the file handle to disk.

Please note that all functions operate on files in the SDK sandbox (which is /mnt/sdk on the host system).

3.11.1 stdin

```
stdin
```

The variable stdin is defined by the library to be the resource for the standard input stream of the script. Usually this means the user's keyboard, but many operating systems allow input redirection from files.

What kind of buffering is performed on stdin is operating system dependent. Under Unix systems, stdin is normally line-buffered at the terminal driver level, so that it is only possible to read from stdin if the user has already pressed the return key.

3.11.2 **stdout**

```
stdout
```

The variable stdout is defined by the library to be the resource for the standard output stream of the script. Usually this means writing to stdout will cause output to appear on the user's screen. However, most operating systems allow users to redirect stdout to a file.

What kind of buffering is performed on stdout is operating system dependent.

3.11.3 **stderr**

```
stderr
```

The variable stderr is defined by the library to be the resource for the standard error stream of the script. Usually this will be the same stream as stdout, but operating systems often allow the user to redirect this to a file, independent of stdout.

What kind of buffering is performed on stderr is operating system dependent.

3.11.4 **is_file_resource**

```
bool is_file_resource(resource res)
```

The is_file_resource returns true if the resource "res" is a file resource. It returns false otherwise.

3.11.5 **fopen**

```
mixed fopen(string name, string mode)
```

The fopen function tries to open the file identified by "name" with the I/O mode given by "mode". If the operation is successful, a file handle resource is returned. If an error occurs or the given mode is invalid, void is returned. The following modes are defined:

```
r      open for reading
r+     open for reading and writing
w      open for writing
w+     open for writing and reading
a      open for appending
```

```
a+      open for appending and reading
```

For "r" modes, it is an error if the file does not exist. For "w" modes, the file will be created if it did not exist and will be truncated to zero size if it did. For "a" modes, the file will be created if it did not exist, but it will not be truncated if it did.

The initial file position for "r" and "w" modes is the beginning of the file. The initial file position for "a" mode is the end of the file. Additionally, all writes in "a" mode append data to the end of the file, irrespective of the file position at the time of the write.

3.11.6 fseek

```
bool fseek(resource handle, int position)
```

The fseek function attempts to set the file position of the file handle "handle" to "position". Positive positions are taken to be from the start of the file, negative positions are taken to be from the end of the file. The fseek function returns true on success and false on failure.

The effects of setting the file position beyond the end of the file are implementation-defined.

3.11.7 ftell

```
mixed ftell(resource handle)
```

The ftell function returns the current file position of the file handle "handle". It returns the file position as an int value on success or void on error.

If the current file position does not fit into an int value, the result is implementation-defined.

3.11.8 fread

```
mixed fread(resource handle, int max)
```

The fread function tries to read at most "max" bytes from the file associated with the file handle "handle". It returns a string containing the bytes read on success or void on failure. It is not a failure if less than the given number of bytes were available.

3.11.9 fgetc

```
mixed fgetc(resource handle)
```

The fgetc function tries to read one byte from the file associated with the file handle "handle". It returns a string containing the byte read on success or void on failure.

3.11.10 fgets

```
mixed fgets(resource handle)
```

The fgets function tries to read a single line of text from the file associated with the file handle "handle". A line is considered to end at the next newline character or the end of the file. The fgets function returns the line read as a string value, including the newline character, if any. On failure, void is returned.

In order to ease the memory management burden, implementations are allowed to specify a maximum length for the returned string. In this case, lines in the file that are longer than the defined maximum length will be returned as multiple lines, but no synthetic newline characters are allowed to be inserted.

3.11.11 fwrite

```
mixed fwrite(resource handle, string data)
```

The fwrite function tries to write the bytes contained in the string "data" to the file associated with the file handle "handle". It returns the number of bytes actually written or void if "handle" is not a valid file handle. If an I/O error occurs, the number of bytes written before the error occurred is returned.

3.11.12 setbuf

```
bool setbuf(resource handle, bool enable)
```

The setbuf function can be called between opening a file and performing the first I/O operation on the file handle. It enables or disables buffering for the file handle "handle". If "enable" is true, the file handle is block buffered with a block size defined by the implementation. If "enable" is false, the file handle is unbuffered, and all reads and writes will go directly to the file.

The setbuf function returns true on success and false on failure.

3.11.13 fflush

```
bool fflush(resource handle)
```

The fflush function flushes all buffers associated with the file handle "handle", meaning all outstanding writes will be written to the associated file before the fflush function returns. It returns true on success and false on failure.

3.11.14 feof

```
bool feof(resource handle)
```

The feof function returns true if the end-of-file indicator of the file handle "handle" is set. It returns false otherwise. The end-of-file indicator is set whenever one of the reading functions has previously encountered the end of the associated file. If no read from the file handle has taken place yet, feof always returns false.

3.11.15 ferror

```
bool ferror(resource handle)
```

The ferror function returns true if the error indicator of the file handle "handle" is set. It returns false otherwise. The error indicator is set whenever an I/O function used on the file handle has encountered an I/O error. If no operation on the file handle has taken place yet, ferror always returns false.

3.11.16 clearerr

```
void clearerr(resource handle)
```

The clearerr function clears both the end-of-file and error indicator of the file handle "handle".

3.11.17 fclose

```
bool fclose(resource handle)
```

The fclose function closes the file associated with the file handle "handle", after which "handle" is no longer associated with the file. If the file was buffered, any pending data not yet written to the file is flushed to disk.

The fclose function returns true on success and false on failure. A failure can result if the file handle was buffered and an I/O error occurred while flushing the contents of the

buffer to disk.

Use of the resource value after calling fclose behaves as if the resource is not a valid file resource.

3.11.18 errno

```
int errno()
```

The errno function returns an int value describing the last I/O error that was encountered. The actual set of possible values depends on the operating system being used.

3.11.19 strerror

```
string strerror(int error)
```

The strerror function takes an operating system "error" value as returned by the errno function and returns a string describing the error in a human-readable form. Language and format of the string depend on the operating system being used.

This function is mainly intended for displaying I/O errors to the user as it is supposed to use the same messages as the operating system and its tools.

3.11.20 open

```
int open (string path, int flags)
```

The open function opens a file and returns a number known as file descriptor.

path:	the file path	
flags:	compliant POSIX flags which can be specified by OR'ing	O_RDONLY

(0) open for reading only O_WRONLY (1) open for writing only O_RDWR (2) open for reading and writing O_CREAT (100) create if not exists O_TRUNC (1000) truncate file O_APPEND (2000) open in append mode O_SYNC (10000) open in synchronous mode

Returns -1 on error, otherwise the corresponding fd.

3.11.21 close

```
int close (int fd)
```

The close function closes an open file descriptor.

```
fd      file descriptor
```

Returns -1 on error.

3.11.22 read

```
string read (int fd, int size)
```

The read function reads size bytes from the specified file descriptor and returns the retrieved text.

```
fd      file descriptor
size    number of bytes to read
```

Returns retrieved text or NULL on error.

