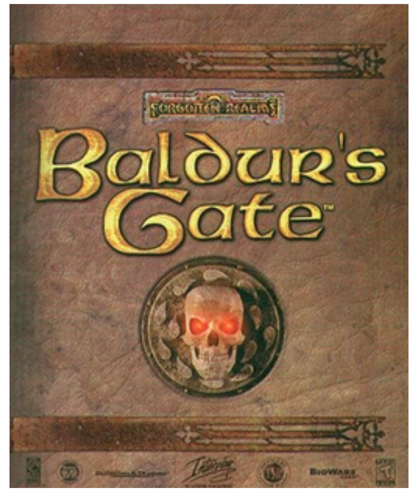
CS448h: Introduction to Lua

Zach DeVito

Dynamically-typed language similar to Javascript:

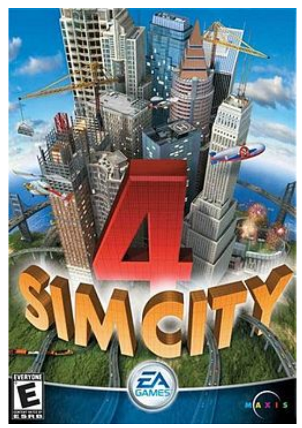
- Garbage-collected
- Objects are really just tables

Activate Your Web Pages
6th Edition
JavaScript
The Definitive Guide
David Flanagan
Unearthing the Excellence in JavaScript
JavaScript:
The Good Parts
O'REILLY
YAHOO!. PRESS
Douglas Crockfo



Baldur's Gate

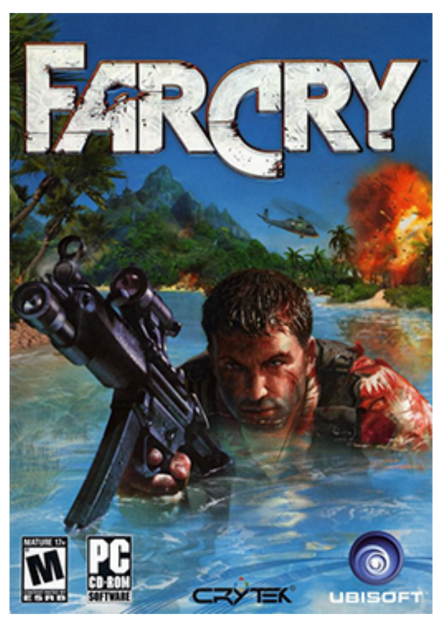
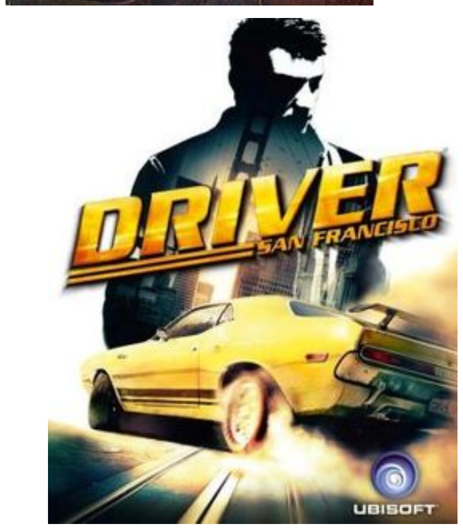
SID MEIER'S
CIVILIZATION V
FIRAXIS
2K GAMES

GRIM FANDANGO
An EPIC Tale of CRIME and CORRUPTION in the LAND OF THE DEAD

ROCKSTAR GAMES PRESENTS
L.A. NOIRE

SIMCITY 4

WORLD OF WARCRAFT
BLIZZARD

FABLE II

Games for Windows
PC CD
COMPANY of HEROES
relic
THQ

FARCRY
PC CD-ROM
CRYTEK
UBISOFT

DRIVER
SAN FRANCISCO
UBISOFT

Goals

Introduce Lua for use in assignments

Demonstrate useful patterns for building DSLs with it

Next Lecture: show how Terra extends Lua with C-level language for generating code

