morpho

Developer Guide
Version 0.6.2

January 10, 2025

Contents

Chapter 1

Introduction

This Developer Guide aims to assist users at all levels who want to learn more about programming with *morpho*, including those who are interested in extending or improving the program and contributing to the project. It is divided into several sections:—

- Part I introduces the *morpho* language in some detail, as well as providing information on data types.
- Part II describes the *morpho* debugger and profiler which are tools to help programmers write better *morpho* code.
- Part III discusses how *morpho* can be extended. Modules can be written in morpho and or in C or similar languages. The *morpho* C API is described. Information about packaging modules and extensions and distributing them is also included.
- Part IV provides documentation of *morpho's* internals for those interested in contributing to the source.

Note that the Guide remains under construction, and we anticipate adding additional information in future releases as we approach morpho v1.0.0. We welcome your feedback on what is most helpful in the below as well as what is missing.

Part I The *morpho* language

Chapter 2

Design goals

Morpho is a dynamic language oriented toward scientific computing that was designed with the following goals:

- **Familiar.** *Morpho* uses syntax similar to other C-family languages.
- **Simple.** The syntax has been kept simple, so there are only a few things to learn; nonetheless, the features provided are very expressive, enabling the programmer to express intent cleanly.
- **Fast.** *Morpho* programs run as efficiently as other well-implemented dynamic languages like wren, lua or python. *Morpho* leverages numerical libraries like BLAS, LAPACK, etc. to provide good performance.
- **Class-based.** *Morpho* is highly object-oriented, which simplifies coding and enables reusability.
- **Extendable.** Functionality is easy to extend add via modules, which can be written in *Morpho* or C and other compiled languages.

This book is intended as a brief introduction to *Morpho*, illustrating its features and providing some advice on how to use them best.

Chapter 3

Variables and types

Like many languages, *Morpho* allows the programmer to define *variables* to contain pieces of information of *values*. A variable is created using the `var` keyword, which is followed by the variable name

```
var a
```

Variable names must begin with an alphabetical character or the underscore character _, and may consist of any combination of alphanumeric characters or underscores thereafter. Variable names are case sensitive, so

```
var a
var A
```

each refer to distinct variables.
After creating a variable, you may immediately store information in it by providing an *initializer,* which can be any value

```
var i = 1
var str = "Hello"
```

3.1 Types

Morpho is a dynamically typed language: Every value has a definite type, and Morpho is always able to tell what type it is, but variables may generally contain values of any type and functions or methods can accept arguments of any type.

There are a number of basic types in Morpho:

nil is a special value that represents the *absence* of information and is different from any other value. Unless an initializer is provided, Morpho variables initially contain `nil` after declaration.

Bool values contain either `true` or `false`.

Int values contain 32 bit signed integer numbers. An integer constant is written intuitively, e.g. `1`, `50`, `1000` and may include a negative sign `-100`.

Float values contain double precision floating point numbers. You can write numeric constants either using a decimal `1.5` or in scientific notation, e.g. `1e10`, `1.6e-19` or `6.625e26`.

In addition to these basic types, Morpho provides a rich collection of **objects**. Discuss immutability i.e. cannot be changed after creation, or not

3.2 Strings

Strings are sequences of unicode UTF8 encoded characters. You specify a literal string using double quotes

```
var h = "Hello"
```

Literal strings can contain a number of special characters, which are introduced by a backslash character as in the table below

Code	Character
\f	Form feed
\n	Newline
\r	Carriage return
\t	Tab
\u	Unicode character (followed by 4 hex digits)
\U	Unicode character (followed by 8 hex digits)
\x	Unicode character (followed by 2 hex digits)
\\	Backslash
\”	Quote

To create a string incorporating the morpho butterfly emoji, for example, you would use

```
"\U0001F98B morpho"
```

Hex digits are not case sensitive.

You may also insert the results of arbitrary expressions (see Chapter 5) into a *Morpho* string using *interpolation* as in the following example

```
print "Hello ${name}! Happy ${age} birthday"
```

The values of the variables `name` and `age` are converted to strings if necessary and joined with the surrounding string.

To convert something to a string, use the `String` constructor

```
print String(1.5)
```

Note that Strings in Morpho are *immutable*. Their contents cannot be changed, and operations on strings always return a new String:

```
print "ABC" + "DEF"
```

3.3 Lists

Lists are ordered sequences of values. They can be created either through the syntax

```
var a = [1,2,3]
```

or using the `List` constructor

```
var a = List(1,2,3)
```

which converts its arguments into a List.

Elements of a list can be accessed using the index notation

```
print a[0] // Prints the first element
```

Note that, like all Morpho collection objects, the index `0` represents the first element. You can change the value of any element in the List using analogous notation

```
a[0] = 10
```

You can access the last element using `-1`

```
print a[-1] // Prints the last element
```

and, more generally, negative numbers can be used to count from the end of the List.

3.3.1 Stacks

One application of a List is to implement a *stack*, which is a data structure to which values can be added to or removed from, following a Last In, First Out (LIFO) protocol.

We create the stack using an empty List

```
var stack = []
```

and can then implement key stack operations as follows:

- **Push** elements onto the stack:

  ```
  stack.append(1,2,3)
  ```

- **Pop** the last element off the stack:

  ```
  print stack.pop()
  ```

- **Peek** at the last element of the stack without removing it:

  ```
  print stack[-1]
  ```

- Check if the stack **is empty**:

  ```
  if (stack.count()==0) {
          // Do something
  }
  ```

We can use these basic operations to implement more complex operations, such as duplicating the last element of the stack

```
a=stack.pop()
stack.append(a,a)
```

or swapping the top two elements

```
a=stack.pop()
b=stack.pop()
stack.append(a,b)
```

3.4 Arrays

Arrays are multidimensional stores, and can be created as part of a variable declaration. Both of these examples create a 2×2 array:

```
var a[2,2]
```

or by a constructor

```
var a = Array(2,2)
```

Use index notation to set and get individual elements

```
a[0,0]=1
print a[0,0]
```

Each array element can contain any content that a regular variable can, so you may store Strings, Lists and even other Arrays in an Array:

```
a[0,0]="Hello"
a[1,0]=[1,2,3]
```

All Array elements are initialized to `nil` by default.

3.5 Dictionaries

Dictionaries, also called *hashtables*, *hashmaps* or *associative arrays*, are data structures that map a set of keys to a set of values. This simple example maps a few state codes to their capitals:

```
var cap = { "MA" : "Boston", "NY" : "Albany", "VT": "Montpellier" }
```

You could also use the Dictionary constructor

```
var cap = Dictionary("MA", "Boston", "NY", "Albany", "VT", "Montpellier")
```

where the arguments are, alternatingly, keys and values.

Access the contents of a Dictionary using index notation

```
print cap["MA"]
```

which also allows you to add additional key-value pairs

```
cap["ME"]="Augusta"
```

Any value can be used as a key in a Dictionary, but there is a subtlety: the behavior depends on whether the object is *immutable*. Basic types are immutable, and hence integers work as expected

```
var a = { 0: "Zero", 1: "One", 2: "Two"}
print a[0]
```

Lists, however, are

3.6 Ranges

Ranges describe a collection of values, denoted using the `..` and `...`. The Range

```
1..10
```

describes the collection of integers from 1 to 10 inclusive; the Range

```
1...10
```

(note the triple dots) represents the integers from 1 to 9; the upper bound is *excluded*. You can represent a range with an increment other than 1 like so

```
1..9:2
```

which comprises the odd numbers from 1 to 9. Ranges work with floating point numbers, e.g.

```
0..1:0.1
```

Ranges are frequently used to define the bounds of loops as will be described in Section X.

3.7 Complex numbers

Morpho supports complex numbers, which can be created using the keyword `im` to denote the imaginary part of a complex number

```
var z = 1 + 1im
```

You can use any number format for either the real or imaginary parts

```
print 0.1im
print 1e-3im
```

Complex numbers work with arithmetic operators just like regular numbers

```
print z + 1/z
```

Get the real and imaginary parts of a complex number using the convenience functions `real` and `imag`

```
print real(z)
print imag(z)
```

or by calling the corresponding methods

```
print z.real()
print z.imag()
```

Find the complex conjugate

```
print z.conj()
```

and obtain the magnitude and phase

```
print z.abs() // Or use the regular abs() function
print z.angle()
```

Note that the value ϕ returned by the `angle` method always lies on the interval $-\pi < \phi \leq \pi$; in some applications you will need to track the correct Riemann surface separately and add multiples of 2π as appropriate.

3.8 Tuples

Tuples, like Lists, represent an ordered sequence of values but are immutable. They are created either using the syntax

```
var t = (1,2,3)
```

or using the `Tuple` constructor

```
var t = Tuple(1,2,3)
```

Elements can be accessed using index notation

```
print t[1] // Prints the second element
```

but an attempt to change them throws an error

```
t[0] = 5 // Throws 'ObjImmutable'
```

Because Tuples are immutable, they can be used as keys in a Dictionary:

```
var dict = { (0,0): true }
print dict[(0,0)] // expect: true
```

Chapter 4

Comments

Documentation of code is one of the most important tasks of the programmer. Code is very hard to reuse, and often hard to understand, without documentation. We urge the programmer to include at least two kinds of comment:

- **Inline comments** are annotations intended to convey the intent of the programmer for what the code should do. They should be compact, clearly written and judiciously inserted. Include such comments wherever a section of code requires the reader to guess, or think deeply about a nontrivial sequence of statements, or convey assumptions that might not be obvious from the code.

- **Block comments** are longer pieces of documentation. It's useful to include a block comment at the start of a program to describe the program, when it was written, who implemented it and how it may be used.

Morpho supports two syntactic constructs to support comments. The first is introduced by the marker `//`. After that, any text *to the end of the current line* is ignored by the *Morpho* compiler. This style is useful for annotating a statement

```
var acceleration // in units of m/s^2
```

The second style of comment is introduced by `/*` and closed by `*/`. Anything between, including line breaks, is ignored by the compiler. Such comments may therefore span multiple lines. The following comment documents a new class and its usage

```
/* A new data structure.
   Brief description
   Usage: ... */
class DataStructure { }
```

Unlike the C language, from which the notation is derived, such comments can be nested within one another

```
/* A comment /* A nested comment */ */
```

which facilitates the common exploratory programming practice of "commenting out" a section of code.

While `//` is most obviously used for inline comments and `/* ... */` is oriented to block comments, either style can be used for either purpose depending on the programmer's sense of aesthetics.

Chapter 5

Expressions

An expression is any construct that produces a value. Expressions include any combination of literal values, function or method calls, assignment expressions, object constructors, as well as arithmetical, relational and logical operations.

5.1 Arithmetic operators

Morpho provides the standard binary arithmetic operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation

Hence, *Morpho* can be used, among other things, as an (over-engineered) pocket calculator like so:

```
print 2^8-1
```

5.2 Comparison operators

Comparisons between expressions can be achieved using relational operators,

Operator	Description
<	Less than
>	Greater than
==	Equal to
!=	Not equal to
<=	Less than or equal to
>=	Greater than or equal to

all of which return `true` or `false`. Not all expressions can be compared; Complex numbers for example, can be tested for equality but not compared so

```
print 1im < 2 // Invalid
```

throws a `InvldOp` error. Equality tests are more complicated than they appear. Comparisons of objects are only equal if they refer exactly to the same object, so

```
print [1,2,3] == [1,2,3]
```

prints `false` because two *different* Lists are created and compared, even though their contents are identical. A few object types support “deep” equality comparison, such as Strings and Tuples. Hence

```
print (1,2,3) == (1,2,3)
print "Hello" == "Hello"
```

both print `true`.

5.3 Logical operators

Finally, *Morpho* provides the logical operators

Operator	Description
&&	AND
||	OR
!	NOT

which implement the Boolean algebra. All of these operators consider the two values `false` and `nil` to be false; *any other value* is considered to be true. This is very different from C and some other languages, where the integer value `0`, and sometime even other values, are also considered to be false. **In *Morpho*, `0` (like any other number) is considered to be true.**

Like C, however, the logical operators do not always cause evaluate both operands to be evaluated. If the left operand of the AND operator is *false*, for example, the right hand operand is not evaluated because the expression is manifestly false. You can see this explicitly in the following example

```
fn f() {
  print "f was evaluated!"
  return true
}

print true && f() // f is evaluated
print false && f() // f is not evaluated
```

Conversely, if the left operand of the OR operator `||` is *true*, the right hand operand isn’t evaluated because it’s clear the composite expression must be true.

```
print true || f() //  f is not evaluated
print false || f() // f is evaluated
```

Since the NOT operator `!` has only one operand, it is always evaluated.

5.4 Assignment

The contents of variables can be changed using assignment expressions. The operator = is used to indicate assignment: the operator on the left hand is called the assignment *target* and is the variable or component to be modified; the operand on the right hand side is the value to be assigned.

An assignment statement can be as simple as assigning a value

```
foo = 1
```

or could involve changing the contents of a collection using index notation

```
foo[0] = 1
```

A common use of assignment is to capture the return value of a function

```
foo = sin(Pi/4)
```

Morpho also provides a number of shorthand assignment operators that retrieve and modify the contents of an assignment target and then store the result back in the same target:

- += Increments the target by a given value, e.g. `foo += 1`
- -= Decrements the target by a given value, e.g. `foo -= 1`
- *= Multiplies the target by a given value, e.g. `foo *= 2`
- /= Divides the target by a given value, e.g. `foo /= 2`

These shorthand operators are provided purely for the convenience of the programmer and each is entirely equivalent to a regular longhand assignment, e.g.

```
foo += 1
```

could equally be written as

```
foo = foo + 1
```

Because assignment operators in *Morpho* are expressions, they evaluate to a value which is always the value assigned. Hence

```
var a = 1
print a+=1
```

prints `2` which is the value of `a+1`

5.5 Other expressions

There are a few other types of expression and associated operators that are presented elsewhere in the book:

- The **Range constructors** `..`, `...` are discussed in Section 3.6.
- **Function calls** are introduced in Chapter 7.
- **Method calls** are denoted by a single dot `.` and are explained in Section XXX.

5.6 Precedence

Compound expressions like `1 + 2 * 3` may appear ambiguous at first sight because it is not obviously clear which of `(1 + 2) * 3 = 9` or `1 + (2 * 3) = 7` is meant. Running this example

```
print 1 + 2 * 3
```

prints 7 rather than 9 because the multiplication operator * binds to its operands with higher *precedence* than the addition operator +. The order of precedence in *Morpho* is as follows

	Operators	
Highest	`.`	Method call
	`^`	Power
	`-, !`	Unary minus, NOT
	`*,/`	Multiplication and division
	`+,-`	Addition and subtraction
	`.., ...`	Range constructor
	<, >, <=, >=	Comparison
	==, `!=`	Equality tests
	&&	AND
	`||`	OR
Lowest	=, +=, -=, *=, /=	Assignment

The programmer is always free to use parentheses to control the order of evaluation, so

```
print (1 + 2) * 3
```

indeed prints 9. It is recommended to do so even if an expression is formally correct, but challenging for the reader to parse.

Chapter 6

Statements

Statements in *Morpho* are the basic unit of a program: they are executed one after another as the program is run. Like other C-family languages, statements are organized into *code blocks* using curly brackets:

```
{
  var foo = "Hello World"
  print foo
}
```

Any variables created in a code block, referred to as *local variables*, cease to exist once the code block is over. Hence this example

```
{
  var foo
}
print foo // Throws an error
```

throws a `SymblUndf` error.

