Sam Discusses Lunacy

Table of contents

INTRODUCTION

Lunacy is my personal fork of Lua 5.1. *Lua* is a tiny scripting language which is suitable for embedding in other applications. While there are newer versions of Lua, Lua 5.1 is the most widely deployed and used. Lua 5.1 is the most recent version supported by LuaJIT, a very fast implementation of Lua; Lua 5.1 is also the basis for Luau, the scripting language used by the very popular gaming platform Roblox, as well as the Scribunto MediaWiki extension which allows the Wikipedia to run Lua scripts.

This book is a guide to programming in Lunacy. While the book is written for Lunacy, it can also be used to learn how to program in Lua.

The book is not an introduction to programming; in particular, it assumes the reader is already familiar with other scripting languages such as AWK, Perl, or Python. The book concentrates more on the parts of Lua which can be difficult for people used to other scripting languages to understand, such as function closures, how Lunacy uses function factories to run `for` loops (and how one can have Lunacy iterate through items in a list using the right function in a `for` loop), metatables, and Lunacy's prototypical class model.

This book uses a large number of code examples to illustrate how code is written in Lunacy, and to provide public domain "battery" code (such as a regular expression splitter and a function which makes regular expressions case insensitive) people can use in their own Lunacy scripts.

This book is not a Lunacy reference; a very terse reference manual is already included with Lunacy. This is a companion to the reference manual to have code examples and longer explanations of the concepts expressed in that manual.

This book concentrates on using Lunacy as a scripting language in the vein of Perl/AWK/Python/etc. That in mind, it does *not* cover concepts which are not used in text processing scripts, such as coroutines or integrating Lunacy in to a larger C or C++ project.

The goal of this book is to be an open content guide to programming in Lunacy (or Lua) which people can use to become familiar with this language.

This book is a work in progress and the latest version of this book can be found at either `https://github.com/samboy/lunacy` or `https://maradns.samiam.org/lunacy/`

Installing Lunacy

To way to install Lunacy depends on the operating system one is using. In Windows (Windows XP, Windows 7, Windows 8, Windows 10, and Windows 11) one can simply use the pre-compiled Lunacy binary. In Linux and MacOS, if one has a C compiler and an implementation of *make*, it is possible to compile Lunacy.

The simplest way to download Lunacy is to get it from GitHub using the *git* command. From a command line, run this:

```
git clone https://github.com/samboy/lunacy
```

One option to obtain Lunacy without git is to go to `https://github.com/samboy/lunacy`, click on "Code" then "download zip". Another option is to go to `https://maradns.samiam.org/lunacy/` and download Lunacy there.

Should GitHub be down, Lunacy is available at other locations on the internet, e.g. `https://codeberg.org/samboy/lunacy` or `https://gitlab.com/maradns/lunacy/`

Once Lunacy is downloaded, Windows users can use the `lunacy.exe` 32-bit x86 compile of Lunacy in the `bin/` folder of the download. People on other platforms can compile Lunacy with `make` and a C compiler (`gcc`, `clang`, etc.) in the `src/` folder. The simplest way to compile Lunacy is to simply run `make` in the `src/` folder of Lunacy; if readline support is desired (to have up arrow history) and the, one can compile Lunacy via `make -f Makefile.readline` If one has the editline library instead (similar to readline, but without the GPL license), one can compile Lunacy via `make -f Makefile.editline`

Once Lunacy is compiled, one can copy it over to one's local path (e.g. `/bin/` or `/usr/bin/` or another directory in one's usual path) as root or admin via the appropriate copy command (e.g. `cp lunacy /usr/bin/`).

The Windows binary of Lunacy has up arrow history support if it's run in Cygwin (available at `https://cygwin.com/`) or PowerShell.

Lunacy is a command line application, so it is best run from a terminal window, such as Cygwin, PowerShell, any Linux terminal application, or the terminal in MacOS.

That said, the `lunacy.exe` binary runs just fine when double clicked on in Windows 10, and even has up arrow history support.

Running Lunacy

Once Lunacy is installed, running the `lunacy` binary (e.g. `lunacy.exe` on Windows) will show something that looks like this:

```
Lunacy 2022-12-06  Copyright 1994-2012 Lua.org PUC-Rio; 2020-2022 Sam Trenholme
>
```

Here we can begin to type in commands. In the examples on this page, text in bold **`like this`** is typed by the user; text not in bold `like this` is returned from the Lunacy binary.

Let's do the "Hello, world!" example:

```
> print("Hello, world!")
Hello, world!
```

We can also use Lunacy as a basic desktop calculator:

```
> 1 + 1
2
```

Note that the trick that makes the answer to an expression appear only works if the first character in the line is a number or the left parenthesis "`(`" character. Other characters need `=` at the beginning of the line, e.g.

```
> = math.pi
3.1415926535898
```

Lunacy has support for functions, e.g.

```
> function hi()
>> print("Hello, world!")
>> end
> hi()
Hello, world!
```

If Lunacy is being run from the command shell, it is possible to tell Lunacy to run a given file as its script contents. For example, let us suppose we have a file `foo.lua` with the following in it:

```
print("Hello, world!")
```

Running Lunacy in the same directory as `foo.lua` gives us a result like this:

```
$ cat foo.lua      This would be "type foo.lua" in a Windows cmd shell terminal
print("Hello, world!")
$ lunacy foo.lua
Hello, world!
```

Where `$` is a Cygwin, Linux terminal, MacOS terminal, or similar prompt.

Desktop Calculator

Lunacy can be used as a desktop calculator. To do so, Lunacy can be started from the command line in a terminal window. For example, anything in bold is typed by the user:

```
$ lunacy
Lunacy 2022-12-06  Copyright 1994-2012 Lua.org PUC-Rio; 2020-2022 Sam Trenholme
> 1 + 1
2
> (1 + 2) * 3
9
```

`+` is addition; `-` is subtraction; `*` is multiplication; `/` is division.

Lines which begin with a number of the `(` character are calculated. Should an expression who which starts with another character is desired, the line needs to start with the `=` character.

```
> =math.pi
3.1415926535898
```

It's possible to assign and use variables in the Lunacy command line:

```
> a = (34 * 3) + 2
> = a * 2
208
```

It's also possible to define functions (or do pretty much anything Lunacy can do when run as a script) when being run as a desktop calculator:

```
> function addOne(x)
>> print(x + 1)
>> end
> =addOne(3)
4
```

A desktop calculator will usually have arrow history support; in other words, if one hits the up arrow character while using Lunacy in desktop calculator mode, the previously entered command will appear again and can be edited with the left and right arrow keys. The 32-bit Windows Lunacy has arrow support when run in Cygwin, PowerShell, or from the Windows cmd shell; the source code to Lunacy has support for both readline and editline which also can give Lunacy this support.

Lunacy does not store the state of its virtual machine; every single time Lunacy is started as a fresh slate; functions and variables stored in previous invocations of Lunacy are lost.

NUMBERS

The version of Lua I use is Lunacy, which is a fork of Lua 5.1.

In Lunacy, all numbers are IEEE 64-bit floats. This means:

- Numbers between -9,007,199,254,740,992 and 9,007,199,254,740,992 have a precision of 1 or higher; numbers in this range can be represented as integers.
- Numbers can be fractional; the closer the number is to zero, the more precision the fraction has.
- The type of floating point number Lunacy uses is binary. This means that fractions which are not a power of two are imprecise in Lua.

Let's show an issue with these floats by looking at a loop in Lua.

To do a simple loop in Lua, we use this form:

```
for counter = start, end, increment do
  print(counter)
end
```

The default value for increment is `1`.

For example:

```
for counter = 1,10 do
  print(counter)
end
```

Which gives us 1,2,3,4,5,6,7,8,9, and finally 10.

Likewise:

```
for counter = 1,11,2 do
  print(counter)
end
```

Will give us 1,3,5,7,9,11

We can count backwards:

```
for counter = 10,1,-1 do
  print(counter)
end
```

This gives us 10,9,8,7,6,5,4,3,2,1

Note that we must have the increment as a negative number to count backwards.

Note that, since Lunacy uses 64-bit IEEE floats, counters really should only use integers. This can give unexpected behavior:

```
for counter = .9,1,.05 do
  print(counter)
end
```

With infinite precision numbers, or heck with decimal floating point numbers, we would get 0.90, 0.95, 1.00, but instead we get 0.90 and 0.95 without the 1.00 (when running as 32-bit x86 code; the behavior is as expected on ARM64 and x86_64). This is caused because .05 is a number ever so slightly higher when represented as a 64-bit IEEE binary floating point number, causing the loop to act in unexpected ways.

The solution to avoid these kinds of issues is to always use integers for Lua loops, then divide the number as needed, e.g.:

```
for counter = 90,100,5 do
  print(counter / 100)
end
```

This will give the expected behavior.

How can we see a large integer in Lunacy?

The simple `print()` command will, with Lunacy, print numbers as high as 99,999,999,999,999. Higher numbers show the number of as "e+XXX" format, e.g. 100,000,000,000,000 is shown as `1e+014`.

```
print(99999999999999)
print(100000000000000)
```

This shows `99999999999999` then `1e+014`

In a 64-bit compile of Lunacy, this will correctly show `100000000000000`:

```
print(string.format("%d",100000000000000))
```

However, in a 32-bit compile of Lunacy, the above code incorrectly shows `-2147483648`. To work around this issue requires a function (see the next page):

```
function numberToString(n)
  out = ""
  isNeg = false
  if(n < 0) then
    n = -n
    isNeg = true
  end
  if(n == 0) then
    return "0"
  end
  while n > 0 do
    out = string.format("%d",n%10) .. out
    n = n - (n % 10)
    n = n / 10
  end
  if isNeg then
    out = "-" .. out
  end
  return out
end
print(numberToString(100000000000000))
```

This will correctly show `100000000000000` in both the 32-bit and 64-bit versions of Lunacy.

Note that this function only works correctly for integers, and only for numbers small enough to correctly divide by 10.

COMMENTS

Lunacy has support for comments. Comments are blocks of code ignored by the Lunacy parser. Unlike traditional UNIX scripts, Lua uses two dashes to indicate the beginning of the comment.

Here is an example comment:

```
-- Here is a comment. It ends at the end of the line
foo = 1
-- Another comment
```

A comment which starts with two dashes ends at the end of a line.

Multi-line comments

Lua also has support for multi-line comments. A multi line comment starts with two dashes, immediately followed by two left square brackets. A multi-line comment ends with two right square brackets.

Here is an example multi-line comment:

```
foo = 1
--[[ Here is a multi-line comment.  It ends with two right square
     brackets, like this: ]]
bar = 2
```

It's also possible to use square bracket comment notation to put a comment in the middle of a line:

```
foo --[[ Here is a mid-line comment ]] = 1
```

In case we want `]]` in a comment, we can use this syntax:

```
foo = 1
--[=[ Here is another multi-line comment.  Since this comment has an equal
      sign between the brackets, ]] no longer ends a comment.  Instead,
      the comment ends with an equals sign between the brackets: ]=]
bar = 2
```

If both `]]` and `]=]` are desired in a comment, we can use `--[==[` or `--[===[` or `--[====[` to begin a comment; the comment is ended by the corresponding number of equals signs between the brackets closing the comment (e.g. `]==]` or `]===]` or `]====]`):

```
foo = 1
--[===[ Comment with three equals between the brackets
  ]===]
bar = 2
```

IF STATEMENTS

Lunacy implements *flow control* via "`if`" statements. The form is:

```
if condition then
  block of code #1
elseif other condition then
  block of code #2
else
  block of code #3
end
```

The "`elseif`" and "`else`" blocks are optional, and it's possible to have multiple `elseif` blocks. The way code is run is as follows:

- If the condition in the initial `if` statement is true, then we execute *block of code #1*
- Otherwise, if a condition in an `elseif` statement is true, when we execute the block of code after the `elseif` condition that is true. We only execute one `elseif` in a block.
- Otherwise, we execute the condition in the `else` block of code.

Condition is a statement that satisfies the `if`/`elseif` condition (so that we can run the block of code right after then condition) if it evaluates to anything besides `nil` or `false`. In particular, the number `0` and the empty string `""` are considered `true` for the purposes of evaluating an `if` or `elseif` block (also used with `while` and `until`, discussed elsewhere).

Condition statements are usually in the form *variable-or-value operation variable-or-value*. For example, `a < 10` evaluates to `true` if `a` is less than `10`. Some operations:

- `a < b` is true if `a` is less than `b`. Strings are lexically compared; for example "`a`" is less than "`b`", "`Z`" is less than "`a`", and the string "`11`" is less than the string "`2`".
- `a <= b` is true if `a` is less than or equal to `b`
- `a > b` is true if `a` is greater than `b`
- `a >= b` is true if `a` is greater than or equal to `b`
- `a == b` is true if `a` is the same as `b`. Strings are never equal to numbers. Tables are equal if they point to the same table; two different tables with the same data are not equal.
- `a ~= b` is true if `a` is not equal to `b`

This example code returns `"Fizzy"`, `"Buzzy"`, `3`, `4`, and finally `5`:

```
for number = 1,5 do -- Iterate so that number is 1, 2, 3, 4, and then 5
  if number < 2 then
    print("Fizzy")
  elseif number < 3 then
    print("Buzzy")
  else
    print(number)
  end
end
```

Compound if conditions

It possible to compile multiple conditions in to a single if statement. If we want a given condition to be true only if all of multiple conditions are true, we use `and`. For example:

```
for number=1,5 do -- Iterate so that number is 1, 2, 3, 4, and then 5
  if number > 2 and number < 5 then
    print(number)
  end
end
```

This will output 3 then 4. This is because the compound `if` statement has two sub-conditions. The compound `if` statement is only true if both:

- `number > 2` i.e. `number` is greater than two **`and`**
- `number < 5` i.e. `number` is less than five

If, on the other hand, we want a condition to be true if any of the sub-conditions is true, we use `or`. For example:

```
for number=1,5 do -- Iterate so that number is 1, 2, 3, 4, and then 5
  if number < 2 or number > 3 then
    print(number)
  end
end
```

This will output 1, 4, then 5. This is because the compound `if` statement, like the above one, had two sub-conditions. The compound `if` statement is true if either:

- `number < 2` i.e. `number` is less than two **`or`**
- `number > 3` i.e. `number` is more than three

The third way to modify an `if` statement is to use the `not` keyword. The `not` keyword makes an `if` condition that is otherwise `true` become `false` and an `if` condition that is otherwise `false` become `true`. For example:

```
for number=1,5 do -- Iterate so that number is 1, 2, 3, 4, and then 5
  if not (number < 3) then
    print(number)
  end
end
```

Here, the output will be 3, 4, then 5. In more detail, `number < 3` is only true when the above loop is run when `number` is 1 or 2, and is false when `number` is 3, 4, or 5. However, the not makes false statements true and vice versa, so now the compound state is true when `number` is 3, 4, or 5.