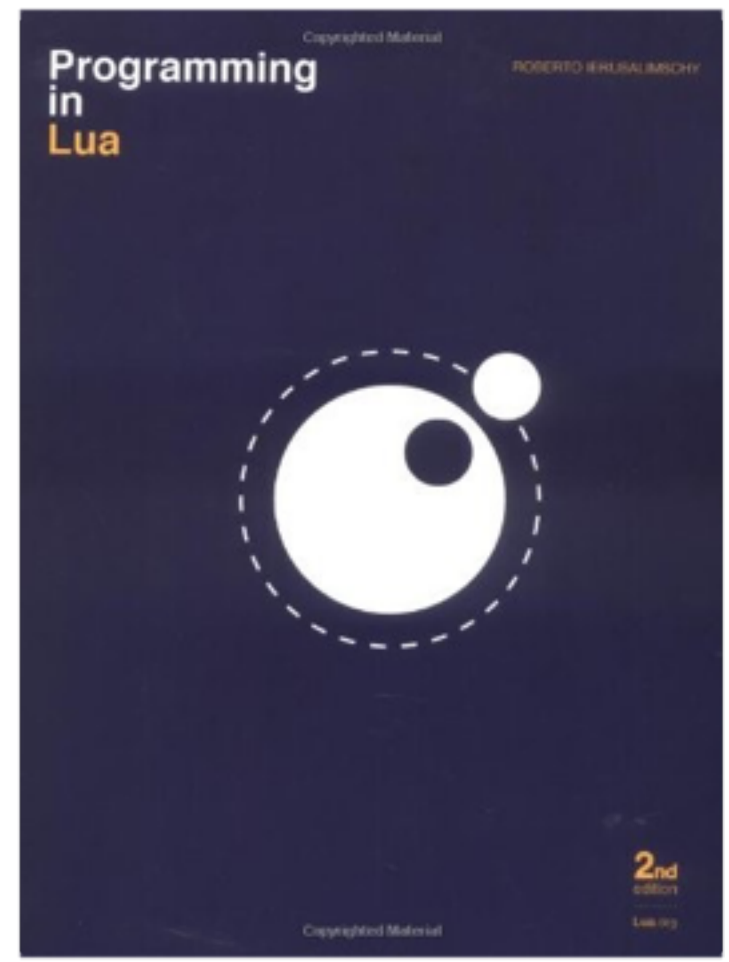
Learning Lua

We will use LuaJIT, which is based on Lua 5.1

- luajit.org/download.html

Additional Resources

- Lua 5.1 reference manual (short and useful)
  www.lua.org/manual/5.1/
- Programming in Lua, Second Edition



Hello, World

hello.lua:

```
print(“hello, world”)
```

```
$ luajit hello.lua
> hello, world
```

Lua’s Design Philosophy

Simplicity

- Lua is a small language
- 51 page reference manual (C++ is 1359)
- “Batteries not included”

Extensibility

- Designed to flexible enough to build your own “batteries”
- “Mechanisms, not policies”
- Embeddable in large systems like game engines

Chunks

Top-level block of code is a **chunk**.

```
print(“hello, world”) -- this is a comment
a = “hello, world”
print(a)
```

Semi-colons are not needed,

Lua syntax makes it possible to know when a statement is finished.

```
a = “hello, world”  print(a)
a = “hello, world”; print(a)
```

But if you want you can put them in for readability

The Interpreter

You can use Lua interactively as well:

```
$ luajit
LuaJIT 2.0.4 -- Copyright (C) 2005-2015 Mike Pall. http://luajit.org/
JIT: ON CMOV SSE2 SSE3 SSE4.1 fold cse dce fwd dse narrow loop abc sink fuse
> print("hello, world")
hello, world
> a = "hello, world"
> = a -- if you want the interpreter to just print a value, prefix with =
```

