The Morpho book
Version 0.6.1

July 8, 2024

Chapter 1

Introduction

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 2

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"
```

2.1 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

2.2 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 4) 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"
```

2.3 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.

2.3.1 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)
```

2.4 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.

2.5 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

2.6 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.

2.7 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.

2.8 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 3

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 4

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.

4.1 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 (overengineered) pocket calculator like so:

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

4.2 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`.

4.3 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.

4.4 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`

4.5 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 2.6.
- **Function calls** are introduced in Chapter 6.
- **Method calls** are denoted by a single dot `.` and are explained in Section XXX.

4.6 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 5

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.

5.1 Declarations

An important category of statements are declarations, which define varius kinds of construct. Variable declarations were already introduced in Chapter 2. Function declarations will be described in Chapter6 and Class declarations in Chapter 8.

5.2 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()
```

5.3 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.

5.4 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 forseen by the programmer.

5.4.1 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 4.3, 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*.

5.4.2 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
}
```

5.4.3 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
        // ..
}
```

5.4.4 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")
```

5.4.5 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
```

5.4.6 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 nonlocally 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.

5.4.7 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 10.

Chapter 6

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 5.4.6, 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.

6.1 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 parametes 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.

6.2 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)
}
```

6.3 Multiple dispatch

While *Morpho* functions, by default, accept any value for each parameter, it's often the case that a function's behaviour 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.

6.4 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 7

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)
```

7.1 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.

7.2 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.

7.3 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 interchangably.

Chapter 8

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 12.1.

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

8.1 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`.

8.2 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 8.4 below.

8.3 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 6.3 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.

8.4 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.

8.5 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 9

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.

9.1 Morpho protocols

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

9.1.1 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.

9.1.2 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.

9.1.3 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.

9.1.4 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)`.

9.1.5 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.

9.2 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 10

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 forseeable 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.")
```

10.1 Handling errors

While not necessarily expected, some errors are at least forseeable. 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 *forseen* errors only.

Chapter 11

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 seach 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.

11.1 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 libary. By making only the components needed accessible, the possibility of conflicts is reduced.

11.2 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.

11.3 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 12

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.

12.1 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 Antipatterns". 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.

12.2 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)
  }
}
```

12.2.1 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.

12.2.2 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.

12.2.3 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.