PL Category: Dynamic PLs
Introduction to Lua

CS 331 Programming Languages

Lecture Slides

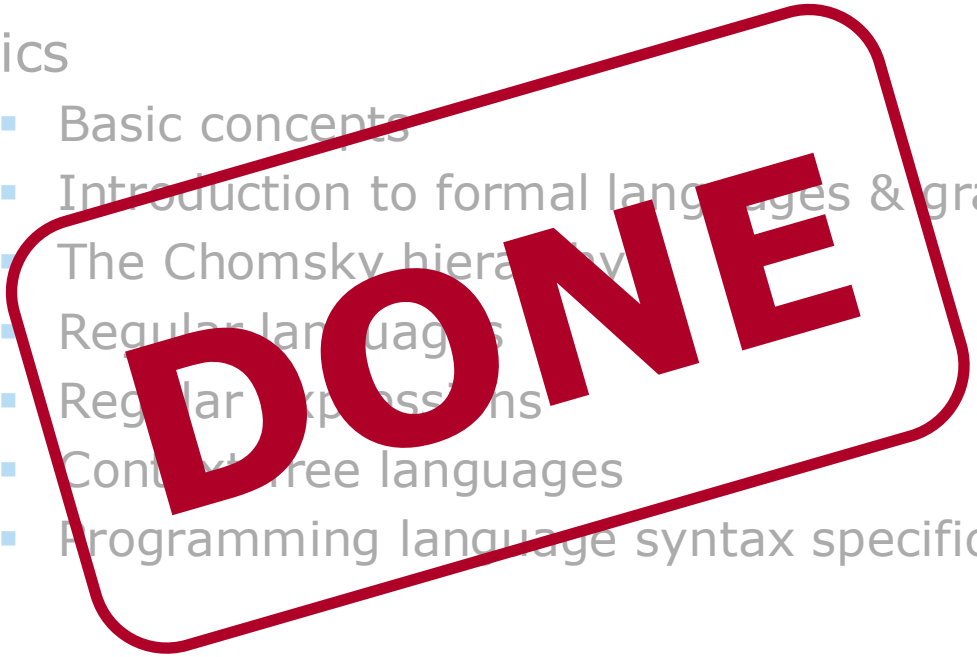
Monday, January 26, 2026

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

Unit Overview
Formal Languages & Grammars

Topics

- ✓ Basic concepts
- ✓ Introduction to formal languages & grammars
- ✓ The Chomsky hierarchy
- ✓ Regular languages
- ✓ Regular expressions
- ✓ Context-free languages
- ✓ Programming language syntax specification



Review

PL Syntax Specification — Backus-Naur Form

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

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

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5"
          | "6" | "7" | "8" | "9"
```

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 "→" 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.

Review
PL Syntax Specification — Extended Backus-Naur Form

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

```
phone_number = [ area_code ] digit7 ;
area_code    = "(" digit digit digit ")" ;
digit7       = digit digit digit "-"
               digit digit digit digit ;
digit        = "0" | "1" | "2" | "3" | "4" | "5"
             | "6" | "7" | "8" | "9" ;
```

BNF is a single standard. EBNF refers to a family of similar standards.

EBNF typically includes two important features.

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

Worth memorizing!

Unit Overview
The Lua Programming Language

Topics

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

PL Category: Dynamic PLs

PL Category: Dynamic PLs
Background [1/7]

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
$ curl -O http://b.us/cat.c
$ curl -O http://b.us/asp.c
$ curl -O http://b.us/bat.c
$ curl -O http://b.us/eel.c
```

Command prompt

Typed in separately.

Batch File

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

In a file. Just run the file.

PL Category: Dynamic PLs
Background [2/7]

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

Batch File

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

Shell Script

```
for n in dog cat asp bat eel
do
    curl -O http://b.us/$n.c
done
```

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**.

PL Category: Dynamic PLs
Background [3/7]

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.

PL Category: Dynamic PLs
Background [4/7]

To improve on scripting languages, small, high-level text-processing 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 }
```

PL Category: Dynamic PLs
Background [5/7]

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.

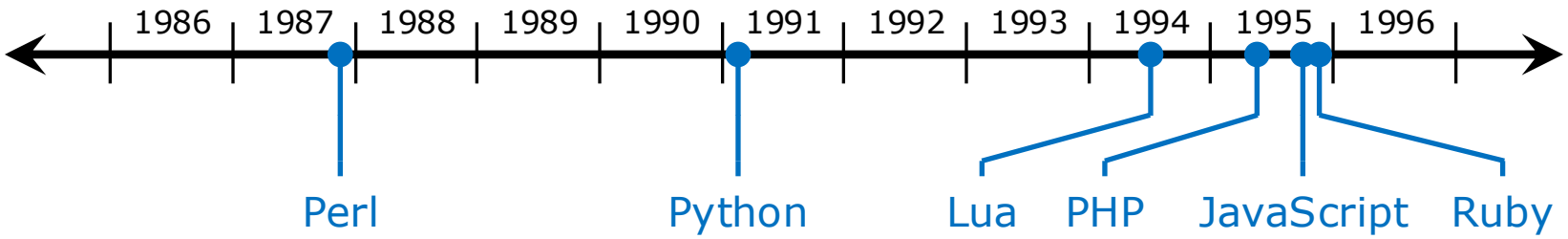
Below is a Perl script that treats the last word in each line of a file as a number and prints the sum of these numbers.

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

PL Category: Dynamic PLs
Background [6/7]

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 come. A number of similar PLs were released in the next few years: **Python** in 1991, **Lua** in 1994, and **PHP**, **JavaScript**, and **Ruby** 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.

PL Category: Dynamic PLs
Background [7/7]

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.

PL Category: Dynamic PLs
Typical Characteristics [1/2]

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 say *what to do*, as in Swift, C++, Java, and Python) and block-structured, with support for object-oriented programming.
- Implementations are mostly interpreters, with compilation to a byte code as an initial step. Native-code executable files are uncommon.

PL Category: Dynamic PLs
Typical Characteristics [2/2]

Below are hello-world programs in five dynamic PLs.

Perl