Types

Types are always dynamically assigned:

```
a = 4
a = “hi” -- ok
```

The function type retrieves the dynamic type of an expression as a string:

```
type(4*3) == “number”
type(true) == “boolean”
type(“hi”) == “string”
type(nil) == “nil”
type(print) == “function”
type(type(X)) == “string”
```

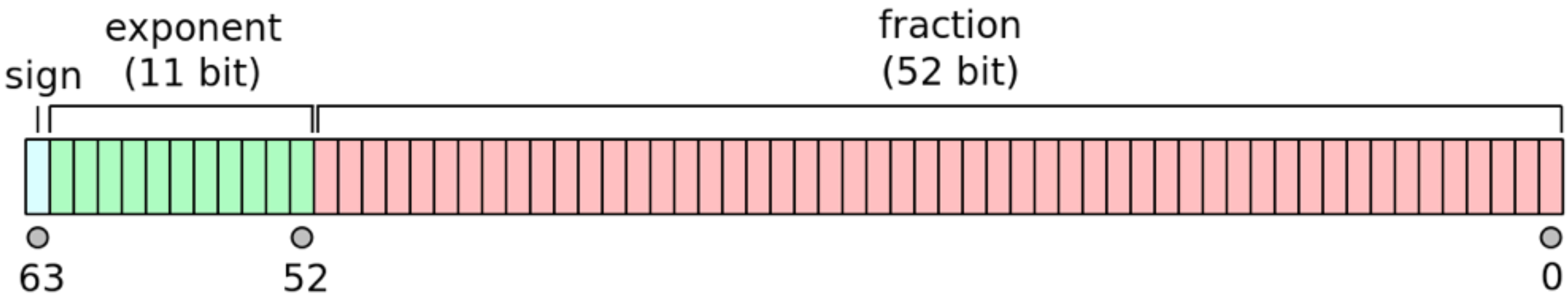
Numbers

`3*4.5+1.3e-4 -- normal math expressions`

All numbers are double precision floating point values. (I told you it was simple!)

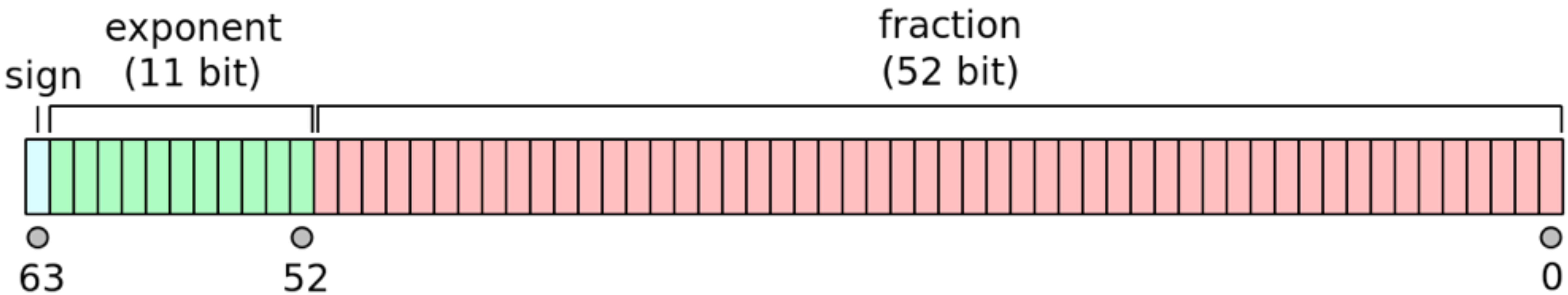
Question How big can an integer get before you can use a floating point number to represent it?

