The Lua Programming Language

By: Myles Duah, Alexander Leo, Daniel Lopez, James Vannicola and Kevin Zheng

An Overview of Lua:

- Lua is an open-source, high-level, multi-paradigm, scripting language designed primarily for embedded systems and applications.
- It is used as an extension language, a configuration language, and a general-purpose scripting language. With a small footprint and a fast execution time, Lua is a popular choice for many developers.

What is Lua?

- Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description.
- Lua is designed to be flexible, fast, and easy to learn. It has a simple syntax, making it an ideal choice for beginners.

How does Lua compare to other programming languages?

- Lua is a powerful, efficient, and lightweight scripting language. It has a much smaller footprint than other popular scripting languages, such as Python, JavaScript, or PHP.
- It is also much faster than these languages, making it ideal for applications that require quick execution. Lua is also embeddable and easy to learn, making it a great choice for beginner developers.

How Lua is being used today

- Lua is used in a wide range of applications, such as game development, web development, mobile applications, and embedded systems.
- It is also used in many popular video games, such as World of Warcraft, Roblox, Garry's Mod, and Counter-Strike. Lua is also used in the popular game engine, Unity.
- In addition, Lua is used to create applications for the Internet of Things (IoT) and artificial intelligence (AI). Moving forward, Lua is expected to be used in more advanced applications, such as autonomic computing and machine learning.

An Overview of Lua: Cont.

- Lua is a powerful and versatile scripting language that has been used in a wide range of applications.
- It is fully extensible and can be embedded into other applications.
- Lua is similar to other popular scripting languages, but it is designed to be easier to learn and use, and it has a simpler syntax and a larger standard library.
- The versatility and power of Lua make it an ideal choice for a wide range of modern applications.

History

- Lua was created in 1993 by Roberto Ierusalimschy, Luiz Henrique de Figueiredo and Waldemar Celes at the the Pontifical Catholic University Rio de Janeiro, known as PUC-Rio, in Brazil.

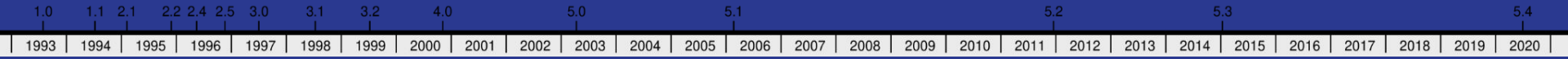
- From the very beginning, Lua was intended to be simple, small, portable, fast, and easily embedded into application.

- Lua has been a continuously evolving language since its initial release and continues to adapt to the communities needs

	1.0	1.1	2.1	2.2	2.4	2.5	3.0	3.1	3.2	4.0	5.0	5.1
constructors	●	●	●	●	●	●	●	●	●	●	●	●
garbage collection	●	●	●	●	●	●	●	●	●	●	●	●
extensible semantics	○	○	●	●	●	●	●	●	●	●	●	●
support for OOP	○	○	●	●	●	●	●	●	●	●	●	●
long strings	○	○	○	●	●	●	●	●	●	●	●	●
debug API	○	○	○	●	●	●	●	●	●	●	●	●
external compiler	○	○	○	○	●	●	●	●	●	●	●	●
vararg functions	○	○	○	○	○	●	●	●	●	●	●	●
pattern matching	○	○	○	○	○	●	●	●	●	●	●	●
conditional compilation	○	○	○	○	○	○	●	●	●	○	○	○
anonymous functions, closures	○	○	○	○	○	○	○	●	●	●	●	●
debug library	○	○	○	○	○	○	○	○	●	●	●	●
multi-state API	○	○	○	○	○	○	○	○	○	●	●	●
`for` statement	○	○	○	○	○	○	○	○	○	●	●	●
long comments	○	○	○	○	○	○	○	○	○	○	●	●
full lexical scoping	○	○	○	○	○	○	○	○	○	○	●	●
booleans	○	○	○	○	○	○	○	○	○	○	●	●
coroutines	○	○	○	○	○	○	○	○	○	○	●	●
incremental garbage collection	○	○	○	○	○	○	○	○	○	○	○	●
module system	○	○	○	○	○	○	○	○	○	○	○	●

	1.0	1.1	2.1	2.2	2.4	2.5	3.0	3.1	3.2	4.0	5.0	5.1
libraries	4	4	4	4	4	4	4	4	5	6	8	9
built-in functions	5	7	11	11	13	14	25	27	35	0	0	0
API functions	30	30	30	30	32	32	33	47	41	60	76	79
vm type (stack × register)	S	S	S	S	S	S	S	S	S	S	R	R
vm instructions	64	65	69	67	67	68	69	128	64	49	35	38
keywords	16	16	16	16	16	16	16	16	16	18	21	21
other tokens	21	21	23	23	23	23	24	25	25	25	24	26

Table 1. The evolution of features in Lua



LUA 1

- Lua 1.0 was fully operational on July 28th, 1993. Lua 1.0 contained 5 built-in functions, 4 libraries, constructors and garbage collection.
- The initial version of of lua was never made available to the public, instead it first major saw use within the Tecgraf department.
- Version 1.1 released July 8th, 1994. This was the first release of Lua that was publicly available. Lua 1.1 functionally is the same as 1.0, the key differences are that 1.1 has 2 additional built-in functions and a further optimized compiler 2x faster than the 1.0 compiler.

LUA 2

- The first iteration of Lua 2 February 7th, 1995 with version 2.1. This version brought support for Object-Oriented Programming and extensible semantics.

- During the development of Lua 2.1, There was high demand for object oriented programming features for Lua. It was denied initially as the developers did not want to make Lua an OOP language as it would no longer be a multi-paradigm language.

LUA 2

- In order to allow for OOP features into Lua without making it a fixed paradigm language, extensible semantics were introduced that allow the user to create whatever model was necessary for a given application.

- Version 2.2 released on November 28th, 1995. It introduced new features to Lua such as a debug API, long strings, extended syntax for function definition, and improved stack tracebacks.

LUA 2

- Lua 2.4 was launched on May 14th, 1996, and with it came an external compiler named Luac. This compiler boasted fast loading and off-line syntax checking.

- This version also included improvements to the debug API by additional functions that allowed access to local variables and hooks, which then allowed the user to call them whenever needed.

LUA 2

- The final iteration of Lua 2 arrived on November 19th, 1996. Labeled Lua 2.5, this iteration introduced new engine into Lua for Pattern Matching and Vararg functions.
- Pattern matching was included in the language due the desirability of heavier text processing in Lua. The feature was added into Lua 2.5 in the form of two functions, strfind and gsub. The Vararg feature allowed for arguments corresponding to the triple dot (‘...’) operator to be collected and stored into a table named ‘arg’.

LUA 3

- The following versions of Lua would begin a trend where the time between major releases would gradually increase. Lua 3.0 would see release on July 1st, 1997.
- This new version featured a new library for creating Lua libraries called auxlib and conditional compilation support, but main feature included was that fallbacks were replaced with tag methods.

LUA 3

- The reason that fallback was replaced was because it was a global mechanic, which meant that there was only one hook per event.This made sharing or reusing code complicated because modules that defined fallbacks for the same event do not coexist well.

- The tags method solved this by attaching the hooks to pairs (E.x. Function test (event, tag)) instead of the event itself.

LUA 3

- One year after the 3.0 release came Lua 3.1. The 3.1 version saw the addition of functional programming with the introduction of anonymous functions and function closures

- Function closures in Lua 3.1 operated by use of upvalues. When the closure function is called, the local variable that was closed is sent to scope of another function. The closure function is also an anonymous function.

LUA 3

- The final version of Lua 3 released roughly a year later in July of 1999, Dubbed Lua 3.2.
- This version was largely considered just a maintenance release, including no new major features. This release did contain a new debug library that allowed debug tools to be written in Lua rather than in C.
- Lua 3.2.2, which was the very last version of 3.2, became publicly available on february 22nd, 2000. This version corresponds only to bug fixes for 3.2.

LUA 4

- Lua 4.0 released in November of 2000 with a brand new API. This API was used to replace the built-in functions that existed previously by writing a standard library for the language.
- Lua 4.0 also brought “for” statements to the language, one of the most requested item in the lua community at the time.
- The final version, Lua 4.0.1, releasing on July 4th, 2002. This version was not originally planned to lose support this early. After the release of 4.0, development for version 4.1 began.