Note the parenthesis in the condition after `not`; they are mandatory.

Loops (While and Repeat)

In this chapter we will look at `while` loops, `repeat` loops, the `break` keyword, working around Lunacy's lack of a "continue" keyword, and show a basic `for` loop.

The While loop

In a `while` loop, we go through a loop while *condition* is true. A `while` loop starts with the `do` keyword and ends with the `end` keyword. The general form is:

```
while condition do
  actions
end
```

When Lunacy hits the `end` keyword, we see if *condition* is true; if so, Lunacy goes back to the `do` keyword which begins the `while` loop.

Let's this look at this code:

```
a = 1
while a < 10 do
  print(a)
  a = a + 1
end
```

The output is 1, 2, 3, 4, 5, 6, 7, 8, and 9.

While loops can nest, which means we can have one while loop in another while loop:

```
a = 1
while a < 4 do
  b = 1
  while b < 3 do
    print(a * b)
    b = b + 1
  end
  a = a + 1
end
```

Output is 1, 2, 2, 4, 3, 6.

Note that it's possible to have a `while` loop which never runs, e.g.:

```
a = 10
while a < 4 do
  print("This will never run")
end
print("We are here")
```

This will only output `We are here`.

The repeat loop

A `repeat` loop is similar to a `while` loop, but it will always goes through the loop once. For example:

```
a = nil
repeat
  if a == nil then
    a = 1
  end
  print(a)
  a = a + 1
until a > 9
print("Hello")
```

This outputs 1,2,3,4,5,6,7,8,9, then the word `Hello`.

Note that a `repeat` loop, unlike other loops, does not terminate with the `end` keyword but instead with an `until` statement.

The break keyword

It's possible to break a `for`, `while`, or `repeat` loop with the `break` keyword. For example:

```
a = 1
while true do
  if a >= 10 then
    break
  end
  print(a)
  a = a + 1
end
```

This will output 1,2,3,4,5,6,7,8, and 9.

With a `repeat` loop:

```
a = 1
repeat
  if a >= 10 then
    break
  end
  print(a)
  a = a + 1
until false
```

This will also output 1,2,3,4,5,6,7,8, and 9.

No continue keyword

Some other languages, such as C, have a "continue" keyword. This keyword would cause the loop to go back to the beginning of the loop, e.g. the `while` keyword if we're in a `while` loop.

Lunacy does not have a continue keyword, but we can simulate it by having a loop inside of a loop:

```
doBreak = false
a = 1
repeat
  while a < 10 do
    doBreak = true
    a = a + 1
    if a % 2 == 0 then
      doBreak = false -- Make break act like continue
      break -- Acts like continue
    end
    if a == 9 then
      break -- Acts like break because doBreak is true
    end
    print(a)
  end
until doBreak
```

This will output 3,5, and 7.

This trick works for `while` and `repeat` loops, but not with `for` loops (a version with the `repeat/until` inside the `for` loop would work; this is left as an exercise for the reader).

The for loop

Lunacy actually has two forms of a `for` loop. One is a simple for loop where we go through numbers; the other, which we will look at elsewhere, uses a function along with `for`. Here is a simple numerical `for` loop:

```
for a=1,10,2 do
  print(a)
end
```

This will output 1,3,5,7, and 9.

The "`1`" above is the value `a` has when we run the loop for the first time; `10` is the maximum value `a` can have and have the loop still run. `2` above is the amount we increment `a` by (or decrement, should this third number be negative) after each iteration of the loop.

Note that it's best in a `for` loop of this form to use only integers; see the chapter on numbers for discussion of why.

Strings

Lunacy has support for strings. For example:

```
a = "Hello, world!"
print(a)
```

This will output "`Hello, world!`".

Single quotes

Strings can be quoted with either single quotes or double quotes; here's single quotes:

```
a = 'Hello, world!'
print(a)
```

The only difference between single and double quotes is that it changes whether a single quote or a double quote needs to be escaped; unlike other languages, Lunacy does not by default expand variables in strings.

Strings are binary

A string is a binary stream; unlike C's default string type, the NUL (0 value) character can be part of a string. Lunacy does not have UTF-8 or any other locale support; high bit strings are stored as is and strings do not have locale information (i.e. what character encoding a given string uses or even what type of line feed is used) attached to them.

Escaping characters

It's possible to escape characters with strings:

- `\"` always returns a double quote without terminating a string. This escape sequence is not needed if the string is single quoted; e.g. `'" is a double quote'` is equivalent to the string `"\" is a double quote"` or `'\" is a double quote'`. In all cases, the resulting string is `" is a double quote`
- `\'` likewise always returns a single quote.
- `\\` returns a backslash
- `\` followed by a newline allows a multi-line string; the `\` is not part of the resulting string.
- `\0`, `\00`, or `\000` represents a NUL character
- Likewise *nnn*, where *nnn* is a one digit, two digit, or three digit decimal number between 0 and 255 represents a character with the corresponding ASCII (or high bit) numerical value. E.g. `\33` or `\033` is an exclamation point ("bang") character and is equivalent to `!`
- `\r`, `\n`, `\a`, `\b`, `\f`, `\t`, and `\v` generate the corresponding character (In order shown here: carriage return, newline, bell, backspace, form feed, tab, and vertical tab).

String concatenation

It's possible to concatenate strings using the `..` operator. For example:

```
a = "Hello, "
b = a .. "world!"
print(b)
```

returns `Hello, world!`

String manipulation functions

Let's look at some of Lunacy's string manipulation methods. String methods can be called either as `string.`*functionName* (the first argument is the string to run the method against) or as *string*`:`*functionName* (where the string name is already given); e.g. given that `a` is a string, `string.sub(a,1,3)` is equivalent to `a:sub(1,3)`. Note that strings in Lunacy are immutable and the functions which generate strings return newly created strings.

string.sub

`string.sub` returns a substring. Its arguments are `string.sub(`*string, start, end*), where *start* is the first character of the resulting string we put in the resulting substring (since Lua tends to be 1-indexed, 1 is the first character in the source string, 2 is the second character, and so on) and *end* is the final character. If *start* or *end* are negative, we count characters from the end of the string; e.g. `-1` is the final character in the string, `-2` is the second final character, and so on. If *end* is not specified, the resulting substring ends with the source string ends.

If a search is out of bounds, `string.sub` returns the empty string (and *not* `nil`).

Examples:

```
a = "1234567890"
print(a:sub(2,5)) -- Output is 2345
print(string.sub(a,3,8)) -- Output is 345678
print(a:sub(8)) -- Output is 890
print(a:sub(7,-2)) -- Output is 789
print(string.sub(a,-5,8)) -- Output is 678
print(a:sub(12,12)) -- Out of bounds, returns ""
```

string.len

`string.len` returns how long a string is. A string can be 0 or more bytes long. Examples:

```
a = "12345"
print(a:len()) -- Returns 5
a = ""
print(a:len()) -- Returns 0
```

string.find

`string.find` looks for a given substring in a string. Its form is `string.find(`*string*, *pattern*, *start*, *plain*`)`. *start* and *plain* are optional. *pattern* is the substring we are looking for; by default *pattern* is a Lunacy regular expression pattern (a regular expression allows various different strings with certain characteristics to match; regular expressions will be discussed elsewhere). *start* is the first character we look at in the string for the matching pattern; if not specified, *start* has a value of `1` which is the start of a string. *plain*, if set to `true`, looks for the literal string in *pattern* instead of performing regular expression matching.

If the pattern is found, `string.find` returns two numbers: Where the matching string starts (as a numeric string index; e.g. `1` is the first character), followed by where the matching string ends. If the pattern is not found, `string.find` returns `nil`.

For example, `string.find(a, "345")` will look for an occurrence of the substring `345` in the string `a`. Some other examples:

```
a = "1234567890"
print(a:find("567")) -- Returns 5,7
print(a:find("111")) -- Returns nil
a = "LoveLoveLove"
print(a:find("Love")) -- Returns 1,4
print(a:find("Love",3)) -- Returns 5,8
print(a:find("Love",7,true)) -- Returns 9,12
print(a:find("Love",11)) -- Returns nil
```

string.gsub

`string.gsub` replaces occurrences of a pattern with another string. It is called as `string.gsub(`*string*`, `*pattern*`, `*newString*`, `*count*`)` or *string*`:gsub(`*pattern*, *newString*, *count*`)`

string is the string we are creating an altered copy of. *pattern* is the regular expression pattern we are looking for (again, regular expressions will be discussed elsewhere). *newString* is the string we replace occurrences of *pattern* with. *count*, which is an optional argument, specifies the maximum number of times we replace *pattern*; if *count* is not specified, we replace all occurrences of *pattern* in *string*.

`string.gsub` returns the altered string as well as the number of times it found *pattern*.

Some examples:

```
a = "1234567890"
print(a:gsub("456","axy")) -- Returns "123axy7890",1
print(a:gsub("111","axy")) -- Returns "1234567890",0
a = "LifeLifeLife"
print(a:gsub("Life","Love")) -- Returns "LoveLoveLove",3
print(a:gsub("Life","Love",1)) -- Returns "LoveLifeLife",1
print(a:gsub("Life","Love",2)) -- Returns "LoveLoveLife",2
```

Recursion and local variables

Lunacy has support for recursion:

```
function factorial(a)
  if a <= 1 then
    return 1
  end
  return a * factorial(a - 1)
end
```

The above code works, e.g. `factorial(5)` returns `120`. This, however, will not work:

```
function sillyFactorial(a)
  b = a
  if b <= 1 then
    return 1
  end
  c = sillyFactorial(b - 1)
  return b * c
end
```

This is because `b` and `c` are global variables. This *does* work:

```
function worksFactorial(a)
  local b = a
  if b <= 1 then
    return 1
  end
  local c = worksFactorial(b - 1)
  return b * c
end
```

Point being, any variables defined in a recursive function should always be `local`.

Local variables are variables which only exist in the block (`function`, `while`, `if`, etc.) they are declared as `local` in. Let's look at a simpler example of `local`:

```
a = 1
function changeA(value)
  local a = value
  print(a)
end
changeA(2)
print(a)
```

This will return `2` then `1` because the instance of `a` in the `changeA` function is a different instance than the `a = 1` which is in the global (i.e. non-local) scope. If the `local` keyword was not present in the above block (i.e. `a = value` instead of `local a = value`) then it would return `2` twice, because then the global `a` would be changed in the `changeA` function.

FUNCTION CLOSURE

Some other languages, such as C, support "static" variables, i.e. variables local to a function which keep their values between calls to the function, e.g.:

```
int stat() {
  static int z = 0;
  return ++z;
}
int main() {
  int a;
  for(a = 0; a < 10; a++) {
    printf("%d ",stat());
  }
  puts("");
}
```

This returns `1 2 3 4 5 6 7 8 9 10` because the value of `z` is remembered between invocations of `stat()`

To do this in Lunacy requires a function which returns a function. Here's the equivalent Lunacy code:

```
function statFactory()
  local z = 0
  return function()
    z = z + 1
    return z
  end
end
stat = statFactory()
for a = 0,9 do
  print(stat())
end
```

Like the C code above, this also returns 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10, since the value of `z` is remembered between invocations of the `stat` function.

This is called *function closure* and it's Lunacy's way of having variables local to a function which keep their value between calls of the function. The advantage of function closure over static variables is that we can have multiple instances of `stat` with their own closure. Given the `statFactory` function above:

```
stat1 = statFactory()
stat2 = statFactory()
print(stat1())
for a=1,5 do print(stat2()) end
print(stat1())
```

This will return 1 (`stat1`), 1,2,3,4, and 5 (`stat2`), then 2 (`stat1` again).

Types

To determine the type of a given variable, we use the `type` keyword. `type` returns a string describing what kind of value a given variable has. Some examples:

```
a = {this = "is", a = "table", n = 321}
b = 1.2345
c = "This is a string"
print(type(a))
print(type(b))
print(type(c))
print(type(a.n))
print(type(a.this))
```

This will return the strings `table`, `number`, `string`, `number`, and `string`.

Another type that can be seen is `userdata`. In the chapter on file input and output, we will see code that looks like this:

```
handle = io.open("foo.txt","wb")
print(type(handle))
```

This will return the string `userdata`. This code will also write a blank file called `foo.txt`, so run this code with care.

We can use the `type` operator to help recursively look at the contents of a table (see the chapter on tables for more discussion of the `table` type):

```
function viewTable(theTable, prefix, depth)
  if not prefix then
    prefix = ""
  end
  if not depth then
    depth = 1
  end
  if depth > 10 then
    return nil
  end
  if type(theTable) ~= "table" then
    return nil
  end
  for k,v in pairs(theTable) do
    if type(v) ~= "table" then
      print(prefix .. k,v)
    else
      viewTable(v, k .. "/", depth + 1)
    end
  end
end
-- Example usage of the viewTable function above
theTable = {a = 1, b = { c = 1, d = 2 }, e = "Hello!" }
viewTable(theTable)
```

Binary Numbers

Numbers in Lunacy are *binary numbers*, which mean they are internally represented as numbers with only 0s and 1s in them. The number 0 is represented as `0`, the number 1 is `1`, but the number 2 is `10`, and the number 3 is `11`, 4 is `100`, and so on (to be pedantic, the representation for numbers in Lunacy are actually IEEE 754 64-bit floating point numbers which have a different representation in binary, but for the purposes of this lesson we can treat the numbers as if they are binary integers). Each digit in a binary number is called a *bit*.

Hexadecimal

While it's not possible in Lunacy to directly represent numbers in binary, we can represent numbers as hexadecimal (sometimes called hex) numbers. This is a base 16 notation and we tell Lunacy the number is hexadecimal by prefixing it with `0x` so `0x9 + 0x1` is `0xA` or equivalently 10. Let's look at numbers in decimal (normal numbers as taught in elementary school math), binary (base 2, as discussed above), and hexadecimal.