Doubles can store integer values accurately to 53 bits



$$(-1)^{\text{sign}}(1.b_{51}b_{50}...b_0)_2 \times 2^{e-1023}$$

Doubles can store integer values accurately to 53 bits



$$(-1)^{\text{sign}}(1.b_{51}b_{50}\ldots b_0)_2 \times 2^{e-1023}$$

1. Take any 53 bit number or smaller, write out its bits:

```
101010101010111110
```

2. Put in a binary point (up to 52 bits can follow it):

```
1.01010101010111110
```

3. Adjust exponent as appropriate:

```
1.01010101010111110 * 2^17
```

Booleans

```
a < b and c > d -- boolean expression, short circuits
```

All values are considered "true" except for nil and false

```
a = a or 5 -- give 'a' the value 5 unless it already has a value
```

Strings

```
a = “a short string”

b = ‘another short string’

c = [[ a
“long” string
]]

d = [=[ a
long string that contains
a ]]
]=]

e = a .. b -- string concat
f = string.sub(“ab”,2) -- “b”
```

Generally no implicit conversions to/from strings

Nil

Represents the absence of a value.

```
print(an_undefined_variable)
```

```
> nil
```

Not “first-class”: they cannot be keys in a hash-table, and they cannot be put in arrays of things.

Functions

Functions are first class. They can be stored in variables and other data structures, defined inside other functions, etc.

```
function add(a,b)
  return a + b
end
```

```
-- desugared:
add = function(a,b) return a + b end
```

```
function addsub(a,b)
  return a + b, a - b -- multiple returns
end
local added,subbed = adsub(3,4)
```

Control Flow

is pretty standard:

```
if a < b then
  a = a + 1
end
```

```
for i = 1,100 do --inclusive
  print(i)
end
```

```
while a < b do
  a = a + 1
end
```

```
repeat
  a = a + 1
until a >= b
```

Control Flow

is pretty standard:

```
if a < b then
  a = a + 1
end
```

```
while a < b do
  a = a + 1
end
```

```
repeat
  a = a + 1
until a >= b
```

```
for i = 1,100 do --inclusive
  print(i)
end
```

Lua is 1-indexed, the original justification was because non-programmers might use it for configuration.

It’s probably not the best choice, but it does not really get in the way a lot.

Local Variables, Lexical Scope

```
local c = 3
function add(a,b)
  return a + b + c + d
end

_G["d"] = 4 -- equivalently: d = 4
add(1,2) -- 1 + 2 + 3 + 4
```

Local keyword can appear whenever a variable is introduced:

```
local function add(a,b)
  return a + b
end
```

Use local variables everywhere, including at the top-level in chunks. Only use global variables when you explicitly want to look something up from the global table.

Closures

Lexical scoping allows functions to be nested in other functions:

```
local function readfile()
  local count = 0
  local function next()
    count = count + 1
    return count
  end
  repeat
    local v = next()
    print(v)
  until v > 100
end
```

Pattern of having many inner functions is used frequently in compiler transformations to decompose large transformations into smaller components.

Tables

```
type(4*3) == “number”
type(true) == “boolean”
type(“hi”) == “string”
type(nil) == “nil”
type(print) == “function”
type(type(X)) == “string”
type( { r = .5, b = .25, g = .25 } ) == “table”
```

Tables are associative arrays (hash tables) that are used in Lua to represent most data-structures:

- arrays
- maps
- user-defined abstract data-types

Tables

```
local t = {} -- a new blank table
t[1] = “one”
t[2] = “two”

print(t[2]) -- “two”
print(t[3]) -- nil

t[“three”] = 3
local three = t[“three”]
```

When the table key is a string, syntax sugar applies:

```
print(t.three) -- “three”
t.four = 4
print(t[“four”]) -- “four”
```

Tables are objects (like objects in Java)

```
local a = { value = 4 }
local b = { value = 4 }
local c = a
assert(a ~= b and a == c) -- equality is referential
```