Code blocks are themselves statements and hence can be nested arbitrarily

```
{
  var foo = 1
  {
    var boo = 2
    print foo + boo
  }
}
```

Variables defined in an outer block are visible to code in an inner block, so both `foo` and `boo` are visible to the print statement, but the converse is not true.

As can be seen in the above example, it is common stylistic practice to *indent* statement within code blocks using tabs or spaces. In *Morpho*, and most other languages with the important exception of Python, indentation is purely aesthetic and done to improve readability; it has no special syntactic meaning.

6.1 Declarations

An important category of statements are declarations, which define various kinds of construct. Variable declarations were already introduced in Chapter 3. Function declarations will be described in Chapter7 and Class declarations in Chapter 9.

6.2 Expression statements

Any expression on its own is also a valid statement. Hence, assignment, function and method calls, etc., which are are all expressions, are also statements

```
a=5
foo("boo")
stack.pop()
```

6.3 Print statements

Morpho provides a simple way of producing output through the `print` keyword. The expression after `print` is output, most commonly to the Terminal if the terminal app is being used

```
print log(10)
```

Some objects are able to display themselves in a user-friendly manner (sometimes called *"pretty printing"*). Printing a List for example

```
print List(1,2,3)
```

displays something a List displayed in the *Morpho* syntax: `[ 1, 2, 3 ]`. Other objects don't provide this

```
print Object()
```

simply displays a placeholder `<Object>`.

Print statements are provided primarily for convenience and *always* follow the output with a newline. For more control over printing, the `System` class provides additional functionality as described in Chapter XXX.

6.4 Control structures

Morpho provides a typical variety of *control structures*, which control the order in which code is executed. Control blocks can conditionally execute code (`if ... else`), repeatedly execute code (`for`, `while`, `do ... while`), `break` out of a loop or `continue` to the next iteration. Morpho also provides a mechanism (`try ... catch`) to handle errors that are foreseen by the programmer.

6.4.1 If...else

An `if` statement evaluates the condition expression, and if it is true, executes the provided statement

```
if (a<0) a*=a
```

Note that, as discussed above in Section 5.3, wherever a condition test is performed in *Morpho*, all values (including the are considered to be equivalent to `true` *other than* `false` or `nil`.

The statement to be executed is often a code block

```
if (a.norm() < epsilon) {
  print "Converged"
}
```

You can provide a second statement using the `else` keyword that is executed is the condition test was false

```
if (q>0) {
  print "Positive definite"
} else {
  print "Not positive"
}
```

It's possible to chain `if` and `else` together to perform multiple tests, one after the other as in this fragment of a calculator

```
if (op=="+") {
  r = a + b
} else if (op=="-") {
  r = a - b
} else if (op=="*") {
  r = a * b
} else if (op=="/") {
  r = a / b
} else {
  print "Unknown operation"
}
```

Note that only one code block will be executed in such an `if ... else` *tree*.

6.4.2 For loops

A `for` loop is used to iterate over elements of a collection. You specify an iteration variable and the collection to iterate over enclosed in parentheses and using the `in` keyword; this is then followed by a statement to be repeatedly executed, the *loop body*. At each iteration, the iteration variable takes on successive values from the collection. This example prints the numbers 1 to 10 using a Range

```
for (i in 1..10) print i
```

where `i` is the iteration variable and here the loop body is a single `print` statement. Any collection can be used, for example this List of functions

```
for (f in [sin, cos, tan]) print f(Pi/2)
```

Loops may also use a code block for the body

```
for (r in collection) {
  // Do some processing
}
```

It's occasionally useful to access an integer index used to iterate over the collection, for example when working with two parallel collections.

```
for (q, k in lst) {
        p[k] = q
}
```

6.4.3 While loops

A `while` loop tests whether a condition test is true; if it is it then executes the loop body and this process is repeated until the condition test fails. They're particularly useful where the loop is modifying something as it iterates. For example, this loop prints the contents of a list in reverse order, popping them off one by one

```
var a = List(1..10)
while (a.count()>0) print a.pop()
```

This example reads the contents of a text file and prints it to the screen

```
var f = File("file.txt", "r") // Open file to read
while (!f.eof())  {
  print f.readline()
}
f.close()
```

Very occasionally, it's useful to make an infinite loop and terminate it based on a condition test somewhere in the middle. To do so, use the `break` keyword as will be discussed later in the chapter

```
while(true) {
        // ..
        if(somethingHappened()) break
        // ..
}
```

6.4.4 Do...while loops

A `do...while` loop is similar to a `while` loop, but the condition test is performed after the loop body has executed. Hence the loop body is always executed *at least once*. This is a skeleton Read-Evaluate-Print loop (REPL) loop that gets input from the user, processes it and displays the result, repeating the process until the user types “quit”:

```
do {
  var in = System.readline()
  // process input
} while(in!="quit")
```

6.4.5 C-style for loops

Morpho also provides a traditional C-style for loop. These are far less commonly used relative to the more modern `for ... in` syntax, but are occasionally useful. They have the following structure

```
for (initializer; test; increment) body
```

incorporating four elements:

- an **initializer** creates iteration variables and sets their initial variables.
- the **test** condition is evaluated, and the loop terminates unless the condition is true or equivalent to true.
- the **increment** is evaluated after each iteration, and is typically used to increment iteration variables.
- the **body** is evaluated each iteration as for other loops.

Hence the C-style loop

```
for (var i=0; i<5; i+=1) print i
```

is equivalent to the `for ... in` loop

```
for (i in 0...5) print i
```

6.4.6 Return, break, continue

There are three keywords that transfer control to a different point in the program. The most commonly used is **return**, which ends execution of the current function, and returns to the calling code. You may optionally provide an expression after **return**, which is the result of the function as returned to the caller. Because **return** is best understood in the context of functions, we defer further discussion of **return** to the next chapter.

The **break** statement exits the control structure and transfers execution to the code immediately *after* the structure. It's usually used to terminate a loop early. In this skeleton example, the programmer wants to perform up to a specified maximum number of iterations for an algorithm, but to finish once the algorithm has converged on the result

```
for (iter in 1...Niter) {
  if (hasConverged()) break
  // Do some work
}
// Execution continues here
```

On the other hand, **continue** is used, exclusively in loops, to skip immediately to the next iteration. It's often useful when processing a collection of data that includes elements that should be ignored. In this example, the condition checks whether an element in the given collection is callable, and if it isn't the **continue** statement causes the loop to go to the next element in the collection.

```
for (f in collection) {
  if (!iscallable()) continue
  var a = f()
  // process the result
}
```

Both **break** and **continue** should be used judiciously because they transfer control non-locally to another point in the program and hence introduce the possibility of confusion. It's always possible to replace them using **if**, but this too can lead to tangled code. The previous code could be written as

```
for (f in collection) {
  if (iscallable()) {
    var a = f()
    // process the result
  }
}
```

but there's a tradeoff—the extra level of indentation could make the code depending on the complexity of the processing code. The programmer should always keep in mind the clarity of the code written, and use one construct or another depending on which is clearer.

6.4.7 Try...catch

Morpho provides a type of statement, denoting using the keywords **try** and **catch**, that enables programs to handle error conditions that may be generated at runtime. This mechanism could be used, for example, by a program to recover if a file isn't found, or a resource is unavailable. The construct will be discussed more fully in Chapter 11.

Chapter 7

Functions

Functions are packages of code that accomplish a specific task. Sometimes called *subroutines* or *procedures* in other languages, the intent is the same: to modularize code into simple, understandable and reusable components. They can also be used to model the mathematical notion of a function, a *map* from parameter values onto results.

Morpho provides a number of useful functions as standard, e.g. trigonometric functions, which are *called* by providing appropriate parameter values or *arguments*.

```
print cos(Pi/4)
```

When Morpho encounters a function call, control is transferred to the function with the parameters initialized to the value of the supplied *arguments*. Once the function has accomplished its task, it *returns* a value that can be used by the code that called it. In our example, the value of $\cos(\pi/4) = 2^{-1/2}$ is returned by the `cos` function and then displayed by the **`print`** statement.

Function calls can occur anywhere where an expression is allowed, including as part of another expression or as an argument to another function call

```
print apply(cos, Pi/3 + arctan(1,0))
```

If the function is called without being used, the return value is simply discarded

```
cos(Pi/2)
```

To define a function, use the **`fn`** keyword. This must be followed by the *name* of the function, a list of parameters enclosed in parentheses and the *function body*, which is specified as a code block. Here's a function that simply doubles its argument

```
fn twice(x) {
  return 2*x
}
```

The **`return`** keyword, introduced above in Section 6.4.6, is used to introduce a return statement that indicates where control should be returned to the calling code. The expression after **`return`** becomes the return value of the function. A function can contain more than one **`return`** statement

```
fn sign(x) {
  if (x>0) {
    return "+"
  } else if (x<0) {
    return "-"
  } else return "0"
}
```

If no **return** statement is provided, the function returns `nil` by default.

Functions can have multiple parameters. Here's another example that calculates the norm of a two dimensional vector

```
fn norm(x, y) {
 var n2 = x^2 + y^2
 return sqrt(n2)
}
```

When `norm` is called, the x and y values must be supplied *in order*. These parameters are therefore referred to as *positional parameters*. They take on their value from the order of the arguments supplied. The calling code must call the function with the correct number of positional parameters, otherwise an `InvldArgs` error is thrown.

7.1 Optional parameters

Functions can also be defined with *optional parameters*, sometimes referred to as *keyword* or *named* parameters in other languages. Optional parameters are declared after positional parameters and must include a *default value*. This rather contrived example raises its argument to a power that can be optionally changed.

```
fn optpow(x, a=2) {
  return x^a
}
```

If `optpow` is called with just one parameter

```
print optpow(3) // Expect: 9
```

the default value `a=2` is used. But `optpow` can also be called specifying a different value of `a`

```
print optpow(3, a=3) // Expect: 27
```

You can define multiple optional parameters as in this template to find a root of a specified function

```
fn findRoot(f, method=nil, initialValue=0, tolerance=1e-6) {
  // ...
}
```

The caller can specify any number, including none, of the optional parameters so any of the following are valid

```
findRoot(f)
findRoot(f, tolerance=1e-8)
findRoot(f, tolerance=1e-6, initialValue=1)
```

Notice that the caller can supply optional parameters in any order; they need not correspond to the order in which they're provided in the function definition. The *Morpho* runtime automatically handles the correct assignment. If positional parameters are defined, however, they must be provided. Calling `findRoot` with no arguments would throw an `InvldArgs` error.

There are some restrictions on the default value of optional arguments. Currently, they may be any of `nil`, a boolean value, an Integer or a Float. For other kinds of values, use `nil` as the default value and check whether an optional argument was provided in the function. For FindRoot, the `method` parameter probably refers to some object or class that provides a user selectable algorithm. If no value of `method` is provided, the function should select a default algorithm like so

```
fn findRoot(f, method=nil, initialValue=0, tolerance=1e-6) {
  var m = method
  if (isnil(m)) m = DefaultMethod()
  // ...
}
```

Optional arguments are best used for functions that perform a complex action with many independent user-selectable parts, particularly those that may only needed infrequently. They alleviate the user of having to remember the order parameters must be given, and allow customization.

7.2 Variadic parameters

Occasionally it is useful for a function to accept a variable number of parameters. A variadic parameter is indicated using the `...` notation, as in this example that sums its arguments:

```
fn sum(...x) {
  var total = 0
  for (e in x) total+=e
  return total
}
```

When called, the parameter `x` is initialized as a special container object containing the arguments provided. It's valid to call `sum` with no arguments provided, in which case the container `x` is empty.

You may only designate one variadic parameter per function. Hence this example

```
fn broken(...x, ...y) { // Invalid
}
```

throws a `OneVarPr` error when compiled.

It's possible to combine positional and variadic parameters, as in this example that computes the L_n norm of its parameters (at least for $n \geq 2$)

```
fn nnorm(n, ...x) {
  var total = 0
  for (e in x) total+=e^n
  return total^(1/n)
}
```

When a function that accepts both positional and variadic parameters is called, the required number of argument values are assigned to positional parameters first, and then any remaining arguments are assigned to the variadic parameter. Hence, you must call a function with at least the number of positional parameters. Calling `nnorm` with no parameters will throw a `InvldArgs` error.

It's also required that the variadic parameter, if any, comes after positional parameters. This example

```
fn broken(...x, y, z) { // Invalid
}
```

would throw a `VarPrLst` error on compilation.

Optional parameters must be defined after any variadic parameter. We could redefine `nnorm` to make it compute the L_2 norm by default like this

```
fn nnorm(...x, n=2) {
  var total = 0
  for (e in x) total+=e^n
  return total^(1/n)
}
```

7.3 Multiple dispatch

While *Morpho* functions, by default, accept any value for each parameter, it's often the case that a function's behavior depends on the type of one or more of its arguments. You can therefore define functions to restrict the type of arguments accepted, and you can even define multiple implementations of the same function that accept different types. The correct implementation is selected at runtime—this is known as *multiple dispatch*—depending on the actual types of the caller. It is sometimes clear to the compiler which implementation will be called, in which case the compiler will select this automatically. *Morpho* implements multiple dispatch efficiently, and so the overhead of this relative to a regular function call is small.

Consider this skeleton intended to compute the gamma function, a mathematical special function that is related to factorials for integer values, and which is also defined for the complex plane:

```
fn gamma(Int x) {
  if (x>0) return Inf
  return factorial(x-1)
}

fn gamma(Float x) {
  // An implementation for Floating point numbers
}

fn gamma(Complex x) {
  // Another implementation
}
```

When `gamma` is called, one of the three implementations is selected depending on whether the argument is an Integer, Float and Complex number. If no implementation is available, the `MltplDsptchFld` error is thrown. This could happen, for example, if `gamma` is called with a List by mistake. Implementations need not have the same number of arguments. Here's a collection of implementations that return the number of arguments provided:

```
fn c() { return 0 }
fn c(x,y) { return 1 }
fn c(x,y,z) { return 2 }
fn c(x,y,z,w) { return 3 }
```

Multiple dispatch can occur on any combination of positional parameters and not all positional parameters need to be typed. The ordered collection of types accepted by the positional arguments of an implementation is known as its *signature*. This enterprising collection joins Strings to Lists, producing a String.

```
fn join(String x, List y) { return x + String(y) }
fn join(String x, String y) { return x + y }
fn join(List x, String y) { return String(x) + y }
fn join(List x, List y) { return String(x) + String(y) }
```

It's an error to define two implementations with the same signature within the same scope.

7.4 Documentation

We highly recommend documenting each function with a comment before (or close to) the function definition. The set of parameters is known as the *interface* of the function, and it's recommended to document the meaning and purpose of each parameter, as well as any restrictions, constraints or required units. There are many valid styles to accomplish this, but the importance of documenting interfaces cannot be emphasized enough.