Decimal	`0`	`1`	`2`	`3`	`4`	`5`	`6`	`7`
Binary	`0000`	`0001`	`0010`	`0011`	`0100`	`0101`	`0110`	`0111`
Hexadecimal	`0x0`	`0x1`	`0x2`	`0x3`	`0x4`	`0x5`	`0x6`	`0x7`

Decimal	`8`	`9`	`10`	`11`	`12`	`13`	`14`	`15`
Binary	`1000`	`1001`	`1010`	`1011`	`1100`	`1101`	`1110`	`1111`
Hexadecimal	`0x8`	`0x9`	`0xA`	`0xB`	`0xC`	`0xD`	`0xE`	`0xF`

Note forms like `0xa` or `0xc` are also acceptable; either upper case or lower case or even mixed case (e.g. `0xfF`) is acceptable.

Viewing numbers in hexadecimal can be done with the `string.format` call as follows:

```
function viewAsHex(n)
  return string.format("%x",n)
end
print(viewAsHex(0xBeadBabe))
```

If we wish to view the hexadecimal number in upper case, use `%X` for the string format:

```
function viewAsHexUpper(n)
  return string.format("%X",n)
end
print(viewAsHexUpper(0xBeadBabe))
```

32-bit numbers

There are certain binary operations Lunacy can perform on the numbers. Since the float point format Lunacy uses to represent numbers allows numbers with up to 53 bits of precision, we can not use 64-bit binary integers in Lunacy. 32-bit is the largest common representation of binary numbers smaller than the 53 bits Lunacy can actually store as a number, so the

functions which this book is about to show representing binary math are done with 32-bit numbers. `bit32.bor(0,foo)` will return the value of `foo` as long as `foo` can fit in 32 bits. That in mind:

```
print(string.format("%x",bit32.bor(4294967295,0)))
```

returns `ffffffff`, the largest number we can represent with 32 bits, but

```
print(string.format("%x",bit32.bor(4294967297,0)))
```

returns `1` (two more than the largest number one can fit in 32 bits; its hex form would be `0x100000001` but 32-bit Lunacy can only handle numbers inputted as hex between `0x00000000` and `0xffffffff`) because the `bit32` operations work on return the lowest 32 bits of the input number.

Binary operations

While Lua 5.1 does not have direct support for bitwise binary operations, Lunacy (as well as other common ports of Lua 5.1 such as LuaJIT, Luau used in Roblox, and the version of Lua 5.1 used by the Wikipedia) includes a library of functions which allow binary operations. The number of this library in Lunacy is `bit32`.

Common binary operations are *and*, called `bit32.band` in Lunacy, *or* called `bit32.bor`, and *exclusive or* called `bit32.bxor`. These operations work on each bit independently. Let's look at the four possible one bit results for these operations, where the input is `0` or `1`

Operation	Result
`bit32.band(0,0)`	`0`
`bit32.band(0,1)`	`0`
`bit32.band(1,0)`	`0`
`bit32.band(1,1)`	`1`
`bit32.bor(0,0)`	`0`
`bit32.bor(0,1)`	`1`
`bit32.bor(1,0)`	`1`
`bit32.bor(1,1)`	`1`
`bit32.bxor(0,0)`	`0`
`bit32.bxor(0,1)`	`1`
`bit32.bxor(1,0)`	`1`
`bit32.bxor(1,1)`	`0`

As mentioned before, these operations work on each bit independently. So, if the two inputs have 3 bits, we perform the relevant operation on the lowest bit of the input numbers and have the determine the lowest bit in the output number, then perform the relevant operation on the second lowest bit of the input numbers to determine the second lowest bit in the output number, and finally perform the operation on the third lowest bit of the input numbers which affects the third lowest bit in the output number.

So, for example, to exclusive or the binary numbers `100` (four) and `110` (six), we do the following steps:

1. The highest (leftmost) bits in the input numbers are `1` and `1`. The exclusive or of `1` and `1` is `0`. So the leftmost bit of the output is `0`.
2. The second (middle) bits in the input numbers are `0` and `1`. The exclusive or of `0` and `1` is `1`, so the middle bit in the output is `1`.
3. The lowest (rightmost) bits in the input numbers are `0` and `0`. The exclusive or of `0` and `0` is `0`, so the lowest bit in the output is `0`.

Let's look at this another way:

`100` input one
`110` input two
`010` result

Since binary `100` is `4` in decimal and binary `110` is `6` in decimal, the result is `2` (`010`).

And, indeed:

```
print(bit32.bxor(4,6))
```

Returns `2`.

Other binary operations

Another binary operation is `bit32.lshift(n,x)`, which shifts a number `n` to the left `x` number of bits. For example, shifting the binary number `110` one bit to the left yields `1100`; shifting `101` two bits to the left results in `10100`. A `lshift` of one bit is equivalent of multiplying a number by 2; a two bit shift is like multiplying a number by 4; a three bit shift like multiplying by 8; and so on.

The other binary operation in the example xoshiro code is `bit32.rrotate(n,x)`, which rotates a number `n` by `x` bits to the right. Since this is an operation on 32-bit numbers, this means that if the low (rightmost) bit is `1`, the high (32nd from left) bit becomes `1` when rotating one bit to the right.

One example of `bit32.rrotate`:

Binary form of `n` `00000010 00000111 10100101 10000001` In hex: `0x0207a581`
`bit32.rrotate(n,2)` `01000000 10000001 11101001 01100000` In hex: `0x4081e960`

Here, the two low bits of the input become the two high bits in the output, and the rest of the bits move to the right by two spots.

A real world example

One real-world algorithm which generates *pseudo-random numbers* (a pseudo-random number is a number which looks random, but is actually generated in a non-random way) using 32-bit number bitwise math is *xoshiro128***. Here is an implementation of the algorithm in Lua:

```
xoshiro128 = {}
xoshiro128.__index = xoshiro128
function xoshiro128:init()
  out = {}
  setmetatable(out, xoshiro128) -- Instance of xoshiro128 class
  out.s1 = 1
  out.s2 = 2
  out.s3 = 3
  out.s0 = 4
  return out
end
function xoshiro128:next()
  local result = self.s1 * 5
  result = result % 4294967296 -- Keep number 32-bit
  result = bit32.rrotate(result,25) -- 7-bit left shift in reference code
  result = result * 9
  result = result % 4294967296
  local t = bit32.lshift(self.s1, 9)
  self.s2 = bit32.bxor(self.s2,self.s0)
  self.s3 = bit32.bxor(self.s3,self.s1)
  self.s1 = bit32.bxor(self.s1,self.s2)
  self.s0 = bit32.bxor(self.s0,self.s3)
  self.s2 = bit32.bxor(self.s2,t)
  self.s3 = bit32.rrotate(self.s3,21)
  return result
end
-- This seeding is not part of the xoshiro128** spec
function xoshiro128:seed(s)
  s = s % 4294967296
  self.s0 = bit32.bxor(s,0x55555555)
  self.s1 = (s * 37 + s * 3 + 1009) % 4294967296
  self.s2 = (s * 223 + s * 7 + 1229) % 4294967296
  self.s3 = (s * 947 + bit32.bxor(s,0xAAAAAAAA)) % 4294967296
  return true
end
x = xoshiro128:init()
x:seed(12345)
for a=1,8 do
  print(string.format("%08x",x:next()))
end
```

This code uses classes, which are discussed in the chapter on metatables.

While stock Lua 5.1 uses a low-quality pseudo random number generator for `math.random()` and the above code generates higher quality numbers, Lunacy's `math.random()` generates high quality random numbers using RadioGatún[32].

Tables

Let's look at Lunacy's tables. A *table* is a way to store multiple entries in a variable, akin to Python's dictionaries, Perl's "hashes", and AWK's associative arrays.

```
foo = {}
foo["bar"] = "Hello, "
foo["baz"] = "world!"
foo["bozzle"] = function() print("Hello, world!") end
```

Then, we can do stuff like this:

```
print(foo["bar"])
print(foo["baz"])
print(foo["bozzle"])
```

Since `foo["bozzle"]` is a function (table members can be numbers, strings, functions, tables, booleans, among other data types, but not `nil`), we can call it directly:

```
foo["bozzle"]()
```

or even

```
print(foo["bozzle"]())
```

We can define the above table like this in Lua:

```
foo = {
   bar = "Hello, ",
   baz = "world!",
   bozzle = function() print("Hello, world!") end
}
```

Tables, as noted above, can have members which are themselves tables:

```
foo = {}
foo["bar"] = { oink = "Hello, world!",
               mister = function() print("Hello, world!") return 1 end
             }
foo["baz"] = "Cool!"

print(foo["bar"]["oink"])
print(foo["bar"]["mister"]())
```

The brackets and quotes can get unwieldy, so this also works:

```
print(foo.bar.oink)
print(foo.bar.mister())
```

We can combine brackets and dot notation (see the next page):

```
print(foo["bar"].oink)
```

Which is useful when using a variable to determine the table index:

```
bozzle = "bar"
print(foo[bozzle].oink)
```

We can even have a sub-table refer to the parent table:

```
foo.bar.loop = foo
```

And then:

```
print(foo.bar.loop.bar.oink)
print(foo.bar.loop.bar.loop.bar.oink)
print(foo.bar.loop.bar.loop.bar.loop.bar.oink)
```

To see the members of a table, we have to iterate through them:

```
for k,v in pairs(foo) do print(k,v) end
```

Which gives us output like this:

```
bar     table: 00F49F08
baz     Cool!
```

We can use recursion to look at subtables, but be careful of loops!

```
function tableView(t, name, depth)
  -- If we call a function with too few arguments, the extra parameters
  -- have the value nil
  if name == nil then
    name = ""
  end
  if depth == nil then
    depth = 0
  end

  -- Infinite loop protection
  if depth > 10 then
    return 0
  end

  -- View this table and recurse with subtables
  for k,v in pairs(t) do
    if type(v) ~= "table" then -- ~= is != in most other languages
      print(name,k,v)
    else
      tableView(v, name .. " " .. k, depth + 1)
    end
  end
end
```

Note that Lua uses "`--`" to start a comment, unlike the "`#`" most UNIX scripting languages use.

We can now invoke it:

```
tableView(foo)
```

Note the function takes three arguments; since we call the function with only one argument, the other two values are set by Lua to be `nil`.

While the above works, the issue is that when we have an element which loops back to the top of the table, we will show that element multiple times. There *is* a way to solve that problem, but it's a little tricky to understand.

The trick is this: Table keys can *also* be tables (well, pointers to tables; Lua internally stores the memory address of the table as the key and a note the pointer is to a table).

Let's check this out:

```
foo[foo] = "Table pointer key example"
```

Here, the *key* is a pointer to a table. This can be useful, because we can use that to know right away if we're going in to a loop with `tableView`:

```
function tableView(t, name, seen)
  -- If we call a function with too few arguments, the extra parameters
  -- have the value nil
  if name == nil then
    name = ""
  end
  if seen == nil then
    seen = {t = true}
  end

  seen[t] = true

  -- View this table and recurse with subtables
  for k,v in pairs(t) do
    if type(v) ~= "table" then -- ~= is != in most other languages
      print(name,k,v)
    else
      if seen[v] then
        print(name, k, "LOOP!")
      else
        tableView(v, name .. " " .. k, seen)
      end
    end
  end
end
```

Arrays

Lunacy does not have an array type *per se* but it can simulate an array by having a table where the keys are integer numeric keys like 1, 2, 3, 4, and so on. Since *any* number can be a table key, Lunacy can have 0-indexed arrays (0, 1, 2, 3, etc.) or even 5-indexed arrays (5, 6, 7, etc.) or any other number (-1 indexed arrays would be possible too).

Some of the routines and functions assume an array is 1-indexed, where the first element in the array is 1, but this is mainly a matter of convenience which isn't hard-coded in to Lunacy arrays.

Array length

Lunacy uses the `#` character to indicate the highest numeric index an array has, as long as said index is 1 or higher. For example:

```
a={} a[0]="a" a[1]="b" a[2]="c"
b={} b[1]="d" b[2]="e" b[3]="f"
print(#a)
print(#b)
```

This will output 2 (the highest element in the array "`a`") then 3 (the highest element in the array "`b`"). Note that a 0-indexed array will have a length value of 0 both when the array has either 0 or 1 elements.

Adding and removing elements from arrays

One can add and remove elements from an array using the built-in `table.insert()` and `table.remove()` methods. To insert an element at the end of an array:

```
a = {}
table.insert(a, "Hello")
for z=1,#a do
  print(z,a[z])
end
```

This will output `1` then `Hello`, the key and value for the one element in the array.

`table.remove()` not only removes an element from the array, it shifts the elements above the removed element down so the array does not have any holes in it. For example:

```
b={} b[1]="d" b[2]="e" b[3]="f" b[4]="g"
table.remove(b,2)
for a=1,#b do
  print(a, b[a])
end
```

This will output (1, "`d`"), (2, "`f`"), (3, "`g`"); the "`e`" element has been removed and the other elements shifted down in the array; e.g. element 3 was "`f`" before and now element 2 is "`f`".

METATABLES

Let's make two tables in Lua:

```
foo = { bar = "Hello", baz = "World" }
someTable = { life = "is", good = "yes" }
print(someTable.bar)
```

This returns "`nil`". When we look for `someTable.bar` (i.e. `someTable["bar"]`), we will get `nil` because the element "`bar`" is not in `someTable`, even though it's in the table `foo`.

Now, let's have it so that, if an element is not in `someTable`, we look in the table `foo` for that element:

```
someTableMeta = {__index = foo}
setmetatable(someTable, someTableMeta)
print(someTable.bar)
```

Now, looking up `someTable.bar` returns the value "`Hello`":

1. We look in `someTable` for the element "`bar`"
2. Since it's not found, we see if we have an index metatable. We do, so we look in the index metatable, in this case the table `foo`, for the element.
3. Since `foo` has the element `foo.bar`, we use the element in the index metatable for `someTable` to have it so asking for `someTable.bar` gives us "`Hello`".

If we don't want a separate metatable, a table can be its own metatable:

```
foo = { bar = "Hello", baz = "World" }
someTable = { life = "is", good = "yes", __index = foo }
setmetatable(someTable, someTable)
print(someTable.bar)
```

`someTable.bar` is now "`Hello`", since `foo.bar` is "`Hello`" and `someTable`'s index metatable is the `foo` table.

Likewise, the index metatable can also be the metatable:

```
foo = { bar = "Hello", baz = "World" }
foo.__index = foo
someTable = { life = "is", good = "yes" }
setmetatable(someTable, foo)
print(someTable.bar)
```

Again, this returns "`Hello`".