LUA 5

- Lua 5.0 was released on April 11th, 2003. This version originally was named Lua 4.1 in its early stages, but due to the amount of changes and features within it, the name was reconsidered.
- 5.0 had brought the following main features to the language:
  - Collaborative multithreading using coroutines
  - Full lexical Scoping - Replaced upvalues
  - Metatables - Replaced tags and tag methods
  - Booleans
  - Weak tables
  - API for Lua Chunks

LUA 5

- Version 5.1 released on February 21st, 2006. Version 5.1 began development so that incremental garbage collection could be implemented per the request of game developers. Its last release was 5.1.5, released February 17th, 2012.
- The following main features were introduced in this release:
  - Incremental garbage collection
  - Module system
  - Mod and Length operators
  - Updated syntax for long strings and comments
  - Metatables for all types

LUA 5

- Version 5.2 of Lua released on December 16th 2011. The last release of 5.2 was on March 7th, 2015 with 5.2.4.
- This release introduced the following main features into the language:
  - Metamethods
  - New Global Lexical scheme
  - Ephemeron tables
  - Bitwise operations library
  - Light C functions
  - Goto Statement

LUA 5

- Lua version 5.3 released on January 12th, 2015 with the final version 5.3.6 releasing on September 25th, 2020.
- This release introduced the following main features into the language:
  - 32-bit/64-bit system support
  - Integers - 64-bit integers by default
  - Bitwise operators
  - Uft-8 library

LUA 5

- Lua version 5.4 is the most recent major release, releasing on June 29th, 2020. As of writing, the most recent version is Lua 5.4.4, released on January 26th, 2022.
- This release introduced the following main features into the language:
  - Generational mode for garbage collection
  - To-be-closed variables
  - Const variables

Distinguishing features and application domain

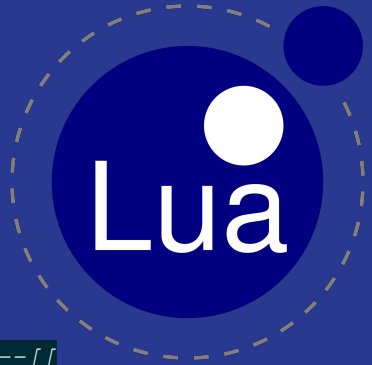
- Industrial Applications
- Robotics
- Literate Programming
- Distributed Business
- Image Processing
- Extensible Text Editors
- Ethernet Switches
- Bioinformatics
- Game development
- Web Development
- More . . .

Distinguishing features and application domain

- Semantically, many similarities with Scheme even though these similarities are not immediately clear because the two languages are syntactically very different.
- Influence of Scheme on Lua has gradually increased during Lua's evolution.
- The main distinguishing feature is that Lua offers tables as its sole data-structuring mechanism.

Data Types

1. Lua is dynamically typed. That is variables do not have types, only values have types. (***nil, boolean, number, string, userdata, function, thread and table***).
2. All values in Lua are first-class values: they can be assigned to global and local variables, stored in tables, passed as arguments to functions, and returned from functions.
3. A table can have any value as key and can store any value



Data Types

```
--[[
"Data Types"

Lua is dynamically typed. Variables do
not have a type indicator such as
C++ where "int" stands for integer. In
Lua, the value you assign to the
variable
is the type of value that variable will
have.

Value types: nil, boolean, number(the
reals), string, function, userdata,
thread,
and table.
```

```
print(type("What is this"))
--string
print(type(1.0)) --number
print(type(10)) --number
print(type(type)) --function
print(type(nil)) --nil
print(type(true)) --boolean
```

```
string
number
number
function
nil
boolean
```

Data Types - Tables

- Tables are the only data structure in Lua
- Tables are used to represent other data structures such as arrays, sets, queues, lists, etc.
- Tables are dynamic objects:
  - They can be created by simply using the constructor: { }
    - Ex: newTable = { }
  - The variable “newTable” is simply a reference to the table object, so calling theSameTable = newTable creates a second reference to the same object.
- Tables are comprised of key - value pairs:
  - Var = 7
  - newTable[var] = 100
  - print(newTable[7]) - - gives 100

- Tables are objects in Lua much like objects in Java or Scheme
- Tables implement associative arrays meaning that the array can be indexed using any data types Lua supports which is what makes tables so useful

Data Types - Numbers

- number in Lua represents real (double-precision floating point) numbers
  - However Lua does not have an integer data type
- Lua is able to represent any long integer without rounding problems
- Standard Lua uses 64-bit int and double precision (64-bit) float
- Can be configured to use 32-bit and single (32-bit) precision
  - useful for small machines and embedded systems
- Number values wrap around per two-complement arithmetic

Data Types - nil

- Used to differentiate between a value having data or not having data
  - This is similar to null in languages such as java
- Global variables in Lua have a default value of nil
  - These variables can be assigned nil to delete them

```
x = 1
print(x)
x = nil
print(x)
```

```
1
nil
```

Data Types - Boolean

- Boolean in Lua is the same as java, c, c++
  - Can carry two values either true or false
- In Lua all values evaluate to true except for false and nil

```
Lua 5.4.2  Copyright (C) 199
> t = false
> if t == true then
>> print("True")
>> elseif t == false then
>> print("False")
>> end
False
```

Data Types - String

- Strings represent both characters and strings
- Strings are defined with double quotes: “ ”

  Or single quotes: ‘ ’

  - Ex

A = “A string”

B = ‘Another String’

- Strings are used for characters
  - Char = “A”
  - Char = ‘A’

Data Types - String

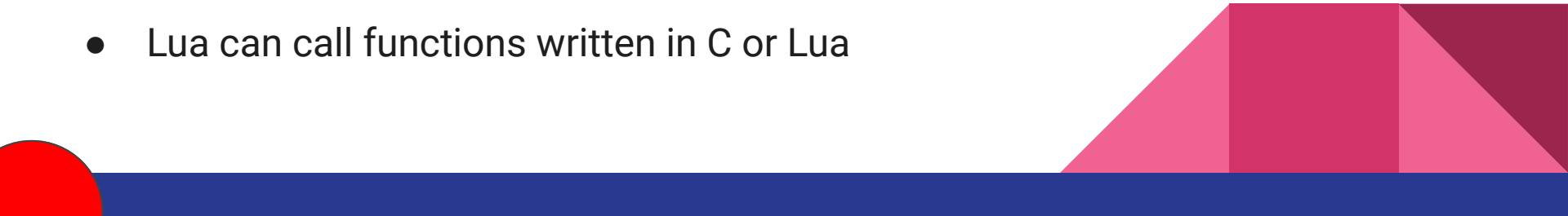
- String concatenation is done with two periods
  - print(“Hello ”.. “World”)
    - Hello World
- Newlines are defined as follows
  - print(“Hello”..”\n”..”World”)
    - Hello
    - World
- There are a few other things you can do with strings

```
> print("one line\nnext line\n\"in quotes\", 'in quotes'")
  one line
  next line
  "in quotes", 'in quotes'
  > print('a backslash inside quotes: \'\\\'')
  a backslash inside quotes: '\'
  > print("a simpler way: '\\'")
  a simpler way: '\'
```

\a	bell
\b	back space
\f	form feed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\"	double quote
\'	single quote
\[	left square bracket
\]	right square bracket

Data Types - Function

- Functions are first class values in Lua
  - This mean that functions are able to be stored in variables, passed as arguments to other functions and returned as results
  - This allows for great flexibility in the Lua language
- Lua offers good support for functional programming, including nested functions with proper scoping
- First class functions play a key role in Lua's object oriented facilities
- Lua can call functions written in C or Lua



Data Types - Userdata

- A userdata is an interesting data type in Lua
- Userdata allows arbitrary C data to be stored in Lua variables
  - A userdata value is a pointer to a block of raw memory
- There are two types of Userdata in Lua
  - Full Userdata:
    - The block of raw memory is managed by Lua
  - Light Userdata
    - The block of raw memory is managed by the host
- Userdata values cannot be created or modified in Lua, only through the C API

Data Types - Threads

- A function for a coroutine is created

```
> function foo()
>> print(“foo”, 1)
>> coroutine.yield()
>> print(“foo”, 2)
>>end
```

- Coroutine is created using the coroutine.create() function:

```
> co = coroutine.create(foo)
>> = type(co)
thread
```

- The object returned by Lua is a thread
- We can find out what state a thread is in using the coroutine.status() function

```
> coroutine.status(co)
suspended
```

- This means the thread is alive, just not doing anything. The thread will only continue once coroutine.resume() function
- It is important to note that when the thread was created it did not start execution. In order to start a thread you must call the coroutine.resume() function

Variable Names

- Variable names can be strings of letters, numbers and underscores, not beginning with a number.
- An underscore followed by one or more CAPITAL letters are reserved for special uses and should not be used as identifiers.
- Several of the expected words are reserved (**and , break, do, else, etc.**) and can not be used for identifiers, however. . .
- Lua is case sensitive so (AND, And, BREAK, DO, Do, etc.) are legal identifiers, although should probably be avoided to avoid confusion.