3.11.23 write

```
int write (int fd, string buf, int size)
```

The write function writes size bytes to the specified file descriptor.

```
fd      file descriptor
buf     buffer to write
size    number of bytes to write
```

Returns number of written bytes or -1 on error.

3.11.24 seek

```
int seek (int fd, int offset, int whence)
```

The seek function repositions the offset of the open file associated with the file descriptor fd relative to the argument whence.

```
fd      file descriptor
offset  new offset to be set
```

```
whence   offset directive
         SEEK_SET (0)     absolute offset
         SEEK_CUR (1)     current location plus offset bytes
         SEEK_END (2)     size of the file plus offset bytes
```

Returns the new offset or -1 on error.

3.11.25 readlink

```
string readlink (string path, int size)
```

Displays the path to which a symlink points to.

```
path    a symlink's path
size    maximum number of bytes to read
```

Returns real path.

3.11.26 unlink

```
int unlink (string path)
```

The unlink function deletes a file specified by path.

```
path    a file/symlink/socket path
```

Returns -1 on error.

3.11.27 remove

```
bool remove(string name)
```

The remove function tries to remove the file with the given name from disk. It returns true on success and false on failure.

3.11.28 rename

```
bool rename(string source, string dest)
```

The rename function tries to rename the file with the name given by "source" to the name given by "dest". It returns true on success and false on failure.

3.11.29 symlink

```
int symlink (string orig, string link)
```

The symlink function creates a symlink for the specified file.

orig	the original file name
link	the name of the symlink

Returns -1 on error.

3.11.30 chown

```
int chown (string path, int owner, int group)
```

The chown function changes the ownership of a file.

```
path    file path
owner   UID of new owner
group   GID of new owner group
```

Returns -1 on error.

3.11.31 chmod

```
int chmod (string path, int mode)
```

The chmod function changes the permissions of a file.

```
path    file path
mode    mode can be specified by OR'ing:
S_IXOTH  (000001)  execute/search by others
S_IWOTH  (000002)  write by others
S_IROTH  (000004)  read by others
S_IXGRP  (000010)  execute/search by group
S_IWGRP  (000020)  write by group
S_IRGRP  (000040)  read by group
S_IXUSR  (000100)  execute/search  by  owner
```

```
S_IWUSR   (000200)   write by owner
S_IRUSR   (000400)   read by owner
S_ISVTX   (001000)   sticky bit
S_ISGID   (002000)   set-group-ID
S_ISUID   (004000)   set-user-ID
```

S_IFIFO (010000) fifo (named piped) S_IFCHR (020000) character device S_IFCHR (040000) directory S_IFBLK (060000) block device S_IFREG (100000) regular file S_IFLNK (120000) symlink S_IFSOCK (140000) socket

Returns -1 on error.

3.11.32 mkdir

```
int mkdir (string path, int mode)
```

The mkdir function creates a new directory.

```
path      directory's full path
mode      mode can be specified by OR'ing:
S_ISUID   (04000)   set-user-ID
S_ISGID   (02000)   set-group-ID
S_ISVTX   (01000)   sticky bit
S_IRUSR   (00400)   read by owner
S_IWUSR   (00200)   write by owner
S_IXUSR   (00100)   execute/search  by  owner
S_IRGRP   (00040)   read by group
S_IWGRP   (00020)   write by group
S_IXGRP   (00010)   execute/search by group
S_IROTH   (00004)   read by others
S_IWOTH   (00002)   write by others
S_IXOTH   (00001)   execute/search by others
```

Returns -1 on error.

3.11.33 rmdir

```
int rmdir (string path)
```

The rmdir function deletes a directory.

```
path    directory's full path
```

Returns -1 on error, otherwise zero.

3.11.34 opendir

```
int opendir (string path)
```

The opendir function opens a directory for reading and returns a handle which can be used for further directory operations.

```
path    directory's full path
```

Returns -1 on error, otherwise a directory handle.

3.11.35 readdir

```
string readdir (int handle)
```

The readdir function can be used to read files in a directory.

```
handle  directory handle
```

Returns directory file name or NULL on end.

3.11.36 closedir

```
int closedir (int handle)
```

The closedir function closes a directory handle.

```
handle  directory handle
```

Returns -1 on error.

3.11.37 file_exists

```
bool file_exists (string path)
```

The file_exists function can be used to check if a particular file is present.

```
path   path to file
```

Returns true if present, otherwise zero.

3.11.38 file_size

```
int file_size (string path)
```

The file_size function returns the sizeof of the file specified by path.

```
path   path to file
```

Returns -1 on error.

3.11.39 file_mtime

```
int file_mtime (string path)
```

The file_mtime function returns the timestamp of the last modification of the file.

```
path   path to file
```

Returns -1 on error.

3.11.40 file_copy

```
int file_copy (string source, string dest)
```

The file_copy functions copies the contents of a file to another.

```
source path to source file
dest   path to destination file
```

Returns -1 on error, otherwise the number of bytes written.

3.12 Socket functions

3.12.1 socket

```
int socket (int domain, int type, int proto)
```

The socket function creates an endpoint for a socket-based communication.

```
domain  specifies the communication domain
 1   AF_UNIX   (UNIX domain)
 1   AF_LOCAL  (UNIX domain/POSIX)
 2   AF_INET   (IPv4)
 10  AF_INET6  (IPv6)
 17  AF_PACKET (low level)
```

```
type    specifies the communication semantics
 1   SOCK_STREAM (stream connection)
 2   SOCK_DGRAM  (connection-less datagram)
 3   SOCK_RAW    (raw socket)
```

```
proto   specifies the protocol to be used
 0   IPPROTO_IP
 1   IPPROTO_ICMP
 6   IPPROTO_TCP
 17  IPPROTO_UDP
```

Returns -1 on error, otherwise a file descriptor.

3.12.2 bind

```
int bind (int fd, int port, string addr)
```

The bind function binds the socket to an address/port.

```
fd        socket file descriptor
port      port number to bind to (must be above 1024)
```

```
addr       address (can be empty for ANY)
```

Returns -1 on error.

3.12.3 select

```
int select (int fd, int timeout)
int select (array fds, int timeout)
```

The select function can be used for synchronous I/O multiplexing. It allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some sort of I/O operation.

```
fd         socket file descriptor
fds        array of socket file descriptors
timeout    timeout in second
```

Returns 0 on timeout, -1 on error and >0 when data is available. If an array of file descriptors has been specified, it returns the corresponding descriptor.

3.12.4 connect

```
int connect (int fd, string hostname, int port)
```

The connect function initiates a connection on a socket.

```
fd         socket file descriptor
hostname   hostname or IP address
port       a port number
```

Returns -1 on error.

3.12.5 listen

```
int listen (int fd, int backlog)
```

The listen function can be used to listen for connections on a socket.

```
fd         socket file descriptor
```

```
backlog    maximum queue length of pending connections
```

Returns -1 on error.

3.12.6 accept

```
int accept (int fd)
```

The accept function can be used to accept a connection on a socket.

```
fd         socket file descriptor
```

Returns -1 on error, otherwise a descriptor for the accepted socket.