Chapter 8

Functions as data

Functions in *Morpho* are objects just like Lists, Strings, etc. Hence, they can be assigned to variables, and called at a later point

```
var P = sin
print P(Pi/10)
```

Functions can also be stored in collections. This example computes the value of several trigonometric functions, which are stored in a list:

```
var lst = [sin, cos, tan]
for (f in lst) print f(Pi/3)
```

Finally, functions can be passed as arguments to other functions. For example, here's a function that applies a given function twice to its second argument

```
fn calltwice(f, x) {
  return f(f(x))
}
```

You can then use `calltwice` with any function, as in this example:

```
fn sqr(x) {
  return x^2
}

print calltwice(sqr, 2)
```

8.1 Anonymous functions

Morpho provides an abbreviated syntax for functions that can be used in assignments or as parameters. There's no need to give the function a name—such functions are hence called *anonymous*—and you may, optionally, provide a single statement as the body in place of the usual code block. The value of the body statement is returned from the function as if a **return** was in front of it. Hence

```
var sq = fn (a) a^2
```

is equivalent to the named function

```
fn sqr(a) {
  return a^2
}
```

The anonymous function syntax is particularly useful for supplying to other functions, because quite often such functions end up being quite short. The List class, for example, provides a `sort` method that can be used to sort the contents. You can optionally provide a sort function that compares two elements of the list a and b; this function should return a negative value if $a < b$, a positive value if $a > b$ and 0 if a and b are equal. This example sorts the list in reverse order

```
var lst = [5,2,8,6,5,0,1,3,4]
lst.sort(fn (a,b) b-a)
print lst
```

In some languages, anonymous functions are referred to as *lambda* functions, referring to the pioneering work of Alonzo Church on the theory of computation.

8.2 Scope

Functions obey scope so functions can be defined locally within a code block as in this example

```
{
  fn f(x) { return x^2 }
  print f(2) // prints 4
}

print f(2) // Raises an error
```

The function `f` remains available for the rest of the code block, but is not visible outside of it. Hence, while the first call works, the second throws `SymblUndf` as `f` is no longer visible.

8.3 Closures

Functions can be *returned* from other functions; such functions are known as *closures* for reasons that will become apparent shortly. Here's an example

```
fn inc(a) {
  fn f(x) { return x + a }
  return f
}

var add = inc(5)
print add(10) // prints 15
```

The function `inc` manufactures a closure that adds a given value `a` to its argument. This can be a bit complicated to follow, so let's trace out the sequence of events:

1. `inc` is called with the argument `5`. During the call, the parameter `a` takes on this value inside the `inc` function.

2. A closure using the local function `f` is created within `inc` using the provided value of `a` (i.e. 5).

3. The closure is returned to the calling code. The value of a remains available to the closure.

4. The user calls the closure with the argument 10; the closure adds 5 to this and returns 15 which is displayed.

Closures are so-named because they *enclose* the environment in which they're created. In this example, the value of a is encapsulated together with the function f, forming the closure. The quantity a as is sometimes called an *upvalue*, because it's not local to the function definition; a is said to be *captured* by the closure.

Upvalues can be written to as well as read from. This closure reports how many times it has been called

```
fn counter(val) {
  fn f() { val+=1; return val }
  return f
}

var c = counter(0)
print c() // prints 1
print c() // prints 2
print c() // prints 3
```

Closures can be called anywhere regular functions can. An important use of closures is to create functions that obey a defined interface, i.e. they have the same signature, but have access to additional parameters. This example creates a function that describes the electric scalar potential due to a point charge with given charge and position

```
fn scalarPotential(q, x0, y0) {
  fn phi(x,y) {
    return q/sqrt((x-x0)^2 + (y-y0)^2)
  }
  return phi
}

var p1 = scalarPotential(1, -1, 0)
var p2 = scalarPotential(-1, 1, 0)

print p1(0,1) + p2(0,1)
```

The closures created can then be called with position of interest, returning the appropriate value. This could be useful in a larger code, where potential functions for many different types of entity are to be created, but each type requires very different data to specify them. Nonetheless, because all such potential functions obey the same interface, they can be used interchangeably.

Chapter 9

Classes and Objects

Morpho, in contrast to many dynamic languages, is strongly oriented towards object oriented programming (OOP). The central idea behind OOP is to encapsulate related data or *properties* into packages called *objects* that also supply a collection of actions or *methods* that can be performed on them. Methods are much like functions—they have parameters and return values—except they are always called with reference to a particular object. A method call is specified in *Morpho* using the `.` operator:

```
var a = [1,2,3]
print a.count()
```

Here, the `count` method returns the number of entries in a List. Many *Morpho* objects provide the same method. The left hand operand of the `.` operator is called the *receiver* of the call and the label `count` is called the *selector*. Like functions, method calls can have parameters and return values.

To define new object types, use the **`class`** keyword. A *class* provides the definition of an object type in that it comprises the methods available to the object; objects themselves are *instances* of a particular class and are made using a *constructor*. Let's create a simple `Pet` class with two methods, `init` and `greet`:

```
class Pet {
  init(name) {
    self.name = name
  }

  greet() {
    print "You greet ${self.name}!"
  }
}
```

To create a `Pet` *instance*, we write a constructor, which uses the same syntax as a function call.

```
var whiskers = Pet("Whiskers")
```

Since *Morpho* classes generally begin with capital letters—although the language doesn't enforce this explicitly—you can usually spot constructors easily in code.

The `init` method that we defined in `Pet` is special: if provided, it's called *immediately* after the object is created, *before* the constructed object is returned to the user, and is intended to prepare the object for use. Here, we store the provided name in the object's `name` property. Note that the `init` method must **not** return a value; if you try to use **`return`** in its definition, the compiler will throw an `InitRtn` error. It's also a good idea to avoid complex code in the `init` method; if your object requires significant setup, or is complicated to create, consider using the Builder pattern described in Section 13.1.

Classes are themselves objects in *Morpho*; it's occasionally useful to call methods on a class rather than on an instance.

9.1 Inheritance

A key goal of object oriented programming is *reuse* of code. In class-based languages like Morpho, you can create a new class that reuses the methods provided by a previously defined class using the `is` keyword; the prior class is called the *parent* of the new class, which becomes its *child*. Here, the `Cat` class inherits from `Pet`:

```
class Cat is Pet {
  hiss() {
    print "${self.name} hisses!"
  }
}
```

Any methods defined in the parent class are copied into the child class, so `Cat` acquires `init` from `Pet`, but also defines a new method `hiss`.

9.2 Multiple inheritance

A programmer may wish to make a class by combining unrelated functionality from different classes, a design strategy known as *composition*. If classes only inherit from one parent—a paradigm called *single inheritance*—this is challenging and requires special techniques to overcome the limitation. To support composition, *Morpho* classes can inherit from multiple parents. Let's create a class, Walker, that describes something that can `walk`.

```
class Walker  {
  walk() {
    print "${self.name} walks!"
  }
}
```

We can define a `Dog` class by composing `Pet` and `Walker`

```
class Dog is Pet with Walker {
  bark() {
    print "${self.name} barks!"
  }
}
```

The `Dog` class inherits `init` from `Pet` and `walk` from `Walker`.

If two parents (or their parents) provide the same method, there is a special mechanism called *method resolution* that determines which implementation is actually inherited by a class; we'll describe it in section 9.4 below.

9.3 Multiple dispatch

Like functions, methods utilize multiple dispatch: You may define multiple implementations that accept different types of argument. In languages like C++, similar functionality is provided by *overloading*, multiple dispatch generalizes this idea.

This sketch implementation of a class provides different services for various kinds of Pet:

```
class PetHotel {
        lodge(Dog x) {
                print "${x.name} gets a private room!"
        }

        lodge(Cat x) {
                print "${x.name} is at home in the Cattery!"
        }

        lodge(Pet x) {
                print "Unfortunately, we lack the facilities to look after
                   ${x.nam}."
        }

        lodge(x) {
                Error("PetRqd", "This hotel is for Pets").throw()
        }
}
```

As described in Section 7.3 for functions, when the `lodge` method is called the correct implementation is selected at runtime. The most *specific* implementation is the one selected: If `lodge` is called with a Dog or Cat, the first or second implementation is used respectively. If another kind of Pet—including subclasses that the implementors of PetHotel don't know about when the class was defined—is used, the third implementation of `lodge` is able to provide a user-friendly message. Any other kind of value triggers the fourth implementation, which raises an error. Multiple dispatch uses the entire signature of a method, i.e. all positional arguments, to select the correct implementation and not just one parameter as in some languages.

9.4 Method resolution

If more than one parent or ancestor classes provide methods with the same name, the following rules apply:

1. **Priority.** Method implementations have priority given by a *linearization* of the class structure; i.e. the compiler computes a ordering of the parent classes consistent with ordering relationships expressed in the class definitions. For each class definition, priority of the parents is given left to right, so the direct parent given after `is` has priority over those parents specified by `with`, which then take priority in the order given. The algorithm used is called C3 linearization[1], and for our simple `Dog` example yields the ordering `Dog, Pet, Walker`: Methods in `Dog` have priority over those in `Pet`, which have priority over those in `Walker`. You can obtain the linearization computed by the compiler for a class by calling the `linearization` method on the class, e.g. `print Dog.linearization()` (an example of *reflection* described in the next section). Not all possible class structures admit a C3 linearization; an error is raised at compile time where a linearization cannot be computed.

2. **Similarity.** Methods with the same *name* and the same *signature* are considered to be *similar*. When assembling a new class's methods from its definition and parents, the implementation whose class that

[1]Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, P. Tucker Withington (1996-06-28). "A Monotonic Superclass Linearization for Dylan". OOPSLA '96 Conference Proceedings. ACM Press. pp. 69–82. CiteSeerX 10.1.1.19.3910. doi:10.1145/236337.236343. ISBN 0-89791-788-X.

comes first in the priority list is used. Hence, an implementation in `Dog` would replace one in `Pet` or `Walker`. An error is raised if a single class definition provides two similar implementations.

While these rules may seem complicated, in most cases they correspond to what the programmer expects. Pathological examples certainly exist, and the programmer is advised to be wary of deep inheritance structures. Favor composition over inheritance is a commonly advised design principle.

9.5 Reflection

Like many dynamic languages, *Morpho* provides facilities for code to discover an object's class, properties and methods at runtime. The methods to do so are:

1. `clss` Returns an object's class. As noted above, classes are themselves objects in *Morpho*.

2. `has` Tests if an object possesses a property. Either supply a property label as a string, e.g `has("foo")` or call with no arguments to get a list of properties.

3. `respondsto` Tests if an object provides a method. Either give a method label as a string, e.g. `respondsto("foo")` or call with no arguments to get a list of methods.

Chapter 10

Protocols

A collection of methods that more than one class implements is called a *protocol*. For example, all standard classes permit cloning by implementing a method called `clone`. Some classes support addition and other arithmetical operations by implementing methods called `add`, `sub` etc. Protocols are a major feature in dynamic languages, because they enable code to work with many kinds of objects regardless of their inheritance hierarchy. A rather trivial function that scales its argument, for example,

```
fn scale(x) {
  return 2*x
}
```

immediately works with Integers, Floats, Matrices, Sparse matrices, Complex numbers, for example, and will also work with any future object that implements a `mul` method.

10.1 Morpho protocols

Morpho's standard collection of types implement a number of protocols as we describe here.

10.1.1 Clone

Objects that can be cloned provide a method called `clone` that makes a copy of the object. Typically, this is a *shallow* copy, namely the object itself is cloned by the contents are merely copied (List is a good example). Since most objects inherit from Object, this method is provided by default.

10.1.2 Count

The `count` method returns the number of constituent values included in a collection. A List returns its length, a Dictionary returns the number of key/value pairs, a Matrix returns the number of entries, etc.

10.1.3 Enumerate

The `enumerate` method enables a collection to participate in `for ... in` loops. The loop code calls `enumerate` repeatedly with a single integer value indicating the requested entry; the collection should return the corresponding value. If a negative value is supplied, the collection object should return the total number of entries in the collection.

10.1.4 Accessing collections

Collection objects may provide two methods that facilitate access to the collection:

- `index(i)` Retrieves the value corresponding to the index `i`. The `index` method can be defined with any number of parameters, or using variadic parameters, to support multi-dimensional collections. The programmer must supply the correct number of indices when the collection is accessed or an `ArrayDim` is thrown.
- `setindex(i, val)` Sets the value in the collection corresponding to index `i` to be `val`. As for `index`, `setindex` can be defined with more than one index parameter; the value to be set is always the *last* parameter.

Retrieving information from a collection using the syntax `a[1,2]` is translated into a call `a.index(1,2)` at runtime; similarly setting a value `a[1,2]=2` is translated into `a.setindex(1,2,2)`.

10.1.5 Arithmetic

Objects may support arithmetic operations. When code like

```
var a = b + c
```

is encountered, *Morpho* first checks if it knows how to perform the operation. If not, it checks to see if the left hand operand, `b`, provides a method called `add`. If it does, the addition is redirected to `b`'s `add` method using `c` as the argument

```
var a = b.add(c)
```

If `b` doesn't provide an `add` method, *Morpho* checks to see if `c` provides an `addr` method. If so, this is called with `b` as the argument

```
var a = c.addr(b)
```

Analogous methods `sub`, `mul`, `div` and "right associated" versions `subr`, `mulr`, `divr` handle subtraction, multiplication and division respectively and allow the programmer to support non-commutative algebras.

10.2 Defining new protocols

Some languages[1] provide a specific *protocol* keyword to define protocols. To keep things simple, *Morpho* doesn't do this but you can achieve much the same effect with classes and multiple inheritance. This class defines a protocol for objects that are cookable, i.e. respond to a `cook` method

```
class Cookable {
  cook() { }
}
```

Now let's define a few objects that implement this protocol. Kale gets most of its properties from its Plant parent class (not shown), but is also cookable:

```
class Kale is Plant with Cookable {
  cook() { print "It wilts deliciously!" }
}
```

So is this Cake, which in the absence of any other parent classes simply inherits directly from Cookable:

[1] Swift is a good example

```
class Cake is Cookable {
  cook() { print "Cooked to perfection!" }
}
```

It’s often helpful to restrict a function or method to accept only objects that implement a particular protocol, which can be done with a type annotation on the relevant parameter. Here’s a function that makes dinner, and accepts only Cookable objects:

```
fn dinner(Cookable w) {
  print "Making dinner"
  w.cook()
}
```

Chapter 11

Errors

Computer programs written in any language may encounter unexpected or challenging situations. The user might request that the program opens a file, for example, but the file doesn't exist. An algorithm might call for the solution of a linear system, but the matrix turns out to be poorly conditioned. Both these examples are foreseeable by the programmer at the time of writing, so an appropriate course of action can be taken by the program. Perhaps the user should be informed that the file didn't exist or that the algorithm didn't succeed, or perhaps an alternative algorithm is available.

There are three kinds of error in *Morpho*:

- **Compilation errors** are thrown when the provided code is incorrect. If the *Morpho* compiler can't find a symbol, it throws `SymblUndf`, for example. Compilation errors prevent the code from running at all; they must be fixed by the programmer rather than handled by the code.

- **Runtime errors** are thrown during execution of the program and indicate that execution cannot proceed further. Execution therefore halts, and the runtime environment displays a message describing the error and where it occurred.

- **Runtime warnings.** Occasionally, a situation occurs that isn't strictly an error, but something is unusual that the user should be told about. An algorithm that solves a problem may wish to report that the quality of the solution is poorer than expected. Or perhaps the user used deprecated functionality, and the program wishes to suggest an alternative. Warnings do not interrupt execution; they simply display a highlighted message to the user.

All errors in Morpho have two components: a short *tag* that identifies the error, e.g. `SymblUndf`, and a longer *description*. This structure is intended to support internationalization, because locale-appropriate descriptions could be loaded, and also support a need for customized error messages that can include useful information.