Comments

- - - Comments begin with a double hyphen.
- - -[[

  Double hyphen followed by two opening square brackets begins multi-line comments.

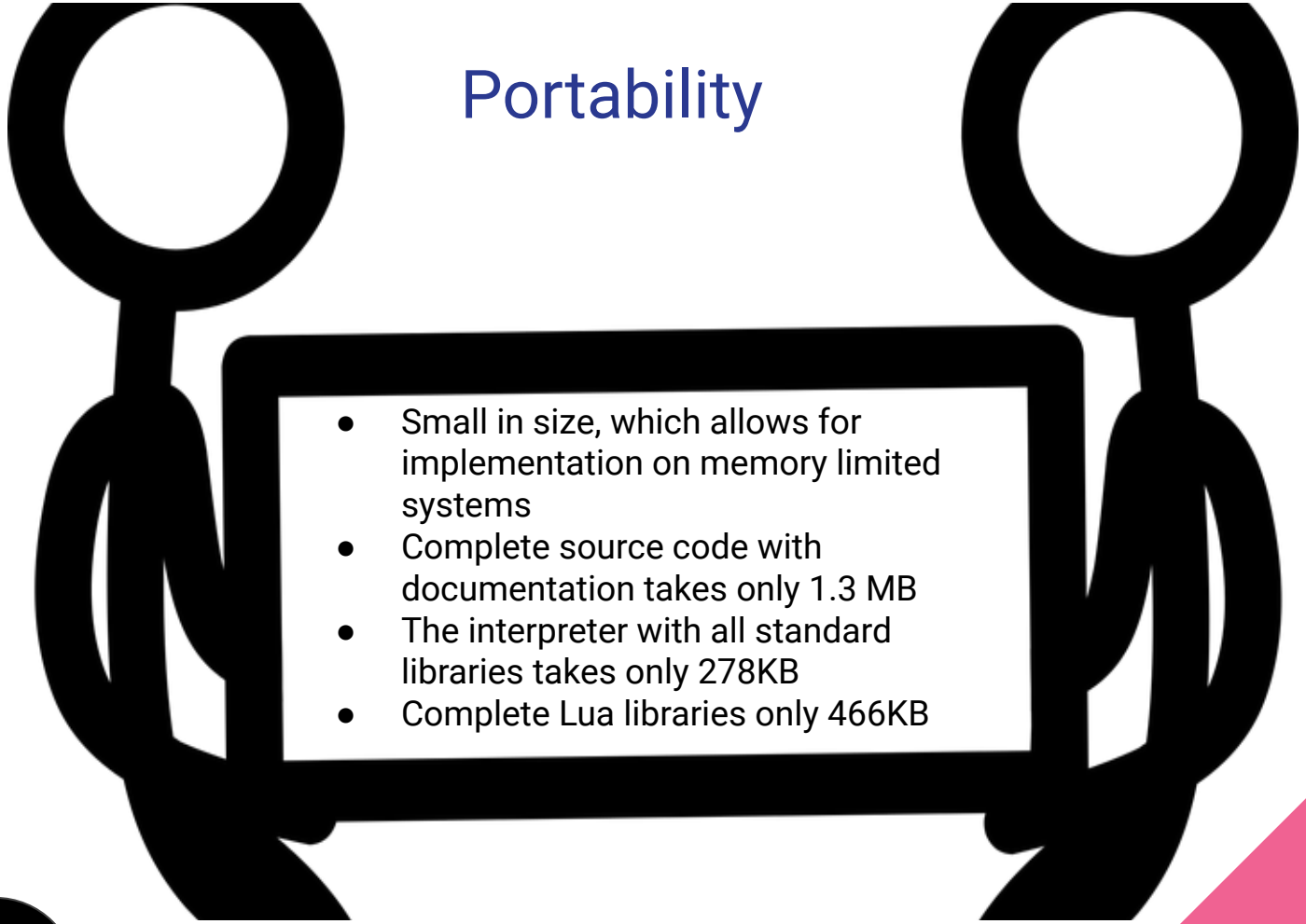
  The double closing square brackets closes the comment section

-]]
- Often multi-line comments will end in - -]] as this allows the opening notation to be disabled by simply adding a hyphen at the start. The closing notation then becomes a single line comment.

Portability

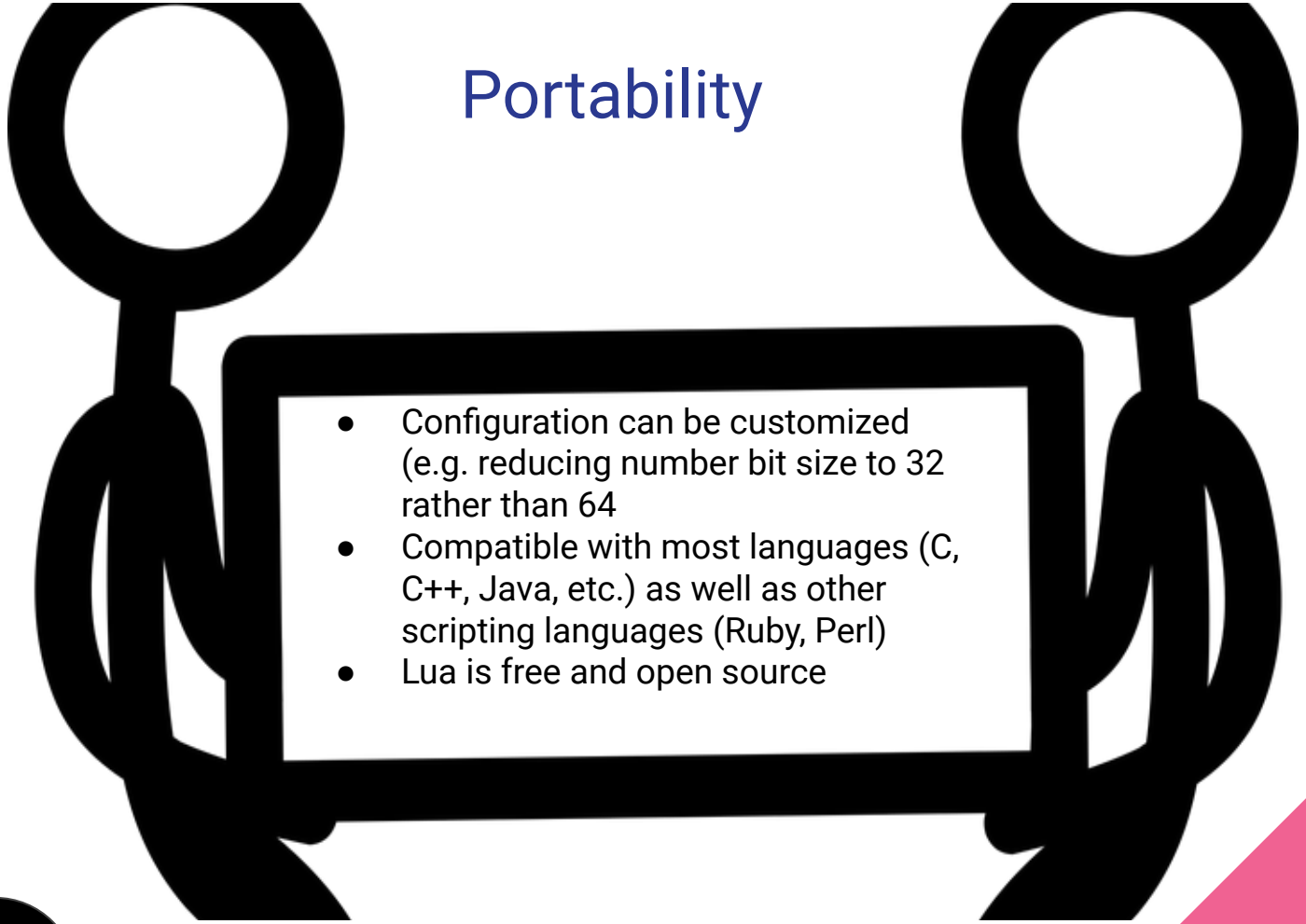
- Lua is easy to build because it is implemented in strict ANSI C
- It compiles out-of-the-box on most platforms (Linux, Unix, Windows, Mac OS X, etc.)
- It runs with at most a few small adjustments in virtually all platforms we know of