Tables as Arrays

Lua internally implements Tables so that continuous integer indexes will be stored efficiently as an array:

```
local a = {“one”,”two”, “three”}
-- same as:
local a = {}; a[1] = “one”; a[2] = “two”; a[3] = “three”
-- same as:
local a = { [2] = “two”, [1] = “one”, [3] = “three” }

for i,v in ipairs(a) do -- generic for loop
  -- ipairs returns an iterator for integer keys
  -- use ‘pairs’ for all keys
  print(string.format(“index %d has value %s”,i,v))
end

assert(#a == 3) -- # is the size of table (max integer entry)
                -- and length of a string
```

Image Processing in Lua

Reading a PPM file

Format:

```
P6
640 480
255
<rgb bytes>
```

```
local function loadppm(filename)
    local F = assert(io.open(filename,"rb"),"file not found")
    local cur
    local function next()
        cur = F:read(1)
    end
    next()
    local function isspace()
        return cur and (cur:match("%s") or cur == "#")
    end
    local function isdigit()
        return cur and cur:match("%d")
    end
    ...
```

Note how even a simple function like loadppm can use many internal functions. These do not have meaning globally since the current token ‘cur’ is part of the local state.

```
local function loadppm(filename)
    local F = assert(io.open(filename,"rb"),"file not found")
    local cur
    local function next()
        cur = F:read(1)
    end
    next()
    local function isspace()
        return cur and (cur:match("%s") or cur == "#")
    end
    local function isdigit()
        return cur and cur:match("%d")
    end
    ...
```

common error checking pattern

Note how even a simple function like loadppm can use many internal functions. These do not have meaning globally since the current token ‘cur’ is part of the local state.

```lua
local function loadppm(filename)
    local F = assert(io.open(filename,"rb"),"file not found")
    local cur
    local function next()
        cur = F:read(1)
    end
    next()
    local function isspace()
        return cur and (cur:match("%s") or cur == "#")
    end
    local function isdigit()
        return cur and cur:match("%d")
    end
    ...
```

common error checking pattern

lexer-style interface, advance
a token stream

Note how even a simple function like loadppm can use
many internal functions. These do not have meaning globally since
the current token 'cur' is part of the local state.

```
local function loadppm(filename)
    local F = assert(io.open(filename,"rb"),"file not found")
    local cur
    local function next()
        cur = F:read(1)
    end
    next()
    local function isspace()
        return cur and (cur:match("%s") or cur == "#")
    end
    local function isdigit()
        return cur and cur:match("%d")
    end
    ...
```

common error checking pattern

lexer-style interface, advance a token stream

return false if EOF

Note how even a simple function like loadppm can use many internal functions. These do not have meaning globally since the current token ‘cur’ is part of the local state.

```lua
local function loadppm(filename)
    local F = assert(io.open(filename,"rb"),"file not found")
    local cur
    local function next()
        cur = F:read(1)
    end
    next()
    local function isspace()
        return cur and (cur:match("%s") or cur == "#")
    end
    local function isdigit()
        return cur and cur:match("%d")
    end
    ...
```

