PL Category: Dynamic PLs Introduction to Lua

CS 331 Programming Languages Lecture Slides Monday, January 27, 2025

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

© 2017–2025 Glenn G. Chappell

Unit Overview Formal Languages & Grammars



Review

Backus-Naur Form (**BNF**) is a notation for writing CFGs.

BNF production for a digit, wrapped (incorrectly!) for lack of space:

Unlike our earlier CFG format, BNF is suitable for specifying programming language syntax.

- It allows for terminals to contain arbitrary characters.
- It does not *require* unusual characters (like our " \rightarrow " and " ε ").
- It is suitable for use as input to a computer program.
- It allows symbols to have descriptive names (e.g., <for-loop>), rather than just single letters.

Some variations on BNF are referred to as **Extended BNF** (EBNF).

phone_number area_code	=	[are "(" d	a_code igit d digit] digi igit di digit	it7 ; igit " "_")";	BNF is a single standard. EBNF refers to a family of similar	
UIGIC/	_	digit	digit	digit	digit	;	standards.	
digit	=	"0"	"1"	"2"	"3"	'' 4 ''	"5"	
		"6"	'' 7''	"8"	"9";			

EBNF typically includes two important features.

- Brackets [...] surround optional sections.
- Braces { ... } surround optional, repeatable sections.

Worth memorizing!

Topics

- Introduction to survey of programming languages
- ✓ PL feature: compilation & interpretation
 - PL category: dynamic PLs
 - Introduction to Lua
 - Lua: fundamentals
 - Lua: modules
 - Lua: objects
 - Lua: advanced flow

PL Category: Dynamic PLs

Entering commands one by one can be tedious. Decades ago, the idea of a file containing a "batch" (that is, a list) of commands was introduced. Such a file is a **batch file**.

Interactive Session curl -O http://b.us/dog.c Ş curl -0 http://b.us/cat.c curl -0 http://b.us/asp.c \$ \$ curl -0 http://b.us/bat.c

```
curl -0 http://b.us/eel.c
```

Batch File

- curl -O http://b.us/dog.c
- curl -0 http://b.us/cat.c
- curl -O http://b.us/asp.c
- curl -0 http://b.us/bat.c
- curl -0 http://b.us/eel.c

Command prompt



In a file. Just run the file.

Variables and control structures were added to the syntax for batch files. A program written in the resulting programming language is a **script**.

done

Batch File

- curl -0 http://b.us/dog.c
- curl -O http://b.us/cat.c
- curl -0 http://b.us/asp.c
- curl -0 http://b.us/bat.c
- curl -0 http://b.us/eel.c

Shell Script

for n in dog cat asp bat eel do

```
curl -0 http://b.us/$n.c
```

This particular script uses the syntax of the Bash shell.

A program that handles command entry is called a **shell**; a program interpreted by the shell is a **shell script**.

This idea turned out to be useful in many kinds of software packages, as a way to extend functionality, automate operations, and allow for customization.

When a software package becomes complex enough, its designers often allow for some of the tasks it performs to be automated through the use of **scripts**. The programming language in which these scripts are written is the package's **scripting language**.

Full-featured word processors, spreadsheets, and similarly complex software packages often allow for scripts. So do web pages, many games, and the windowing environments that form the front ends for major desktop operating systems. To improve on scripting languages, small, high-level textprocessing PLs appeared. One of these, written at Bell Labs in the 1970s, was **AWK**—after its authors, A. Aho, P. Weinberger, and B. Kernighan.

Below is an AWK script that treats the last word in each line of a file as a number and prints the sum of these numbers.

```
BEGIN { total = 0 }
{
    total += $NF
}
END { print "Total:", total }
```

1987 saw the release of **Perl**, a PL designed by Larry Wall and based on AWK, various shells, and other text-processing tools. While aimed at the same kinds of problems as these tools, Perl was a full-featured programming language, with sophisticated data structures and access to operating-system services.

Below is a Perl script that treats the last word in each line of a file as a number and prints the sum of these numbers.

```
$total = 0;
while (<>) {
    @_ = split;
    $total += $_[-1];
}
print "Total: $total\n";
```

Perl was soon used for many other tasks. For example, in the early days of the Web, Perl dominated server-side web programming.