Portability



- Small in size, which allows for implementation on memory limited systems
- Complete source code with documentation takes only 1.3 MB
- The interpreter with all standard libraries takes only 278KB
- Complete Lua libraries only 466KB

Portability



- Configuration can be customized (e.g. reducing number bit size to 32 rather than 64
- Compatible with most languages (C, C++, Java, etc.) as well as other scripting languages (Ruby, Perl)
- Lua is free and open source


```
--[[
"Basic Syntax"

Identifiers:
Starts with A...Z | a..z | _
then followed with zero or more letters, underscores, or
digits(0-9).

Keywords:
and, break, do, else, elseif, end, false, for, function, if,
in, local,
nil, not, or, repeat, return, then, true, until, while, print.

]]
```


```
--[[
"Basic Syntax"

Whitespace:
The interpreter for Lua ignores whitespace. Must have
a space after keywords!

Indentation: Not needed but makes the programs look neater.

Pass by reference: function, table, userdata, thread(coroutine),
strings.

Optional: You can use ; to end statements in lua but it is not
required. Usually
used if one wants multiple statements all in one line.
Examples:
]]
```

Valid Identifiers

```
hello;
```

```
Hello
```

```
_hello;
```

```
_hel_lo123_
```

I/O Functionality

- io.read() reads in a file
- io.write() keeps writing to the same line in a terminal
- print() after print is done it starts a new line in a terminal
- Simple I/O model
  - Reads and Writes to current files
  - Can change files wit io.input(filename)
- Complete I/O Model uses file handles
  - Can open files to read, write, append
  - Can open files specifically as binary

Variables

```
--[[
"Variables"

Types of variables:
global, local, and table fields.

Definition:
type variable_list

Examples:
]]
```

```lua
local x = 1
local y
print("value of x: ", x, "\nvalue of y: ", y, "\n\n")
--global variable from line 52
GLOBAL = 1
```

```
value of x:     1
value of y:     nil
```

Strings

- Strings in Lua are as expected: a sequence of characters
- 8-bit clean: strings my contain any character or numeric value
- Strings can hold binary data
- Strings are immutable - you may not change the value in the string, although you can change what value the assigned variable is pointing to.
  - i.e. str = “hello”
    str = gsub(str, “h”, “j”) - - changes the assignment of str

Strings

\ Escape character goes before the character that needs escaping or used for

\a Bell

\b Backspace

\f Formfeed

\n New line

\r Carriage return

\t Tab

\v Vertical tab

\\ Backslash

\" Double quotes

\' Single quotes

\[Left square bracket

\] Right square bracket

Strings

String Manipulation:

string.upper(argument)

Returns a capitalized representation of the argument.

string.lower(argument)

Returns a lower case representation of the argument.

string.gsub(mainString,findString,replaceString)

Returns a string by replacing occurrences of findString with replaceString.

Strings

String Manipulation:

string.find(mainString,findString,optionalStartIndex,optionalEndIndex)
Returns the start index and end index of the findString in the main string and nil if not found.

string.reverse(arg)
Returns a string by reversing the characters of the passed string.

Strings

string.format(...)

Returns a formatted string.

string.char(arg) and string.byte(arg)

Returns internal numeric and character representations of input argument.

string.len(arg)

Returns a length of the passed string.

string.rep(string, n))

Returns a string by repeating the same string n number times.

..

Thus operator concatenates two strings.

Strings

```
--String formatting
local str1, str2, str3 = "Lua", 'Lua', [[Lua]]
print(" \"str1: \" ", str1, "\n") --Outputs
```

```
"str1: "        Lua
```

Strings

```
print(
"\n\n",string.upper(str2),"\n",
string.lower(str2),"\n",
string.gsub(str2, "Lua", [[luA]]),"\n",
string.find(str2, 'Lua'),"\n",
string.reverse(str2),"\n",
string.format("%s %s %s", str1, str2, str3),"\n",
string.byte(str1, 1),"\n",  --Byte representation of the 1
or nth char
string.char(76),"\n",
string.len(str2),"\n",
string.rep(str3.." ", 3),"\n"
)
```

```
LUA
lua
luA
1
auL
Lua Lua Lua
76
L
3
Lua Lua Lua
```

Strings

```
--Strings may be re-assigned
str1, str2, str3 = [[lu]], 'hello', "LUA"
print(str1, " ", str2, " ", str3,"\n\n")
```

```
lu hello LUA
```

Operators

- Mathematical Operators operate on real numbers
  - Binary (+, -, *, /)
  - Unary ‘-’ (negation)
  - Partial support for ^ (exponentiation) with the C mathematical library (not part of Lua core)

Operators

- Relational Operators (<, >, <=, >=, ==, ~=)
  - Always result in true or false
  - Considers different types different values
    - I.e. 0 ~= “0”
  - Nil is only equal to itself
- Functions, tables and userdata compared by reference
  - They are **only** considered equal if they are the **same object**
- Order operators compare strings alphabetically
- Mixed data types can cause errors if compared

Operators

```
--[[
"Operators"

Operators: Arithmetic, Relational,
Logical, and Misc.

Precedence:
(right to left)
unary: not, #, -
Concatenation: ..
(left to right)
Mult: *, /, %
Add: +, -
Relational: <, >, <=, >=, ==, ~=
Logical: and
al: or
```

```
--Arithmetic
local x, y = 2, 3
print(x + y)
print(x - y)
print(x * y)
print(x / y)
print(x % y)
print(x ^ y)
print(-y, "\n\n")

--Relational
print(x == y)
print(x ~= y)
print(x > y)
print(x < y)
print(x >= y)
print(x <= y, "\n\n")
```

```
5
-1
6
0.66666666666667
2
8.0
-3


false
true
false
true
false
true
```

Boolean Operators

- Logical Operators (and , or, not)
  - **False** and **nil** are considered false
  - **Everything** else is true (yes, even 0 and “”)

Return values for “and”

- Returns the first argument if the ***first argument*** is false
  - i.e. (false and 27) = false
  - (nil and 27) = nil
- Otherwise it returns the second argument
  - I.e. (27 and false) = false
    (27 and 1) = 1

Return values for “or”

- Returns the first argument if the ***first argument*** is **not** false
  - i.e. (27 or false) = 27
    (27 or 1) = 27
- Otherwise it returns the second argument
  - I.e. (false or 27) = 27
    (false or false) = false

Boolean Operators

- Logical Operators (and , or, not)
  - Lua only evaluates the second variable when necessary (short-cut evaluation)
  - “and” has higher precedence than “or”:
    - 1 and 2 or 3 is equivalent to (1 and 2) or 3
  - Example getting the max of two numbers:
    - Max = (x > y) and x or y

Precedence

- Binary operators are left associative ***except*** ^ and .. which are right associative. i.e. a ^ b ^ c is equal to a ^ (b ^ c)
- Decreasing Precedence
  - ^
  - not
  - * /
  - + -
  - .. (concatenation)
  - < > <= >= == ~=
  - and
  - or

Operators

```lua
--Logical
local x, y = true, false
print(x and y)
print(x or y)
print(x and not y, "\n\n")

--Misc
local x, y = "hello ", "world"
print(x..y)
print(#(x..y), "\n\n")
```

```
false
true
true


hello world
11
```

Scoping and Parameter Passing

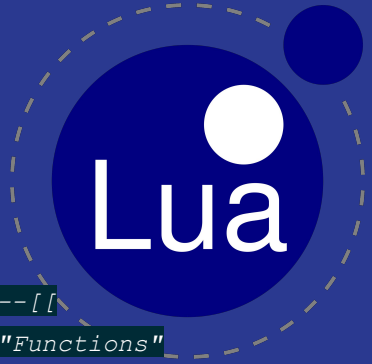
- No error if calling a variable before or without it being declared
  - Calling an un-initialized variable gives nil
- Global variables
  - Any variable declared without “local” in front is a global variable
  - Global variables can be effectively deleted by setting the value to nil
- Local variables are declared by preceding the with the word “local”
  - scope is limited to the function in which it is declared
  - As with global variables, undeclared local variables are nil and can be effective deleted by setting to nil
  - Access to local variables is faster than to globe variables