3.12.7 send

```
int send (int fd, string msg)
```

The send function sends a message from a socket to its peer (used for TCP).

```
fd         socket file descriptor
msg        message to be sent
```

Returns -1 on error or number of bytes written.

3.12.8 sendto

```
int sendto (int fd, string msg, string hostname, int port)
```

The sento function sends a message on a socket to a particular host (used for UDP).

```
fd         socket file descriptor
msg        message to be sent
hostname   hostname or IP address
port       port number
```

Returns -1 on error or number of bytes written.

3.12.9 sendbuf

```
int sendbuf (int fd, string buf, int len)
```

The sendbuf function sends a buffer with a specific length from a socket to its peer (used for TCP).

```
fd        socket file descriptor
buf       buffer to be sent
len       length of buffer
```

Returns -1 on error or number of bytes written.

3.12.10 sendbufto

```
int sendbufto (int fd, string buf, int len, string hostname, int port)
```

The senbufto function sends a buffer with a specific length on a socket to a particular host (used for UDP).

```
fd        socket file descriptor
buf       buffer to be sent
len       length of buffer
hostname  hostname or IP address
port      port number
```

Returns -1 on error or number of bytes written.

3.12.11 recv

```
string recv (int fd)
```

The recv function receives a message from a socket (used for TCP).

```
fd        socket file descriptor
```

Returns the received message or NULL on error.

3.12.12 recvfrom

```
string recvfrom (int fd)
```

The recvfrom function receives a message from a socket.

```
fd        socket file descriptor
```

Returns the received message or NULL on error.

3.12.13 recvmsg

```
string recvmsg (int fd, int timeout)
```

The recvmsg function receives a message from a socket and returns a struct including the address from where the message came from.

```
fd        socket file descriptor
timeout   timeout in seconds to wait for a message
```

Returns the received message or NULL on error. Returns void on error, otherwise it returns a msg struct containing the fields:

```
data    received data (max 4096 bytes)
from    the address of the sender
```

3.12.14 getsockopt

```
int getsockopt (int fd, int level, int optname)
```

The getsockopt function can be used to retrieve options for a socket descriptor.

```
fd        socket file descriptor
```

level socket level optname option name

The following socket options exist:

SOL_SOCKET (socket level): SO_BROADCAST SO_DONTROUTE SO_KEEPALIVE SO_REUSEADDR SO_RCVBUF SO_RCVTIMEO SO_SNDBUF SO_SNDTIMEO

SOL_IP (IP level): IP_OPTIONS IP_TOS IP_TTL IP_MULTICAST_IF IP_MULTICAST_TTL
SOL_TCP (TCP level): TCP_NODELAY TCP_MAXSEG TCP_KEEPIDLE TCP_KEEPINTVL
TCP_KEEPCNT TCP_SYNCNT TCP_LINGER2 TCP_INFO TCP_CONGESTION

3.12.15 setsockopt

```
int setsockopt (int fd, int level, int optname, int optval)
```

The setsockopt function can be used to set options for a socket descriptor.

```
fd        socket file descriptor
```

level socket level optname option name optval option value

Returns 0 on success or -1 on error.

3.13 Date and time functions

The date and time functions provide a way to query the operating system for the current date and time. There are also function to create string representations of dates and/or times.

3.13.1 Date and time structure

Most of the date and time functions return or accept time given as a struct value with the following fields, all containing int values:

```
tm_sec       seconds (0-59)
tm_min       minutes (0-59)
tm_hour      hours (0-23)
tm_mday      day of month (1-31)
tm_mon       month (0-11, 0 = January)
tm_year      year (number of years since 1900)
tm_wday      day of week (0-6, 0 = Sunday)
tm_yday      day of year (0-365)
tm_isdst     daylight savings (1 = yes)
```

The tm_sec field can actually have the range 0-61 to allow for leap seconds.

3.13.2 time

```
int time()
```

The time function returns the number of seconds that have passed since midnight on 1970-01-01 (the Unix time epoch).

3.13.3 gmtime

```
struct gmtime(time)
```

The gmtime function returns a date and time struct from the given time which can be derived from time(). The struct contains the current time in the UTC time zone.

3.13.4 localtime

```
struct localtime(time)
```

The localtime function returns a date and time struct from the given time. The struct contains the current time in the local time zone, as defined by the operating system.

3.13.5 mktime

```
int mktime(struct datetime)
```

The mktime function takes the date and time structure "datetime" and returns the corresponding number of seconds since midnight on 1970-01-01 (the Unix time epoch). The result is undefined if the given struct is not a valid date and time struct.

3.13.6 asctime

```
string asctime(struct datetime)
```

The asctime function returns a string containing an ASCII representation of the time given by the date and time structure "datetime". The format used by asctime results in strings like "Wed Jun 30 21:49:08 1993\n", that is it uses English three-letter abbreviations for the day of the week and the month; it also terminates the string with a newline character.

The result is undefined if the given struct is not a valid date and time struct.

3.13.7 ctime

```
string ctime(int seconds)
```

The ctime function interprets its "seconds" arguments as the number of seconds that have passed since midnight on 1970-01-01 (the Unix time epoch) in the local time zone. It returns a string containing an ASCII representation of that date and time in the same format as the asctime function.

The result is undefined is the given number of seconds is negative.

3.13.8 strftime

```
string strftime(string format, struct when)
```

The strftime function formats the date given by the date and time structure "when" according to the format string "format". The format string can contain conversion specifiers starting with a "%" (percent sign) character and other characters. Conversion specifiers are expanded in the output string while all other characters are copied to the output string unmodified. The following conversion specifiers are defined:

```
%%      literal percent sign
%a      abbreviated weekday name
%A      full weekday name
%b      abbreviated month name
%B      full month name
%c      preferred date and time
%d      day of month (01-31)
%H      hour on 24-hour clock (00-23)
%I      hour on 12-hour clock (01-12)
%j      day of year (001-366)
%m      month (01-12)
%M      minute (00-59)
%p      AM or PM (upper case)
%S      second (00-61)
%U      week number (00-53, Sunday first)
%w      day of week (0-6, 0 = Sunday)
%W      week number (00-53, Monday first)
%x      preferred date format without time
%X      preferred time format without date
%y      year without century (00-99)
%Y      year with century
```

```
%Z      time zone name or abbreviation
```

An implementation may define additional conversions.

3.13.9 strptime

```
struct strptime(string str, struct fmt)
```

The strptime function converts a string representation of time to a time structure given by a specific format (see strftime specifiers).

3.14 Locale functions

The locale functions provide a way to query and set the operating system locale. This usually has an influence on the language, date and time formats, and numerical formats used by the operating systems and some standard library functions.

The following is a table of standard library functions that can be influenced by the current locale setting of the operating system:

```
strcoll  tolower  toupper  isalnum  isalpha
islower  ispunct  isupper  strftime  strerror
```

3.14.1 getlocale

```
string getlocale()
```

The getlocale function returns the name of the currently active operating system locale. It is expected that the returned name is a valid argument for the setlocale function.

