# Lua: Fundamentals

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Wednesday, January 30, 2019

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

## Review
## PL Categories: Dynamic PLs

**Script**: program associated with a software package, used to automate or configure operations. Written in the package's **scripting language**.

Out of early shell scripting languages and associated text-processing tools grew the full-featured PL **Perl** (1987).

Similar PLs: **Python** (1991), **Lua** (1993), **Ruby** & **JavaScript** (1995). These are **dynamic programming languages**.

Typical Characteristics

- Dynamic type checking.
- Little text overhead in code.
- Just about everything is modifiable at runtime.
- High-level.
- A *batteries-included* approach.
- **Imperative** and block-structured, with support for OOP.
- Mostly interpreted, with compilation to byte code as an initial step.

## Review
## Introduction to Lua — History, Characteristics

The **Lua** PL originated in 1993 at the Pontifical Catholic University in Rio de Janeiro, Brazil.

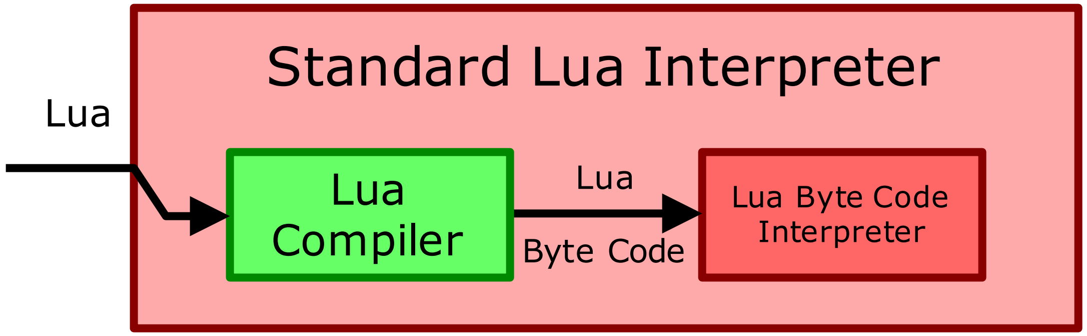
Lua's **source tree** is small, easy to include in other projects. It is a popular scripting language for games, LaTeX, Wikipedia, etc.

Characteristics

- Dynamic PL.
- Simple syntax. Very little **punctuation**. Small, versatile feature set.
- **Imperative**.
- Insulated from machine.
- Typing: **dynamic**, **implicit**. **Duck typing**.
- Only eight types: `number`, `string`, `boolean`, `table`, `function`, `nil`, `userdata`, `thread`.
- **First-class functions**.
- Function definitions are executable statements.
- Uses **eager** evaluation (opposite: **lazy** evaluation).

## Review
## Introduction to Lua — Build & Execution [1/2]

Lua is nearly always interpreted. The interpreter in the standard Lua distribution compiles Lua to **Lua byte code**, which is system-independent. This byte code is then interpreted directly by the runtime system.



Standard filename suffix for Lua source files: "`.lua`".

Standard Lua interpreter includes an **interactive environment**.

## Review
## Introduction to Lua — Build & Execution [2/2]

Three ways to specify an interpreter in a Unix-derived OS.

(1) Command line: *INTERPRETER SOURCE_FILE*

```
lua zzz.lua
```

(2) First line of source: **shebang** + *INTERPRETER_PATH*

- Requires knowing where the interpreter is. ☹

```
#!/usr/local/bin/lua
```

(3) First line of source: shebang + `/usr/bin/env` *INTERPRETER*

```
#!/usr/bin/env lua
```

## Lua: Fundamentals
## Overview

Subtopics

- Lexical structure.
- Variables, values, expressions.
- Functions.
- Tables.
  - Lua’s only built-in data structure: a hash table.
- Arrays.
  - Tables with keys 1, 2, 3, …
- Flow of control.

*The material for this topic is covered in a Lua source file, with lots of comments.*

*See* `fund.lua`.

## Lua: Fundamentals
## Lexical Structure — Comments

Like many other PLs, Lua has both single-line comments and multiline comments.

Single-line comments: “`--`” to end-of-line:

```
--This is a comment
```

Multiline comments: “`--[`” + zero or more equals-signs (=) + “`[`”. End with “`]`” + same number of equals-signs + “`]`”.

```
--[==[This is a
comment]==] this_is_not_a_comment = 3
```

## Lua: Fundamentals
## Lexical Structure — Identifiers

Lua has 21 **keywords** (words with special meaning):

```
and break do else elseif end false for function if in
local nil not or repeat return then true until while
```

A Lua **identifier** (the name of something) is like a "C" identifier: it begins with a letter or underscore (_), contains only letters, underscores, and digits, and is not a keyword.