Functions

- Functions - like all values in Lua are first class values:
  - Can be stored as values
  - Passed as parameters to other functions
  - Returned from other functions
- Functions can be re-defined or even erased!
  - This may be desirable for creating a secure environment.
- Lua supports functional programming
  - Functions can be nested
  - Lexical scoping of variables (variables scope within the block where it was defined)

Functions

- C compatibility
  - Lua can run functions written in Lua as well as C or any other language used by the host application
- No big surprise:
  - Lua interpreter is written in ANSI C
  - The standard Lua library is also written in C
  - Functions for I/O, system operations, string manipulation, table manipulation and mathematical operations are all written in C
- Users may define their own functions in C



Functions

```
--[[
"Functions"

 (scope) function (name)(arg1, arg2, arg3, ..., argn)
 (body)
 return v1, v2, v3, ..., vn

 scope: Can be local. If global, do not use the local keyword
 name: using the conventions for naming a variable.
 argn: Arguments are optional
 body: a bunch of statements
 return: you can return multiple values separating by commas or return a single
```

```
function min(n1, n2)
  local result
  if(n1 < n2) then
     result = n1;
  else
     result = n2;
  end
  return result
end

--Function call
print("Is the minimum number 5 or 4?: ", min(5,4), "\n\n") --Formal parameters
```

```
Is the minimum number 5 or 4?:  4
```

Functions

```lua
--Assigning functions as variables
local min_max = function(min_num,
max_num)
  print("The min is: ", min_num, "The
max num is: ", max_num, "\n\n")
end

local function max(n1, n2)
  local result
  if(n1 > n2) then
     result = n1
  else
     result = n2
  end
  min_max(min(1,2), result)
--Recursive call on max
end

--Function call
max(1,2)
```

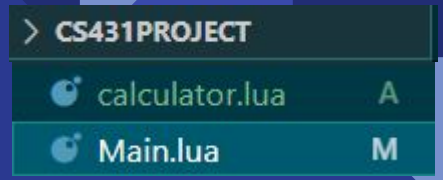
```
The min is:     1       The max num is:         2
```

Functions

```lua
--global variable from line 52
GLOBAL = 1

--Global variable access all scopes
local function global_output()
  print("Global variable is: ", GLOBAL)
end
global_output()
```

```
Global variable is:     1
```

Modules



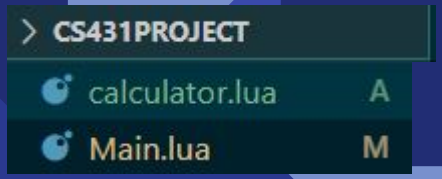

```lua
--Accessing the calculator module
local calc = require("calculator") --written on the same folder in file
named calculator.lua
calc.add(1,2)
calc.subt(1,2)
calc.mul(1,2)
calc.div(1,2)
print("\n\n")
```

```
3

-1

2

0.5
```

Iterators and Closures

- Function that can be instantiated like a class
- Used to iterate over elements of a collection
- Closures are like anonymous functions or lambda functions.
- Closure can be assigned to variables, pass to other functions, and returning a value(s).
- Closures have access to all local variables.
- Upvalues are saved when function is terminated

Modules




```
--[[
"Modules"

Modules are like libraries
loaded in using the require
keyword.

Just a bunch of functions
wrapped under a variable and do
things when
invoked
```

```lua
local calculator = {}

function calculator.add(a,b)
    print(a+b, "\n")
end

function calculator.subt(a,b)
    print(a-b, "\n")
end

function calculator.mul(a,b)
    print(a*b, "\n")
end

function calculator.div(a,b)
    print(a/b, "\n")
end

return calculator
```

Closures

```
--[[
"Closures"

Closures are functions that
'closes' over those local
variables.
The local variable that has been
closed over by that function
reads up
into the new scope of the other
func which is why it's called an
value.
```

```lua
local function counter()
  local i = 1
  return function() --closure
function (anonymous)
    i = i + 1
    return i
  end
end

local v1 = counter()
print("Value of v1: ", v1())
local v2 = counter()
print("Value of v1: ", v1())
--Value is saved
print("\n\n")
```

```
Value of v1:    2
Value of v1:    3
```

Control Structures including Recursion

- If then
- else
- elseif
- While do end
- Repeat until
- Numeric for
- Generic for
- Break and return
- Recursion functionality yes
- Chunks

Decision

```
--[[
"Decision"

!!!:In Lua, zero and empty strings are true in condition checks.

Just like loops, Decisions can be nested inside any number of times.

Statements: if() then, elseif , else,
```

```
if(false) then --then statements can go here
  print("false\n\n")

  --Else and elseif statements are optional but only one end is needed
  elseif(0 and "")
    then--then statements can go here
    print("true\n\n")

  elseif("")
    then
      if(0) then
      print("true but will not run since prev is true\n\n")
      end

  else
    print("This statement will never run\n\n")
end --Notice only one end for each block of code. In this case, the blocks are
-- the first if statement and the second if statement inside the second elseif.
```

```
true
```

Loops

```
--[[
"Loops"

Types of loops: while, for,
repeat...until, nested loops

To exit a loop you can use
"break"

!!!: Any type of loop can be
nested inside another loop type
 number of times
```

```lua
local x = 10

print("while loop:")
while(x >= 0)
do
  print("x is: ", x)
  x = x - 1
end
print("\n\n")

print("for loop:")
for i = 0, 10, 1  --Where for(x =
initial real #, final real #,
decrease/increase real #)
do
  print("i is: ", i)
end
print("\n\n")
```

```
while loop:
x is:   10
x is:   9
x is:   8
x is:   7
x is:   6
x is:   5
x is:   4
x is:   3
x is:   2
x is:   1
x is:   0


for loop:
i is:   0
i is:   1
i is:   2
i is:   3
i is:   4
i is:   5
i is:   6
i is:   7
i is:   8
i is:   9
i is:   10
```

Loops

```
print("repeat...until loop:")
local x = 10
repeat
  print("x is: ", x)
  x = x - 1
until(x == 0)
print("\n\n")
```

```
repeat...until loop:
x is:   10
x is:   9
x is:   8
x is:   7
x is:   6
x is:   5
x is:   4
x is:   3
x is:   2
x is:   1
```

```
--Showcasing multiplication and do
statement can go in either places.
print("nested loop multiplication table: ")
for i = 1, 10, 1 do
  local x = 1
  while(x <= 10) do
    print(i, "X", x, "is: ", i*x)
    x = x + 1
  end
end
print("\n\n")
```

```
nested loop multiplication table:
1       X       1       is:     1
1       X       2       is:     2
1       X       3       is:     3
1       X       4       is:     4
1       X       5       is:     5
1       X       6       is:     6
1       X       7       is:     7
1       X       8       is:     8
1       X       9       is:     9
1       X       10      is:     10
2       X       1       is:     2
2       X       2       is:     4
2       X       3       is:     6
2       X       4       is:     8
2       X       5       is:     10
2       X       6       is:     12
2       X       7       is:     14
2       X       8       is:     16
2       X       9       is:     18
2       X       10      is:     20
3       X       1       is:     3
3       X       2       is:     6
3       X       3       is:     9
3       X       4       is:     12
3       X       5       is:     15
3       X       6       is:     18
3       X       7       is:     21
3       X       8       is:     24
3       X       9       is:     27
3       X       10      is:     30
4       X       1       is:     4
4       X       2       is:     8
4       X       3       is:     12
4       X       4       is:     16
4       X       5       is:     20
4       X       6       is:     24
4       X       7       is:     28
4       X       8       is:     32
4       X       9       is:     36
4       X       10      is:     40
5       X       1       is:     5
5       X       2       is:     10
5       X       3       is:     15
5       X       4       is:     20
5       X       5       is:     25
5       X       6       is:     30
5       X       7       is:     35
5       X       8       is:     40
5       X       9       is:     45
5       X       10      is:     50
6       X       1       is:     6
6       X       2       is:     12
6       X       3       is:     18
6       X       4       is:     24
6       X       5       is:     30
6       X       6       is:     36
6       X       7       is:     42
6       X       8       is:     48
6       X       9       is:     54
6       X       10      is:     60
7       X       1       is:     7
7       X       2       is:     14
7       X       3       is:     21
7       X       4       is:     28
7       X       5       is:     35
7       X       6       is:     42
7       X       7       is:     49
7       X       8       is:     56
7       X       9       is:     63
7       X       10      is:     70
8       X       1       is:     8
8       X       2       is:     16
8       X       3       is:     24
8       X       4       is:     32
8       X       5       is:     40
8       X       6       is:     48
8       X       7       is:     56
8       X       8       is:     64
8       X       9       is:     72
8       X       10      is:     80
9       X       1       is:     9
9       X       2       is:     18
9       X       3       is:     27
9       X       4       is:     36
9       X       5       is:     45
9       X       6       is:     54
9       X       7       is:     63
9       X       8       is:     72
9       X       9       is:     81
9       X       10      is:     90
10      X       1       is:     10
10      X       2       is:     20
10      X       3       is:     30
10      X       4       is:     40
10      X       5       is:     50
10      X       6       is:     60
10      X       7       is:     70
10      X       8       is:     80
10      X       9       is:     90
10      X       10      is:     100
```