Errors are instances of the `Error` class, which can be used to create new errors. You supply the tag and a default error message in the constructor

```
var myErr = Error("MyTag", "Default error message")
```

Once the error is created, call the `throw` method to throw the error

```
myErr.throw()
```

Any error can also be used as a *warning* as described above by using the `warning` method

```
myErr.warning()
```

You can also call `throw` or `warning` with a custom message provided as a String, which is useful if you want to provide more information to the user about what happened

```
myErr.throw("File ${foo} was missing.")
```

11.1 Handling errors

While not necessarily expected, some errors are at least foreseeable. A well written program should handle such errors gracefully, and provide an alternate course of action if possible. If the user requests a file that isn't available, the program can notify them of this fact and request a different file, for example.

Morpho provides a control structure to handle errors using the `try` and `catch` keywords

```
var f

try {
  f = File(fname, "r")
} catch {
  "FlOpnFld":
    print "File ${fname} not found"
}
```

The `try` statement describes code to be executed that is anticipated may throw an error. The `catch` statement defines an *error handler*, a collection of errors that can be handled and code to handle them. You provide the appropriate tag for each error to handle, here just `FlOpnFld`, and a corresponding statement to execute if and only if the error is generated. You may use the `break` keyword within a `catch` statement, which allows you to escape from the error handler entirely.

If the code in the `try` block generates `FlOpnFld`, control is transferred to the corresponding `catch` statement, which in this case prints an error. If the `try` statement doesn't thrown any errors, code in the `catch` statement isn't executed.

What if the `try` block throws an error that isn't handled by the corresponding `catch` statement? Error handlers work on a stack, so every time `try` is executed, the *Morpho* runtime adds the corresponding handler to the stack; once the `try ... catch` statement is finished executing the error handler is removed from the stack. More than one error handler can be active if `try ... catch` statements are nested, for example, or if a function called within a `try` statement provides its own error handler. In any case, when an error is thrown, the *Morpho* runtime looks at the most recent error handler, i.e. the handler from the most recently executed `try` statement. If the error can be handled by the handler, then control is transferred to it; if not, the runtime walks back along the stack of error handlers until one is found that can handle the error. If *no* suitable handler is found, the error is reported the user and interrupts execution as in the absence of `try`.

Because error handlers can transfer control very non-locally—jumping outside of multiple nested function or method calls in extreme cases—they should be used with care. It's a good idea to document errors each function might throw, and to consider carefully possible errors that might need to be handled by your code.

Note that *Morpho* does **not** provide a "default" error handler, i.e. a section of a `catch` block that catches all errors. This is because the intent of `try ... catch` is to handle *foreseen* errors only.

Chapter 12

Libraries

Morpho is a modular environment. A number of standard libraries[1] are provided, and you can easily write and distribute your own. To use a library, use the `import` keyword

```
import color
```

When compiling this code, the *Morpho* compiler looks for a library called `color`—really a file names `color.morpho`—in a number of standard locations that depend on the platform and how *Morpho* was installed; the collection of search locations is called the *environment*. *Morpho packages*, bundles of code, help files, documentation etc. can be added to the environment by the user and provide a structured mechanism to extend its capabilities.

Once a library has been imported, anything it defines becomes available to the user: classes, functions and even global variables. These collectively are referred to as the *symbols* defined by the library. The `color` module provides a few named colors, a `Color` class, as well as `ColorMap` objects that map a scalar parameter to a range of colors (useful for plotting).

You can also load a *Morpho* file using `import`

```
import "mylibrary.morpho"
```

In this case, the compiler looks for the file `"mylibrary.morpho"` in the same folder as your code; it doesn't search the environment for the file. You can load files in subfolders using UNIX path syntax

```
import "folder/mylibrary.morpho"
```

If necessary, the *Morpho* compiler will translate the UNIX path into a platform specific descriptor.

Unlike in some languages, `import` can *only* be used in the global scope, but it can be used at any suitable point in the program. You may also import more than one library per line

```
import color, meshtools
```

and these are then imported in the order specified.

12.1 Importing selected symbols

Sometimes, the programmer only wants to use a subset of the features of a library. To do this, use the `for` keyword together with `import` to select which symbols to import. This statement imports a special color-accessible `ColorMap` from the `color` module

[1]The *Morpho* runtime distinguishes between *modules*, which are written in Morpho, and *extensions*, which are written in C or another compiled language. Both are used in the same way and the distinction is transparent to programmers other than those seeking to implement their own module or extension.

```
import color for ViridisMap
```

Note that *no other* symbols from `color` are imported, not even `ColorMap` from which `ViridisMap` inherits.

Judicious use of `import ... for` improves the robustness of code. As libraries are updated, new symbols may be added that could conflict with a symbol in the code using the library. By making only the components needed accessible, the possibility of conflicts is reduced.

12.2 Namespaces

Avoidance of conflicts between libraries motivates a more general construct called a *namespace*. These are containers for symbols defined at compile time; any library can be imported into a designated namespace using the `as` keyword, e.g.

```
import color as col
```

where `col` is the name of the namespace created to contain the `color` libraries definitions. Once a namespace has been defined, its contents can be accessed using the `.` operator, similar to a property lookup:

```
var c = col.Red()
```

A namespace lookup can be used *anywhere* where a regular identifier could be used, e.g. in constructors, function definitions and function calls, type specifiers, etc. The compiler attempts to locate the given the symbol in the namespace, and throws an error `SymblUndfNmSpc` if it cannot be found.

12.3 Multiple includes

Morpho will *never* load the same library twice, even if it is imported more than once, and even if it is imported using `for` or `as` as described below. Different `import` statements can even use the library in different ways—Morpho will automatically ensure the correct symbols are available.

Chapter 13

Advanced Topic: Design Patterns

Object oriented code often utilizes *design patterns*, code templates that show how to achieve a desired effect. The term is inspired by *pattern books*, volumes that contain plans or diagrams to aid in making something. Such books are still used nowadays to sew or weave material, and in the 19th century were a common way of building a house without the expense of hiring an architect. A very famous computer science book, *"Design Patterns: Elements of Reusable Object-Oriented Software"*[1] popularized the idea and provided a number of design patterns in widespread use. In this chapter, we show how to implement a selection of these patterns that we have found particularly useful in writing *Morpho* code. Certain *Morpho* libraries use these patterns, and knowledge of the terminology can help the reader understand their design. In some cases, the *Morpho* implementation differs from the original pattern because of the language's features. Also noteworthy is the idea of an *anti-pattern*[2], a design approach that may be obvious, but contains undesirable features and is to be avoided. We do not dwell on anti-patterns here—they are to be avoided after all!—but a good programmer should read one of the many excellent texts on the subject.

13.1 Builder

The Builder pattern facilitates creation of objects that are complicated or expensive to create—geometric data structures such as Meshes are a good example—or must be constructed in multiple stages. They accomplish this by separating initialization code into a separate class. Another use case is where the type of object created may depend on parameters the user supplies: a MatrixBuilder, for example, might create a regular Matrix or a ComplexMatrix depending on whether the contents are real or complex numbers. A Builder is initialized with parameters describing the object to be created, and provides one method:

- **build()** Constructs and returns the requested object.

```
class Builder {
  init(parameter) {
    self.parameter = parameter
  }

  build() {
    var obj = Object() // Make object
```

[1]Gamma, Erich, et al. *"Design Patterns: Elements of Reusable Object-Oriented Software"*. Germany, Addison-Wesley, 1995.

[2]Originally coined in Koenig, Andrew (March–April 1995). "Patterns and Anti-patterns". Journal of Object-Oriented Programming. 8 (1): 46–48; there have been many books on the subject of anti-patterns in software design and project management.

```
    // Initialize it
    return obj
  }
}
```

If the object requires multiple steps to create it, `build` can be replaced by appropriate build stages.

13.2 Visitor

The Visitor pattern accomplishes the task of processing a collection of heterogeneous objects. A Visitor object is constructed with the collection to process, and provides two methods:

- **visit()** Processes a single element of the collection. Multiple implementations of this method are provided that process different kinds of object.
- **traverse()** Loops over the collection, calling the **visit()** method for each object in turn. The correct implementation is selected automatically by multiple dispatch.

```
class Visitor {
  init(collection) {
    self.collection = collection
  }

  visit(Type1 a) {
    // Do something
  }

  visit(Type2 b) {
    // Do something
  }

  visit(x) {
    // Default
  }

  traverse() {
    for (p in self.collection.contents()) self.visit(p)
  }
}
```

13.2.1 Example: SVG export for vector graphics

To illustrate use of the Visitor class, let's create an example of a collection that might benefit from it. A 2D vector graphics image comprises various graphics primitives, e.g. lines, circles, polygons, etc. Here's a simple implementation that incorporates a couple of basic primitives:

```
class GraphicsPrimitive {}

class Circle is GraphicsPrimitive {
  init(x,y,r) { self.x = x; self.y = y; self.r = r }
}

class Line is GraphicsPrimitive {
  init(x1,y1,x2,y2) { self.start = [x1,y1]; self.end = [x2,y2] }
}

class Graphics2D {
  init() { self.contents = [] }

  append(GraphicsPrimitive x) { // Adds a primitive to the collection
    self.contents.append(x)
  }

  contents() { return self.contents }
}
```

The user can then build up an image[3] by creating a blank Graphics2D object and adding elements one by one. Here, we create a disk enclosed by a square:

```
var g = Graphics2D()
g.append(Circle(50,50,50))
g.append(Line(0,0,0,100))     // Left
g.append(Line(100,0,100,100)) // Right
g.append(Line(0,0,100,0))     // Top
g.append(Line(0,100,100,100)) // Bottom
```

Now imagine that we would like to export our Graphics2D object to a file, e.g. SVG, PDF, postscript, etc. The exporter class must traverse the contents of the `Graphics2D` object one by one and build up the output. Here's an example of a working SVG Exporter that accomplishes this using the Visitor pattern:

```
class SVGExporter {
  init(Graphics2D graphic) {
    self.graphic = graphic
  }

  visit(Circle c, f) {
    f.write("<circle cx=\"${c.x}\" cy=\"${c.y}\" r=\"${c.r}\"/>")
  }

  visit(Line l, f) {
    f.write("<line x1=\"${l.start[0]}\" y1=\"${l.start[1]}\"
       x2=\"${l.end[0]}\" y2=\"${l.end[1]}\" stroke=\"black\"/>")
  }

  visit(x, f) { } // Do nothing for unrecognized primitives

  export(file) {
    var f = File(file, "w")
```

[3]A good potential use of a Builder pattern!

```
    f.write("<svg xmlns=\"http://www.w3.org/2000/svg\">")
    for (p in self.graphic.contents()) self.visit(p, f)
    f.write("</svg>")
    f.close()
  }
}
```

To use the exporter, the user first creates an `SVGExporter`, passing the `Graphics2D` object to the constructor, and then calls the `export` method with a desired filename:

```
var svg = SVGExporter(g)
svg.export("pic.svg")
```

The `export` method creates the requested file, generates header information, and then loops over the content of the Graphics2D object, calling the visit method for each object.

13.2.2 Advantages

The Visitor pattern enforces modularity by separating representation from processing of data. Since Graphics2D is intended to represent a vector image abstractly it shouldn't also provide the unrelated functionality of export facilities to particular filetypes. This separation makes it much easier to extend the program. A new graphics primitive could be defined without changing Graphics2D; only the SVGExporter would need to be modified, by adding an additional implementation of `visit`. Similarly, support for additional graphics formats could be achieved without modifying the existing code above *at all*. All that needs to be done is to create a new class analogous to SVGExporter for the desired filetype.

13.2.3 Comparison with other languages

Readers familiar with OOP design patterns may notice that the Visitor pattern described here is slightly different from the traditional implementation. In languages that lack multiple dispatch, the Visitor's `export` method first calls a method called `accept` on each object in the collection, rather than calling `visit` on the Visitor itself:

```
export(file) {
  // Create file and write header
  for (p in self.graphic.contents()) p.accept(self, f)
  // Write footer and close file
}
```

Each Graphics primitive must provide an `accept` method that in turn calls an appropriate method on the Visitor. For the Circle primitive, `accept` might call a method called `visitCircle`:

```
accept(visitor, f) {
  visitor.visitCircle(self, f)
}
```

The Visitor must therefore provide a different method for each object type. Here's the implementation of `visitCircle`:

```
visitCircle(c, f) {
  file.write("<circle cx=\"${c.x}\" cy=\"${c.y}\" r=\"${c.r}\"/>")
}
```

The procedure described, where the Visitor calls `accept` on the object being processed that then dispatches the call to the correct method on the Visitor, is called *double dispatch*. It's more convoluted and less efficient, requiring two method calls per processed object, than the multiple dispatch implementation. It's also less compact and less modular: with double dispatch, each primitive must provide an `accept` method which is unnecessary in the multiple dispatch version. Indeed, all the functionality of the Visitor with multiple dispatch is contained within the Visitor, which could be highly advantageous if a collection is from a library designed without the Visitor pattern in mind and where the code can't easily be modified.

Another advantage of the multiple dispatch is that a fallback `visit` method for unrecognized object types is trivially implemented. In other languages, adding a new primitive requires immediately modifying *all* Visitor objects to avoid potentially raising an ObjLcksPrp error; in the multiple dispatch implementation, unknown primitives are simply routed to the fallback `visit` method and ignored.

Part II Tools for *morpho*

Chapter 14

The *morpho* debugger

Morpho provides a debugger that allows you to pause execution of a program, examine the state of variables and registers, and continue. To enable it run *morpho* with the `-debug` command line switch. Note that there is a performance penalty for running with debugging enabled.

You may set hard breakpoints in your code—places where morpho will always pause—by inserting an `@` symbol. For example:

```
@
print "Hello World"
```

will break immediately before executing the print statement. When morpho reaches one of these breakpoints, it enters debugging mode:

```
---Morpho debugger---
Type '?' or 'h' for help.
Breakpoint in global at line 5 [Instruction 15]
@>
```

The `@>` prompt reminds you that you're in the debugger rather than in interactive mode. You can then perform a number of commands to understand the current state of the virtual machine, set additional breakpoints, examine the contents of variables and registers, etc. Most commands have a long form, e.g. "break" or "clear" and a short form "b" or "x" respectively. Debugger commands are largely consistent with those for the `gdb` tool.

Some of the debugging features require knowledge of how *morpho's* virtual machine works, which is documented in Chapter 21.

14.1 Debugging commands

14.1.1 Break

The `b` command sets a breakpoint:

`b lineno` Break at a given line number.

`b *instruction` Break at a given instruction.

`b functionname` Break at a given function.

`b Class.methodname` Break at a given method.

14.1.2 Continue

Continues program execution, leaving the debugger.

14.1.3 Disassemble

Displays disassembly for the current line of code.

14.1.4 Garbage collect

Forces a garbage collection.

14.1.5 Clear

Clears a breakpoint. The syntax is the same as for `b`. Note the abbreviation is `x` not `c`.

14.1.6 Info

Info reports on various features of the virtual machine.

`i address n` Displays the physical address of the object in register *n*. *[This is primarily useful when debugging morpho itself]*

`i break` Displays all active breakpoints

`i globals` Displays the value of all globals

`i global n` Displays the contents of global *n*

`i registers` Displays registers for the current function call

`i stack` Displays the current stack

`i help` Displays a list of valid info commands

14.1.7 List

Prints a program listing of the lines around the current execution point.

14.1.8 Print

Prints the value of variables.

`p symbol` Prints the value of a given symbol

`p` Print all currently visible symbols

14.1.9 Set

Prints the value of a variable or register.

`set r n = <expr>` Sets the value of register *n* to be <*expr*>.

`set <symbol> = <expr>` Sets the value of variable <*symbol*> to be <*expr*>.