3.14.2 setlocale

```
mixed setlocale(string locale)
```

The setlocale function attempts to set the current operating system locale to "locale". What names are valid depends on the operating system being used. The setlocale function returns the name of the new effective locale as a string value on success, or void on failure.

Passing an empty string to setlocale instructs the operating system to switch to the user's preferred locale.

3.14.3 localeconv

```
struct localeconv()
```

The localeconv function returns a struct containing the numerical formatting conventions of the current operating system locale. The following fields are expected to be set:

```
decimal_point
thousands_sep
grouping
int_curr_symbol
currency_symbol
mon_decimal_point
mon_thousands_sep
mon_grouping
positive_sign
negative_sign
int_frac_digits
frac_digits
p_cs_precedes
p_sep_by_space
n_cs_precedes
n_sep_by_space
p_sign_posn
n_sign_posn
```

All values are strings. Some strings are actually vectors of integers, meaning they contain sequences of integers between 0-9 which are encoded as ASCII characters of the same value.

The "decimal_point" field indicates the decimal point character to use for non-monetary values.

The "thousands_sep" field indicates the separator characters to place between groups of digits before a decimal point. It should only be used for non-monetary values.

The "grouping" field indicates how to group digits before a decimal point in non-monetary values. It is a vector of bytes indicating how much digits to put in each group. If the vector terminates with a character of value 255, no further grouping is to take place beyond that specified. If the vector does not terminate with a 255 character, the last grouping given is to be repeated for all remaining digits. For example, assuming a "thousands_sep" of "," (comma), a string containing the byte vector (3,2,3) means to print numbers like this: "300,00,000".

The "int_curr_symbol" field specifies the standardized international currency symbol to

use for monetary values.

The "currency_symbol" field specifies the local currency symbol to use for monetary values.

The "mon_decimal_point", "mon_thousands_sep", and "mon_grouping" fields behave the same as their cousins with the "mon_" prefix stripped, but they specify the same information for monetary values.

The "positive_sign" field specifies the symbol to be used to indicate positive monetary values.

The "negative_sign" field specifies the symbol to be used to indicate negative monetary values.

The "int_frac_digits" field specifies how many digits to print after the decimal point in international-style monetary values.

The "frac_digits" field specifies how many digits to print after the decimal point in local-style monetary values.

The "p_cs_precedes" field contains a character of value 1 (one) if the currency symbol should precede positive monetary values. In any other case, the currency symbol should follow positive monetary values.

The "p_sep_by_space" field contains a character of value 1 (one) if a space character should be placed between the currency symbol and a positive monetary value. In any other case, no space character is desired.

The "n_cs_precedes" and "n_sep_by_space" fields work like the two preceding fields, but contains the information for formatting negative monetary values.

The "p_sign_posn" field specifies where to place the sign of positive monetary values with respect to the currency symbol. An empty string specifies to put parenthesis around the whole monetary string. The following table lists the other possibilities:

\x01	before the whole string
\x02	after the whole string
\x03	just before currency symbol
\x04	just after currency symbol

The "n_sign_posn" field works like the "p_sign_posn" field, but gives information for negative monetary values.

Any field may contain a one-byte string with a byte of value 255 to indicate that the field is not available in the current locale.

Note that the accuracy and availability of the above information depends entirely on the operating system being used.

3.15 Dictionary functions

The following functions provide dictionaries. A dictionary associates string names with arbitrary values. Dictionaries are represented by resource values, which are cheap to pass into functions that need access to the dictionary.

Implementations can limit the strings that are acceptable as names for dictionary entries. For example, strings containing characters of value 0 (zero) may not be acceptable.

When the resource value of a dictionary is unset or the namespace containing containing it is destroyed, the memory associated with the dictionary is freed.

3.15.1 is_dict_resource

```
bool is_dict_resource(resource res)
```

The is_dict_resource returns true if the resource "res" is a dictionary resource. It returns false otherwise.

3.15.2 dopen

```
mixed dopen(int order)
```

The dopen function creates a new, empty dictionary and returns a resource for the new dictionary. The "order" parameter specifies a power of two that gives the average number of elements that the programmers expects to put into the dictionary. Thus, an order of 5 means an expected average of 32 elements in the dictionary. The order is only a hint that can be used by the implementation to decide on an appropriate internal representation of the dictionary.

The dopen function returns void if the new dictionary could not be created, for example when not enough free memory is available.

3.15.3 dread

```
mixed dread(resource handle, string name)
```

The dread function looks for an entry with the given name in the dictionary associated with the handle "handle". If the entry is found, its value is returned. Otherwise, void is returned.

3.15.4 dwrite

```
bool dwrite(resource handle, string name, mixed value)
```

The dwrite function creates a new entry in the dictionary associated with the handle "handle". The name for the new entry is "name" and the value is "value". If the dictionary already contains an entry for the same name, the old entry is discarded.

The dwrite function returns true on success or false if the handle is invalid or the name is not an acceptable dictionary entry name.

3.15.5 dremove

```
bool dremove(resource handle, string name)
```

The dremove function removes the entry with the given name from the dictionary associated with the handle "handle". Subsequent reads from the dictionary using the name will return void.

The dremove function returns true on success or false if the handle is invalid or the name not an acceptable dictionary entry name.

3.15.6 dexists

```
bool dexists(resource handle, string name)
```

The dexists function returns true if the given name exists in the dictionary associated with the handle "handle". It returns false if no such entry is found in the dictionary.

3.15.7 dclose

```
void dclose(resource handle)
```

The dclose function closes the dictionary associated with the handle "handle". All memory used by the dictionary is released.

Use of the resource value after calling dclose behaves as if the resource is not a valid dictionary resource.

3.16 Memory management functions

While Arena does provide automatic memory management, sometimes it is necessary to manually allocate and deallocate memory. This is especially true for calling into C libraries from an Arena script (see the next section), since these will often need pointers to memory and you cannot let this memory be automatically freed while the C library may still be using it.

The memory management functions work with resources that represent memory allocated from the operating system. Arena memory resources automatically expand as needed, so they can't overflow when used from Arena script code.

Memory resources can be read-write or read-only. Resources created by an Arena script are always writable, but memory resources returned by calls to C functions are normally read-only.

3.16.1 is_mem_resource

```
bool is_mem_resource(resource res)
```

The is_mem_resource returns true if the resource "res" is a memory resource. It returns false otherwise.

3.16.2 malloc

```
mixed malloc(int size)
```

The malloc function allocates "size" bytes of memory. It returns a memory resource on success and void on failure. The contents of the allocated memory are undefined.

When the resulting memory resource is unset or goes out of scope, the associated operating system memory is freed.

3.16.3 calloc

```
mixed calloc(int number, int size)
```

The calloc function allocates memory sufficient to store "number" elements of "size" bytes each. The allocated memory is always zero-filled. The calloc function returns a memory resource on success and void on failure.