Note that the "`__index`" element needs to be set *after* the table is initially declared.

Index metatables can also be functions:

```
someTable = { life = "is", good = "yes",
              __index = function() return 404 end }
setmetatable(someTable, someTable)
print(someTable.bar)
```

This returns the number `404`. Here, any element which is not in the table will return, instead of `nil`, the number `404`.

We can use a function to have the same behavior we have when an index metatable is a table:

```
foo = { bar = "Hello", baz = "World" }
someTable = { life = "is", good = "yes",
              __index = function(self, i) return foo[i] end }
setmetatable(someTable, someTable)
print(someTable.bar)
```

This will return "`Hello`". The second argument to an index metatable function is the element we tried to access in the table.

The first argument, `self`, is a reference to the table calling the function, e.g. this will return "`yes`" for any element not otherwise in `someTable`:

```
someTable = { life = "is", good = "yes",
              __index = function(self, i) return self.good end }
setmetatable(someTable, someTable)
print(someTable.bar)
```

This returns "`yes`".

All of this in mind, we can use metatables to have classes:

```
simpleClass = {}
simpleClass.__index = simpleClass
function simpleClass.init(self, x, y)
  self.x = x
  self.y = y
end
function simpleClass.add(self, x, y)
  self.x = self.x + x
  self.y = self.y + y
end
function simpleClass.values(self)
  return self.x, self.y
end
```

Now that we have created a simple class, we can make instances of that class via metatables. Give the above class, let's make an instance of it:

```
objectInstance = {}
setmetatable(objectInstance, simpleClass)
objectInstance.init(objectInstance, 1, 2)
objectInstance.add(objectInstance, 3, 3)
print(objectInstance.values(objectInstance))
```

This will return `4` then `5`.

This works but the grammar is a bit clunky. Lua has some synatic sugar to make things look a little nicer. This is equivalent to the above example class code:

```
simpleClass = {}
simpleClass.__index = simpleClass
function simpleClass:init(x, y)
  self.x = x
  self.y = y
end
function simpleClass:add(x, y)
  self.x = self.x + x
  self.y = self.y + y
end
function simpleClass:values()
  return self.x, self.y
end
objectInstance = {}
setmetatable(objectInstance, simpleClass)
objectInstance:init(1,2)
objectInstance:add(3,3)
print(objectInstance:values())
```

This also returns `4` then `5`.

"`self`" is a magic word in Lua used with Lua's colon syntax.

Let's change the class so that, for addition, we have a method which returns a new object which is the sum of two other objects:

```
simpleClass = {}
simpleClass.__index = simpleClass
function simpleClass:init(x, y)
  self.x = x
  self.y = y
end
function simpleClass:add(other)
  newObject = {}
  setmetatable(newObject, simpleClass)
  newObject.x = self.x + other.x
  newObject.y = self.y + other.y
  return newObject
end
function simpleClass:values()
  return self.x, self.y
end
```

```
objectInstance = {}
setmetatable(objectInstance, simpleClass)
anotherObject = {}
setmetatable(anotherObject, simpleClass)
objectInstance:init(1,2)
anotherObject:init(3,3)
thirdObject = objectInstance:add(anotherObject)
print(thirdObject:values())
```

This again returns `4` then `5`.

This in mind, we can use metatables to define what "`+`" does with a given table:

```
simpleClass = {}
simpleClass.__index = simpleClass
function simpleClass:init(x, y)
  self.x = x
  self.y = y
end
function simpleClass:add(other)
  newObject = {}
  setmetatable(newObject, simpleClass)
  newObject.x = self.x + other.x
  newObject.y = self.y + other.y
  return newObject
end
simpleClass.__add = simpleClass.add
function simpleClass:values()
  return self.x, self.y
end
objectInstance = {}
setmetatable(objectInstance, simpleClass)
anotherObject = {}
setmetatable(anotherObject, simpleClass)
objectInstance:init(1,2)
anotherObject:init(3,3)
thirdObject = objectInstance + anotherObject
print(thirdObject:values())
```

Like the other examples, this returns `4` then `5`.

As we can see, Lua has operator overloading.

We can make it a little easier to make new objects. In addition, the `__add` metatable can be a function instead of a pointer to a function (see the next page):

```
simpleClass = {}
simpleClass.__index = simpleClass
function simpleClass:init(x, y)
  out = {}
  setmetatable(out, simpleClass)
  out.x = x
  out.y = y
  return out
end
function simpleClass:__add(other)
  newObject = {}
  setmetatable(newObject, simpleClass)
  newObject.x = self.x + other.x
  newObject.y = self.y + other.y
  return newObject
end
function simpleClass:values()
  return self.x, self.y
end
objectInstance = simpleClass:init(1, 2)
anotherObject = simpleClass:init(3, 3)
thirdObject = objectInstance + anotherObject
print(thirdObject:values())
```

This also returns `4` then `5`.

Note that while the `init` method gets a "`self`" argument, we ignore that so that we can retain the colon syntax for object methods.

FOR LOOPS WITH FUNCTIONS

It's possible to use a function with a `for` loop. E.g:

```
a = {z = 1, y = 2}
for k,v in pairs(a) do
  print(k,v)
end
```

This will output in varying order (z,1) and (y,2).

We can make our own function which can be used as the iterator in a `for` loop:

```
function range(low, high)
  local index = low
  return function()
    local rvalue = index
    index = index + 1
    if(rvalue > high) then
      return nil
    end
    return rvalue
  end
end
```

Here, `range` is a function which returns a function. What does the function generated by `range` do? Let's look:

```
twoToNine = range(2,9)
for a=1,11 do
  print(twoToNine())
end
```

We get this output:

```
2
3
4
5
6
7
8
9
nil
nil
nil
```

Here is what is happening: The function `twoToNine` is able to remember the value of the variable `index` between its invocations. `index` is a variable whose value gets stored in a namespace local to the `twoToNine` function; this ability is called *function closure*.

That in mind, we can use this `range` function in a `for` loop:

```
for a in range(2,9) do
  print(a)
end
```

Given `range` as defined above, the above function will return this:

```
2
3
4
5
6
7
8
9
```

Under the hood, Lunacy calls the `range` function once. The `range` function returns an anonymous function which we will call *instanceOfRange*. This *instanceOfRange* function is then called at the start of the `for` loop. If *instanceOfRange* returns `nil` then the loop terminates; otherwise we go through the loop with the variable `a` set to be the output value of the *instanceOfRange* function. We do this over and over until *instanceOfRange* returns `nil`.

Sorted table keys

Let's make a `for` iterator function which will return a function that returns the key for each element in a table in sorted order:

```
function sortedTableKeys(inputTable)
  local keyList = {}
  local index = 1
  for k,_ in pairs(inputTable) do
    table.insert(keyList,k)
  end
  table.sort(keyList)
  return function()
    rvalue = keyList[index]
    index = index + 1
    return rvalue
  end
end
```

Here, `table.insert` is used to add a key in `inputTable` to `keyList`; it is run over and over again to make `keyList` a list of all of the keys in `inputTable`. Once we have this list as an array, `table.sort` sorts the list for us.

Next, the `sortedTableKeys` function creates a function which will simply return, when called over and over again, the keys to `inputTable` in sorted order. That in mind, let's make a table and compare how the `sortedTableKeys` function runs versus how `pairs` runs (on next page):

```
someTable = {a=12,b=14,c=19,d=23,e=17,f=11}

print("Unsorted:")
for a in pairs(someTable) do
  print(a) -- Letters in random order
end

print("")
print("Sorted:")
for a in sortedTableKeys(someTable) do
  print(a) -- Letters correctly sorted
end
```

This will first output the keys to `someTable` in random order (as an aside, the order changes each time Lunacy is started), then output the keys to `someTable` in sorted order, namely a, b, c, d, e, and f.

We can even have a function which determines how the keys are sorted:

```
function sortedTableKeysSF(inputTable, sortFunction)
  local keyList = {}
  local index = 1
  for k,_ in pairs(inputTable) do
    table.insert(keyList,k)
  end
  table.sort(keyList, sortFunction)
  return function()
    rvalue = keyList[index]
    index = index + 1
    return rvalue
  end
end

function revSort(a, b)
  return a > b
end

someTable = {a=12,b=14,c=19,d=23,e=17,f=11}

for a in sortedTableKeysSF(someTable, revSort) do
  print(a) -- Letters sorted in reverse
end
```

Here, the above code returns f, e, d, c, b, and a: The letters reverse sorted. The default (forwards, i.e a, b, c, d, e, and f) sorting would be done as follows:

```
function normalSort(a, b)
  return a < b
end

for a in sortedTableKeysSF(someTable, normalSort) do
  print(a)
end
```

List iteration

Lua has simple numerical iteration via `for foo=1,10 do` and Lua has function iteration via `for foo in something(bar) do` but Lua does not have built in iterators for a list the way, say, Perl or Python do.

However, we can work around this by making functions which let Lua go through a list.

First, let's have a function that, when given a table, returns its keys as a list:

```
function tableKeys(inputTable)
  local out = {}
  for k,_ in pairs(inputTable) do
    table.insert(out,k)
  end
  return out
end
```

Now, let's have a function that will allow us to use `for` to iterate through a list:

```
function iterateList(inputList)
  local index = 1
  return function()
    local rvalue = inputList[index]
    index = index + 1
    return rvalue
  end
end
```

And another function which sorts a list and returns the sorted list

```
function sorted(inputList)
  table.sort(inputList)
  return inputList
end
```

We can also have a function to reverse a list:

```
function reverse(inputList)
  local out = {}
  for a=#inputList,1,-1 do
    table.insert(out,inputList[a])
  end
  return out
end
```

With these four functions, we can iterate through a table with the keys reversed sorted:

```
someTable = {a=12,b=14,c=19,d=23,e=17,f=11}
for k in iterateList(reverse(sorted(tableKeys(someTable)))) do
  print(k, someTable[k])
end
```

MULTIPLE INHERITANCE

In the chapter on metatables we looked at simple inheritance. We can also have multiple inheritance:

- A class can have child classes
- Child classes can have multiple parents

Children

A given class can have a derived (a.k.a. child) class. Let's look at the xoshiro128** example we used in the chapter on binary numbers, but now with a child class. First, the parent class:

```
xoshiro128 = {}
xoshiro128.__index = xoshiro128

function xoshiro128:init()
  out = {}
  setmetatable(out, self)
  out.s1 = 1
  out.s2 = 2
  out.s3 = 3
  out.s0 = 4
  return out
end

function xoshiro128:next()
  local result = self.s1 * 5
  result = result % 4294967296 -- Keep number 32-bit
  result = bit32.rrotate(result,25) -- 7-bit left shift in reference code
  result = result * 9
  result = result % 4294967296
  local t = bit32.lshift(self.s1, 9)
  self.s2 = bit32.bxor(self.s2,self.s0)
  self.s3 = bit32.bxor(self.s3,self.s1)
  self.s1 = bit32.bxor(self.s1,self.s2)
  self.s0 = bit32.bxor(self.s0,self.s3)
  self.s2 = bit32.bxor(self.s2,t)
  self.s3 = bit32.rrotate(self.s3,21)
  return result
end
```

We can run this:

```
x = xoshiro128:init()
print(string.format("%x",x:next()))
```

Now that we have this class, we can make a derived class which adds the seeding method. We do this because the method to seed a xoshiro128** state is not part of its specification. On the next page is how that code will look:

```
-- Let's make a class which is the child class
-- This seeding is not part of the xoshiro128** spec
xoSeed = {}
xoSeed.__index = xoSeed
setmetatable(xoSeed,xoshiro128)

function xoSeed:seed(s)
  s = s % 4294967296
  self.s0 = bit32.bxor(s,0x55555555)
  self.s1 = (s * 37 + s * 3 + 1009) % 4294967296
  self.s2 = (s * 223 + s * 7 + 1229) % 4294967296
  self.s3 = (s * 947 + bit32.bxor(s,0xAAAAAAAA)) % 4294967296
  return true
end
```

Now that we have a child class, we can run it:

```
x = xoSeed:init()
x:seed(12345)
for a=1,8 do
  print(string.format("%08x",x:next()))
end
```

Let's look at the `init()` method which is used by both the parent and child class again:

```
function xoshiro128:init()
  out = {}
  setmetatable(out, self)
  out.s1 = 1
  out.s2 = 2
  out.s3 = 3
  out.s0 = 4
  return out
end
```

Here, the `setmetatable` function call has it so the class the object is derived from, `self` in the code, is whatever class we are calling the function in. If we're calling this method from the `xoshiro128` class, then `self` will be `xoshiro128`. If we're calling this method from the `xoSeed` class, then `self` is `xoSeed`.

Multiple parents

Since the `__index` member of a metatable can be either a table or a function, we can have multiple parents for a class in Lua. Let's see how that would look. Note that the `xoshiro128` class is the same as it was before:

```
xoshiro128 = {}
xoshiro128.__index = xoshiro128

function xoshiro128:init()
  out = {}
  setmetatable(out, self)
  out.s1 = 1
  out.s2 = 2
  out.s3 = 3
  out.s0 = 4
  return out
end

function xoshiro128:next()
  local result = self.s1 * 5
  result = result % 4294967296 -- Keep number 32-bit
  result = bit32.rrotate(result,25) -- 7-bit left shift in reference code
  result = result * 9
  result = result % 4294967296
  local t = bit32.lshift(self.s1, 9)
  self.s2 = bit32.bxor(self.s2,self.s0)
  self.s3 = bit32.bxor(self.s3,self.s1)
  self.s1 = bit32.bxor(self.s1,self.s2)
  self.s0 = bit32.bxor(self.s0,self.s3)
  self.s2 = bit32.bxor(self.s2,t)
  self.s3 = bit32.rrotate(self.s3,21)
  return result
end

seeder = {}
seeder.__index = seeder
function seeder:seed(s)
  s = s % 4294967296
  self.s0 = bit32.bxor(s,0x55555555)
  self.s1 = (s * 37 + s * 3 + 1009) % 4294967296
  self.s2 = (s * 223 + s * 7 + 1229) % 4294967296
  self.s3 = (s * 947 + bit32.bxor(s,0xAAAAAAAA)) % 4294967296
  return true
end

mix={}
mix.__index = function(self,element)
  if xoshiro128[element] then
    return xoshiro128[element]
  end
  if seeder[element] then
    return seeder[element]
  end
  return nil
end
setmetatable(mix, mix)

x = mix:init()
x:seed(12345)
for a=1,8 do print(string.format("%08x",x:next())) end
```

Private functions (Includes RadioGatún[32] class)

Lunacy has support for private functions, e.g.:

```
repeat

  local function a()
    return 3
  end
  local function b()
    return 4
  end
  function c()
    return a() + b()
  end

  print(a)
  print(b)
  print(c())

until true

print(a)
print(b)
print(c())
```

This will print two function pointers, the number 7, `nil` twice, then the number 7 again. Point being, the functions `a()` and `b()` are only visible in the `repeat/until true` block, but the function `c()` which can see `a()` and `b()` can always see those private functions, even in contexts where `a()` and `b()` are not visible.