Expressions must be simple constant values.

14.1.10 Quit

Terminates program execution.

14.1.11 Step

Continues execution, but returns to the debugger at the next line.

14.1.12 Trace

Shows the current execution trace, i.e. the list of functions and method calls that the program has made to get to the current point.

Chapter 15

The *morpho* profiler

Morpho provides a simple profiler to help identify bottlenecks in the program. To use it run morpho with the `-profile` command line switch. As the program runs, a separate monitor thread runs independently and samples the state of the morpho virtual machine at regular intervals, deducing at each time which function or method is in use. At the end of program execution, the profiler prints a report. A sample run[1] might produce something like:

```
===Profiler output: Execution took 51.019 seconds with 272450 samples===
issame                              32.98% [89866 samples]
Delaunay.dedup                      15.63% [42580 samples]
(garbage collector)                 13.41% [36528 samples]
List.ismember                       7.16% [19518 samples]
Delaunay.triangulate                6.40% [17450 samples]
List.enumerate                      3.58% [9750 samples]
Show.trianglecomplexobjectdata      2.59% [7065 samples]
Circumsphere.init                   2.24% [6091 samples]
OneSidedHookeElasticity.integrandfn 1.77% [4834 samples]
Matrix.column                       1.25% [3412 samples]
(anonymous)                         1.25% [3406 samples]
List.count                          1.01% [2758 samples]
Range.enumerate                     0.90% [2451 samples]
...
```

On the first line, the profiler reports the time elapsed between the start and end of executing the program (which does not include compilation time) and the total number of samples taken. In subsequent lines, the profiler reports the name of a function or method, the number of samples in which the virtual machine was observed to be in that function, and the overall fraction of samples as a percentage. The list is sorted so that the most common function is reported first. The profiler reports on both user-implemented functions and morpho functions and methods that are implemented in C (but visible to the user).

There are some special entries: anonymous functions are reported as `(anonymous)`; time in the global context, i.e. outside of a function or method is reported as `(global)`; time spent in the garbage collector is reported as `(garbage collector)`, here on the third line. Garbage collection in this example is frequently $\sim 10\%$ of execution time; if it becomes significantly higher, this may suggest your program is creating too many temporary objects.

How to interpret and act on profiler data is something of an art form. In the above example, the largest

[1]on the *morpho* example `examples/meshgen/sphere.morpho`

fraction of execution time was spent in a relatively function, `issame`, that compared two objects. An obvious strategy would have been to simply reimplement the function in C, which would have undoubtedly improved the performance. However, on inspecting the code it was realized that `issame` was actually being called by `Delaunay.dedup` to remove entries from a data structure, and that by using a different data structure this step could be entirely eliminated providing a significant performance gain.

Hence, optimization involves not only thinking about the performance of individual pieces of code, but also the data structures and algorithms being used. The profiler simply directs the programmer's attention to the most time consuming bits of code to avoid optimizing sections of code that aren't called frequently.

Part III Extending *morpho*

Chapter 16

Extending *morpho*

Developers interested in adding new features to *morpho* can utilize one of the following mechanisms for expansion:

- **Modules** are written in the *morpho* language and are loaded with the `import` keyword. Creating a module is no different than writing a *morpho* program!
- **Extensions** are written in C or C++ using the *morpho* C API. These are also loaded with the `import` keyword; the distinction between modules and extensions is purposefully not visible to the user; a module could be reimplemented as an extension with the same interface, for example.
- **Contributing to the *morpho* source code.** Changes to the core data types, improvements to the compiler, etc. could be incorporated into the *morpho* source directly. We highly recommend connecting to the *morpho* developers before doing this to check if the idea is already being worked on, or whether there is guidance or advice on how specific features should work. Bug reports, suggestions for new features are welcome; please see `CONTRIBUTING.md` in the *morpho* git repository.

Modules and extensions can be distributed in their own git repository as a package. See Chapter 17 for further information.

We also recommend contributors look at the `CONTRIBUTING.md` and `CODE_OF_CONDUCT.md` documents in the *morpho* git repository for information and advice about contributing to *morpho*, as well as how ethical standards for participation in our community.

This developer guide is only partially complete and represents a work in progress: We are gradually adding in helpful information to assist developers.

Chapter 17

Morpho packages

To facilitate convenient distribution, *morpho* supports a lightweight notion of a *package*, which is a folder containing files that collectively provide some enhancement or functionality. A package is simply a git repository that contains some or all of the following file structure:

/share/modules for `.morpho` files that define a module

/share/help for `.md` files containing interactive help (see below).

/bin for compiled executables (these may be produced during installation).

/lib for compiled extensions (these may be produced during installation).

/src for source files

/examples for examples

/test for test files

/manual if a package is sufficiently complex to require a manual

README.md information about the package, installation etc.

Morpho searches both its base installation and all known packages when trying to locate resources. A simple and experimental package manager, *morphopm*, has been created to help users obtain and install packages; the structure above is intended to be sufficiently simple that different installation approaches could be supported.

We recommend naming your package with the *morpho*- prefix. Please let the morpho development team know about interesting packages!

17.1 *Morpho* help files

Morpho's interactive help system utilizes a subset of the Markdown plain text formatting system. Help files should be put in the `/share/help` folder of your package so that *morpho* can find them.

Each entry begins with a heading, for example:

```
# Entry
```

Using different heading levels indicates to *morpho* that a topic should be included as a subtopic. Here, the two heading level 2 entries become subtopics of the main topic, which uses heading level 1:

```
# Main topic

## Subtopic 1

## Subtopic 2
```

Morpho's help system supports basic formatting, including emphasized text:

```
This is *emphasized* text.
```

and lists can be included like so:

```
* List entry
* Another list entry
```

Code can be typeset inline,

```
Grave accents are used to delimit ‘some code‘
```

or can be included in a block by indenting the code:

```
    for (i in 1..10) print i
```

The terminal viewer will syntax color this automatically.

Morpho (ab)uses the Markdown hyperlink syntax to encode control features. To specify a tag or keyword for a help entry, create a hyperlink where the label begins with the word `tag`, and include the keyword you'd like to use in the target as follows:

```
## Min
[tagmin]: # (min)

Finds the minimum value...
```

This unusual syntax is necessary as Markdown lacks comments or syntax for metadata, and we use hyperlinks to encode text in a way that is valid Markdown but remains transparent to regular Markdown viewers. The # is a valid URL target, and the construction in effect 'hides' the text in parentheses. Since hyperlink labels (the part in square brackets) must be unique per file, add any text you like, typically the name of the tag, after the word `tag`.

Similarly, to tell the help viewer to show a table of subtopics after an entry, add a line like this:

```
[showsubtopics]: # (subtopics)
```

Any characters after `showsubtopics` in the label are ignored, so you can add additional characters to ensure a unique label.

Chapter 18

Extensions

Morpho extensions are dynamic libraries that are loaded at runtime. From the user's perspective, they work just like modules through the `import` statement:

```
import myextension
```

When the compiler encounters an import statement, it first searches to see if a valid extension can be found with that name. If so, the extension is loaded and compilation continues.

Extensions are implemented in C or any language that can be linked with C. A minimal extension looks like this:

```
// myextension.c

#include <stdio.h>
#include <morpho/morpho.h>
#include <morpho/builtin.h>

value myfunc(vm *v, int nargs, value *args) {
    printf("Hello world!\n");
    return MORPHO_NIL;
}

void myextension_initialize(void) {
    builtin_addfunction("myfunc", myfunc, BUILTIN_FLAGSEMPTY);
}

void myextension_finalize(void) {
}
```

All *morpho* extensions **must** provide an initialize function, and it **must** be named `EXTENSIONNAME_initialize`. In this function, you should call the morpho API to define functions and classes implemented by your extension, and set up any global data as necessary. Here, we add a function to the runtime that will be visible to user code as `myfunc`.

Morpho extensions **may** but are not required to provide a finalize function, with a similar naming convention to the initializer. This function should deallocate or close anything created by your extension that isn't visible to the *morpho* runtime. Here, the function data structures are handled by the morpho runtime so there's no finalization to do.

The remaining code implements your extension. Here, we implement a very simple function that conforms to the interface for a "builtin" function. The function just prints some text and returns `nil`.

Compiling an extension manually

To compile the above code, it's necessary to ensure that the morpho header files are visible to your compiler. They could be copied from the morpho git to `/usr/local/include/morpho` for example, but may be found in other places if morpho has been installed with homebrew or another package manager.

You need to compile this code as a dynamic library. For example on the macOS with clang,

```
cc -undefined dynamic_lookup -dynamiclib -o myextension.dylib myextension.c
```

The `-dynamiclib` option indicates that the target should be a dynamic library. The `-undefined dynamic_lookup` option indicates to the linker that any undefined references should be resolved at runtime.

Compiling an extension manually

We highly recommend using the CMake build system for extensions. Examples of how to do this are to be found e.g. in the ZeroMQ extension. We aim to add additional examples in future.

Packaging an extension

As for *morpho* modules, we advise hosting your extension in a git repository with *morpho-* as the prefix and with the file structure as suggested in chapter 17. We recommend including the C source files in `/src` and compiling your extension to `/lib`, where it can be found by *morpho*. We highly recommend including interactive help files in `/share/help` and examples in `/examples` as well. All extensions should have a `README.md` explaining what the extension is for and how the user should install it.

The new morphopm package manager provides an automated way to build extensions that should help with installation. For now, the above recommendations should ensure your basic file structure is future-proof.

We also note that the C API is not yet entirely stable; this will occur at v1.0.0. As we gain experience writing extensions and identify common needs, we anticipate improving the API. We welcome your feedback.

Chapter 19

The *morpho* C API

Morpho, like many languages, is implemented in C for performance reasons. Extensions to *morpho* can also be written in C, or in any language that can link with C.

19.1 Basic data types

19.1.1 Value

A `value` is the most basic data type in *morpho*. At any time, a value can contain any *one* of:

- A signed integer, equivalent to an `int32`.
- A `double` precision floating point number.
- A pointer to an object.
- A boolean value indicating `true` or `false`.
- The value `nil` representing no information.

The structure of a value is kept opaque for performance reasons; setting and getting a value must be done through macros and functions provided by *morpho:*

- **Initialize a `value` with a literal.** Macros provided include `MORPHO_NIL`, `MORPHO_TRUE`, `MORPHO_FALSE`. You can also use `MORPHO_INTEGER()` and `MORPHO_FLOAT()` to create integers and floats respectively.
- **Convert a `value` to a C type.** `MORPHO_GETINTEGERVALUE`, `MORPHO_GETFLOATVALUE`, `MORPHO_GETBOOLVALUE`, `MORPHO_GETOBJECT`. In general, it's important to be sure that a value contains the type you expect before using these. Use `morpho_valuetoint` or `morpho_valuetofloat` for example to convert a generic value to an integer or double respectively; these functions return true if they succeed.
- **Test whether a value is of a certain type.** Use `MORPHO_ISNIL`, `MORPHO_ISINTEGER`, `MORPHO_ISFLOAT`, `MORPHO_ISOBJECT`, `MORPHO_ISBOOL`. Do **not** use a direct comparison with a literal, because the `value` implementation is intentionally opaque and such comparisons may fail. In other words, do this

  ```
  if (MORPHO_ISNIL(val)) ... // Correct
  ```

  and not

  ```
  if (val==MORPHO_NIL) ... // Incorrect
  ```

Additionally, a number of utility functions exist to compare values:

- `MORPHO_ISEQUAL(a, b)` tests if two values are equal. For strings, etc. this involves a detailed comparison.
- `MORPHO_ISSAME(a, b)` tests if two values refer to the same object (or are equal if they are not objects). This macro is intended to be faster than ISEQUAL.
- `morpho_ofsametype(value a, value b)` returns true if a and b have the same type.
- `MORPHO_ISNUMBER(value a)` returns true is a is a number (i.e. integer or float).
- `morpho_valuetoint`, `morpho_valuetofloat`, `MORPHO_INTEGERTOFLOAT`, `MORPHO_FLOATTOINTEGER` provide conversion between types.
- `MORPHO_ISFALSE`, `MORPHO_ISTRUE` test if a value is true or false.

19.1.2 Objects

Objects are data types that require memory allocation, and are implemented as C `struct`s that always begin with a field of type `object`. This design enables *type munging*, i.e. casting any object to a generic `object` type, but with the ability to infer the type of an object at a later point. To store a pointer to an object in a `value`,

```
value v = MORPHO_OBJECT(objectpointer)
```

Many macros are provided to detect what kind of object is present, for example `MORPHO_ISSTRING`, `MORPHO_ISLIST`, `MORPHO_ISMATRIX`, `MORPHO_ISSPARSE`. Once you have determined the type of an object, you can then use macros like `MORPHO_GETSTRING`, `MORPHO_GETLIST` or similar to retrieve a pointer of the correct type. For some types, convenience macros such as `MORPHO_GETCSTRING` are provided to enable easy access to object fields from a `value`.

New types of object can be defined; see Section 19.4.

You **must not** store a *generic* pointer in a value using MORPHO_OBJECT, only a suitably defined `struct` will work. Passing a generic pointer to the virtual machine will likely cause a segfault, and it may occur at a random point (i.e. when the garbage collector runs).

19.1.3 Varrays

Variable length arrays are arrays that dynamically adjust in size as new members are added. They're a very useful type that differs only by the type contained in them. Hence, `varray.h` provides two convenient macros to create them for a specific type. Suppose we want to define a varray of integers: to do so, we would include in an appropriate `.h` file the statement:

```
DECLARE_VARRAY(integer, int)
```

and then

```
DEFINE_VARRAY(integer, int)
```

in our `.c` file. These definitions would create

```
void varray_integerinit(varray_integer *v); // Initialize a varray
bool varray_integeradd(varray_integer *v, int data[], int count); // Add a
   specified number of elements
```

```c
bool varray_integerresize(varray_integer *v, int count); // Resize the varray
int varray_integerwrite(varray_integer *v, int data); // Add one element,
   returns number of elements
void varray_integerclear(varray_integer *v); // Clear the varray
```

Where we want to use an integer varray, we would write something like

```c
varray_integer v;
...
/* Initialize the varray */
varray_integerinit(&v);
...
/* Write an element to the varray */
varray_integerwrite(&v, 1);
...
/* Deinitialize the varray */
varray_integerclear(&v);
...
```

Check the *morpho* source before defining a new varray type for a C type in an extension—it may cause conflicts. Note that `varray.h` defines varrays for `char`, `int`, `double` and `ptr_diff`; value.h defines varrays for `value`. You should not redefine these.

19.2 Implementing a new function

Creating a new builtin morpho function requires the programmer to write a function in C with the following interface:

```c
value customfunction(vm *v, int nargs, value *args);
```

Your function will be passed an *opaque reference*[1] to the virtual machine `v`, the number of arguments that the function was called with `nargs`, and a list of arguments `args`. You must **not** access the argument list directly, but rather use the macro `MORPHO_GETARG(args, n)` to get the `n`th argument. You **must** return a `value`, which may be `MORPHO_NIL`. For example, a simple implementation of the `sin` trigonometric function might look like this:

```c
value sin_fn(vm *v, int nargs, value *args) {
        if (nargs!=1) /* Raise error */;
        double input;
        if (morpho_valuetofloat(MORPHO_GETARG(args, 0)) {
                return MORPHO_FLOAT(sin(input));
        } else /* Raise error */
        return MORPHO_NIL;
}
```

To make the function visible to morpho, call `builtin_addfunction` in your extension's initialization function, e.g.

```c
builtin_addfunction("sin", sin_fn, BUILTIN_FLAGSEMPTY);
```

[1] i.e. a pointer of type `void *`. This is specifically intended to prevent developers from relying on a particular implementation of the virtual machine.