Iterators and the Generic For vs Iterative

- Allows for the traversal through elements tables.
- Generic for iterator is always a key value pair for each element in the table.
- Generic for allows the traversal of all values returned by the iterator function

Iterators

```
--[[
"Iterators"

Allows traversal through
elements of data structures such
as arrays.
keys always start at value 1 and
increments by one.

in ipairs is the keyword for
iterator format.
```

```
--for iterator for(name, name, in
ipairs(arg))
local array = {1, 2, 3}
for key, value in ipairs(array)
do
  print(key, value)
end
print("\n")
```

```
1       1
2       2
3       3
```

Data Structures (including arrays, ADTs)

- Arrays
- Matrices and Multi-Dimensional Arrays
- Queues and Double Queues
- Sets and Bags
- Strings and Buffers
- Metatables and Metamethods (sec 2.4)

Tables

```
--[[
"Tables"

Tables are the only data structure in Lua.
Functionally, it is exactly
like an array. It can also be a dictionary,
queue, stack, or mapping.
!!!: Can have a mix of values like numbers
and strings in all permutations
!!!: Table sizes are not fixed and is
scalable and start at index 1
!!!: Tables can be negative in Lua
!!!: You can have multiple returns values
from a function
!!!: When assigning a table to another
table, the memory is both the same
!!!: Tables can have any kind of index
strings, boolean, real numbers, etc.
```

```lua
local array = {"Lua", "Array"}
--index 0 is nil as show below
for i = 0, 2 do
  print(array[i])
end

print("\n")

--Arrays can be negative!
local array = {}
for i = -1, 1 do
  array[i] = i
  print(array[i])
end

print("\n")
```

```
nil
Lua
Array


-1
0
1
```

Tables

```lua
--Multidimensional Table
local multiplication_table = {}
for i=1,10 do
  multiplication_table[i] = {}
  for j=1,10 do
    multiplication_table[i][j] = i * j
    io.write(multiplication_table[i][j], " ")
  end
end
```

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Tables

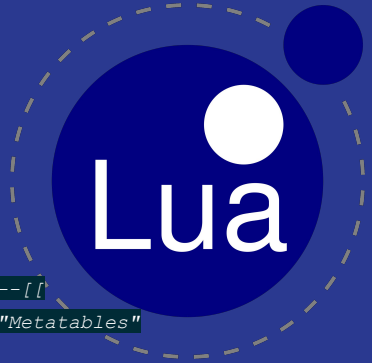
```lua
--Interesting indexes
local table1 = {}
local table2 = {1, 2, 3}


table1[1] = 1
table1["what_the"] = 2
table1[false] = 3

print(table1[1], table1["what_the"],
table1[false], "\n")
```

```
1	2	3
```

Tables

```lua
--Table manipulation
print(
"\n",table.concat(table2),"\n",
table.concat(table2, ", ", 2, 3),"\n"
)
table.insert(table2,4) --adds at the end of the array
table.remove(table2, 1)
table.sort(table2)
print(table.concat(table2, ", "))

print("\n\n")
```

```
123
2, 3

2, 3, 4
```



Metatables

```
--[[
"Metatables"

An auxiliary table that helps modify behaviors of
tables along with key est and meta methods
such as changing/adding functionalities to
operators on tables and looking up metatables
when
no key is available in the table.

Important methods to set and get metatables:
setmetatable(table, metatable) which sets the
metatable
getmetatable(table) which is used to get the
metadata of a table

Meta methods:
__index which looks up a metatable
__newindex new keys will be defined in the
metamethod which
then transfers over to the main table

Table operator behaviors:
1
__add

Changes the behavior of operator '+'.

2
__sub

Changes the behavior of operator '-'.

3
__mul

Changes the behavior of operator '*'.

4
__div

Changes the behavior of operator '/'.

5
__mod

Changes the behavior of operator '%'.

6
__unm

Changes the behavior of operator '-'.

7
__concat

Changes the behavior of operator '..'.

8
__eq

Changes the behavior of operator '=='.

9
__lt

Changes the behavior of operator '<'.

10
__le

Changes the behavior of operator '<='.
]]
```

Metatables

```lua
--How to set a table as a metatable and check table for index
local mytable = {1,2,3,4,5}
setmetatable(mytable,{
   __index = function(atable, index)
    print("No such data")
    return 1 --prevents table search from crashing
  end
})
print(mytable[100], "\n")
```

```
No such data
1
```

Prototype not Classes

- There is no Class in Lua, however class-like behavior is easy to implement
- Lua uses metatables to simulate OOP

Prototype not Classes

```lua
--[[
OOP and inheritance with metatables
]]
Rectangle = {area = 0, length = 0, width = 0}

-- Derived class method new

function Rectangle:new (length,width)
   setmetatable(o, self)
   self.__index = self
   self.length = length or 0
   self.width = width or 0
   self.area = length*width;
   return o
end

--  method printArea

function Rectangle:printArea ()
   print("The area of Rectangle is ",self.area)
end

--Create object
local rectangle_object = Rectangle:new(10, 10)

print(rectangle_object)
rectangle_object:printArea() --colin to access
methods
```

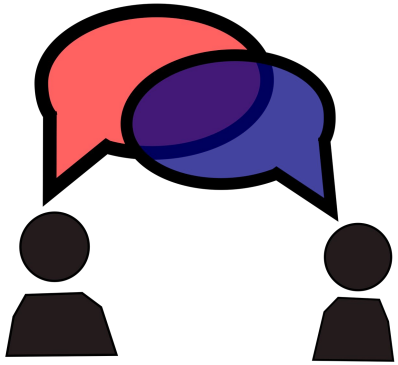
```
table: 00000000006dd280
10
The area of Rectangle is        100
```

Recursion

```lua
--[[
"Recursion"

A function that calls itself until
ended by a base case
]]
local function factorial(n)
  if n == 0 then
    return 1
  end
  return n * factorial(n-1)
end

print(factorial(5))
```

```
120
```

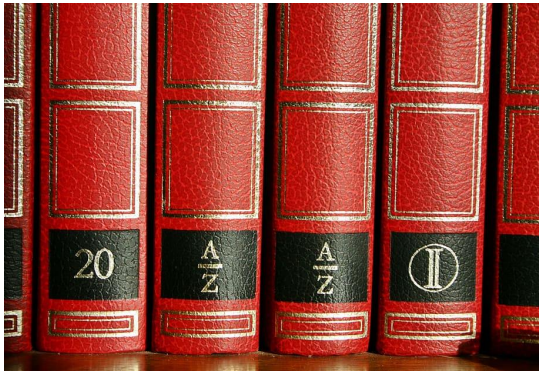
Tuckers Criteria



Expressivity

Great

- Variable names are fairly un-restricted which means they can be given descriptive names
- Mathematical operations follow familiar rules and simple precedence
- Rich in control structures (if, then, else, for, while) allowing for logical program design



Well-definedness

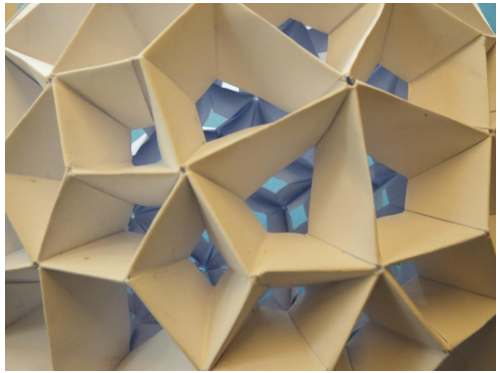
Good

- Overall well defined, however:
- Functions are anonymous. They are assigned to variable names, and that assignment can be changed.
- Tables use key value pair - this can be confusing when dealing with array like functionality.
- Re-typing of variables

Data Types and Structures

Great

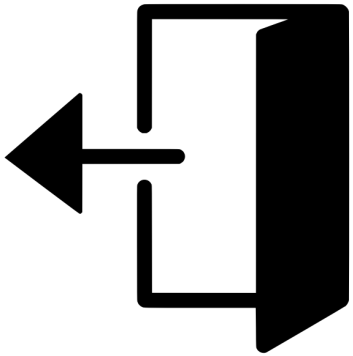
- Lua has a variety of data types (nil, boolean, number, string, userdata, function, thread and table)
- Variables are dynamically typed and can be re-typed
- Table are dynamic data structure used to efficiently and effectively represent all other structures



Modularity

Great

- Lua was designed to be embedded with other languages. It is inherently modular from the start.
- Functions can be written in either Lua or the language that Lua is embedded in
- Recursive functionality allows for elegant and dynamic solution
- Lua is sometimes called a Glue language.