Private functions are useful when writing object oriented code. Let's see that by looking at a Lunacy implementation of RadioGatún[32], the random number generator Lunacy uses:

```
repeat
```

The class is public, and we make it its own metatable:

```
rg32 = {}
rg32.__index = rg32
```

"`local`" here indicates that the function is private, i.e. it's not visible outside of the "`repeat`"/"`until true`" block of code. This function implements the "belt" and "mill" part of RadioGatún, which is a cryptographic protocol originally introduced in 2006 and remains secure. The exact design of a cryptographic protocol requires very specialized math and cryptanalyst to get right; programmers without knowledge of that kind of math should implement a cryptographic protocol *exactly* as specified. The specification for RadioGatún is available at `https://radiogatun.noekeon.org/RadioGatun.pdf`

```
-- Note that belt and mill are 1-indexed here
local function beltMill(belt, mill)
```

The mill to belt feedforward is on page 9 of the RadioGatún specification, in the “algorithm 3” section.

```
  -- Mill to belt feedforward
  for z = 0, 11 do
    offset = z + ((z % 3) * 13) + 1
    belt[offset] = bit32.bxor(belt[offset],mill[z + 2])
  end
```

The mill core is as per page 9 of the RadioGatún specification, in the “algorithm 4” section.

```
  -- Mill core
  local rotate = 0
  local millPrime = {}
  for z = 0, 18 do
    rotate = rotate + z
    local view = (z * 7) % 19
    local num = mill[view + 1]
    view = (view + 1) % 19
    local viewPrime = (view + 1) % 19
    num = bit32.bxor(num,bit32.bor(mill[view + 1],
                    bit32.bnot(mill[viewPrime + 1])))
    num = bit32.rrotate(num,rotate)
    millPrime[z + 1] = num
  end
  for z = 0, 18 do
    local view = (z + 1) % 19
    local viewPrime = (z + 4) % 19
    mill[z + 1] = bit32.bxor(millPrime[z + 1],
        millPrime[view + 1],millPrime[viewPrime + 1])
  end
```

The belt rotate is part of the “algorithm 3” section of the RadioGatún specification.

```
  -- Belt rotate
  for z = 39, 1, -1 do
     belt[z + 1] = belt[z]
  end
  for z = 0, 2 do
     belt[(z * 13) + 1] = belt[((z + 1) * 13) + 1]
  end
```

The belt to mill feedforward is also part of the “algorithm 3” section.

```
  -- Belt to mill
  for z = 0, 2 do
     mill[14 + z] = bit32.bxor(belt[(z * 13) + 1],mill[14 + z])
  end
```

The iota step is actually in the “algorithm 4” section, described as “asymmetry”.

```
  mill[1] = bit32.bxor(mill[1],1) -- Iota
end
```

The belt and mill are initialized with zeros, as per the first line of the “algorithm 5” section.

```
local function initBeltMill()
  local belt = {}
  local mill = {}
  for z = 1, 40 do
    belt[z] = 0
  end
  for z = 1, 19 do
    mill[z] = 0
  end
  return belt, mill
end
```

This is the rest of the input mapping as per the “algorithm 5” section of the specification.

```
-- RadioGatun input map; given string return belt, mill
function RG32inputMap(i)
  local belt, mill
  belt, mill = initBeltMill()
  local phase = 0;
  for a = 1, string.len(i) do
    local c = string.byte(i, a)
    local b
    c = bit32.band(c,0xff)
    c = bit32.lshift(c,8 * (phase % 4))
    b = math.floor(phase / 4) % 3
    belt[(13 * b) + 1] = bit32.bxor(belt[(13 * b) + 1],c)
    mill[17 + b] = bit32.bxor(mill[17 + b],c)
    phase = phase + 1
    if phase % 12 == 0 then
      beltMill(belt, mill)
    end
  end
```

The padding byte and blank rounds are as per section 6.2 on page 9 of the specification. While the specification says there are 16 blank rounds, we actually need to run 18 blank rounds to get the same output as the reference test vectors.

```
  -- Padding byte
  local b = math.floor(phase / 4) % 3
  local c = bit32.lshift(1,8 * (phase % 4))
  belt[(13 * b) + 1] = bit32.bxor(belt[(13 * b) + 1],c)
  mill[17 + b] = bit32.bxor(mill[17 + b],c)

  -- Blank rounds
  for z = 1, 18 do
    beltMill(belt,mill)
  end
  return belt, mill
end
```

We can initialize the RadioGatún[32] state now that we have implemented the specification:

```
-- Initialize a RG32 state we can get 32-bit PRNGs from
local function RG32init(i)
  local belt, mill = RG32inputMap(i)
  return {belt = belt, mill = mill, phase = 1}
end
```

Note that the actual test vectors have an endian swap, which we do not perform here to keep the code as simple as possible. The "`phase`" is which word we look at in the mill to get the desired pseudo-random number, as per "algorithm 6", the output mapping, on page 9 of the specification.

```
-- This returns a 32-bit pseudo-random integer from a RG32 state
local function RG32rand32(state)
  if state.phase == 1 then
    state.phase = 2
  elseif state.phase == 2 then
    state.phase = 3
  else
    state.phase = 2
    beltMill(state.belt,state.mill)
  end
  return state.mill[state.phase]
end
```

Now that all of the private ("`local`") functions are defined, we have public functions for the RadioGatún[32] class, starting with the one which initializes a new object using the class.

```
-- Public functions
function rg32:init(seed)
  if not seed then
    seed = tostring(os.time())
  end
  out = {}
  setmetatable(out, self)
  out.state = RG32init(seed)
  return out
end
```

This returns a raw (endian-swapped compared to the reference test vectors) 32-bit number from RadioGatún:

```
function rg32:rand32()
  return RG32rand32(self.state)
end
```

This returns a random number. If run without arguments, we get a floating point number between 0 and just below 1; if run with one argument, we get an integer between 1 and that argument; if run with two arguments, we get an integer between those two numbers. There is a lot of error checking and correction for cases like `high` being less than or equal to than `low` and what not. This function is on the next page:

```
function rg32:random(low, high)
  if not low then
    return RG32rand32(self.state) / 4294967296
  end
  if not high then
    high = low
    low = 1
  end
  if high < low then
    local temp = low
    low = high
    high = temp
  end
  if high == low then
    return low
  end
  return (RG32rand32(self.state) % (high - low)) + low
end

until true -- End block containing private functions
```

Now that we have made a class, complete with private functions, we can use that class:

```
randomState = rg32:init(1)
print(string.format("%x",randomState:rand32()))
print(randomState:random(10))
print(randomState:random(1,7))
```

This will output the following:

```
e586c89
3
6
```

We can implement this in pure Lunacy, which uses the same random number generator:

```
function endianSwap(number)
  return bit32.bor(bit32.lshift(number, 24),
             bit32.band(bit32.lshift(number, 8),0xff0000),
             bit32.band(bit32.rshift(number, 8),0xff00),
             bit32.band(bit32.rshift(number, 24),0xff))
end
rg32.randomseed(1)
print(string.format("%x",endianSwap(rg32.rand32())))
print(endianSwap(rg32.rand32()) % 9 + 1)
print(endianSwap(rg32.rand32()) % 6 + 1)
```

The `rg32` library Lunacy uses to generate these random numbers is available at `https://github.com/samboy/LUAlibs`

Regular expressions

In the chapter on strings, we mentioned regular expressions. Let's look at those in more detail.

To review, here's an example of `string.gsub`:

```
a = "Hello for the world."
b = "Life is good. Life is amazing!"
a = a:gsub("Hello","Love")
b = b:gsub("Life","Love")
print(a,b)
```

At the point, the output is "`Love for the world.`" and "`Love is good. Love is amazing!`".

What if we could simply replace the first word in a given sentence with the word `Love`? That is where *regular expressions* come in; they allow a number of different strings to match a given *pattern*. For example, here's the Lunacy regular expression which means "the first word in a sentence":

```
%.%s*%w+
```

In more detail:

- "`%.`" means "the '`.`' character". This is followed by....
- "`%s*`" means "any number, including zero, of spaces, tabs, or other whitespace characters"
- That is followed by "`%w+`" which means "one or more letters and/or numbers"

So we match against a period, then any amount of space, then a word. In particular:

- "`%.`" means "the literal '`.`' character"
- "`%s`" means "any whitespace character (space, tab, etc.)"
- "`*`" means "zero or more instances of the character (or character type) we just saw"
- "`%w`" means "any letter or number"
- "`+`" means "one or more instances of the previous character (or character type)"

So, let's see this in action:

```
b = "Life is good. Life is amazing!"
b = b:gsub("%.%s*%w+","Love")
print(b)
```

We get this:

```
Life is goodLove is amazing!
```

This is a start but there are some problems (see next page):

- We do not replace the word at the beginning of the string
- We do not preserve the punctuation starting the sentence

So, let's see if we can account for these. Let's look at this regular expression:

```
^%w+
```

This regular expression means "the first word in the string":

- "^" means "the beginning of the string"
- "%w" means "any letter or number"
- "+" means "one or more instances of the previous character (or character type)"

```
b = "Life is good. Life is amazing!"
b = b:gsub("^%w+","Love")
print(b)
```

Gives us this:

```
Love is good. Life is amazing!
```

So now we need another regular expression which lets us capture words at the beginning of sentences, while preserving the punctuation which begins a sentence:

```
(%.%s*)%w+
```

The part of the regular expression between the parenthesis is noted; "%1" in the second argument to gsub is replaced with the part between the parenthesis in the first argument, e.g.

```
b = "Life is good. Life is amazing!"
b = b:gsub("(%.%s*)%w+","%1Love")
print(b)
```

Gets this:

```
Life is good. Love is amazing!
```

It would also be nice if we could have more characters than "." matched, because sentences sometimes end with "?" or "!". Let's look at how we can match multiple characters:

```
[%.%?!]
```

- The left square bracket "[" indicates we begin a set of multiple possible characters and/or character types which the regular expression can match against.
- "%." indicates the literal "." character. We need the "%" character because "." otherwise indicates that any character can match that position in a regular expression.
- "%?" likewise matches against the literal "?" character. We need "%" here because "?" matches against zero or one occurrences of the previous character (or character group) in the regular expression.

- "`!`" does not have any special meaning in Lunacy regular expressions. However, should we forget this when formulating regular expression, "`%!`" matches against a literal "`!`" as does just "`!`".
- While not used in the above regular expression, we use "`%%`" to match against "`%`".
- "`]`" ends the set of possible characters we can match against.

Let's see the above regular expression in action:

```
a = "!@*$.?%^&"
a = a:gsub("[%.%?!]","#")
print(a)
```

This outputs:

```
#@*$##%^&
```

As one can see, the "`.`", "`?`", and "`!`" characters are all replaced with "`#`".

That in mind, let's replace every word at the beginning of sentences with "`Love`". Let's first look at the Lunacy code which does this:

```
a = "Life is good. Someone is nice! What is amazing? Iron is strong."
a = a:gsub("^%w+","Love") -- Make first word on line "Love"
a = a:gsub("([%.%?!]%s*)%w+","%1Love") -- Words at sentence start
print(a)
```

This outputs:

```
Love is good. Love is nice! Love is amazing? Love is strong.
```

The "." character

In regular expressions, the "`.`" character matches against any possible character in the regular expression. For example, the regular expressions "`Sp..k`" matches against the words "`Spook`", "`Speak`", "`Spork`", and "`Spock`", but doesn't match against, say, "`Splook`" or "`Spilt`".

Example code:

```
a = "The Spook Speak Spork about Spock, Splook, and Split"
a = a:gsub("Sp..k","Spunk")
print(a)
```

Will output:

```
The Spunk Spunk Spunk about Spunk, Splook, and Split
```

Since anything in the form of S-p-*any*-*any*-k matches, but other patterns do not match against the "`Sp..k`" regular expression.

If we want to see if a regular expression is in a string, we can use `string.find()`.

File input and output

Let's create a file with the name "`foo.txt`" and put it in the same directory we are running a Lunacy script in:

```
handle = io.open("foo.txt","wb")
if not handle then
  error("Can not open foo.txt for writing")
  os.exit(1)
end
handle:write("Hello, world!\n")
handle:close()
```

This writes a file called `foo.txt` and puts the string "`Hello, world!`" in that file. Should it not be possible to open a file, the above code returns an error.

To read that file:

```
handle = io.open("foo.txt","rb")
if not handle then
  error("Can not open foo.txt for reading")
  os.exit(1)
end
contents = handle:read("*a") -- Read all the file
handle:close()
print(contents)
```

If both are run, one will see "`Hello, world!`" after the second block of code above is run.