From morpho, the user can then use the new `sin` function as if it were a regular function.

There are a few important things to note about defining your own *morpho* C-function. First, these functions are naturally variadic; if your function is called with the wrong number of arguments, you are responsible for raising an error as described in Section 19.6. If you create any new objects in your function, you **must** bind them to the virtual machine as described in Section 19.7.

You can call *morpho* code from within your function with certain restrictions, see Section 19.8 for details.

19.2.1 Optional parameters

Your function may accept optional parameters just as regular *morpho* functions do. The library function `builtin_options` enables you to retrieve these values, as in the below example:

```
static value fnoption1;
static value fnoption2;

init(void) { // Initialization function called when your extension is run
        fnoption1=builtin_internsymbolascstring("opt1");
        fnoption2=builtin_internsymbolascstring("opt2");
}

value opt_fn(vm *v, int nargs, value *args) {
        int nfixed; // Number of fixed args.
        value optval1 = MORPHO_NIL;        // Declare values to receive
        value optval2 = MORPHO_INTEGER(2); // optional parameters
        if (!builtin_options(v, nargs, args, // Pass through
                                  &nfixed,        // Number of fixed
                                     parameters is returned
                            2,               // Number of possible optional args
                            fnoption1, &optval1, // Pairs of symbols and values
                               to receive them
                            fnoption2, &optval2) return MORPHO_NIL;

        // ...
}
```

Symbols for the optional parameters **must** be declared in your extension's initialization function by calling `builtin_internsymbolascstring`; this is for performance reasons and enables the runtime to detect optional arguments. You must keep a record of the returned value; typically this is done in a global variable (`fnoption1` and `fnoption2` in the example above).

In your function implementation, you call `builtin_options` with, `v`, `nargs` and `args` as passed to you by the runtime, an pointer of type int* to receive the number of fixed parameters detected, and then a list of optional parameters and their associated symbols. If `builtin_options` detects that a particular optional argument has been supplied by the user, the corresponding `value` is updated.

You must check the return value of `builtin_options`, which returns `false` on failure. Where this occurs, you must return as quickly as possible from your function, cleaning up any memory you allocated that has not been bound to the virtual machine as per Section 19.7 and returning `MORPHO_NIL`.

19.3 Implementing a new class

There are two implementation patterns to define a new `morpho` class:

1. The new class reuses `objectinstance` and all information is stored in properties of the object. These are visible to the user, can be edited by the user using the property notation, and are accessible from within C code using `objectinstance_getproperty` and `objectinstance_setproperty`. The user may subclass a class implemented in this way and override method definitions. Many functionals use this strategy, as it's very lightweight. It does have some limitations: you may **not** use this strategy if you need to store C pointers that refer to something that isn't an `object` or if you want the very fastest possible performance since property access is relatively expensive.

2. You create a new object type (see Section 19.4) which can include arbitrary information including C pointers to non-objects. You then create what is referred to as a *veneer class,* (see Section 19.5), a *morpho* class that that defines user-accessible methods. While more cumbersome, this pattern provides the fastest possible performance, but object properties are *not* visible to the user and the resulting class cannot be subclassed by the user. The *morpho* source uses this pattern extensively: many examples are to be found in `src/classes/` in the git repository; the List or Range classes are good places to look.

Both of these use a similar approach to define the class, which is in effect simply a collection of method implementations. Indeed, methods have exactly the same interface as C-functions.

```
value mymethod(vm *v, int nargs, value *args);
```

and are written in the same way. From within a method definition, the macro `MORPHO_SELF(args)` returns the object itself as a `value`.

Once you have defined your method implementations, you must tell the *morpho* runtime about your class. First, a set of macros are provided to create the appropriate class definition, e.g. for the Mesh class,

```
MORPHO_BEGINCLASS(Mesh)
MORPHO_METHOD(MORPHO_PRINT_METHOD, Mesh_print, BUILTIN_FLAGSEMPTY),
    MORPHO_METHOD(MORPHO_SAVE_METHOD, Mesh_save, BUILTIN_FLAGSEMPTY),
/* ... */
MORPHO_ENDCLASS
```

You call the `MORPHO_BEGINCLASS` macro with a name for your class (this need not be the user-facing name). You then use `MORPHO_ METHOD` repeatedly to specify each method. The first argument is the user-facing method label, which is often a macro. The second argument is the C function that implements the method. The final argument is a list of flags that can be used to inform morpho about the method. These are reserved for future use and `BUILTIN_FLAGSEMPTY` is sufficient. Finally, you use `MORPHO_ENDCLASS` to finish the class definition.

Having defined the available methods you must then call `builtin_addclass` in your initialization code to actually define the class. For the Length functional, this would look like:

```
builtin_addclass(LENGTH_CLASSNAME, MORPHO_GETCLASSDEFINITION(Length),
    objclass);
```

The first argument, `LENGTH_CLASSNAME`, is the user-visible name for the class. The second argument is the class definition. Use the macro `MORPHO_GETCLASSDEFINITION` to retrieve this, supplying the name you used with `MORPHO_BEGINCLASS`. The final argument is the parent class. Often, we want this to be Object, and we can retrieve this like so:

```
objectstring objclassname = MORPHO_STATICSTRING(OBJECT_CLASSNAME);
value objclass = builtin_findclass(MORPHO_OBJECT(&objclassname));
```

19.4 Implementing a new object type

Objects are heap-allocated data structures with a special format that enables the *morpho* runtime to infer type information. Other than the simple types that are held in a value, most *morpho* data structures including strings, lists, dictionaries, matrices, etc. are implemented as objects. The *morpho* object system is readily extensible, and it's easy to create new ones. Note that objects are not necessarily visible to the user, to facilitate this it's also necessary to define what is known as a *veneer class* (Section 19.5 which defines methods that can be called from *morpho* code).

To illustrate the definition and implementation of a new object type, in this section we'll implement a new objectfoo type. It's a good idea to declare new object types in a separate pair of implementation (.c) and header (.h) files. Many similar examples are to be found in `src/classes` in the morpho source.

In the header file, we'll begin by declaring a global variable as `extern` that will later contain the objecttype. We'll also declare a macro to refer to the objecttype later.

```
extern objecttype objectfootype;
#define OBJECT_FOO objectfootype
```

Now we can declare an associated structure and type for objectfoos. Let's make them as simple as possible, simply storing a single `value`.

```
typedef struct {
        object obj;
        value foo;
} objectfoo;
```

Note that we **must** start the structure declaration with a field of type `object` that is reserved for *morpho* to use. All object structures are **required** to have the first field be an `object`, because *morpho* uses this field to detect the object type. The remainder of the structure can contain arbitrary information; here we just declare the value.

It's a good idea to implement convenience macros to check if a value contains a particular type of object, and to retrieve the object from a value with the correct pointer type. For example:

```
/** Tests whether an object is a foo */
#define MORPHO_ISFOO(val) object_istype(val, OBJECT_FOO)

/** Gets the object as a foo */
#define MORPHO_GETFOO(val)   ((objectfoo *) MORPHO_GETOBJECT(val))
```

You may also declare other macros to retrieve fields

```
/** Gets the foo value from a foo */
#define MORPHO_GETFOOVALUE(val) (((objectfoo *) MORPHO_GETOBJECT(val))->foo)
```

In the implementation file, we will create a global variable to hold the `objectfootype`; this will be filled in by initialization code.

```
objecttype objectfootype;
```

We can then begin defining an objectfoo's functionality. The first step is to implement a constructor function, which should allocate memory for the object and initialize it. This will involve calling,

```
object *object_new(size_t size, objecttype type)
```

to create a new object with a specified size and type. If `object_new` returns a non-NULL pointer, allocation was successful and you can initialize the object. The constructor for an `objectfoo`, for example, might be:

```
objectfoo *object_newfoo(value foo) {
        objectfoo *new = (objectfoo *) object_new(sizeof(objectfoo),
           OBJECT_FOO);
        if (new) new->foo=foo;
        return new;
}
```

You could provide more than one constructor to create your object from different kinds of input. For example, we could declare `object_foofromllist` to create a foo from an objectlist. Prototypes for the constructors should be added to the header file.

To interface our new object type with the morpho runtime, we need to define several functions. Where these are listed as optional, they can be set to `NULL` in the definition.

- `void objectfoo_printfn (object *obj, void *v)` Called by *morpho* to print a brief description of the object, e.g. `<Object>`. If the object's contents are short and can be conveniently displayed (as for a string), printing the contents is allowable. Detailed information or printing of complicated objects (e.g. a matrix) should *not* be implemented here; it should go in a veneer class (see Section 19.5). *Note that in Morpho 0.6 and higher the virtual machine is passed as an argument, and you should use* `morpho_printf` *to print to morpho's output stream.*

  ```
  void objectfoo_printfn (object *obj, void *v) {
          morpho_printf(v, "<Foo>");
  }
  ```

- `void objectfoo_freefn(object *obj)` [Optional] Called when the object is about to be free'd, providing an opportunity to free any private data, i.e. data that is otherwise invisible to the virtual machine. Almost always, objects that are referred to by a `value` are not required to be free'd. For example anything that has been passed to you by the virtual machine, or that you have created and bound to the virtual machine with `morpho_bindobjects`, should **not** be free'd. Rather, this is for memory that your object has allocated independently with `MORPHO_ALLOC`. Since our objectfoo doesn't have any private data, we can actually skip this function.

- `void objectfoo_markfn(object *obj, void *v)` [Optional] Called by the garbage collector to find references to other objects. You should call `morpho_markobject`, `morpho_markvalue`, `morpho_markdictionary` or `morpho_markvarrayvalue` as appropriate to inform the garbage collector of these references. Since we have a reference to a value in the foo field, we just need to call `morpho_markvalue`. Failing to inform the garbage collector correctly of references your object holds can cause random crashes; to help identify these compile with `MORPHO_DEBUG_STRESSGARBAGECOLLECTOR`.

  ```
  void objectfoo_markfn (object *obj) {
          morpho_markvalue( ((objectfoo *) obj)->foo );
  }
  ```

- `size_t objectfoo_sizefn(object *obj)` Should return the size of the object. You **should** include the size of any private data you hold, but **should not** include the size of anything in a value that has been passed to you or bound to the virtual machine. Hence, we simply return the size of the struct. If the estimates returned by this function are incorrect, *morpho* programs using your object will still most likely run correctly, but the garbage collector may run either too frequently, impacting performance, or not frequently enough, potentially causing the program to run out of memory.

```
void objectfoo_sizefn (object *obj) {
        return sizeof(objectfoo);
}
```

- `hash objectfoo_hashfn(object *obj)` [Optional] Called by the dictionary data structure to compute a hash from the object. You **must only** define this if your object type is immutable, i.e. cannot be modified once created. If this were the case for an objectfoo, we could then hash the contents using one of the functions in `src/datastructures/dictionary.h`. Note that if you define this function, you must also define a comparison function that compares the contents of the object.

```
hash objectfoo_hashfn (object *obj) {
        return dictionary_hashvalue( ((objectfoo *) obj)->foo );
}
```

- `int objectfoo_cmpfn(object *a, object *b)` [Optional if no hash function] Called by the morpho runtime to test for equality between two objects. If this isn't defined, *morpho* assumes by default that two values are only equal if they are identical (i.e. refer to the same object). This function should return one of `MORPHO_EQUAL`, `MORPHO_NOTEQUAL`, `MORPHO_BIGGER` or `MORPHO_SMALLER` depending on whether an ordered comparison is meaningful or not. The library function `morpho_comparevalue` is available to compare two values.

```
int objectfoo_cmpfn(object *a, object *b) {
        objectfoo *af = (objectfoo *) a;
        objectfoo *bf = (objectfoo *) b;

        return morpho_comparevalue(af->foo, bf->foo);
}
```

These functions should be collected together in a `objecttypedefn` structure, which is normally declared statically:

```
objecttypedefn objectfoodefn = {
        .printfn=objectfoo_printfn,
    .markfn=objectfoo_markfn,
    .freefn=NULL,
    .sizefn=objectfoo_sizefn,
        .hashfn=objectfoo_hashfn, // Or NULL if we want a mutable type
        .cmpfn=objectfoo_cmpfn // Or NULL to prevent deep comparisons
};
```

Now all these functions have been defined, we must add the following line to initialization code,

```
objectfootype=object_addtype(&objectfoodefn);
```

which registers the objectfoodefn with the morpho runtime and returns an objectype, which we record for use elsewhere.

Nothing requires us to expose a new object type to the user; we can use such an object purely for internal purposes. Most objecttypes, however, provide a veneer class as we'll discuss in the following section.

19.5 Veneer classes

A veneer class is a morpho class definition that the runtime uses whenever the corresponding object is encountered. Such a class provides a “veneer” over a regular *morpho* object that enables the user to interact with it like any other object. If the user calls `clone()` on a value that contains an `objectmatrix,` morpho uses the veneer class to select the method to call. This can sometimes happen implicitly: For example, if morpho tries to add a float to an `objectmatrix`, morpho looks up the veneer class for an `objectmatrix` then tries to invoke the `add` or `addr` method as appropriate.

Veneer classes are defined in the same way as regular C classes. The programmer must provide method implementations, define the class and register it with the runtime as described in Section 19.3. The only difference, of course, is that `MORPHO_SELF` contains a reference to the specific object type. Most *morpho* data structures have veneer classes, and many examples can be found in `src/classes/` in the git repository; `list.c` and `range.c` are good places to look. These files follow a common pattern: they begin with the object definition, then provide C functions to work with the new object type, and then define the veneer class. We recommend extensions adopt a similar pattern for new object types and associated veneer classes.

To register a class as a veneer class, one additional step is required in initialization code. For example, in `list_initialize` in the file `src/classes/list.c` is the line:

```
object_setveneerclass(OBJECT_LIST, listclass);
```

which registers the class `matrixclass` (this was the return value of a previous call to `builtin_addclass`) as the veneer class for objects of type `OBJECT_LIST`. The user therefore sees an `objectlist` referred to in a value as a *morpho* `List` and can interact with it accordingly.

19.6 Error handling

When an error occurs, you can report this to the user by setting the virtual machine to an error state.

In order to do so, it’s first necessary to define the error. *Morpho* errors are defined by a *tag*, a short label that is used to identify the error and a detailed *message* for the user. The separation of tags from messages permits locale-dependent display of error messages to be implemented in future versions of *morpho*. The tag and a default message are usually defined as macros in a header file. For example, this common error comes from `src/datastructures/error.h`:

```
#define VM_OUTOFBOUNDS                      "IndxBnds"
#define VM_OUTOFBOUNDS_MSG                  "Index out of bounds."
```

To define the error with the morpho runtime, you need a line like this in your initialization function:

```
morpho_defineerror(VM_OUTOFBOUNDS, ERROR_HALT, VM_OUTOFBOUNDS_MSG);
```

The first argument is the tag and the last is the message. The second argument is the error type, which can be any of:—

- `ERROR_WARNING` Indicates a warning, rather than an error.
- `ERROR_HALT` Indicates an error that should halt execution.
- `ERROR_LEX` Indicates a lexing error; used by lex.c/.h [Use only if extending the lexer]
- `ERROR_PARSE` Indicates a parser error; used by parse.c/.h [Use only if extending the parser]
- `ERROR_COMPILE` Indicates a compiler error. [Use only if extending the compiler]

The values ERROR_USER, ERROR_DEBUGGER and ERROR_EXIT are reserved for use by the runtime.