common error checking pattern

lexer-style interface, advance a token stream

method call

return false if EOF

Note how even a simple function like loadppm can use many internal functions. These do not have meaning globally since the current token 'cur' is part of the local state.

```
...
local function parseWhitespace()
    assert(isspace(), "expected at least one whitespace character")
    while isspace() do
        if cur == "#" then -- handle comments
            repeat
                next()
            until cur == "\n"
        end
        next()
    end
end
local function parseInteger()
    assert(isdigit(), "expected a number")
    local n = ""
    while isdigit() do
        n = n .. cur
        next()
    end
    return assert(tonumber(n),"not a number?")
end
...
```

```
...
-- magic numbers
assert(cur == "P", "wrong magic number")
next()
assert(cur == "6", "wrong magic number")
next()

-- image dimensions
local image = {}
parseWhitespace()
image.width = parseInteger()
parseWhitespace()
image.height = parseInteger()
parseWhitespace()
image.precision = parseInteger()
assert(image.precision > 0 and image.precision < 2^16)
assert(isspace(), "expected whitespace after precision")
next()
...
```

```
...
-- helpers for reading the data
local function parseNumber()
     assert(cur ~= nil, "early EOF")
     local n = string.byte(cur)
     next()
     if image.precision >= 256 then --handle higher dynamic range
         n = n * 256
         n = n + string.byte(cur)
         next()
     end
     return n
 end
 -- turn raw numbers in to RGB triple
 local function parseRGB()
     return { r = parseNumber(), g = parseNumber(), b = parseNumber() }
 end
 ...
```

```
    ...
    -- read the image data
    image.data = {}
    for i = 0,image.width*image.height - 1 do
        image.data[i] = parseRGB()
    end
    assert(cur == nil, "expected EOF")
    return image
end
-- all the helper functions go out of scope
```

```lua
local headerpattern = [[
P6
%d %d
%d
]]
local function saveppm(image,filename)
    local F = assert(io.open(filename,"w"),
                    "file could not be opened for writing")
    F:write(string.format(headerpattern,
                          image.width, image.height, image.precision))
    local function writeNumber(v)
        if image.precision >= 256 then
            F:write(string.char(v/256),string.char(v % 256))
        else
            F:write(string.char(v))
        end
    end
    for i = 0, image.width*image.height - 1 do
        local p = image.data[i]
        writeNumber(p.r)
        writeNumber(p.g)
        writeNumber(p.b)
    end
    F:close()
end
```

* Parsing things is generally harder than producing them since when writing things you can choose a subset to work with.

Objects in Lua

So far we have used tables as containers. Lua also uses them as abstract data types.

Note: batteries not included. it is up to you to provide a "class" system if you want operations like inheritance.

```
object:method(argument)
```

is just syntax sugar for

```
object.method(object,argument)
```

Images as Objects

```
local function saveppm(image,filename)
   ...
end

local image = loadppm("foo.ppm")
image.save = saveppm
image:save("foo2.ppm")
```

For defining "methods" Lua also provides syntax sugar:

```
function image:save(filename)
  return saveppm(self,filename)
end
```

becomes

```
image.save = function(self,argument)
  return saveppm(self,filename)
end
```

Image Operators

```
function image:add(rhs)
    local result = { width = self.width,
                     height = self.height,
                     precision = self.precision,
                     data = {} }
    assert(self.width == rhs.width and
           self.height == rhs.height, "images different size")
    for i = 0, self.width * rhs.height - 1 do
        local l,r = self.data[i],rhs.data[i]
        result.data[i] = { r = l.r + r.r, g = l.g + r.g, b = l.b + r.b }
    end
    return result
end
function ConstantImage(width,height,const)
    local result = { width = self.width,
                     height = self.height,
                     precision = self.precision,
                     data = {} }
    for i = 0, result.width * result.height - 1 do
        result.data[i] = { r = const, g = const, b = const }
    end
    return result
end
```

Image Operators

```
function image:add(rhs)
    local result = { width = self.width,
                     height = self.height,
                     precision = self.precision,
                     data = {} }
    assert(self.width == rhs.width and
           self.height == rhs.height, "images different size")
    for i = 0, self.width * rhs.height - 1 do
        local l,r = self.data[i],rhs.data[i]
        result.data[i] = { r = l.r + r.r, g = l.g + r.g, b = l.b + r.b }
    end
    return result
end
function ConstantImage(width,height,const)
    local result = { width = self.width,
                     height = self.height,
                     precision = self.precision,
                     data = {} }
    for i = 0, result.width * result.height - 1 do
        result.data[i] = { r = const, g = const, b = const }
    end
    return result
end
```

Problem: we need to add all the methods for the image to this new image as well

Meta-methods

Behavior of tables can be overridden by a special table known as a “meta-table.”