When the resulting memory resource is unset or goes out of scope, the associated operating system memory is freed.

3.16.4 realloc

```
bool realloc(resource mem, int size)
```

The realloc function changes the size of the memory associated with the memory resource "mem" to "size" bytes. When the memory resource is extended, the contents of the newly allocated bytes are undefined.

The realloc function returns true if the memory resource could be resized successfully or false on failure. On failure, the input memory resource retains its original size and contents.

3.16.5 free

```
void free(resource mem)
```

The free function frees up the operating system memory associated with the resource "mem". Further attempts to read or write the memory resource will fail.

3.16.6 cnull

```
resource cnull()
```

The cnull function returns a memory resource that is equivalent to a NULL pointer in C. The resource points to no memory and has a size of zero. It cannot be read or written by other Arena memory functions.

This function is mainly useful for creating a resource that can be passed to a C function that expects a NULL pointer.

3.16.7 is_null

```
bool is_null(resource res)
```

The is_null function returns true if the resource "res" is a memory resource that points to no memory. In this case, the resource will act like a C NULL pointer when passed into a call to a C function.

This function is mainly useful for checking memory resources returned from C function calls.

3.16.8 cstring

```
mixed cstring(forced string x)
```

The cstring function creates a memory resource that contains a C string (a sequence of characters terminated by a \0 character) that is a copy of the input string. The resulting resource is returned.

The cstring function returns void when the input string cannot be converted to a meaningful C string. This is the case when the Arena string contains embedded 0 (zero) bytes.

When the resulting memory resource is unset or goes out of scope, the associated operating system memory is freed.

3.16.9 mputchar

```
bool mputchar(resource mem, int offset, int val)
```

The mputchar function stores a character into the memory resource "mem", using "offset" as a byte offset from the start of the memory resource. The input value "val" is truncated to fit into a C "char" value and stored into the memory resource. If the memory resource is smaller than needed by the offset and stored value, it is expanded to fit.

The mputchar function returns true on success and false on failure.

3.16.10 mputshort

```
bool mputshort(resource mem, int offset, int val)
```

The mputshort function stores a short integer into the memory resource "mem", using "offset" as a byte offset from the start of the memory resource. The input value "val" is truncated to fit into a C "short" value and stored into the memory resource. If the memory resource is smaller than needed by the offset and stored value, it is expanded to fit.

The mputshort function returns true on success and false on failure.

3.16.11 mputint

```
bool mputint(resource mem, int offset, int val)
```

The mputint function stores an integer into the memory resource "mem", using "offset" as a byte offset from the start of the memory resource. The input value "val" is truncated

to fit into a C "int" value and stored into the memory resource. If the memory resource is smaller than needed by the offset and stored value, it is expanded to fit.

The mputint function returns true on success and false on failure.

3.16.12 mputfloat

```
bool mputfloat(resource mem, int offset, float val)
```

The mputfloat function stores a float value into the memory resource "mem", using "offset" as a byte offset from the start of the memory resource. The input value "val" is truncated to fit into a C "float" value and stored into the memory resource. If the memory resource is smaller than needed by the offset and stored value, it is expanded to fit.

The mputfloat function returns true on success and false on failure.

3.16.13 mputdouble

```
bool mputdouble(resource mem, int offset, float val)
```

The mputdouble function stores a float value into the memory resource "mem", using "offset" as a byte offset from the start of the memory resource. The input value "val" is converted to a C "double" value and stored into the memory resource. If the memory resource is smaller than needed by the offset and stored value, it is expanded to fit.

The mputdouble function returns true on success and false on failure.

3.16.14 mputptr

```
bool mputptr(resource mem, int offset, resource ptr)
```

The mputptr function stores a pointer to the memory associated with the memory resource "ptr" into the memory resource "mem", using "offset" as a byte offset from the start of the target memory resource. If the memory resource is smaller than needed by the offset and stored pointer, it is expanded to fit.

If the resource value "ptr" is freed (by going out of scope or being unset) before the resource value "mem", the pointer stored inside "mem" will point to garbage. It is the responsibility of the programmer to ensure that memory resources pointing to each other are not freed up in the wrong order.

The mputptr function returns true on success and false on failure.

3.16.15 mgetchar

```
mixed mgetchar(resource mem, int offset)
```

The mgetchar function tries to read one character from the memory resource "mem", using "offset" as a byte offset from the start of the memory resource. If this is possible, the character is returned as an int value. If "mem" is not a valid memory resource or the offset does not fall into the allocated memory, void is returned.

3.16.16 mgetshort

```
mixed mgetshort(resource mem, int offset)
```

The mgetshort function tries to read one short integer from the memory resource "mem", using "offset" as a byte offset from the start of the memory resource. If this is possible, the short integer is returned as an int value. If "mem" is not a valid memory resource or the offset does not fall into the allocated memory, void is returned.

3.16.17 mgetint

```
mixed mgetint(resource mem, int offset)
```

The mgetint function tries to read one integer from the memory resource "mem", using "offset" as a byte offset from the start of the memory resource. If this is possible, the integer is returned as an int value. If "mem" is not a valid memory resource or the offset does not fall into the allocated memory, void is returned.

3.16.18 mgetfloat

```
mixed mgetfloat(resource mem, int offset)
```

The mgetfloat function tries to read one "float" value from the memory resource "mem", using "offset" as a byte offset from the start of the memory resource. If this is possible, the C float is returned as a float value. If "mem" is not a valid memory resource or the offset does not fall into the allocated memory, void is returned.

3.16.19 mgetdouble

```
mixed mgetdouble(resource mem, int offset)
```

The mgetdouble function tries to read one "double" value from the memory resource "mem", using "offset" as a byte offset from the start of the memory resource. If this is possible, the C double is returned as a float value. If "mem" is not a valid memory resource or the offset does not fall into the allocated memory, void is returned.

3.16.20 mgetptr

```
mixed mgetptr(resource mem, int offset, bool free)
```

The mgetptr function tries to read one pointer from the memory resource "mem", using "offset" as a byte offset from the start of the memory resource. If this is possible, the pointer is returned in the form of a new memory resource that points to the same memory as the original pointer. If the "free" argument is true, the memory pointed to by the memory resource will be freed when the memory resource goes out of scope or is unset. If it is "false", the memory will not be freed up.

If "mem" is not a valid memory resource or the offset does not fall into the allocated memory, void is returned.

The resulting memory value does not have a known size (msize returns zero) and is read-only.

3.16.21 mstring

```
mixed mstring(resource mem, int offset)
```

The mstring function tries to create a string based on a C string (zero-terminated character array) found in the memory resource "mem" and starting at byte offset "offset" from the start of the memory resource. On success, the resulting Arena string is returned. If "mem" is not a valid memory resource or the offset does not fall into the allocated memory, void is returned.

3.16.22 is_rw

```
bool is_rw(resource mem)
```

The is_rw function returns true if the memory resource "mem" is a valid memory resource and writable. It returns false otherwise.