It was an idea whose time had code. A number of similar PLs were released in the next few years: **Python** in 1991, **Lua** in 1993, and **Ruby**, **JavaScript**, and **PHP** in 1995.

These PLs continue to be used today. We call them **dynamic programming languages**. They are heavily used in web programming, and increasingly in scientific computing. To clarify:

- I refer to the programming-language category as dynamic programming languages.
- I use "scripting language" to describe a role a programming language can play in a software package.

For example, Lua is a dynamic programming language. It can be used as a scripting language for Wikipedia pages. A typical dynamic programming language has the following features/characteristics.

- **Dynamic typing** (types are determined and checked at runtime).
- Just about everything is modifiable at runtime.
 - For example, we might be able to create new class members at runtime.
- Little text overhead in code.
- A program starts by executing the code at global scope, not "main".
- High-level.
 - Programmers do not deal with resource management, access memory directly, or implement the details of data structures.
- A batteries-included approach.
 - For example, web access might be included in the standard library.
- Code is basically imperative (we tell the computer what to do, as in C++ and Java) and block-structured, with support for objectoriented programming.
- Implementations are mostly interpreters, with compilation to a byte code as an initial step. Native-code executable files are uncommon.

Below is the hello-world program in C++ again, along with equivalent *complete* programs in five dynamic PLs.

C++ #include <iostream> using std::cout; int main() { cout << "Hello, world!\n"</pre> }

Perl
print "Hello, world!\n";
Python
print("Hello, world!")

Lua io.write("Hello, world!\n")

Ruby puts "Hello, world!"

PHP
echo "Hello, world!\n";

Introduction to Lua

The **Lua** programming language originated in 1993 at the Pontifical Catholic University in Rio de Janeiro, Brazil. It was created in response to the strong trade barriers that Brazil had at the time, which made using software from other countries difficult. Its creation was led by Luiz Henrique de Figueiredo and Waldemar Celes, of the Computer Graphics Technology Group.

Lua was partly based on the existing programming language SOL (Simple Object Language). In Portuguese, *sol* means sun; *lua* means moon.

Lua continues to be actively developed. It is now freely available via the web, in robust implementations that are highly consistent across platforms. The standard Lua implementation is very **lightweight**: its **source tree**—the directory structure holding the source code for the various components of Lua—is unusually small, and executing Lua code is generally a low-cost operation.

Lua is mostly used as a scripting language, with the entire Lua source tree included in the source of some other software. Lua is a common scripting language for games—its first major use being in *World of Warcraft*. Lua scripts can also be executed within Wikipedia pages and the LaTeX typesetting system.

I estimate that, today, Lua is the fifth most popular dynamic PL, after Python, JavaScript, PHP, and Ruby. Lua gets less publicity than other PLs, because it is generally included as part of some other software package. Lua is often used, but few large projects are written *entirely* in Lua. Lua is a dynamic PL with a simple syntax. A small but versatile feature set supports most common programming paradigms: object-oriented programming, functional programming, etc.

Lua code is generally organized similarly to C++ & Java. Code is largely **imperative**: we write **statements** that tell the computer what to *do*. Code is encapsulated in functions and the equivalent of classes (but different terminology is used).

Lua programs are insulated from the machine on which they execute. They have no direct access to raw memory. The runtime system does all memory allocation and deallocation.

Lua was designed to interact with code written in other PLs. Various **foreign function interfaces** (**FFI**s) are available. Lua uses less **punctuation** than C++ & Java. It has no semicolons and fewer parentheses. Where C++ uses braces to delimit a **block**, Lua usually marks the end of a block with the keyword end. As for semicolons, Lua has a carefully designed grammar that makes end-of-statement markers unnecessary.

Some example Lua code that defines a function:

```
function fibo(n)
    local currfib, nextfib = 0, 1
    for i = 1, n do
        currfib, nextfib = nextfib, currfib + nextfib
    end
    return currfib
end
```

2025-01-27

Lua has **dynamic typing**: types are determined and checked at runtime.

Lua's typing is largely **implicit**: types do not need to be explicitly stated. Only values have types. Lua variables are merely references to values, and do not themselves have types.

n = 4 -- Set variable n to value of type number
n = "abc" -- Same variable set to value of type string

These two lines can be executed one right after the other.