- prototype style inheritance
- operator overloading

```
local imageprototype = {}
function imageprototype:add() ... end
...

local imagemetatable = { __index = imageprototype }


function ConstantImage(width,height,const)
    local result = { width = self.width,
                     height = self.height,
                     precision = self.precision,
                     data = {} }
    return setmetatable(result,imagemetatable)
end
```

Meta-methods

Behavior of tables can be overridden by a special table known as a “meta-table.”

- prototype style inheritance
- operator overloading

```
local imageprototype = {}
function imageprototype:add() ... end
...

local imagemetatable = { __index = imageprototype }


function ConstantImage(width,height,const)
    local result = { width = self.width,
                     height = self.height,
                     precision = self.precision,
                     data = {} }
    return setmetatable(result,imagemetatable)
end
```

If table does not have a key, look up the key in this table instead

```
local imageprototype = {}
function imageprototype:add() ... end
...

local imagemetatable = { __index = imageprototype }

function imageprototype:__add(rhs)
    local result = { width = self.width,
                     height = self.height,
                     precision = self.precision,
                     data = {} }
    assert(self.width == rhs.width and
           self.height == rhs.height, "images different size")
    for i = 0, self.width * rhs.height - 1 do
        local l,r = self.data[i],rhs.data[i]
        result.data[i] = { r = l.r + r.r, g = l.g + r.g, b = l.b + r.b }
    end
    return setmetatable(result,imagemetatable)
end

-- now images can be added:
local a,b = loadppm("a.ppm"),loadppm("b.ppm")
local c = a + b
-- with other operators defined:
local d = .4*a + .6*b
```

A simple pattern for objects

```
local image = {} -- the image metatable
-- we will also use it as the prototype for images
image.__index = image

function image.isinstance(x) return getmetatable(x) == image end

-- define methods
function image:save(filename) ... end
-- define operators
function image:__add(rhs) ... end

---------
-- we can make the initial setup a function itself:
function newclass()
  local metatable = {}
  metatable.__index = metatable
  function metatable.new(tbl) return setmetatable(metatable,tbl) end
  function metatable.isinstance(x)
      return getmetatable(x) == metatable
  end
  return metatable
end
```

Some More Methods for our Image object

```
-- Support constant numbers as images
function toimage(w,h,x)
  if image.isinstance(x) then
    return x
  elseif type(x) == "number" then
    return ConstantImage(w,h,const)
  else .. other possible conversions
end
```

Modify things like __add to call toimage first, lifting numbers into the image language.

Some More Methods for our Image object

```
-- generate an image that translates the pixels in the new image
function image:shift(sx,sy)
    local result = { width = self.width,
                     height = self.height,
                     precision = self.precision,
                     data = {} }
    for x = 0,width-1 do
        for y = 0,height-1 do
            local fx,fy = x - sx,y - sy
            local p = { r = 0, g = 0, b = 0 }
            if fx >= 0 and fx < width and fy >= 0 and fy < height then
                p = self.data[fy*width+fx]
            end
            result.data[y*width+x] = p
        end
    end
    return result
end
```

Image processing

Do an Image Blur

```
local r =   (a + a:shift(-1,0)
               + a:shift(0,1)
               + a:shift(0,-1)
               + a:shift(1,0)) / 5.0
```

Our image language is slow!

Our Lua implementation: 0.27 MP/s

Naive C loop doing the same thing: 48.2 MP/s

Why?

Our image language is slow!

Our Lua implementation: 0.27 MP/s

Naive C loop doing the same thing: 48.2 MP/s

Why?

- Our storage of the image is inefficient Lua data structures and operations
- We are doing individual operations on the entire image, the C code just does it in one pass

Next Time: How do we fix this?

Bonus: Loading and organizing Lua code

Bonus: Debugging Tips for Lua