I/O Facilities

Good

Multiple I/O facilities

- Simple I/O Model
  - Used or stdin and stdout
  - Can change to read / write to specified files instead
- Complete I/O Model
  - File handles
  - Open as read, write, append
  - Open files as binary

Portability

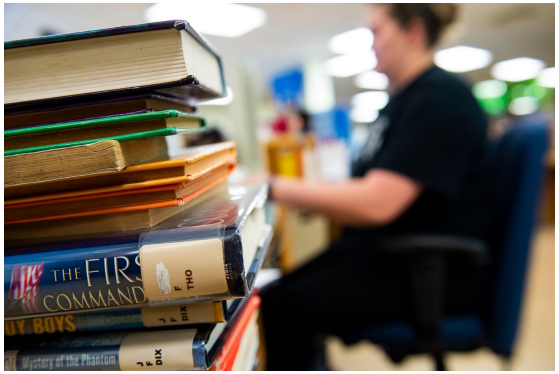
Great

- Lua’s main design was for the language to be simple, small, portable, fast and easily implemented into applications
- Lua is distributed in a small package and builds out-of-the-box in all platforms that have a standard C compiler
- Lua runs on all different kinds of Unix, Windows mobile devices, etc…
- Can be optimized for embedded systems

Efficiency

Great

- Lua is an interpreted language
- Not as efficient as compiled languages
- One of the fastest scripting languages (much faster than python / rust)
- LuaJIT uses just-in-time compiler to make even faster
- From the start Lua development has been focused on efficiency



Petagogy

Good

- Lua is very similar to other languages so as a second language is very fast to learn
- As an interpreted language, the stand-alone interpreter allows for rapid feedback - would make a good first language
- Debugging is somewhat limited
- Tables can be confusing
- Some of the logic rules can be confusing

Generality

Great

- Procedural Programming
- OOP
- Functional Programming
- Data-driven programming
- Embedded systems
- Stand alone interpreter
- “Glue language”

Tuckers Criteria

Overall: Great

- Great general purpose programming language
- Fairly easy to learn and incorporate with other languages
- Extensible and customizable
- Fast

Lua use case: Roblox

Bloxia Life

Bloxia Life is still in BETA

🗺️Explore the vast world of Bloxia Life.🗺️
🤸Roleplay with friends.🤸

Active	Visits
0	892

High Level

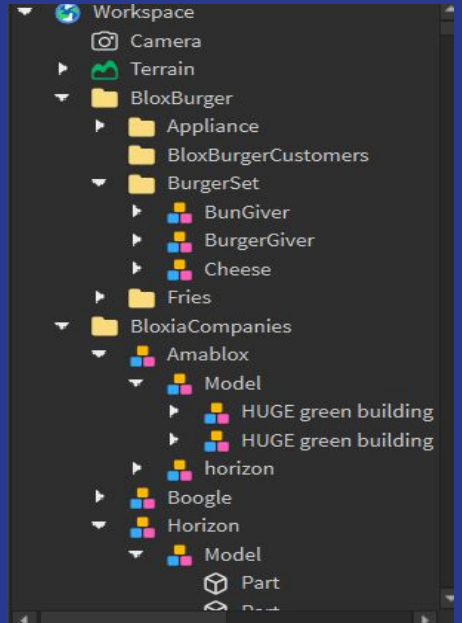
- The World(Bloxia Life)
- Blox Burger & All other restaurants
- Objects of Blox Burger
- Modules(A bunch of reusable funcs)
- Functions(behaviors)
- Tables(attributes) in ipairs key, value
- Variables(identity)

Low Level

Module

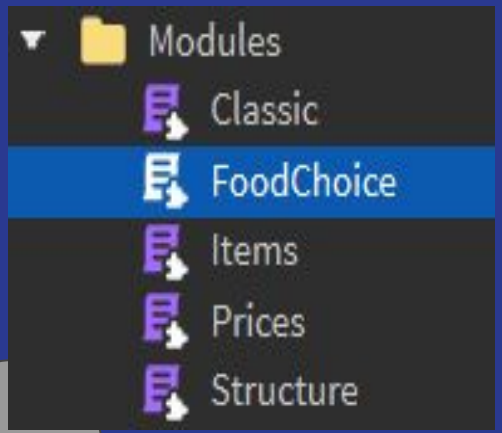
```lua
local choiceItem

local module = {
    --BloxBurger customers options--
    BloxBurgerChoice = function()
      local randNum = math.random(3)

      if randNum == 1 then
          choiceItem = "Fries"
      elseif randNum == 2 then
          choiceItem = "Cheeseburger"
      elseif randNum == 3 then
          choiceItem = "Plain Hamburger"
      end
      return choiceItem
    end,

    --CafeBloxe customers options--
    CafeBloxeChoice = function()
      local randNum = math.random(5)

      if randNum == 1 then
          choiceItem = "Plain Coffee"
      elseif randNum == 2 then
          choiceItem = "Chocolate Coffee"
      elseif randNum == 3 then
          choiceItem = "Strawberry Coffee"
      elseif randNum == 4 then
          choiceItem = "Caramel Coffee"
      elseif randNum == 5 then
          choiceItem = "Coffee w Cream"



      end
      return choiceItem
    end,
```

- Powers all dialogs and correct items to give to customer.
- DRY(Do not repeat yourself) is why modules are used
- Or else there will be more scripts to traverse through the hierarchy. No one on a developer team wants to find multiple function scripts in this mess This is just a small snippet example. Imagine this picture but 1000 times larger GTA5 scale. Do not keep code coupled to objects if possible)---------->



Instead use modules…

- Easy to find in a tree hierarchical structure
- Easy to extend modules
- Easy to reuse code




```lua
--CafeBloxe customers options--
   CafeBloxeChoice = function()
    local randNum = math.random(5)

    if randNum == 1 then
        choiceItem = "Plain Coffee"
    elseif randNum == 2 then
        choiceItem = "Chocolate Coffee"
    elseif randNum == 3 then
        choiceItem = "Strawberry Coffee"
    elseif randNum == 4 then
        choiceItem = "Caramel Coffee"
    elseif randNum == 5 then
        choiceItem = "Coffee w Cream"



    end
    return choiceItem
end,

--IceBlox customers options--
 IceBloxChoice = function()
    local randNum = math.random(6)

    if randNum == 1 then
        choiceItem = "Mint IceCream"

    elseif randNum == 2 then
        choiceItem = "Purple Madness IceCream"

    elseif randNum == 3 then
        choiceItem = "Vanilla IceCream"

    elseif randNum == 4 then
        choiceItem = "Strawberry IceCream"

    elseif randNum == 5 then
        choiceItem = "Black Pepper IceCream"

    elseif randNum == 6 then
        choiceItem = "Prime Peach IceCream"
    end
    return choiceItem
end,
```

```lua
--PizzaPalace customers options--
 PizzaPalaceChoice = function()
    local randNum = math.random(2)

    if randNum == 1 then
        choiceItem = "Cheese Pizza"

    elseif randNum == 2 then
        choiceItem = "Pepperoni Pizza"
    end
    return choiceItem
end,

--Pretzies customers options--
 PretziesChoice = function()
    local randNum = math.random(3)

    if randNum == 1 then
        choiceItem = "Pretzel"

    elseif randNum == 2 then
        choiceItem = "Pretzel w Sugar"

    elseif randNum == 3 then
        choiceItem = "Pretzel w Salt"

    elseif randNum == 4 then
        choiceItem = "Pretzel w Cinnamon"
    end
    return choiceItem
end,

--Smuth customers options--
 FruityChoice = function()
    local randNum = math.random(3)

    if randNum == 1 then
        choiceItem = "Pancake"

    elseif randNum == 2 then
        choiceItem = "Orange Juice"

    elseif randNum == 3 then
        choiceItem = "Apple Juice"

    end
    return choiceItem
end,
```