A word that has the general form of an identifier, but is not legal as an identifier, is called a **reserved word**.

So, in Lua, the reserved words are the same as the keywords. This is true in many other PLs as well—but not in all PLs.

## Lua: Fundamentals
## Lexical Structure — String Literals

A **literal** is a bare value.

Two kinds of string literals:

- Quoted strings
- Multiline strings

Quoted strings use single or double quotes.

```
aa = "hi"
ba = 'ho'
```

Multiline strings use brackets & equals, like multiline comments.

```
cc = [===[Hello
there! Here is a quote mark inside a string: "]===]
```

## Lua: Fundamentals
## Lexical Structure — Whitespace

Separating lexemes with whitespace is allowed, but not required …

```
xyz=12     -- Same as xyz = 12
```

… except where it clearly matters.

```
do return  -- "do" keyword followed by "return" keyword
doreturn   -- Identifier
```

Newlines and indentation are not syntactically significant.

Newlines are treated the same as other whitespace, except:

- A newline ends a single-line comment
- A newline is illegal in a quoted string or before "`(`" in a function call.
- A newline represents itself inside a multiline string.

# Lua: Fundamentals
Variables, Values, Expressions

## Summary

- Only values have types; variables are references to values.
- Arithmetic expressions & comparisons are mostly as usual.
  - Inequality operator: "`~=`".
- **Multiple assignment**.
- Booleans: `true`, `false`, `and`, `or`, `not`.
- Function `print` does quick & dirty output. `io.write` is preferred.
- "`..`" operator does string concatenation, with automatic number-to-string conversion.
- Type errors are flagged at runtime, when statement is executed.

## Lua: Fundamentals
## Functions

Summary

- Main program is code at global scope.
- A function definition begins with the keyword `function`. This is an executable statement.
- Call functions as usual.
  - If passing a single table literal or string literal, then you may leave off the parentheses: `foo "abc"`
- A parameter that is not passed gets the value `nil`.
- Return values from functions as usual.
  - Multiple values can be returned. Capture these with multiple assignment.
- Variables inside a function default to global—except parameters & loop counters. Declare local variables using `local`.
- First-class functions.
- Also use keyword `function` to create an unnamed function.
- We can define a local function inside a function.

## Lua: Fundamentals
## Tables

### Summary

- Maps/dictionaries, arrays, objects, classes, modules implemented using a single PL feature: **table**, a key-value structure implemented internally as a hash table.
- Table literals use braces, entries separates by commas. Key-value pair is key in brackets, equals sign, value: `{ …, ["a"]=56, … }`
- Access values using brackets for index syntax, as in C++/Java.
- Delete a key from a table by setting the associated value to `nil`.
- Can mix types of keys, values.
- If a key looks like an identifier, then we can use dot syntax: `t["abc"]` and `t.abc` are the same.
- Can put functions in tables. There is syntax for declaring a function as a table member.
- Loop over all key-value pairs in a table with `pairs`.
- Tables are also used to implement operator overloading (not covered right now).

## Lua: Fundamentals
## Arrays

Summary

- When a table’s keys that are consecutive positive integers starting at one (not zero!), we call it an **array**.
- Array literal: list values in braces without keys. Indices start at one.
  `arr = { 7, "abc", fibo, 5.34 }`
- Length of array `arr`: `#arr`
- Loop over array items, in order, with `ipairs`.

# Lua: Fundamentals
# Flow of Control

## Summary

- `if` *COND* `then` *STMTS* `end`
- `if` *COND* `then` *STMTS* `else` *STMTS* `end`
- `if` *COND* `then` *STMTS* `elseif` *COND* `then` *STMTS* … `end`
  - "`else if`" is legal, but requires an extra "`end`", which "`elseif`" avoids.
- `while` *COND* `do` *STMTS* `end`
- `repeat` *STMTS* `until` *COND*

  No "end"
  - Like C/C++/Java `do … while`, except the condition is flipped.
- `for` *VAR*`=`*FIRST*`,` *LAST* `do` *STMTS* `end`
- `for` *VAR*`=`*FIRST*`,` *LAST*`,` *STEP* `do` *STMTS* `end`
- `break`: as in C/C++/Java (there is no "continue").
- Iterator-based for-in loop. Examples:
  - `for k, v in pairs(`*TABLE*`) do` *STMTS* `end`
  - `for k, v in ipairs(`*TABLE_AS_ARRAY*`) do` *STMTS* `end`

  We will eventually write our own iterators.
- Other (not covered right now): exceptions, coroutines, threads.