3.16.23 msize

```
mixed msize(resource mem)
```

The msize function returns the number of bytes currently allocated for the memory resource "mem". This value is 0 (zero) if the size is not known, which can happen if the resource was obtained by reading a C pointer. If "mem" is not a valid memory resource, msize returns void.

3.16.24 memcpy

```
bool memcpy(resource dst, int dst_off, resource src,
             int src_off, int count)
```

The memcpy function copies "count" bytes from the memory resource "src" to the memory resource "dst". "src_off" is used as an offset into the source resource and "dst_off" is used as an offset into the destination resource. If the destination memory resource is not large enough to hold the new data, it is extended accordingly. If the destination offset is beyond the end of the destination resource, the bytes between the previous end of the resource and the destination offset are undefined after the operation.

If the source and destination resources overlap, the resulting bytes in the destination resource have undefined values.

The memcpy function returns true if the operation succeeded, or false if one of the input resources was not a memory resource.

3.16.25 memmove

```
bool memmove(resource dst, int dst_off, resource src,
             int src_off, int count)
```

The memmove function copies "count" bytes from the memory resource "src" to the memory resource "dst". "src_off" is used as an offset into the source resource and "dst_off" is used as an offset into the destination resource. If the destination memory resource is not large enough to hold the new data, it is extended accordingly. If the destination offset is beyond the end of the destination resource, the bytes between the previous end of the resource and the destination offset are undefined after the operation.

If the source and destination resources overlap, the memmove function takes care that the correct bytes are copied to the destination resource and offset.

The memmove function returns true if the operation succeeded, or false if one of the input resources was not a memory resource.

3.16.26 memcmp

```
mixed memcmp(resource one, int one_off, resource two,
             int two_off, int count)
```

The memcmp function compares the bytes stored in the memory resources "one" and "two". At most "count" bytes are compared and "one_off" is used as an offset into memory resource "one" while "two_off" is used as an offset into memory resource "two". If any of the resources is not large enough to contain "count" bytes after taking the offset into account, or if one of the input resources is not a memory resource, the memcmp function returns void.

The bytes in the two memory resources are compared one by one. If memcmp encounters a byte in memory resource "one" that is smaller than the corresponding byte in resource "two", a negative int value is returned. If memory resource "one" contains a byte that is larger than the corresponding byte in resource "two", a positive int value is returned. Otherwise, the comparison moves on to the next byte in both resources. If the bytes in both resources prove to be identical, an int value of 0 (zero) is returned.

3.16.27 memchr

```
mixed memchr(resource mem, int offset, int what, int count)
```

The memchr function searches the memory resource "mem" for a byte containing the value "what", with "what" truncated to eight bits. The search starts at the offset "offset" and searches a maximum of "count" bytes starting from the given offset.

The memchr function returns void if "mem" is not a memory resource, the offset and count exceed the size of the memory resource, or the given byte is not found inside the resource. If the given byte is found inside the memory resource, the memchr function returns an int value giving the offset from the start of the memory resource where the byte was found.

3.16.28 memset

```
bool memset(resource mem, int offset, int what, int count)
```

The memset function truncates "what" to eight bits and fills the memory resource "mem" with the resulting byte. Filling starts at the offset "offset" inside the resource and continues for "count" bytes. The memory resource is automatically resized if its current size is not large enough to hold the generated bytes.

The memset function returns true if the operation succeeded, or false if the given resource

was not a memory resource.

3.17 Foreign function calls

This group of functions provides a way for an Arena script to dynamically load C libraries and call C functions contained in those libraries. Foreign function calls are not necessarily possible on all operating systems and architectures supported by an Arena implementation.

Care needs to be taken when calling C functions that use pointers. Arena cannot automatically manage memory for called C functions, so it is the responsibility of the programmer to allocate and free memory used for calls into C library functions. This can be done by using the memory management functions described in the preceding section.

3.17.1 dyn_supported

```
bool dyn_supported()
```

The dyn_supported function returns true if foreign function calls are possible in the currently running Arena implementation. It returns false otherwise.

If foreign function calls are not supported, all other foreign function call operations simply return void without trying to do anything.

3.17.2 is_dyn_resource

```
bool is_dyn_resource(resource res)
```

The is_dyn_resource returns true if the resource "res" is a resource for a loaded dynamic library. It returns false otherwise.

3.17.3 dyn_open

```
mixed dyn_open(string name)
```

The dyn_open function tries to dynamically load the C library called "name". What names constitute valid library names depends on the operating system in use. On Unix systems, is is normally possible to specify both an absolute path and a relative one. Relative paths are then searched for in a set of pre-defined library directories.

On success, the dyn_open function returns a resource that represents the loaded library.

On failure, dyn_open returns void.

3.17.4 dyn_close

```
void dyn_close(resource lib)
```

The dyn_close function tries to unload the C library represented by the resource "lib". Whether the library actually is unloaded from memory depends on the operating system used. In any case, the resource is no longer usable as a valid library resource.

3.17.5 dyn_fn_pointer

```
mixed dyn_fn_pointer(resource lib, string name)
```

The dyn_fn_pointer function searches the library pointed to by the resource "lib" for a function called "name". If it is found, a memory resource pointing to the function is returned. This memory resource has an unknown size and is read-only. If "lib" is not a valid library resource or the function "name" is not found, void is returned.

This function is mainly useful for creating memory resources that can be passed into C function that expect function pointers.

3.17.6 dyn_call_void

```
void dyn_call_void(resource lib, string name, ...)
```

The dyn_call_void function tries to call a C function called "name" from the library pointed to by the resource "lib". The rest of the arguments are passed to the C function. The following conversions take place for the arguments:

```
Arena type      C type
void            int (zero)
bool            int (0 or 1)
int             int
float           double
string          char *
resource        void *
```

For resources, file resources created by the fopen function are passed as "FILE *" pointers and memory resources are passed as "void *" pointers. This does the right thing for both types of resource. For other resources the conversion is not meaningful.

Values of type void, int, bool, int, and float are passed as copies. Values of type string and resource are passed as pointer references, so the called C function can directly access or even change their contents.

It is not allowed to directly pass arrays or structures into C functions. You can use the memory management functions to manually create in-memory representations of C arrays and structures and pass them as resources.

The dyn_call_void function assumes that the called C function does not return a value and thus returns void itself.

3.17.7 dyn_call_int

```
mixed dyn_call_int(resource lib, string name, ...)
```

The dyn_call_int function works almost exactly like the dyn_call_void function, but assumes that the called C function returns the C type "int" or a compatible, shorter integer type such as "short" or "char". This value is returned in the form of an Arena int value. If the called C function does not return an "int", the resulting int value is undefined.

If the resource "lib" is not a valid library resource or the function "name" cannot be found, void is returned.

3.17.8 dyn_call_float

```
mixed dyn_call_float(resource lib, string name, ...)
```

The dyn_call_float function works almost exactly like the dyn_call_void function, but assumes that the called C function returns the C type "double" or a compatible, shorter floating point type such as "float". This value is returned in the form of an Arena float value. If the called C function does not return a "double", the resulting float value is undefined.