When the error occurs in your code, you can then report this fact by calling morpho_runtimerror:

```
morpho_runtimeerror(v, VM_OUTOFBOUNDS);
```

You provide the virtual machine reference that was passed to you and the error tag. Use any existing error tags defined in the morpho header files rather than duplicating the error.

Once you have reported the error to the VM, your function should return as swiftly as possible, returning MORPHO_NIL. You are responsible for freeing any newly created objects or other allocated memory, and undoing side-effects like open files unless you have already registered these with the virtual machine via morpho_bindobjects.

In come circumstances, your program may detect some condition that, while not an error that would prevent further progress, nonetheless should be raised with the user. An example scenario might be where a numerical problem is poorly conditioned, and hence the solution is likely to be of poor quality. Morpho provides an analogous function to raise a given error as a warning. You can use this for any error, even if it was not declared with ERROR_WARNING; errors declared with ERROR_WARNING are always raised as warnings.

```
morpho_runtimewarning(v, VM_OUTOFBOUNDS);
```

If your program detects a state that arises from incorrect programming, you may use the UNREACHABLE macro (see Section 20.2.5) to terminate *morpho* immediately. An example where this might be desirable is if an inconsistency or impossible condition is detected in one of your data structure*s*. Use this primarily for your own debugging purposes; the user should never see an internal consistency error.

19.7 Memory management

The *morpho* runtime provides garbage collection: the user need not worry about deallocating any object. The actual garbage collector implementation is intentionally opaque and a target of continuous improvement. The C programmer typically interacts with the garbage collector in two ways: First, new object types must provide a markfunction to enable the garbage collector to see any values stored within an object as described in Section 19.4. Second, where new objects are created **and returned to the user** these should typically be bound to a virtual machine as will be described below.

Morpho uses the following model for memory management. Generic blocks of memory can be allocated, free'd and reallocated using the following macros:

```
x = MORPHO_MALLOC(size)
MORPHO_FREE(x)
MORPHO_REALLOC(x, size)
```

If your code allocates memory using MORPHO_MALLOC, you are responsible for freeing it. For example, if you allocate additional memory when an custom object is created, you should free it in the appropriate freefn. If you create an *object*, you are responsible for freeing that object by calling object_free.

An important exception occurs when you create an object that is intended for the user to work with. You might return the object from a C function or method, or you might store the object in another data structure that the runtime or user provided you with. In this case, you must tell the virtual machine about the object(s) and the garbage collector then becomes responsible for their management.

To do so, you bind the object(s) to a virtual machine by calling:

```
morpho_bindobjects(vm *v, int nobj, value *obj);
```

with a list of objects. A simple example:

```
objectfoo *foo = objectfoo_new();
// Check for success and raise an error if allocation failed
value new = MORPHO_OBJECT(foo)
morpho_bindobjects(v, 1, &new);
```

Once an object is bound to a virtual machine, the garbage collector is responsible for determining whether it is in use and can be safely deallocated.

The "weak" garbage collection model used by morpho has a number of advantages: It reduces pressure on the garbage collector by reducing the number of blocks that need to be tracked, because data associated with the runtime environment does not have to be managed. It facilitates a number of virtual machine features such as reentrancy, because morpho objects can be created, used and even returned to the user without the garbage collector being involved. Nonetheless, because of the non-deterministic nature of garbage collection, various classes of subtle bugs can arise:

1. An incorrectly programmed custom object may fail to inform the garbage collector about `values` or other structures it has access to in its `markfn`. It is **essential** that this function works correctly, otherwise the garbage collector may think that an object is no longer in use (and hence deallocate it) when in fact it is.

2. When creating an object that contains other objects, you must take care to bind all objects at once (or at least bind the outermost objects first) so that the garbage collector can see the interdependency.

3. When calling *morpho* code from C it is necessary to be careful to ensure that bound objects remain visible to the garbage collector or they may be incorrectly free'd. See Section 19.8 for further information.

In both cases, when an incorrectly free'd object is next used, it will cause a segmentation fault. To help debug such errors, it's possible to build morpho with the option `MORPHO_DEBUG_STRESSGARBAGECOLLECTOR`. This forces garbage collection on every bind operation, and will tend to cause segmentation faults to occur much sooner after the problematic code has executed. There is also a macro `MORPHO_DEBUG_LOGGARBAGECOLLECTOR` that logs object creation and destruction that can be of assistance.

19.8 Re-entrancy

The morpho virtual machine is *re-entrant*, i.e. C function and method implementations can themselves call user code written in *morpho*, re-entering the virtual machine to execute it and then making use of the results open return. The API for this is described in this section. Re-entrant calls can be recursive: during such a call, the user's *morpho* code itself may call back C functions, only limited by the depth of the call stack, which is currently fixed by the macro `MORPHO_CALLFRAMESTACKSIZE` in `build.h`.

In order to use the API, you first need a value that contains a callable object. These could include:

- A C function, as returned by `builtin_addfunction,` and contained in an `objectbuiltinfunction`.
- A morpho function contained in an `objectfunction`.
- An invocation on an object, which contains both method and receiver, and is contained in an `objectinvocation`.
- A closure contained in an `objectclosure`.

All of these are called in the same way, and the runtime automatically takes care of any setup necessary to execute the call. You can check whether a value contains a callable object with the macro MORPHO_ISCALLABLE().

Once you have a callable object, you can have the virtual machine execute it with given parameters by calling:

```
bool morpho_call(vm *v, value fn, int nargs, value *args, value *ret);
```

This function requires a list of arguments, given as a C array of values. You must check the return value: If the call is successful, morpho_call returns true and the value ret is filled out. Otherwise, the virtual machine is returning in an error state and you should return as quickly as possible.

Another common scenario is that you wish to invoke a method on a given object. You can do so with morpho_invoke:

```
bool morpho_invoke(vm *v, value obj, value method, int nargs, value *args,
   value *ret);
```

The method label should be given in the parameter method, and the parameters work similarly to morpho_call.

If you wish to call a method on the same object many times with different parameters, you can avoid the cost of method lookup by getting a callable invocation from the runtime using the function,

```
bool morpho_lookupmethod(value obj, value label, value *method);
```

and supplying the resulting value in method as the input to morpho_call. Note that you must check to ensure morpho_lookupmethod was successful; it will return false if the method wasn't found for example.

You can count the number of fixed parameters for a callable object by using,

```
bool morpho_countparameters(value f, int *nparams);
```

19.8.1 Restrictions on re-entrant code

When the virtual machine is entered, arbitrary code may be executed that can reconfigure the state of the runtime environment. As a result, functions that use the reentrant API are subject to a number of restrictions so that they do not refer to information that is no longer valid across a call to morpho_call or morpho_invoke.

- The value of the args pointer passed to your function may no longer be valid after a re-entrant call. You *must* obtain all parameters using MORPHO_GETARG before making the call.

- You must use morpho_bindobjects carefully. If you make a re-entrant call after using morpho_bindobjects, it is possible that the garbage collector will run without being able to "see" a valid reference to the object held in your C code (the garbage collector cannot check the C stack, for example) and will erroneously destroy the object leading, ultimately, to a segfault. To avoid this, either use morpho_bindobjects after making all re-entrant calls, or call morpho_retainobjects to ensure they're retained across a re-entrant call. If you do the latter, you **must** call morpho_releaseobjects with the handle returned by morpho_retainobjects before your code returns.

Part IV *Morpho* internal documentation

Chapter 20

libmorpho source code

The *morpho* source code has been significantly reorganized in version 0.6 to improve its modularity and reusability. *Morpho* has been split into a dynamic library, *libmorpho*, that provides the compiler, virtual machine and runtime, and is designed to be embeddable in other applications. The command line interface is now provided by a separate program, *morpho-cli*, and other kinds of interface may be provided in future[1].

The libmorpho source is in the src folder within the git repository. Each file is intended to provide one piece of functionality, and is organized into subfolders as follows:—

Folder	Purpose
core	Compiler and virtual machine.
debug	Debugger and profiler.
classes	Morpho object types and veneer classes.
linalg	Support for dense and sparse linear algebra.
builtin	Support construction of builtin classes and functions.
support	Various utility packages
datastructures	Low level data structures used by *morpho*.

A description of files within each of these folders is provided in Table 20.1.

20.1 Compiling *morpho*

As of version 0.6, *morpho* is now compiled with cmake, an automated build utility, facilitating much improved cross-platform building. To install from source, clone the git repository to a convenient place,

```
git clone https://github.com/Morpho-lang/morpho-libmorpho.git
```

and then

```
cd morpho-libmorpho
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make install
```

You may need to use `sudo make install`.

You can also build project files for various IDEs. For example you can create an Xcode project by using

```
cmake -GXcode ..
```

[1]A prototype Jupyter interface exists, for example.

Folder	File	Description
core	compile.c/.h	Compiler
	core.h	Type definitions for VM core
	gc.c/.h	Garbage collector
	optimize.c/.h	Optimizer
	vm.c/.h	Virtual machine.
debug	debug.c/.h	Debugger
	profile.c/.h	Profiler
classes	classes.h/.h	List of classes
	array.c/.h	Array class
	closure.c/.h	Closure class
	clss.c/.h	class object type
	cmplx.c/.h	Complex number
	dict.c/.h	Dictionary class (see dictionary.c/.h for underlying implementation)
	file.c/.h	File and Folder classes
	function.c/.h	Function class
	instance.c/.h	Object class
	invocation.c/.h	Invocation class
	list.c/.h	List class
	range.c/.h	Range class
	upvalue.c/.h	upvalue object type
	strng.c/.h	Strings
	system.c/.h	System class
linalg	matrix.c/.h	Matrix class
	sparse.c/.h	Sparse class
builtin	builtin.c/.h	Support for builtin functions and classes
	functiondefs.c/.h	Builtin functions
support	common.c/.h	Contains utility functions
	extensions.c/.h	Extensions
	lex.c/.h	Lexer
	memory.c/.h	Basic memory allocation
	parse.c/.h	Parser
	random.c/.h	Random number generator
	resources.c/.h	Resource locator
	threadpool.c/.h	Thread pool
datastructures	dictionary.c/.h	Dictionary struct
	error.c/.h	Error messages
	object.c/.h	Fundamental object type
	program.c/.h	Program data structure
	syntaxtree.c/.h	Syntax tree struct used by compiler
	value.c/.h	Basic value type
	varray.c/.h	Variable length arrays

Table 20.1: Map of the *morpho* source code.

20.2 Coding standards

The *morpho* source, being somewhat large (around 40k lines as of 0.6), aims to obey the following coding standards, which may be refined as the project evolves.

20.2.1 File contents

- Header files (`.h`) should include type definitions and function prototypes, and only include other header files necessary for the correct *declaration* of the contents, not necessarily their *implementation*. This helps reduce interdependencies.
- Implementation files (`.c`) must `#include` all necessary header files for successful compilation.

20.2.2 Comments

Functions, typedefs and other constructs in *Morpho* should be accompanied by comments in doxygen format describing the purpose of the construct and its interface, for example:

```
/** myfunc
 * @brief      What the function does
 *
 * @details    A more detailed description.
 *
 * @param[in]  x An input parameter
 * @param[out] y An output parameter
 * @return     What the function returns.
 */
```

Occasionally, it is useful to include ASCII diagrams in the source code to illustrate algorithms or data structures. They can formatted so that doxygen preserves the spacing as in the below example:

```
/** @detail
 * <pre>
 *      A
 *     / \
 *    B   C
 * </pre>
 */
```

20.2.3 Naming conventions

1. Types should be named lower case all one word, e.g `dictionaryentry`.
2. Functions should be named `module_functionname` where `module` refers to the conceptual unit that the function belongs to (typically the same as the filename).
3. Method definitions should be of the form `Classname_methodname` noting the capitalization.

20.2.4 Scoping

Macros not intended for use outside a particular context should be `#undef`'d.

20.2.5 Unreachable code

Where appropriate, mark unreachable code with `UNREACHABLE(x)`. The parameter of this macro is a short `static char` that should explain to the reader where the unreachable code is to be found in the source. The morpho virtual machine immediately halts and reports an “Internal consistency error:” followed by your explanation.

20.2.6 C99 features

In this section we note specific C99 features used in the implementation of *Morpho*.

- We use the `bool` type and `true` and `false`.
- Compound literals are used to implement `value` literals.
- Flexible array members are used to implement some object types. These look like this:

  ```
  struct mystruct {
          int len;
          double arr[]; /* Note lack of size here */
  };
  ```

  where the flexible array member must be at the end of the `struct` definition.
- Pointers are converted into `uintptr_t` to hash them. This type is defined so that conversion from a pointer and back yields the same value.

Chapter 21

The Virtual Machine

21.1 Virtual machine structure

The virtual machine operates on a progam that comprises a sequence of instructions, described in subsequent sections, which perform various functions. While running the program, the VM maintains (at least) the following state:

- **Program counter.** This points to the next instruction to be executed.
- **Data stack**. The data stack contains information the program is acting on. It is an ordered list of values that can grow as needed. A subset of these values are visible to the VM at any one time, referred to as the *register window*. Because morpho instructions can refer to any available register, not just the top of the stack, *morpho*'s VM is a *register machine*.
- **Call stack**. The call stack keeps track of function calls and grows or shrinks as functions are called or return. Each entry, called a *call frame*, contains information associated with a function call: which parts of the data stack are visible, the value of the program counter when the call took place, a table of constants, etc.
- **Error handler stack**. Programs may provide code to handle certain kinds of errors. This stack keeps track of error handlers currently in use.
- **Garbage collector information**. As objects are created at runtime, the VM keeps track of them and their size and periodically removes unused objects.

21.2 *Morpho* instructions

Each *Morpho* VM instruction is a packed `unsigned int`, with the following possible formats
This permits up to 64 separate opcodes and up to 256 individually addressable registers per frame. Parameter A denotes the register that is used to store the result of the operation, or more generally the register that is affected. Parameters B and C are used to denote the input registers. Larger operands are possible in the last three instruction formats.

21.3 *Morpho* opcodes

Each instruction has 1-3 operands. Lower case letters indicate registers, upper case represents literals or constant ids.

Type	Bits				Description
	0-7	8-15	16-23	24-31	
1	Opcode	A	B	C	Operation with three byte parameters; B & C can be marked as constants.
2	Opcode	A	Bx		Operation with one byte and one short parameter & two flags.
3	Opcode	A	sBx		Operation with one byte and one signed short parameter & two flags.
4	Opcode	Ax			Operation with one 24 bit unsigned parameter

Table 21.1: *Morpho* instruction formats.

Category	Opcode	Operands	Explanation
	`nop`		No operation.
	`mov`	a, b	Moves reg. b into reg. a
	`lct`	a, Bx	Moves constant B into reg. a
Arithmetic	`add`	a, b, c	Adds register b to c and stores the result in a.
	`sub`	a, b, c	Subtracts register c from b and stores the result in a.
	`mul`	a, b, c	Multiplies register b with register c and stores the result in a.
	`div`	a, b, c	Divides register b with register c and stores the result in a.
	`pow`	a, b, c	Raises register b to the power of register c and stores the result in a.
Logical	`not`	a, b	Performs logical not on register b and stores the result in a.
Comparison	`eq`	a, b, c	Sets reg. a to boolean b==c
	`neq`	a, b, c	Sets reg. a to boolean b!=c
	`lt`	a, b, c	Sets reg. a to boolean b<c
	`le`	a, b, c	Sets reg. a to boolean b<=c
Branch	`b`	sBx	Branches by (signed) B instructions
	`bif`	a, sBx	Branches by (signed) B instructions if a is true.
	`biff`	a, sBx	Branches by (signed) B instructions if a is false.
Function calls	`call`	a, B, C	Calls the function in register a with B positional arguments and C optional arguments. Arguments are stored in register a+1 onwards.
	`return`	A, b	Returns from the current function If parameter A>0, register b is returned, otherwise `MORPHO_NIL`.