Function calls are checked via **duck typing**: an argument may be passed to a function as long as the operations the function performs are defined on that argument. ("If it looks like a duck, swims like a duck, and quacks like a duck, then it's a duck.") Lua's type system includes exactly eight types:

- number—a floating-point number. Since version 5.3, Lua guarantees that some operations will produce exact whole-number answers.
- string
- boolean
- table—a hash table. Tables are the only nontrivial data structure. A table is versatile, functioning as map/dictionary, array, object, and the equivalent of a C++/Java class. Tables are also used to support operator overloading.
- function
- nil—a "nothing" type. The type of a nonexistent value.
- userdata—an opaque blob that Lua cannot look inside, used when Lua is an intermediary, passing data between code in other PLs.
- thread—a thread of execution.

Lua does not allow new types to be defined.

Lua has first-class functions.

A type is **first-class** if its values can be created, stored, operated on, and passed & returned with the same ease and facility as types like int in C++. Examples of types that are *not* first-class are functions and built-in arrays in C & C++.

So in Lua, a function is an ordinary value.

Definitions of functions and the equivalent of classes are executable statements in Lua. New functions can be defined at runtime. Indeed, functions can *only* be defined at runtime.

Like C++, Java, and Python, Lua does eager evaluation: an expression is evaluated when it is encountered during execution. (An alternative is lazy evaluation: an expression is evaluated only when its value is needed. More on this later in the course.) 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.



There are other Lua implementations, including **LuaJIT**, a Lua interpreter that uses a JIT compiler.

The standard Lua interpreter has an **interactive environment**, allowing statements to be typed in for immediate execution.



We can define functions and call them. Support for multi-line constructions may vary, depending on the environment in use.

2025-01-27

Lua programs can also be stored in files to be executed; the standard filename suffix is ".lua".

In an interactive environment, we can execute a Lua source file by passing the filename, enclosed in quotes, to function dofile.

> dofile("zzz.lua")

Lua is supported by many **IDEs** (Integrated Development Environments). **ZeroBrane Studio** is a Lua-specific IDE. On Unix-derived operating systems (MacOS, Linux distributions; I will say "***ix**"), there is a standard way to specify an interpreter for a program.

First, some background. At a *ix command line, I can execute a Lua program (say, "zzz.lua") by typing

lua zzz.lua

Above, "lua" is the name of the Lua interpreter, a program stored (on my machine) at /usr/local/bin/lua. When executing a file under *ix, we can specify an interpreter by starting the file with "#!", followed by The shebang line is not the path of the interpreter. This is the specific to Lua. (Indeed, sharp-bang or shebang line.

it does not really have much to do with Lua.)

#!/usr/local/bin/lua

I begin zzz.lua as above, and I set execute permission for the file.

When I execute this file, operating system sees the shebang, reads the path of the interpreter, and executes the interpreter with the filename of the program as an argument, just as if I had typed:

/usr/local/bin/lua zzz.lua

When the Lua interpreter executes the Lua code in the file, it knows to ignore a first line that begins with "#!".

The shebang line is useful, but it depends on the interpreter being in a specific directory.

To solve this problem, there is often a program called "env". Its job is to know where the interpreters are. File env is always in the directory /usr/bin. Now I can use the following first line.

#!/usr/bin/env lua

When I execute the file, it is as if I typed:

/usr/bin/env lua zzz.lua

Then the env program does:

/usr/local/bin/lua zzz.lua

TO DO

- Try out the Lua interactive environment.
- Write a hello-world program in Lua and execute it in various ways, including dofile, using a shebang line, and using an IDE.

Done. See hello.lua.

The **Fibonacci numbers** are the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ... Each entry is the sum of the previous two. 233 + 377 = 610

We can define the Fibonacci numbers formally using a recurrence:

$$F_0 = 0$$
; $F_1 = 1$; for $n \ge 2$, $F_n = F_{n-2} + F_{n-1}$.

I have written a Lua program that computes and prints Fibonacci numbers: fibo.lua.

TO DO

• Run fibo.lua.

See fibo.lua.

2025-01-27