Modules and Coroutines in

```lua
local foodChoiceModule = require(ReplicatedStorage:WaitForChild("FoodChoice"))


local head = script.Parent.Head
local debounce = 0


--Make a coroutine that destroys the NPC after 30 seconds--
local deleteNPC = coroutine.create(function()
    wait(40)
    NPC:Destroy()
end)



local flag = true
--Repeat until npc is inside a workspace folder
while flag do
    wait(1)

    ----------------------------------------Positions for BloxBurger-----------------------------------------------------------
    if NPC.Parent == inBloxBurgerCustomers then
        flag = false--to cancel the big boi while loop
        local choice = foodChoiceModule.BloxBurgerChoice()
        ChatService:Chat(head,"A " .. choice .. " Please!","White")

        local randNum = math.random(2)
```

```lua
--Duplicates pathfinding into every child of Customer--


for i, v in pairs(game.ServerStorage.Customers:GetChildren()) do
    if v.ClassName == "Model" then
        local PathFindingScript = game.ServerStorage.ScriptsDup.PathFinding:Clone()
        PathFindingScript.Parent = v --Make the model the parent of script
    end
end
```

- First use the require keyword to access the module
- As things scale up, one has to make use of space efficiently. If I made the deleteNPC function in a separate script and wanted to change the time, I would have to go to every script to change the time.
- One solution is coroutines or duplicated the function into every customer. pairs is used for (key,values) and ipairs is used for (index, value)

More use cases:

Corona SDK Corona SDK is a cross platform mobile game engine that supports iPhone, iPad, and Android platforms. There is a free version of Corona SDK that can be used for small games with limited features. You can upgrade to other versions when needed.

Corona SDK provides a number of features which includes the following –

- Physics and Collision handling APIs
- Web and Network APIs
- Game Network API
- Ads API
- Analytics API
- Database and File System APIs
- Crypto and Math APIs
- Audio and Media APIs

And many more game engines found on the link below.

From: https://www.tutorialspoint.com/lua/lua_game_programing.htm

Roblox Video ex.

Thank you

References

- https://www.lua.org/manual/5.4/manual.html
- https://www.lua.org/pil/contents.html
- https://www.tutorialspoint.com/lua/index.htm
- https://www.bmc.com/blogs/lua-programming-language/#:~:text=Lua%20is%20not%20directly%20interpreted,on%20a%20multitude%20of%20devices.

Blue Page

Pink Page

White Page with Images

1. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua
2. Incididunt ut labore et dolore
3. Consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua
4. Incididunt ut labore et dolore

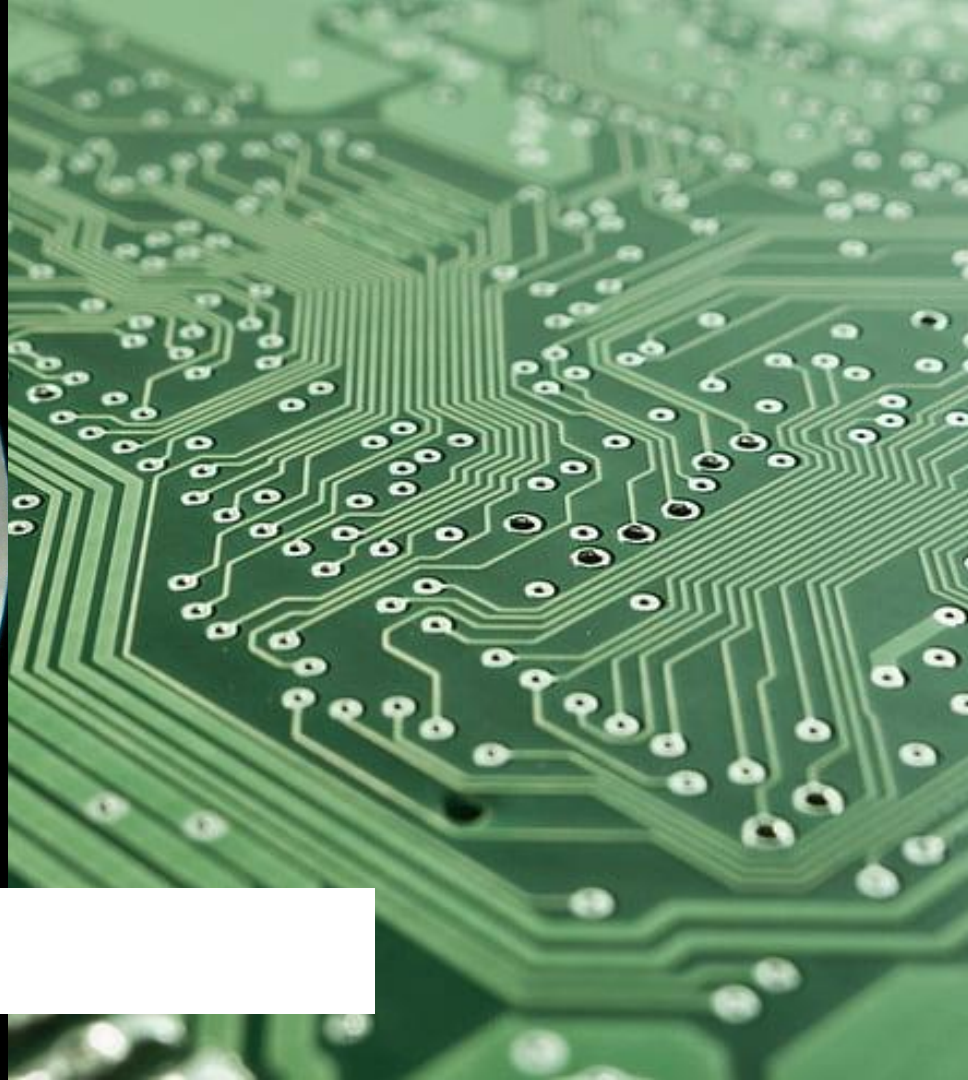
White Page

Column 1

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip.

Column 2

- Lorem ipsum dolor sit amet, consectetur adipiscing elit
- Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua



Text over photos page

Strings

```
--[[
"Strings"

\ Escape character goes before the
character that needs escaping or used for
\a   Bell
\b   Backspace
\f   Formfeed
\n   New line
\r   Carriage return
\t   Tab
\v   Vertical tab
\\   Backslash
\"   Double quotes
\'   Single quotes
[    Left square bracket
]    Right square bracket
```

```
String Manipulation:
1
string.upper(argument)

Returns a capitalized representation of the argument.

2
string.lower(argument)

Returns a lower case representation of the argument.

3
string.gsub(mainString,findString,replaceString)

Returns a string by replacing occurrences of findString with replaceString.

4
string.find(mainString,findString,

optionalStartIndex,optionalEndIndex)

Returns the start index and end index of the findString in the main string and nil if not
found.

5
string.reverse(arg)

Returns a string by reversing the characters of the passed string.
```

Strings

```
6
string.format(...)

Returns a formatted string.

7
string.char(arg) and string.byte(arg)

Returns internal numeric and character representations of input argument.

8
string.len(arg)

Returns a length of the passed string.

9
string.rep(string, n))

Returns a string by repeating the same string n number times.

10
..

Thus operator concatenates two strings.
]]
```

Strings

```lua
--String formatting
local str1, str2, str3 = "Lua", 'Lua', [[Lua]]
print("\n\n \"str1: \" ", str1, "\n") --Outputs
quotes around "Lua"
```

```
"str1: "        Lua
```

Strings

```lua
--String manip
--1-10
print(
"\n\n",string.upper(str2),"\n",
string.lower(str2),"\n",
string.gsub(str2, "Lua", [[luA]]),"\n",
string.find(str2, 'Lua'),"\n",
string.reverse(str2),"\n",
string.format("%s %s %s", str1, str2, str3),"\n",
string.byte(str1, 1),"\n", --Byte representation of the 1 or nth char
string.char(76),"\n",
string.len(str2),"\n",
string.rep(str3.." ", 3),"\n"
```

```
LUA
lua
luA
1
auL
Lua Lua Lua
76
L
3
Lua Lua Lua
```

Strings

```lua
--Strings are mutable
str1, str2, str3 = [[lu]], 'hello', "LUA"
print(str1, " ", str2, " ", str3,"\n\n")
```

```
lu hello LUA
```