If the resource "lib" is not a valid library resource or the function "name" cannot be found, void is returned.

3.17.9 dyn_call_ptr

```
mixed dyn_call_ptr(resource lib, string name, ..., bool free)
```

The dyn_call_ptr function works almost exactly like the dyn_call_void function, but assumes that the called C function returns a C pointer. This value is returned in the form of an Arena memory resource pointing at the same memory as the C pointer. If

the last argument to the dyn_call_ptr function is "true", this memory is freed when the memory resource is unset or goes out of scope. When the last argument to dyn_call_ptr is not a bool value or "false", the memory is not freed by the Arena implementation.

If the called C function does not return a pointer, the resulting memory resource points at an undefined memory location. In any case, the resulting memory resource has an unknown size and is read-only.

If the resource "lib" is not a valid library resource or the function "name" cannot be found, void is returned.

3.18 PCRE functions

The Arena interpreter can be built with PCRE support on systems that have the PCRE (Perl Compatible Regular Expressions) library available. This group of functions provides versatile support for matching regular expressions against strings.

3.18.1 pcre_supported

```
bool pcre_supported()
```

The pcre_supported function returns true if the currently running language implementation provides PCRE support. It returns false otherwise.

3.18.2 PCRE_ANCHORED

```
PCRE_ANCHORED
```

The int variable PCRE_ANCHORED is initialized to contain the option value for forcing a regular expression to match only at the first possible matching point in subject string. It can be passed to all functions that take PCRE options.

3.18.3 PCRE_CASELESS

```
PCRE_CASELESS
```

The int variable PCRE_CASELESS is initialized to contain the option value for forcing a regular expression to perform a case-insensitive match. This option value can only be used when compiling a regular expression.

3.18.4 PCRE_DOLLAR_ENDONLY

```
PCRE_DOLLAR_ENDONLY
```

The int variable PCRE_DOLLAR_ENDONLY is initialized to contain the option value for forcing the dollar meta character in a regular expression to match only at the end of a subject string. Normally it also matches directly before the last character of the string if the last character is a newline character. This option value can only be used when compiling a regular expression.

3.18.5 PCRE_DOTALL

```
PCRE_DOTALL
```

The int variable PCRE_DOTALL is initialized to contain the option value for making the dot meta character in a regular expression match all characters. Normally it doesn't match newline characters. This option value can only be used when compiling a regular expression.

3.18.6 PCRE_EXTENDED

```
PCRE_EXTENDED
```

The int variable PCRE_EXTENDED is initialized to contain the option value for making PCRE ignore whitespace characters in a regular expression and ignore all characters between the "#" (hash) character and the next newline following it. This makes it possible to embed comments in regular expressions. This option value can only be used when compiling a regular expression.

3.18.7 PCRE_MULTILINE

```
PCRE_MULTILINE
```

The int variable PCRE_MULTILINE is initialized to contain the option value for making a regular expression treat a subject string as consisting of multiple lines. Normally the whole subject string is treated as a single line. This option value can only be used when compiling a regular expression.

3.18.8 PCRE_UNGREEDY

`PCRE_UNGREEDY`

The int variable PCRE_UNGREEDY is initialized to contain the option value for reversing the greediness of quantifiers in a regular expression so that they are not greedy by default. This option value can only be used when compiling a regular expression.

3.18.9 **PCRE_NOTBOL**

`PCRE_NOTBOL`

The int variable PCRE_NOTBOL is initialized to contain the option value that makes a regular expression not consider the beginning of a subject string to be the beginning of a line. This option value can only be used when matching a regular expression.

3.18.10 **PCRE_NOTEOL**

`PCRE_NOTEOL`

The int variable PCRE_NOTEOL is initialized to contain the option value that makes a regular expression not consider the end of a subject string to be the end of a line. This option can only be used when matching a regular expression.

3.18.11 **PCRE_NOTEMPTY**

`PCRE_NOTEMPTY`

The int variable PCRE_NOTEMPTY is initialized to contain the option value that makes a regular expression and its subexpressions not match empty subject strings. This option can only be used when matching a regular expression.

3.18.12 **is_pcre_resource**

`bool is_pcre_resource(resource res)`

The is_pcre_resource returns true if the resource "res" is a resource for a compiled regular expression. It returns false otherwise.

3.18.13 **pcre_compile**

```
mixed pcre_compile(string pattern, int options)
```

The pcre_compile function compiles the regular expression "pattern" into an internal format used by the PCRE library. Additional options can be passed in the "options" argument. If the pattern can be compiled successfully, a resource for the compiled pattern is returned. On error, void is returned.

When the resource value returned by pcre_compile is unset or goes out of scope, the compiled regular expression is freed.

For the exact syntax and semantics of the supported regular expressions, please refer to the PCRE documentation which can be found online at:

```
http://www.pcre.org/
```

3.18.14 pcre_match

```
bool pcre_match(resource reg, string x, int options)
```

The pcre_match function returns true if the regular expression associated with the resource "reg" matches the string "x". Additional options can be passed in the "options" argument. If the regular expression does not match the string, false is returned.

3.18.15 pcre_exec

```
mixed pcre_exec(resource reg, string x, int options)
```

The pcre_exec function returns detailed information about how the regular expression associated with the resource "reg" matches the string "x". Additional options can be passed in the "options" argument.

If there is an error while matching the regular expression, void is returned. If the regular expression does not match the string, an empty array is returned.

If the regular expression matches the string, an array with information about the exact matching parts of the string is returned. The first element of the resulting array contains the part of the input string that matches the whole regular expression.

If the regular expression contains subexpressions in parenthesis, these are counted starting from one and the part of the string that matched each subexpression is contained in the resulting array at the same index. Subexpressions from the end of the expression that were not used in matching are not included in the result array. If subexpression n is not used for matching but subexpression n+1 is, the value used for subexpression n in the result array is a void value. This allows the programmer do distinguish between

a subexpression not matching at all and matching an empty string.

3.18.16 pcre_free

```
void pcre_free(resource reg)
```

The pcre_free function frees the regular expression associated with the resource "reg". Later calls to PCRE functions with the same resource will act as if the resource is not a valid regular expression resource.

3.19 Other functions

3.19.1 sleep

```
int sleep (int sec)
```

The sleep function can be used to stop execution and wait for the specified period of time.

```
sec     number of seconds to wait
```

Returns -1 on error.

3.19.2 usleep

```
int usleep (int microseconds)
```

The usleep function can be used to stop execution and wait for the specified period of time.

```
microseseconds     number of microseconds to wait
```

Returns -1 on error.

3.19.3 mkstemp

```
int mkstemp (string template)
```

The mkstemp function creates a unique temporary file.

```
template        path template (X will be substituted)
```

Returns -1 on error.

3.19.4 base64_encode

```
string base64_encode (string str)
```

The base64_encode function will encode a memory string to a base64 encoded string.

```
str     string to encode
```

Returns an empty string on failure, otherwise the encoded string.