Line by line

It's also possible to read a file line by line. For example, let's write a file with a number of lines, then read the file and output it with the line numbers added to the left:

```
handle = io.open("foo.txt","wb")
if not handle then
  error("Can not open foo.txt for writing")
  os.exit(1)
end
handle:write("This\nhas\na\nlot\nof\nlines\nin\nit!")
handle:close()
handle = io.open("foo.txt","rb")
if not handle then
  error("Can not open foo.txt for reading")
  os.exit(1)
end
lineNumber = 1
for line in handle:lines() do
  print(lineNumber .. " " .. line)
  lineNumber = lineNumber + 1
end
handle:close()
```

The code on the bottom of the previous page will output this:

```
1 This
2 has
3 a
4 lot
5 of
6 lines
7 in
8 it!
```

Function iteration

Let's look at this code again:

```
lineNumber = 1
for line in handle:lines() do
  print(lineNumber .. " " .. line)
  lineNumber = lineNumber + 1
end
```

This code could be written like this:

```
lineNumber = 1
iteratorFunction = handle:lines()
line = iteratorFunction()
while line do
  print(lineNumber .. " " .. line)
  lineNumber = lineNumber + 1
  line = iteratorFunction()
end
```

Likewise, we can write our own implementation of `handle:lines()` as follows:

```
function myLines(fileHandle)
  canReadFile = true
  return function()
    if not canReadFile then return nil end
    thisLine = fileHandle:read("*l") -- read one line from file
    if not thisLine then canReadFile = false end
    return thisLine
  end
end
```

This works as before. Given the code on the previous page and the above function:

```
lineNumber = 1
for line in myLines(handle) do
  print(lineNumber .. " " .. line)
  lineNumber = lineNumber + 1
end
```

The read() function

A file handle has, as seen in the previous code, a function called `read`. This function can be called either in the form of `fileHandle:read("*a")` or `fileHandle.read(fileHandle,"*a")`; the two are equivalent. `read()` takes one argument which determines how it reads from a file:

- The string `*a` means to read the rest of the file; e.g. `fileHandle:read("*a")`
- The string `*l` means to read one line from the file; e.g. `fileHandle:read("*l")`
- If the `read()` function is given a number *n*, Lunacy will read up to *n* bytes from the file.

`read()` will return a string containing the contents of what was read from a file, or `nil` if either the file handle is invalid (e.g. unable to open the file) or if it's at the end of the file.

Before showing examples of `read()`, let's write an example file to read:

```
handle = io.open("foo.txt","wb")
if not handle then
  error("Can not open foo.txt for writing")
  os.exit(1)
end
handle:write("This\nhas\na\nlot\nof\nlines\nin\nit!")
handle:close()
```

Given this file, let's read the first line of the file:

```
handle = io.open("foo.txt","rb")
if not handle then
  error("Can not open foo.txt for reading")
  os.exit(1)
end
contents = handle:read("*l") -- Read one line of the file
handle:close()
print(contents)
```

If `handle:read("*l")` was called again before closing the file, we would read the second line of the file. If `handle:read("*l")` was called a third time, we would read the third line, and so on.

We can also read only nine bytes, which are the first two lines of the example file:

```
handle = io.open("foo.txt","rb")
if not handle then
  error("Can not open foo.txt for reading")
  os.exit(1)
end
contents = handle:read(9) -- Read nine bytes from the file
handle:close()
print(contents)
```

Example code reading the entire file, i.e. `handle:read("*a")`, is on a previous page.

Standard input

The standard input is available via the `io.stdin` descriptor, e.g.:

```
print("Type something then press enter")
line = io.stdin:read("*l")
print("You typed " .. line)
```

This will read a line typed in then print back what was just typed in.

Standard output

Standard output is likewise available via the `io.stdout` descriptor. This is useful when a line without a linefeed needs to be output, e.g.:

```
io.stdout:write("Type something here: ")
line = io.stdin:read("*l")
print("You typed " .. line)
```

Like the previous example, this will read a line the user types in and then print it back.

Standard error

Likewise, Lunacy has support for outputting to standard error:

```
io.stdout:write("Standard output\n")
io.stderr:write("Standard error\n")
```

This will output `Standard output` on standard output and `Standard error` on standard error.

Flushing buffered writes

When writing to a file, the writes may be buffered, which means the contents of the file may not be updated in a timely fashion. Writes can be forced to occur at once with the `flush` method, as can be seen in the following example (run `tail -f foo.txt` while running this):

```
handle = io.open("foo.txt","wb")
if not handle then
  error("Can not open foo.txt for writing")
  os.exit(1)
end
for a=1,1000000 do
  for b=1,10000 do
    for c=1,10000 do
      if b * c == 18016129 then
        handle:write(b .. " " .. c .. " ")
        handle:flush()
      end
    end
  end
end
```

Math

Lunacy has a full scientific calculator library allowing it to perform a number of different numerical calculations.

Arithmetic

Lunacy uses `+` for addition and `-` for subtraction:

```
print(1 + 7 - 3)
```

Outputs `5`.

Lunacy uses `*` for multiplication and `/` for division:

```
print(6 / 2 * 3)
```

This outputs 9 (and *not* 1) because multiplication and division have the same *operator preference*: When multiplies and divides are in an expression, they are evaluated left to right.

However, multiplication and division have a higher operator preference than addition and subtraction. For example, this returns `12` and not `24`:

```
print(6 + 2 * 3)
```

That is because we perform all multiplication and division operations before addition and subtraction are done.

Exponents and roots

Exponentiation is done with the `^` operator:

```
print(6 ^ 2)
```

Returns `36`, i.e. six squared.

Roots are performed by fractional exponentiation; e.g. going to the power `1 / 2` (equivalent to `.5`) gives us the square root. Code example:

```
print(16 ^ (1 / 2))
```

Returns `4`, the square root of 16. We need to use parenthesis because exponentiation takes precedence over multiplication, division, as well as addition and subtraction.

Trigonometry

Lunacy has support for trigonometry functions.

For example:

```
print(math.sin(math.pi / 6))
```

Returns `0.5`, i.e. the sine of 30 degrees. Note that `math.sin` assumes the input is in radians (so a circle has `2 * math.pi` radians and 360 degrees, where `math.pi` is approximately `3.14159`); if inputting the angle in degrees is preferred, we can use the `math.rad` function which converts degrees to radians:

```
print(math.sin(math.rad(30))
```

This also returns `0.5`.

`math.cos` returns the cosine.

That in mind, we can write a function where, given a polygon where each side has a length of 1, we list the coordinates of each point in the polygon:

```
function polygonPoints(sides)
  local x = 0
  local y = 0
  local angle = 0
  for a=1,sides do
    print(string.format("%f %f",x,y))
    x = x + math.cos(angle)
    y = y + math.sin(angle)
    angle = angle + math.pi * 2 / sides
  end
end
```

To see where the points are around an equilateral triangle:

```
polygonPoints(3)
```

Around a square:

```
polygonPoints(4)
```

A pentagon:

```
polygonPoints(5)
```

And so on.

The function uses `string.format()` to make the output more attractive.

Logarithms

Lunacy also has support for logarithms. To get the natural logarithm of a number, use `math.log`, for example:

```
print(math.log(10))
```

Which is around 2.3; i.e. it is the exponent of *e* (approximately 2.718) to get 10:

```
print(2.718 ^ 2.3)
```

Which gives us a number ever so slightly lower than 10 — the number is not quite 10 because the value of *e* and the `math.log(10)` values are approximations.

Lunacy has support for base 10 logarithms. For example, `print(math.log10(10))` returns 1, and `print(math.log10(100))` returns 2.

To get the logarithm in another base, we can use this form:

```
function log2(n)
  return math.log(n) / math.log(2)
end
```

Here, `log2(2)` returns `1`, `log2(4)` returns `2`, `log2(8)` returns `3`, and so on.

A table of logarithms

That in mind, we can make a simple ASCII table of logarithms:

```
out = "\t"
-- Header
for a=0,9 do
  out = out .. string.format("%.1f-..-",a / 10)
end
-- Body
for a=1,9 do
  out = out .. string.format("\n%.1f\t",a)
  for b=0,9 do
    out = out .. string.format("%.4f ",math.log10(a + b/10))
  end
end
out = out .. "\n"
print(out)
```

The above code creates a table which can be used to determine the common logarithm for a given number between 1.0 and 9.9 with a precision of 0.1; rows are the first digit of the number and columns are the second digit of the number we want the logarithm for.

MORE ON REGULAR EXPRESSIONS

Lunacy regular expressions do not have have all the features other regular expression libraries, such as PCRE2, have. This is because Lunacy is a tiny language designed to be embedded in other programs or to be used as a very small stand alone scripting language. The PCRE2 shared library is about twice the size of the entire Lunacy binary when both are compiled and stripped using the same toolchain. Doing an apples to apples comparison, the PCRE2 library is 32 times the size of Lunacy's string library (`lstrlib.o` in Lunacy's source code) which implements the Lunacy regular expression engine.

While there *are* library bindings which allow Lua to use PCRE2 for regular expression processing, it's also possible to write some much more compact functions which allow Lunacy to support enough regular expression processing to have it be a reasonable replacement for Perl or another language with a more extensive regular expression library. Indeed, I recently converted a number of Perl scripts which extensively used regular expressions in to Lunacy scripts.

Let's look at some functions which cover some of the gaps Lunacy's regular expression library has compared to a larger regular expression engine.

Case insensitive matching

Here's a function which converts regular expressions in to a case insensitive form:

```
function mixedCaseRegex(pattern)
  local out = ""
  local afterPercent = false
  for a=1,pattern:len() do
    seek = pattern:sub(a,a)
    -- Do not mangle anything right after a %
    if seek == "%" and not afterPercent then
      out = out .. seek
      afterPercent = true
    -- Only letters not after a % get mangled
    elseif seek:find("%a") and not afterPercent then
      out = out .. "[" .. seek:upper() .. seek:lower() .. "]"
    -- Other stuff is copied as is
    elseif seek and not afterPercent then
      out = out .. seek
    -- Everything right after a % is passed as-is
    elseif afterPercent then
      out = out .. seek
      afterPercent = false
    end
  end
  return out
end
```

The reason we have this function is because Lunacy's regular expression library does not have support for case insensitive pattern matching.

To have case insensitive matching, we match against both the upper case and lower case version of a letter.

For example, `string.gsub(someString,"[Jj][Oo][Hh][Nn]","Joe")` will replace all instances of "`John`", regardless of case, in to "`Joe`". Example code:

```
z = string.gsub("John JOHN JoHn john was here", "[Jj][Oo][Hh][Nn]", "Joe")
print(z)
```

This returns "`Joe Joe Joe Joe was here`".

The function on the previous page allows us to avoid the awkward syntax of case insensitive regular expressions. Given that function:

```
z = string.gsub("John JOHN JoHn john was here", mixedCaseRegex("John"), "Joe")
print(z)
```

This also returns "`Joe Joe Joe Joe was here`".

The function `mixedCaseRegex` (defined on the previous page) converts a string in to its case insensitive form, e.g.:

```
print(mixedCaseRegex("John %w+ happy"))
```

returns "`[Jj][Oo][Hh][Nn] %w+ [Hh][Aa][Pp][Pp][Yy]`". The `%w+` pattern, which matches against any word or number, is unaltered, while the letters in the pattern are made case insensitive. Note that `mixedCaseRegex` assumes that the regular expression it receives doesn't have letters in bracketed parts (parts between "`[`" and "`]`").

Regular expression splitting

Lunacy also doesn't have support for splitting up a string by using a regular expression. That in mind, here's another function:

```
function regexSplit(subject, splitOn)
  if not splitOn then splitOn = "," end
  local place = true
  local out = {}
  local mark
  local last = 1
  while place do
    place, mark = string.find(subject, splitOn, last, false)
    if place then
      table.insert(out,string.sub(subject, last, place - 1))
      last = mark + 1
    end
  end
  table.insert(out,string.sub(subject, last, -1))
  return out
end
```

The input to this function is the string we want to split up (`subject`) and the regular expression we will use to split up the string (`splitOn`). The output is a list of substrings that are the `subject` split up via `splitOn`.

For example, if we have the string "`Life; Love; Happiness`" and split on the regular expression "`;%s*`" (or, equivalently, "`%;%s*`"), the `regexSplit` function on the previous page will return, as a list, `{"Life", "Love", "Happiness"}`

For example, if `regexSplit` is defined:

```
t = regexSplit("Life; Love; Happiness",";%s*")
for a = 1, #t do print(t[a]) end
```

This will output, on three lines, "`Life`", "`Love`", then "`Happiness`".

The `%s*` part of the regular expression means "match against zero or more whitespace characters". `regexSplit("Life;Love;  Happiness", ";%s*")` would generate the same output, namely `{"Life", "Love", "Happiness"}`, since the splitting regular expression splits on the "`;`" character optionally followed by any number of whitespace (space, tab, etc.) characters.

An AWK like script

With the `regexSplit` function, we can make a Lunacy script which acts like the AWK script *{print $2 " " $1},* i.e. print the second word then the first word of every input line.

```
line = io.read() -- Default io device is standard input
while line do
  -- Remove leading whitespace
  line = string.gsub(line,"^%s+","")
  -- AWK-style split: fields[1] is $1, etc.
  fields = regexSplit(line,"%s+")
  if fields[2] then -- Avoid raising error
    print(fields[2] .. " " .. fields[1]) -- print $2 " " $1
  else
    print(" " .. fields[1]) -- Act like AWK script
  end
  line = io.read() -- Read next line.
end
```

DIRECTORY TRAVERSAL

Lua 5.1 does not have support for directory traversal out of the box, but there are a number of ways of adding this support. Lunacy adds support for directory traversal via the LuaFileSystem library, which is available for stock Lua 5.1 at `https://github.com/lunarmodules/luafilesystem` or `https://github.com/samboy/LUAlibs` (the second link has a version with a script to compile LuaFileSystem without needing luarocks).

Once LuaFileSystem support is in place (or one simply uses a version of Lunacy compiled with LuaFileSystem support), this script lists all the files in the current working directory:

```
for filename in lfs.dir(".") do
  print(filename)
end
```

Let's see what kind of files and/or folders we're looking at:

```
for filename in lfs.dir(".") do
  local attr = lfs.attributes(filename)
  if attr and attr.mode then
    spaces = ""
    for a=1,17 - attr.mode:len() do
      spaces = spaces .. " "
    end
    print(attr.mode .. spaces .. filename)
  end
end
```

In the resulting output, the first column is the type of file and/or directory we're looking at, and the second column is the filename (or directory name, if you will).

Let's list all files in the current working directory and all sub-folders:

```
function find(path, depth)
  if depth > 20 then return true end
  for filename in lfs.dir(".") do
    local attr = lfs.attributes(filename)
    if attr and filename ~= "." and filename ~= ".." then
      if attr.mode == 'directory' then
        lfs.chdir(filename)
        find(path .. "/" .. filename, depth + 1)
      else
        print(path .. "/" .. filename)
      end
    end
  end
  lfs.chdir("..")
end
find(".",1)
```

TWO-WAY PIPES

While Lua 5.1 does not include support for two-way pipes with sub-processes (i.e. the ability to spawn a child process and both write to the child's standard input and read from the child's standard output), Lunacy has this support via the open source `spawner` library. Lua 5.1 users can add this support by downloading the `spawner` libraries from `https://github.com/samboy/LUAlibs`

These examples assume a UNIX-like environment (e.g. Linux) with standard UNIX commands. Assuming we have the "`wc`" command available:

```
write, read = spawner.popen2("wc")
write:write("Hello, world!\n")
write:flush()
write:close()
print(read:read())
```

This will return something like `1 2 15`, the output of `wc` when given the string "`Hello, world!`" followed by a newline.