Objects	`invoke`	a, B, C	Invokes a method with label in register a on an object in register a+1 with with B positional and C optional arguments. Arguments are stored in register a+2 onwards.
	`method`	a, B, C	nvokes a method with the **method itself** in register a on an object in register a+1 with with B positional and C optional arguments. Arguments are stored in register a+2 onwards.
	`lpr`	a, b, c	Looks up property c in object b, storing the result in a.
	`spr`	a, b, c	Stores value c in object a with property b.
Closures	`closure`	a B	Encapsulates the function in register a into a closure using prototype number B from the enclosing function object.
	`lup`	a, B	Loads the contents of upvalue number B into register a.
	`sup`	A, b	Stores the contents of b in upvalue number A
	`closeup`	A	Closes upvalues beyond register number A
Indices	`lix`	a, b, c	Loads an element from array a. Indices to use are stored in registers b..c, and the result is stored in register b.
	`six`	a, b, c	Stores value c in array a with indices stored in registers b..c-1.
	`array`	a, b, c	Creates an array with dimensions in registers b..c and stores it in register a. *(Should be deprecated)*
Globals	`lgl`	a, Bx	Loads global number Bx into register a.
	`sgl`	a, Bx	Stores the contents of register a into global number Bx.
Error handlers	`pusherr`	Bx	Pushes the error handler in constant Bx onto the error handler stack
	`poperr`	sBx	Pops the current error handler off the error handler stack and branch by (signed) B instructions
	`print`	a	Prints the contents of register a.
	`cat`	a, b, c	Concatenates the contents of registers b-c and stores the result in register a. *(Should be deprecated?)*
	`break`		Breakpoint
	`end`		Denotes end of programs

21.4 How function calls work

When a function or method call takes place, the VM:

1. Records the program counter, register index and stack size in the *current* call frame.

2. Advances the frame pointer.
3. If the called object is a closure, pulls out the function to be called and records the closure in the new call frame.
4. Records the function to be called in the new call frame.
5. Sets up the constant table in the new call frame.
6. Advances the register window and clears the contents of registers to be used by the function.
7. Sets register r0 to contain either the function object OR the value of `self` if this is a method call.
8. Sets register r1 onwards to contain the arguments of the function.
9. Moves the program counter to the entry point of the function.

Method calls are identical to function calls, except that *r0* contains the object. Invocations are unpacked before calling.

21.5 How error handling works

Ordinarily, when a runtime error is generated execution immediately stops and the error is reported to the user. Sometimes a possible error can be forseen by the programmer and the program can be written to take an alternative course of action. To achieve this, the Morpho VM provides for error handlers.

Morpho keeps track of error handlers on a special stack. As execution proceeds, the program may add an error handler to the stack using the `pusherr` opcode; it can then be removed again by using `poperr`. Only the most recent error handler can be removed in this way.

When an error occurs, the VM searches the error handlers currently in use from the top of the error handler stack downwards to find an error handler that matches the ErrorID tag. If none is found, execution terminates and the error is reported to the user as normal. If suitable handler is found, however, execution resumes at a point specified by the handler. The callframe is reset to that of the error handler and any open upvalues beyond that frame are closed.

21.5.0.1 Re-entrancy

In searching for an error handler, the VM checks whether an intermediate call frame requires it to return. This happens if the VM has been re-entered from a C function (using `morpho_call` for example). In such a case, the VM returns from the intermediate frame rather than handling the error (and so `morpho_call` returns false). The calling C function **must** check for this case and either handle the error itself or exit as quickly as possible. If the error isn't handled, once the intermediate C function returns, the outer VM that called it will detect the error and resume the search for an error handler.

Chapter 22

The compiler

22.1 Overview

The *Morpho* compiler takes a string of *Morpho* input and converts to bytecode by a three stage process (Fig. 22.1.1): The source code is first divided into *tokens*, basic units like number, identifier etc., by the *lexer*. Tokens are then converted into a *syntax tree*, an abstract representation of the programs syntactic structure, by the *parser*. These two components are in `lex.c/.h` and `parse.c/.h`. Both of these elements can adapted to parse things other than *morpho* code; see `json.c/.h` for an example of how this is done for the JSON interchange language.

Finally, the syntax tree is converted to bytecode by the bytecode compiler (referred to hereafter simply as the compiler) in `compile.c/.h`. The resulting bytecode can then be run by the virtual machine described in chapter 21.

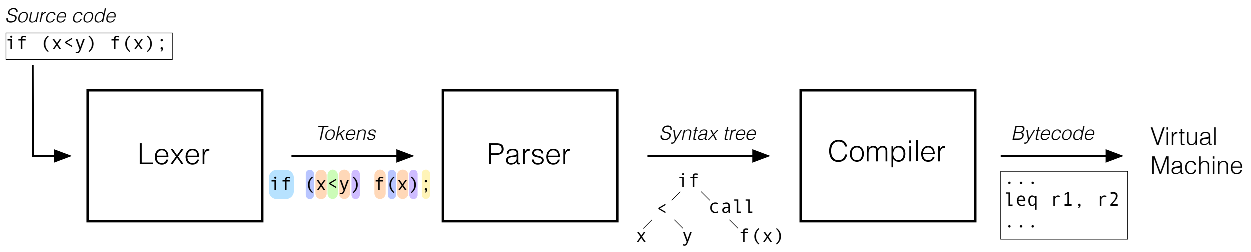

Figure 22.1.1: Structure of the *Morpho* compiler.

22.2 Extending the compiler

This section provides a very brief guide to how the lexer, parser and compiler may be extended. When modifying the compiler, it's a good idea to do so in the order suggested by Fig. 22.1.1: First modify the lexer to produce any new token types required, then the parser to parse new syntax correctly, then finally the compiler. The file `build.h` contains a number of macros that can be defined that cause morpho to generate output to help debug compiler features, for example `MORPHO_DEBUG_DISPLAYSYNTAXTREE` displays a syntax tree after compilation.

22.2.1 Lexer

The lexer takes a raw C source string as input and sequentially splits it into *tokens* demarcated by their starting point, length and type. It is defined in support/lex.c and support/lex.h and has been rewritten for v0.6 to be usable for lexing applications beyond the compiler.

To use a lexer, it must first be initialized by calling `lex_init` with a pointer to the source `start` and a starting line number `line`:

```
void lex_init(lexer *l, const char *start, int line)
```

You can obtain tokens by repeatedly calling `lex`:

```
bool lex(lexer *l, token *tok, error *err)
```

which fills out a `token` structure with the next token and returns `true` on success. If an error occurs, it will fill out the provided `error` structure and return `false`. Once the source string is exhausted, the lexer returns a special token (the token type can be configured) indicating end of file.

Once you're done with a lexer, you must call

```
void lex_clear(lexer *l)
```

to free any attached data.

New token types (for example, for a new operator or keyword) can be implemented by adding a new entry into the enum in `lex.h`. You **must** make a corresponding entry into the parser definition table (see `parserule rules[]` in `parse.c`), even if the token type is marked `UNUSED` for now. The overall order of token types isn't significant. You should then modify the lexer to generate the token. Tokens are identified and generated by the lexer in multiple ways:—

- A preprocessor function gets the first look at a token, and identifies things like symbols and numbers. You can replace this with a custom version. The current preprocessor checks whether symbols match keywords (you can disable this with `lex_setmatchkeywords`).
- If the preprocessor function doesn't claim the token, the lexer tries to identify the token from a table of `tokendefn`'s in `lex.c`. This is used to identify keywords and operators. You can easily extend the token definition table, or replace it with your own. Each token definition can specify a function to call after the token is identified to perform further processing; this is done to lex strings, for example.

You can create custom lexers, entirely replacing morpho tokens. See the JSON parser in `classes/json.c` for an example.

Note that syntax highlighting in the CLI also depends on a `linedit_colormap` between token types and colors; you would need to add these there.

22.2.2 Parser

The parser implements the Pratt parsing algorithm[1]. You typically need to create a parse rule, or edit an existing one if appropriate, to parse the new token type. Parse functions all have the form

```
bool parse_MY(compiler *c, void *out)
```

and call:

1. `parse_advance` and `parse_consume` to obtain tokens.

[1]Pratt, Vaughan. "Top down operator precedence." Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (1973). A very good discussion is also to be found in `https://journal.stuffwithstuff.com/2011/03/19/pratt-parsers-expression-parsing-made-easy/`

2. `parse_expression` or `parse_precedence` to parse subexpressions.

3. `parser_addnode` to add nodes to the syntax tree.

4. `parse_error` to record errors.

Once the parse rule is created, it may be included in the parser definition table (see `parserule rules[]` in `parse.c`) or called from another function. New operators, for example, are typically inserted directly into the definition table, and macros are available to denote the token as `PREFIX`, `INFIX` or `MIXFIX`. Keywords that introduce a statement, e.g. `var` or `fn`, require inserting an appropriate text and call into `parse_statement`.

You can create a custom parser by replacing the parser definition table.

22.2.3 Syntax tree

Once the parser has been modified, it may be necessary to create new syntax tree node types:

1. Create any new syntaxtreenodetypes necessary. A node type is defined by adding a new entry into the `syntaxtreenodetype` enumerated type definition. The order *does* matter: new node types should be grouped with leafs, unary operators or operators as is documented in the source. It is imperative to make a corresponding entry in the compiler definition table (see `compiler_nodefn noderules[]` in `compile.c`); it can be marked `NODE_UNDEFINED` for now. As for the parser definition table, it's essential that these structures match. It's also important to add a corresponding entry into the `nodedisplay` array in `syntaxtree.c`; this is used to display syntax trees for debugging purposes.

2. Modify the parse rule generated in the previous section to emit appropriate syntax tree nodes.

22.2.4 Compiler rule

1. Create a compiler rule for any new node types. Insert it into the compiler definition table at the corresponding place. Compiler rules call `compiler_nodetobytecode` to compile child nodes, and `compiler_writeinstruction` to create bytecode. Macros are available to encode instructions.

2. Creation of new instruction types is possible (by modifying the VM) but strongly discouraged; contact the developers if you have ideas about improved VM functionality.

Chapter 23

Debugger

The debugger is a challenging component because it interacts both with the virtual machine—and hence needs to know about its inner workings—and also the user. The overall design of *morpho* exposes virtual machines as opaque pointers to as much of the code as possible. Code that needs to work directly with the virtual machine must define `MORPHO_CORE` and import `core.h` and is referred to as *privileged*.

To enforce the separation, we divide debugging into multiple components:—

1. debugannnotations, which are a data structure that encodes annotations about the bytecode, e.g. associating registers with symbols and other metadata. This data structure is in libmorpho in `datastructures/debugannotation`, and is not privileged.

2. debugger, defined in debug/debug.c. This data structure maintains the state of the debugger, and provides a debugging API for debugger interfaces. Requests for information are fulfilled using `morpho_printf`. This code is privileged.

3. clidebugger, defined in `morpho-cli`. This implements an interactive REPL interface that allows the user to work with the debugger. It is not privileged, and exclusively works through the debugger API. The interface is described above in part 1.

Chapter 24

Morphoview

Morpho 0.5 onwards separates out UI code into a separate application, *morphoview*. This enables, for example, running a program on a cluster and seeing the results on another computer, or having different client dependent implementations (e.g. a Metal client for macOS, an OpenGL client for Linux, etc.). The *morpho* language runtime communicates with morphoview via temporary files, but this will be replaced by a client/server model via ZeroMQ or similar.

24.1 Command language

Morpho communicates with *morphoview* via an imperative scommand language designed to be simple and human readable. Valid commands are given in table 24.1 together with their syntax. Commands and their elements may be separated by arbitrary whitespace. Additional commands may be provided in the future.

24.1.1 Usage

A typical file will begin will begin by specifying a scene,

```
S 0 2
```

and (completely optionally) setting the window title

```
W "Triangle"
```

Following this, objects in the scene may be defined,

```
o 1
```

which include vertex data and elements of the object. For instance, a simple triangle could include

```
v x
1 0 0
0 1 0
1 1 0
f 0 1 2
```

providing a list of three vertices and then specifying that a single facet is to be drawn from these vertices.

After objects have been specified, the scene can be drawn. Objects are positioned in the scene by using transformation matrices: Morphoview maintains a current object transformation matrix at all times. Initially, this is set to the identity matrix (so that the object is placed in the scene using the coordinates at which it is defined), but the matrix can be modified by the `i`, `r`, `s` and `t` commands. e

For example, to scale, rotate translate and then draw object 1,

Command	Syntax	Description
`c`	`c id r g b ...`	Color buffer declaration
`C`	`C id`	Use color or map object *id*. If no *id* is provided, clears current color object.
`d`	`d id`	Draw object *id* using current transformation matrix and color *id*.
`o`	`o id`	Object `id` is an integer that may be used to refer to the object later; unique per scene.

`v`	`v format`
	`f1 ...`
	
	`...  fn`
		Vertex array
		`format` is a string that contains at least any or all of the letters x, n and c
		and is used to specify the information present and the order, e.g.
		`xnc` — vertex entries contain position, normal and color
		`x` — only position information is present
		`xc` — indicates position and color information
		This is then followed by the appropriate number of floats

`p`	`p`
	`v1 ...`
		Points
		A list of integer indices into the object's vertex array to be drawn as points

`l`	`l`
	`v1 v2 ...`
		Lines
		A list of integer indices into the object's vertex array to be drawn as
		a continuous sequence of lines

`f`	`f`
	`v1 v2 v3`
		Facets
		A list of triplets of integer indices into the object's vertex array to be drawn as facets.

`i`	`i`	Set the current transformation to the identity matrix
`m`	`m m11 m21 ..  m44`	Multiply the current transformation by the given 4x4 matrix, given in column major order
`r`	`r phi x y z`	Rotate by phi radians about the axis (x, y, z) [(x, y) in 2d]
`s`	`s f`	Scale by factor f
`S`	`S id dim`	Scene description `id` is an integer that may be used to refer to the scene later `dim`$\in \{2, 3\}$ is the dimension of the scene
`t`	`t x y z`	Translate by *(x,y, z)*
`T`	`T fontid string`	Draw text "string" with font `id` using current transformation matrix and color id
`F`	`F fontid file size`	Declare font `file` the filename and path of the desired font `size` the size in points `fontid` an integer that will be used by T commands to refer to the font
`W`	`W title`	Window features `title` is the window title

Table 24.1: **Morphoview command language.**

```
s 0.5
r 1 0 0 1
t 0.5 0.5 0
d 1
```

You can draw multiple objects using the same transformation matrix just by issuing subsequent draw commands.

Once the scene is specified, *Morphoview* will open a viewer window displaying the specified scene.

24.1.2 Morphoview internal structure

1. **Parser.** The parser processes the command file and builds up a Scene object from the program.
2. **Scene.** Morphoview programs describe a Scene. Once the scene is described, it is then prepared for rendering.
3. **Renderer.** This module takes a scene and prepares OpenGL data structures for rendering.
4. **Display.** Manages windows, user interface etc.