3.19.5 base64_decode

```
string base64_decode (string str)
```

The base64_decode function will decode a base64 encoded string to a memory string.

```
str     the base64 encoded string to decode
```

Returns an empty string on failure, otherwise the decoded memory string.

3.19.6 json_encode

value json_encode (value val)
The json_encode function will encode a specific value to a JSON string.

```
val     the value to encode
```

Returns an empty string on failure, otherwise the encoded JSON string.

3.19.7 md5sum

```
string md5sum (string file)
```

The md5sum function returns the MD5 hash of the specfied file.

```
file     the file to be hashed
```

Returns an empty string on failure, otherwise the calculated hash.

3.19.8 md5hash

```
string md5hash (string str)
```

The md5hash function returns the MD5 hash of the specfied string.

```
str      the string to be hashed
```

Returns an empty string on failure, otherwise the calculated hash.

3.19.9 sha256sum

```
string sha256sum (string file)
```

The sha256sum function returns the SHA256 hash of the specfied file.

```
file     the file to be hashed
```

Returns an empty string on failure, otherwise the calculated hash.

3.19.10 sha256hash

```
string sha256hash (string str)
```

The sha256hash function returns the SHA256 hash of the specfied string.

```
str      the string to be hashed
```

Returns an empty string on failure, otherwise the calculated hash.

3.19.11 sysinfo

```
struct sysinfo ()
```

The sysinfo function returns a struct containing information on overall system statistics. The following elements are provided:

uptime	seconds since boot
loads	array of 1, 5, and 15 minute load averages
totalram	total usable main memory size
freeram	available memory size
sharedram	amount of shared memory
bufferram	memory used by buffers
procs	number of current processes
totalhigh	total high memory size
freehigh	available high memory size

3.19.12 basename

string	basename (string path)

The basename function returns the filename of the given path.

3.19.13 dirname

string	dirname (string path)

The dirname function returns the directory of the given path.

3.19.14 uptime

int	uptime (void)

The uptime function returns the number of seconds since boot.

4 Changes

This section describes the changes that were made between different versions of the language or library.

Language and library can be improved independent of each other, so each part has a separate listing of changes.

4.1 Language changes

This section describes changes to the language syntax and semantics.

4.1.1 Version 1.0 to 2.0

Version 2.0 of the language added the resource datatype and associated syntax for declaring function return types and arguments of type resource. This is an incompatible change because the word "resource" is no longer available as an identifier.

Negative array indices were changed so that using an index too large refers to the first element of the array when used in an assignment. Previously this was a fatal error.

4.1.2 Version 2.0 to 2.1

Version 2.1 of the language documented the fact that an index expression in an assignment cannot have side effects on the same structure or array as the whole assignment expression.

The meaning of equality tests on fn values was changed to be defined by this manual instead of being defined by an implementation of the language.

4.1.3 Version 2.1 to 2.2

Version 2.2 of the language added hexadecimal integer literals.

4.2 Library changes

This section describes changes to the standard library of functions.

4.2.1 Version 1.0 to 1.1

The following variables were added in version 1.1: FLT_DIG, FLT_EPSILON, FLT_MANT_DIG, FLT_MAX, FLT_MAX_EXP, FLT_MIN, FLT_MIN_EXP, FLT_RADIX, INT_MAX, INT_MIN, RAND_MAX.

The following functions were added in version 1.1: cast_to, cons, drop, drop_while, elem, explode, head, implode, init, intersperse, last, length, ltrim, nil, null, replicate, rtrim, tail, take, take_while, trim.

4.2.2 Version 1.1 to 2.0

The prototypes of most file I/O function and of all dictionary functions were changed to make use of the new resource datatype. This is an incompatible change because scripts depending on the argument or return types of these functions can no longer work.

The following functions were added in version 2.0: calloc, cnull, cstring, dyn_call_float, dyn_call_int, dyn_call_ptr, dyn_call_void, dyn_close, dyn_fn_pointer, dyn_open, dyn_supported, free, is_resource, is_rw, malloc, mgetchar, mgetint, mgetptr, mgetshort, mputchar, mputint, mputptr, mputshort, msize, mstring.

4.2.3 Version 2.0 to 2.1

The following functions were added in version 2.1: pcre_compile, pcre_exec, pcre_free, pcre_match, pcre_supported.

4.2.4 Version 2.1 to 2.2

The following functions were added in version 2.2: is_dict_resource, is_dyn_resource, is_file_resource, is_mem_resource, is_pcre_resource, mgetdouble, mgetfloat, mputdouble, mputfloat.

4.2.5 Version 2.2 to 2.3

The following function was added in version 2.3: realloc.

4.2.6 Version 2.3 to 2.4

The following functions were added in version 2.4: memchr, memcmp, memcpy, memmove, memset.

4.2.7 Version 2.4 to 2.5

The prototypes of the following functions were changed in version 2.5 to allow extra arguments to be passed into the function given as the first argument: drop_while, map, filter, foldl, foldr, take_while.
The following function was added in version 2.5: is_null.

4.2.8 Version 2.5 to 2.5.1

The following functions have been added:

```
open close read write seek
readlink unlink symlink chown chmod
mkdir mkstemp opendir readdir closedir
select socket connect bind listen accept
send sendto sendmsg recv recvfrom recvmsg
sleep strptime checksum ping base64_encode base64_decode
```

Those functions have been removed:

```
system fcmp atof
```

4.2.9 Version 2.5.1 to 2.5.2

The serial device has been renamed from ttySX to serialX. The gethostbyname, sendbuf and sendbufto functions have been added.

4.2.10 Version 2.5.2 to 2.5.3

The dirname, basename, setsockopt, getsockopt, nb_snmp_trap, nb_etherwake, nb_arp_ping and nb_arp_gratuitous functions have been added. The ifc_address function can be called as nb_ifc_address, the gethostbyname as nb_gethostbyname and ping as nb_ping. Not only internal interfaces can be specified but also external ones. We have further added the functions nb_transfer_list and nb_transfer_delete.

4.2.11 Version 2.5.3 to 2.5.4

Voice functions have been added.

4.2.12 Version 2.5.4 to 2.5.5

Remote mail functions have been added. SNMP agent functions have been added.

4.2.13 Version 2.5.5 to 2.5.6

CAN functions have been added. MD5 and SHA256 function have been added. The nb_event_msg function has been added. The file_exists, file_mtime, file_size and file_copy functions have been addded.

4.2.14 Version 2.5.6 to 2.5.7

The nb_reset_factory and nb_reset_statistics functions have been added.

4.2.15 Version 2.5.7 to 2.5.8

The nb_snmp_inform function and support for SNMPv3 traps have been added.

4.2.16 Version 2.5.8 to 2.5.9

The uptime function has been added.

4.2.17 Version 2.5.9 to 2.5.10

The nb_wanlink_activate, nb_wanlink_deactivate, nb_wanlink_priorize and nb_wanlink_weight functions have been added. The nb_transfer_post function is now able to store the response. The nb_email_send function can now send a file as attachment.

5 Acknowledgements