We can make a more complicated script which acts like the UNIX pipeline `ls | sort -n | head` as seen in the following example:

```
w1, r1 = spawner.popen2("ls")
w1:close()
lsOutput = ""
line = r1:read()
while line do
  lsOutput = lsOutput .. line .. "\n"
  line = r1:read()
end
w2, r2 = spawner.popen2("sort -n")
w2:write(lsOutput)
w2:close()
sortOutput = ""
line = r2:read()
while line do
  sortOutput = sortOutput .. line .. "\n"
  line = r2:read()
end
w3, r3 = spawner.popen2("head")
w3:write(sortOutput)
w3:close()
line = r3:read()
while line do
  print(line)
  line = r3:read()
end
```

A TEXT ADVENTURE: PART 1

Let's look at a simple text adventure game, based on the classic *Cloak of Darkness*:

```
foyer = {
name = "foyer",
description = "You are standing in a spacious hall, splendidly decorated in"
.. "\nred and gold, with glittering chandeliers overhead. The entrance from"
.. "\nthe street is to the north, and there are doorways south and west.",
n = "You've only just arrived, and besides, the weather outside\n" ..
"seems to be getting worse.", e = "You can not go that way."
}

cloakroom = {
name = "cloakroom",
description = "The walls of this small room were clearly once lined with " ..
"hooks,\nthough now only one remains. The exit is a door to the east.",
e = foyer, s = "You can not go that way.", n = "You can not go that way.",
w = "You can not go that way."
}

bar = {
name = "bar",
description = "The bar, much rougher than you'd have guessed after the "
.. "opulence\nof the foyer to the north, is completely empty. There seems to"
.. "\nbe some sort of message scrawled in the sawdust on the floor.",
n = foyer, e = "You can not go that way.", w = "You can not go that way.",
s = "You can not go that way."}

foyer.w = cloakroom
foyer.s = bar

location = foyer

print("Hurrying through the rain swept November night, you're glad to see the"
.. "\nbright lights of the Opera House. It's surprising that there aren't more"
.. "\npeople about but, hey, what do you expect in a cheap demo game...?\n")

repeat
  print("You are in the " .. location.name .. "\n\n" .. location.description)
  io.stdout:write("Tell me what to do> ")
  command = io.stdin:read("*l")
  if command then
    command = command:sub(1,1)
    command = command:lower()
  end
  if location[command] and type(location[command]) == "string" then
    print(location[command] .. "\n")
  elseif location[command] and type(location[command]) == "table" then
    location = location[command]
  else
    print("I'm sorry but I do not understand your command.\n")
  end
until false
```

This is a very simple text adventure. In particular, we do not have a multi-word parser and the only thing we can do is move around the tiny map this adventure has. A *text adventure* is a type of video game which is now called *interactive fiction*. In interactive fiction games, rooms and objects in the game are described using text, and commands are given to the computer via simple text sentences typed in. Let's see what's it like to play the game on the previous page; here bold text (**`like this`**) is typed in by the player:

```
Hurrying through the rain swept November night, you're glad to see the
bright lights of the Opera House. It's surprising that there aren't more
people about but, hey, what do you expect in a cheap demo game...?

You are in the foyer

You are standing in a spacious hall, splendidly decorated in
red and gold, with glittering chandeliers overhead. The entrance from
the street is to the north, and there are doorways south and west.
Tell me what to do> north
You've only just arrived, and besides, the weather outside
seems to be getting worse.

You are in the foyer

You are standing in a spacious hall, splendidly decorated in
red and gold, with glittering chandeliers overhead. The entrance from
the street is to the north, and there are doorways south and west.
Tell me what to do> w
You are in the cloakroom

The walls of this small room were clearly once lined with hooks,
though now only one remains. The exit is a door to the east.
Tell me what to do> go east
I'm sorry but I do not understand your command.
```

As people familiar with interactive fiction can see, the usual commands to move around, "`north`", "`south`", "`east`", and "`west`" all work, as well as the abbreviated forms "`n`", "`s`", "`e`", and "`w`". However, the form "`go east`" which is standard for interactive fiction parsers to understand, does not work with our code. Let's look at the parser for this very simple engine:

```
command = command:sub(1,1)
command = command:lower()
```

This code takes the very first letter of the command they typed in and converts it in to lower case. So, "`north`" becomes "`n`", "`South`" becomes "`s`", "`go east`" becomes "`g`", and so on. Once we make the command this one-letter abbreviation, we then look at the table for the location the player is in for that one letter command. If found, we see if the entry is a string; if the entry is a string, we output the string. Otherwise, if the entry in the table for the location the player is located is a table, we move the player to that location ("room") and describe the new location the player is in.

The only thing this simple game engine can do is move the player around. A fully implemented interactive fiction game probably needs more.

A TEXT ADVENTURE: PART 2

Let's add some features to the game. Here we will add:

- A two word parser, with the option to make it multi-word later
- Default exits, e.g. we won't need to specify "`You can not go that way`" for each non-exit in the room.
- An object in the player's inventory which we can pick up or drop
- The ability to see the player's inventory and objects in a room

Let's start with the rooms. They are more or less the same as before; each room is a table with attributes (name, description, exits) detailed using table members:

```
foyer = {
name = "foyer",
description = "You are standing in a spacious hall, splendidly decorated in"
.. "\nred and gold, with glittering chandeliers overhead. The entrance from"
.. "\nthe street is to the north, and there are doorways south and west.",
n = "You've only just arrived, and besides, the weather outside\n" ..
"seems to be getting worse.", items = {}
}

cloakroom = {
name = "cloakroom",
description = "The walls of this small room were clearly once lined with " ..
"hooks,\nthough now only one remains. The exit is a door to the east.",
e = foyer, items = {}
}

bar = {
name = "bar",
description = "The bar, much rougher than you'd have guessed after the "
.. "opulence\nof the foyer to the north, is completely empty. There seems to"
.. "\nbe some sort of message scrawled in the sawdust on the floor.",
n = foyer, items = {} }

-- Since the cloakroom and bar are now defined, we can give the foyer exits
foyer.w = cloakroom
foyer.s = bar
```

Observe that the rooms can now have objects ("`items`") in them.

Now that we have three rooms described, let's create an object which can be carried, the cloak of darkness which is this adventure's namesake:

```
cloak = {
name = "A velvet cloak", noun = "clo",
description = "A handsome cloak, of velvet trimmed with satin, and slightly\n"
            .. "spattered with raindrops. Its blackness is so deep that it\n"
            .. "almost seems to suck light from the room."
}
```

The cloak has to be somewhere in the game, and as per the Cloak of Darkness specification and reference implementation, that location is the player's inventory. So let's give the player an inventory and put the cloak in said inventory:

```
inventory = { cloak }
```

Let's make the cloak a simple item we can carry or drop anywhere, ignoring for now that the Cloak of Darkness specification only allows the cloak to be dropped in the cloak room.

As mentioned before, we're going to make the parser multi-word, so let's add a function, described in the "more on regular expressions" chapter, to split words:

```
function regexSplit(subject, splitOn)
  if not splitOn then splitOn = "," end
  local place = true
  local out = {}
  local mark
  local last = 1
  while place do
    place, mark = string.find(subject, splitOn, last, false)
    if place then
      table.insert(out,string.sub(subject, last, place - 1))
      last = mark + 1
    end
  end
  table.insert(out,string.sub(subject, last, -1))
  return out
end
```

Let's have a function which allows common interactive fiction abbreviations to be used by the player, such as "`n`" for "`go north`" or "`x`" for "`examine`".

```
function expand(command)
  if command:find("^%s*n%s*$") then -- Maybe space - "n" - maybe space
    return "go north"
  elseif command:find("^%s*s%s*$") then -- Maybe space - "s" - maybe space
    return "go south"
  elseif command:find("^%s*e%s*$") then -- Maybe space - "e" - maybe space
    return "go east"
  elseif command:find("^%s*w%s*$") then -- Maybe space - "w" - maybe space
    return "go west"
  elseif command:find("^%s*x%s+") then -- "x" short for "examine"
    return command:gsub("^%s*x%s+","examine ")
  end
  return command
end
```

Each verb in the game should have its own function. The only verbs we will have for now are "`go`", "`get`", "`drop`"/"`put`", and "`inventory`".

The first function we have is one which moves the player around from location to location (see next page):

```
-- Go to a location.  Reads and affects the global variable "location"
function go(noun)
  local direction = nil
  if noun == "north" or noun == "south" or noun == "east" or noun == "west" then
    direction = noun:sub(1,1) -- First letter of noun
  else
    print("I do not know how to go " .. noun)
    return nil
  end
  if not location[direction] then
    print("You can not go that way.\n")
    return nil
  elseif type(location[direction]) == "string" then
    print(location[direction] .. "\n")
    return nil
  end
  location = location[direction]
  return true
end
```

Now, let's have another function to get an item in the room:

```
function get(noun)
  subNoun = noun:sub(1,3)
  for item = 1,#location.items do
    if subNoun == location.items[item].noun then
      table.insert(inventory,location.items[item])
      table.remove(location.items,item)
      print("Carried\n")
      return true
    end
  end
  print("I can not see the " .. noun .. " here.")
end
```

The reason the above function has `subNoun` is because we match against the first three letters of the noun they type in, to both have more tolerance for typos and to allow the player to more quickly type in the noun they are looking for. It allows "`clo`" to mean "`cloak`", since the first three letters match.

Like we have a function to pick up items, we also can have a function which drops items:

```
function drop(noun)
  subNoun = noun:sub(1,3)
  for item = 1,#inventory do
    if subNoun == inventory[item].noun then
      table.insert(location.items,inventory[item])
      table.remove(inventory,item)
      print("Dropped\n")
      return true
    end
  end
  print("I am not carrying the " .. noun .. ".")
end
```

We also need two functions to handle management of items. One function shows the items the player is carrying (`seeInventory`) and another function lists what items are in the location where the player is (`viewItems`):

```
-- Print out what the character is carrying
function seeInventory()
  local seen = false
  print("You are carrying: ")
  for counter = 1, #inventory do
    if type(inventory[counter]) == "table" then
      print(inventory[counter].name)
      seen = true
    end
  end
  if not seen then
    print("Nothing")
  end
  print("")
end

-- Print out visible items in the room
function viewItems(place)
  if not place or not place.items or #place.items < 1 then
    return nil -- No items seen
  end
  print("")
  print("You can see: ")
  for counter = 1, #place.items do
    print(place.items[counter].name)
  end
  return true
end
```

With these functions, we can now implement the kernel of the interactive fiction engine. The first thing we will do is show an introduction message, then put the player in the first location (the foyer):

```
print("Hurrying through the rain swept November night, you're glad to see the"
.. "\nbright lights of the Opera House. It's surprising that there aren't more"
.. "\npeople about but, hey, what do you expect in a cheap demo game...?\n")

location = foyer
```

Now that we have done that, let us enter a loop which is the interactive part of the game. In the loop, the first thing we do is describe the room they are in. Then we list any objects which are in the room. We then display a prompt they can reply to, then read a line from the terminal.

Once the command is entered, to make it easier to parse, we make the command lower case, then we expand any common interactive fiction abbreviations (“`n`” for “`go north`”, etc.), remove leading whitespace, then split the command word by word. After that, we perform an action based on the command they typed in. This code is on the next page.

```
repeat
  print("You are in the " .. location.name .. "\n\n" .. location.description)
  viewItems(location)
  io.stdout:write("Tell me what to do> ")
  command = io.stdin:read("*l")
  command = command:lower() -- Case insensitive
  command = expand(command) -- process common interaction fiction abbreviations
  command = command:gsub("^%s+","") -- Remove leading whitespace
  words = regexSplit(command,"%s+")
  if #words == 1 and (words[1] == "i" or words[1]:sub(1,3) == "inv") then
    seeInventory()
  elseif #words < 2 then
    print("Sorry, I can not understand you.\n")
  elseif words[1] == "go" or words[1] == "move" then
    go(words[2])
  elseif words[1] == "get" or words[1] == "carry" then
    get(words[2])
  elseif words[1] == "put" or words[1] == "drop" then
    drop(words[2])
  else
    print("Sorry, I can not understand you.\n")
  end
until false
```

With all of this, it is now possible to move around, as well as drop and pick up one item (the cloak) Commands typed in are bold **`like this`**:

```
Hurrying through the rain swept November night, you're glad to see the
bright lights of the Opera House. It's surprising that there aren't more
people about but, hey, what do you expect in a cheap demo game...?

You are in the foyer

You are standing in a spacious hall, splendidly decorated in
red and gold, with glittering chandeliers overhead. The entrance from
the street is to the north, and there are doorways south and west.
Tell me what to do> i
You are carrying:
A velvet cloak

You are in the foyer

You are standing in a spacious hall, splendidly decorated in
red and gold, with glittering chandeliers overhead. The entrance from
the street is to the north, and there are doorways south and west.
Tell me what to do> drop cloak
Dropped

You are in the foyer

You are standing in a spacious hall, splendidly decorated in
red and gold, with glittering chandeliers overhead. The entrance from
the street is to the north, and there are doorways south and west.
```

Transcript continued on next page

```
You can see:
A velvet cloak
Tell me what to do> get cloak
Carried

You are in the foyer

You are standing in a spacious hall, splendidly decorated in
red and gold, with glittering chandeliers overhead. The entrance from
the street is to the north, and there are doorways south and west.
Tell me what to do> w
You are in the cloakroom

The walls of this small room were clearly once lined with hooks,
though now only one remains. The exit is a door to the east.
Tell me what to do> drop cloak
Dropped

You are in the cloakroom

The walls of this small room were clearly once lined with hooks,
though now only one remains. The exit is a door to the east.

You can see:
A velvet cloak
Tell me what to do> go east
You are in the foyer

You are standing in a spacious hall, splendidly decorated in
red and gold, with glittering chandeliers overhead. The entrance from
the street is to the north, and there are doorways south and west.
Tell me what to do> inventory
You are carrying:
Nothing

You are in the foyer

You are standing in a spacious hall, splendidly decorated in
red and gold, with glittering chandeliers overhead. The entrance from
the street is to the north, and there are doorways south and west.
Tell me what to do> s
You are in the bar

The bar, much rougher than you'd have guessed after the opulence
of the foyer to the north, is completely empty. There seems to
be some sort of message scrawled in the sawdust on the floor.
```

As one can see, the game is a little more fleshed out but there is still more that needs to be done before it's a complete implementation of *Cloak of Darkness*.

A TEXT ADVENTURE: PART 3

Let's flesh out *Cloak of Darkness* and make it a complete game. While this is not a general purpose interactive fiction engine, this provides enough code to provide a framework for making other interactive fiction games. First of all, let's look at the objects in the game:

```
cloak = {
name = "A velvet cloak", noun = "clo",
description = "A handsome cloak, of velvet trimmed with satin, and slightly\n"
            .. "spattered with raindrops. Its blackness is so deep that it\n"
            .. "almost seems to suck light from the room.",
carry = false, -- Because it can only be dropped/picked up in cloakroom
carryWhy = "This isn't the best place to leave a smart cloak laying around"
}
```

The main change here compared to the previous version is that the cloak can not normally be dropped, as per the *Cloak of Darkness* specification.

```
message = {
name = "A message on the floor", noun="mes",
description = "The message, neatly marked in the sawdust, reads\n" ..
  "You have won",
carry = false
}

hook = {
name = "A small brass hook", noun="hoo",
description = "It's just a small brass hook, screwed to the wall.",
carry = false
}
```

We now have two other objects: A message in the bar, and a hook in the cloakroom. Now that the engine has support for objects which can not be carried, we can add the other two objects in *Cloak of Darkness* to our implementation.

```
inventory = { cloak }
```

The cloak, as before, starts off in our personal inventory.

```
foyer = {
name = "foyer",
description = "You are standing in a spacious hall, splendidly decorated in"
.. "\nred and gold, with glittering chandeliers overhead. The entrance from"
.. "\nthe street is to the north, and there are doorways south and west.",
n = "You've only just arrived, and besides, the weather outside\n" ..
"seems to be getting worse.", items = {}
}
```

This is as before, but we can see in the cloak room that we now support running functions when someone goes through a given exit in the room:

```
cloakroom = {
name = "cloakroom",
description = "The walls of this small room were clearly once lined with " ..
"hooks,\nthough now only one remains. The exit is a door to the east.",
e = function()
  location = foyer
  cloak.carry = false
end,
items = { hook }
}
```

We also support darkness in rooms, as seen with the bar:

```
bar = {
name = "bar",
dark = "It is too dark to see!",
light = "The bar, much rougher than you'd have guessed after the "
.. "opulence\nof the foyer to the north, is completely empty. There seems to"
.. "\nbe some sort of message scrawled in the sawdust on the floor.",
isLight = false,
n = foyer, items = { message }
}
```

We now have handlers for custom game processing when an item is dropped or picked up:

```
cloak.onDrop = function()
  bar.isLight = true
  hook.description = "It's just a small brass hook, with a cloak on it."
  return "You put the cloak on the hook."
end
cloak.onGet = function()
  bar.isLight = false
  hook.description = "It's just a small brass hook, screwed to the wall."
end
```

We also can have a function which is run whenever someone does something in a given room:

```
-- We need function closure to track how many times we do stuff in the
-- bar
function makeBarMoveFunction()
  local barMoveCount = 0
  return function()
    if not bar.isLight then
      barMoveCount = barMoveCount + 1
      if barMoveCount < 3 then
        print("In the dark? You could easily disturb something!")
      else
        print("Blundering around in the dark isn't a good idea!")
        message.description = "The message has been carelessly trampled, "
                  .. " making it difficult to read.\n" ..
                  "You can just distinguish the words...\nYou have lost"
      end
    end
```

```
  end
end
bar.onAction = makeBarMoveFunction()
```

Here is what happens:

- We create a function, `makeBarMoveFunction()`, that itself generates and returns a function.
- We use `makeBarMoveFunction()` to generate the function `bar.onAction()`
- Since Lunacy has *function closure*, any variables declared in the `makeBarMoveFunction()` function are remembered by the `bar.onAction()` function; functions in the `makeBarMoveFunction()` namespace are remembered between invocations of `bar.onAction()`
- This allows us to keep track of how many times the player does something in the dark bar, since `barMoveCount` is remembered between `bar.onAction()` invocations. `barMoveCount`, if you will, is like a static variable in C. Indeed, function closure is the only way Lunacy can have non-global variables remembered between invocations of a function.

Once that is done, we then make the path from the foyer to the cloakroom have a custom handler which makes it possible to drop the cloak (since the cloak can only be dropped in the cloakroom).

```
-- Since the cloakroom and bar are now defined, we can give the foyer exits
foyer.w = function()
  location = cloakroom
  cloak.carry = true
end
```

As before, the bar is to the south of the foyer:

```
foyer.s = bar
```

Here is a “battery” utility function for splitting a string on a regular expression, returning an array of sub-strings. This is the same as before:

```
function regexSplit(subject, splitOn)
  if not splitOn then splitOn = "," end
  local place = true
  local out = {}
  local mark
  local last = 1
  while place do
    place, mark = string.find(subject, splitOn, last, false)
    if place then
      table.insert(out,string.sub(subject, last, place - 1))
      last = mark + 1
    end
  end
  table.insert(out,string.sub(subject, last, -1))
  return out
end
```

As before, we have a function which converts common interactive fiction abbreviations in to

their complete form. This code uses a lot of regular expressions; see the chapter on Lunacy's regular expressions for details.

```
function expand(command)
  if command:find("^%s*n%s*$") then -- Maybe space - "n" - maybe space
    return "go north"
  elseif command:find("^%s*s%s*$") then -- Maybe space - "s" - maybe space
    return "go south"
  elseif command:find("^%s*e%s*$") then -- Maybe space - "e" - maybe space
    return "go east"
  elseif command:find("^%s*w%s*$") then -- Maybe space - "w" - maybe space
    return "go west"
  elseif command:find("^%s*x%s+") then -- "x" short for "examine"
    return command:gsub("^%s*x%s+","examine ")
  end
  return command
end
```

The regular expression `^%s*n%s*$` means "The letter `n` by itself, possibly with whitespace either before and/or after the letter `n`". We have similar regular expressions for the letters "`s`", "`e`", and "`w`". Then we have the regular expression `^%s*x%s+` which means "starting at the beginning of the string, we optionally have any amount of whitespace, followed by the letter `x`, followed by one or more whitespace characters". `command:gsub("^%s*x%s+","examine ")`, for example, converts "`x hook`" (or, say, " `x  hook`") into "`examine hook`".

Next, we implement the verbs in the game. "`go`" is almost the same as before, but it now supports running a function when we go from one location to another:

```
-- Go to a location.  Reads and affects the global variable "location"
function go(noun)
  local direction = nil
  if noun == "north" or noun == "south" or
     noun == "east" or noun == "west" then
    direction = noun:sub(1,1) -- First letter, e.g. "north" becomes "n"
  else
    print("I do not know how to go " .. noun)
    return nil
  end
  if not location[direction] then
    print("You can not go that way.\n")
    return nil
  elseif type(location[direction]) == "string" then
    print(location[direction] .. "\n")
    return nil
  elseif type(location[direction]) == "function" then
    return location[direction]()
  elseif type(location[direction]) ~= "table" then -- Error detection
    print("Internal error trying to go " .. noun)
    return nil
  end
  location = location[direction]
  return true
end
```

The "`get`" verb has been expanded. The code now checks to see if the object is one which can be carried, and it also allows a function to be run when we pick up an item. See `cloak.onGet()` to see how a custom `get` function works. Here is the "`get`" code:

```
-- This reads the global "location" and affects the table for
-- the room the player is in, as well as the player inventory
function get(noun)
  if not location.items then -- Error correction
    print("I can not get anything in this location")
    return nil
  end
  subNoun = noun:sub(1,3)
  for item = 1,#location.items do
    if subNoun==location.items[item].noun and location.items[item].carry then
      local thisItem = location.items[item]
      table.insert(inventory,location.items[item])
      table.remove(location.items,item)
      if type(thisItem.onGet) == "function" then
        thisItem.onGet()
      end
      print("Carried\n")
      return true
    elseif subNoun==location.items[item].noun then -- Can not be carried
      if location.items[item].carryWhy then
        print(location.items[item].carryWhy)
      else
        print("That item can not be carried")
      end
      print("")
      return nil
    end
  end
  print("I can not see the " .. noun .. " here.")
end
```

The "`drop`" verb has been expanded in the same ways "`get`" has been expanded, i.e. the code now checks to see if the object is one which can be carried, and it also allows a function to be run when we drop an item. The custom drop function also has the ability to change the message seen when an item is dropped. See `cloak.onDrop` to see how a custom `drop` function works:

```
-- This reads the global "location" and affects the table for
-- the room the player is in, as well as the player inventory
function drop(noun)
  if not location.items then
    print("I can not drop anything in this location")
    return nil
  end
  subNoun = noun:sub(1,3)
  for item = 1,#inventory do
    if subNoun == inventory[item].noun and inventory[item].carry then
      local thisItem = inventory[item]
      table.insert(location.items,inventory[item])
      table.remove(inventory,item)
```

```
      message = "Dropped"
      if type(thisItem.onDrop) == "function" then
        message = thisItem.onDrop()
        if type(message) ~= "string" then
          message = "Dropped"
        end
      end
      print(message .. "\n")
      return true
    elseif subNoun == inventory[item].noun then -- Can not be dropped
      if inventory[item].carryWhy then
        print(inventory[item].carryWhy)
      else
        print("That item can not be dropped")
      end
      print("")
      return nil
    end
  end
  print("I am not carrying the " .. noun .. ".")
end
```

The functions for viewing a player's inventory and listing items in a room are as before:

```
-- Print out what the character is carrying
function seeInventory()
  local seen = false
  print("You are carrying: ")
  for counter = 1, #inventory do
    if type(inventory[counter]) == "table" then
      print(inventory[counter].name)
      seen = true
    end
  end
  if not seen then
    print("Nothing")
  end
  print("")
end

-- Print out visible items in the room
function viewItems(place)
  if not place or not place.items or #place.items < 1 then
    return nil -- No items seen
  end
  print("")
  print("You can see: ")
  for counter = 1, #place.items do
    print(place.items[counter].name)
  end
  return true
end
```

We have added the ability to examine an item:

```
-- Examine an item
function examine(noun)
  local fullNoun = noun
  noun = noun:sub(1,3) -- We match on first three letters
  -- Look in the player's inventory for the item
  for counter = 1, #inventory do
    if inventory[counter].noun == noun then
      print(inventory[counter].description .. "\n")
      return true
    end
  end
  -- Look in the room for the item
  for counter = 1, #location.items do
    if location.items[counter].noun == noun then
      print(location.items[counter].description .. "\n")
      return true
    end
  end
  print("I can not see the " .. fullNoun .. "\n")
  return false
end
```

The code which describes the room is now in a separate function because we support having darkness in rooms:

```
-- Describe a room
function describeRoom(place)
  if place.description then
    print("You are in the " .. place.name .. "\n\n" .. place.description)
    viewItems(location)
  elseif place.isLight then
    print("You are in the " .. place.name .. "\n\n" .. place.light)
    viewItems(location)
  elseif place.dark then
    print(place.dark)
  else -- Error detection and handling
    print("This location has no description.")
  end
end
```

The kernel has been expanded to separate the input sentence in to verbs, nouns, and preposition phrases.

```
print("Hurrying through the rain swept November night, you're glad to see the"
.. "\nbright lights of the Opera House. It's surprising that there aren't more"
.. "\npeople about but, hey, what do you expect in a cheap demo game...?\n")

location = foyer

repeat
```

The first thing we do is describe the room the player is located in:

```
  describeRoom(location)
```

We now optionally run a function if someone stays in a room and performs any action in that room. This is used in the bar when it is dark. Here is that code:

```
if lastLocation == location and type(location.onAction) == "function" then
  location.onAction()
end
lastLocation = location
```

We now prompt the user to type in something.

```
io.stdout:write("Tell me what to do> ")
command = io.stdin:read("*l")
```

Now that they typed in a command, we process it to make their command easier to parse.

```
command = command:lower() -- Case insensitive
command = expand(command) -- process common interaction fiction abbreviations
command = command:gsub("^%s+","") -- Remove leading whitespace
```

We then split up the command they typed in in to words:

```
words = regexSplit(command,"%s+")
```

Now that we have separate words, let's see if there are any prepositional phrases in the command they typed in. Right now, while we note any prepositional phrase typed in, we simply discard the preposition as if they didn't type it; e.g. "`put cloak on hook`" is processed as if they simply typed in "`put cloak`".

```
-- Look for prepositional phrase
local preposition = nil
local prepObject = nil
for counter = 1, #words - 1 do
  if words[counter] == "on" then -- The only preposition this game has
    preposition = words[counter]
    table.remove(words,counter)
    prepObject = words[counter]
    table.remove(words,counter)
```

We handle phrasal verbs like "`look at`" as if they simply did not type in "`at`", e.g. "`look at hook`" is processed so it simply becomes "`look hook`".

```
  elseif words[counter] == "at" then -- Allow "look at" to be "look"
    table.remove(words,counter)
  end
end
```

Now that we have handled phrasal verbs and prepositional phrases, we grab the verb and noun from the command they typed in.

```
-- Now, grab the verb and noun
local verb = nil
```

```
  local noun = nil
  if #words >= 2 then
    verb = words[1]
    verb = verb:sub(1,3)
    noun = words[2]
```

The only intransitive verb (verb without a noun object) we support is "`inventory`"; we have a preprocessor which converts commands like "`n`" into "`go north`".

```
  elseif #words == 1 and (words[1] == "i" or words[1]:sub(1,3) == "inv") then
    seeInventory()
  elseif #words < 2 then
    print("Sorry, I can not understand you.\n")
  end
```

We now process the verbs by calling the appropriate function based on the verb they typed in.

```
  if verb == "go" or verb == "mov" then -- mov: Move
    go(noun)
  elseif verb == "get" or verb == "car" then -- car: Carry
    get(noun)
  elseif verb == "put" or verb == "dro" then -- dro: drop
    drop(noun)
  elseif verb=="loo" or verb=="exa" or verb=="rea" then -- look/examine/read
    examine(noun)
  elseif verb then
    print("Sorry, I can not understand you.\n")
  end
```

We continue to read and process commands until the user quits the Lunacy process running this text adventure game.

```
until false
```