```
print "Hello, world!\n";
```

Python

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

Lua

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

PHP

```
echo "Hello, world!\n";
```

Ruby

```
puts "Hello, world!"
```

Introduction to Lua

Introduction to Lua
History [1/2]

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.

The first public Lua release came in 1994.

Today, Lua continues to be actively developed. It is now freely available via the web, in robust implementations that are highly consistent across platforms.

History [2/2]

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* in 2004. Lua scripts can also be executed within Wikipedia pages and the LaTeX documents.

I estimate that, today, Lua is the sixth most popular dynamic PL, after Python, JavaScript, Perl, 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.

Characteristics — Basics

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 Swift, C++, and Java. Code is largely **imperative**: we write **statements** that say *what to do*. Code is encapsulated in functions and the equivalent of classes.

What else could we do? Later in the semester, we will discuss **declarative** code, which says *what is true*.

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++, Swift, and Java. It has few semicolons and parentheses. Instead of using 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
```

Introduction to Lua
Characteristics — Type System [1/3]

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 in 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.

Lua 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.")

Introduction to Lua
Characteristics — Type System [2/3]

Lua's type system includes exactly eight types:

1. `number`—a floating-point number. Since v. 5.3, Lua guarantees that some operations will produce exact whole-number answers.
2. `string`
3. `boolean`
4. `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 Swift/C++/Java class. Tables are also used to support operator overloading.
5. `function`
6. `nil`—a "nothing" type. The type of a nonexistent value.
7. `userdata`—an opaque blob that Lua cannot look inside, used when Lua is an intermediary, passing data between code in other PLs.
8. `thread`—a thread of execution.

Lua does not allow new types to be defined.

Characteristics — Type System [3/3]

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 Swift and `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.

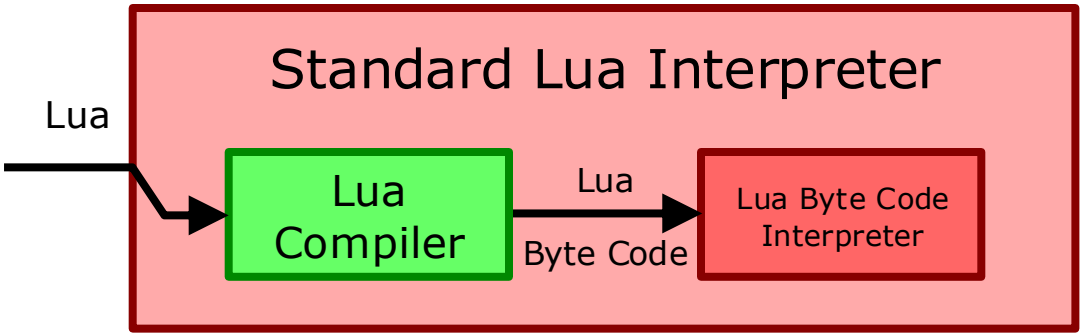
Definitions of functions and the equivalent of classes are executable statements in Lua. New functions can be defined at runtime. Indeed, functions can *only* be defined at runtime.

Like Swift, 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 semester.

Build & Execution — Basics [1/3]

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.

Introduction to Lua
Build & Execution — Basics [2/3]

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

```
> a = 3
> =a
3
> a = a+100
> =a
103
> for i = 1,10 do io.write(" ", i)
>> end io.write("\n")
 1 2 3 4 5 6 7 8 9 10
```

This is an interactive session. Text `like this` is printed by the system.

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

Build & Execution — Basics [3/3]

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.

Build & Execution — The Shebang Convention [1/3]

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/bin/lua`.

Build & Execution — The Shebang Convention [2/3]

When executing a file under *ix, we can specify an interpreter by starting the file with "`#!`", followed by the path of the interpreter. This is the **sharp-bang** or **shebang** line.

The shebang line is not specific to Lua. (Indeed, it does not really have much to do with Lua.)

```
#!/usr/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/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 "`#!`".

Build & Execution — The Shebang Convention [3/3]

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/bin/lua zzz.lua
```

Some Programming [1/2]

TO DO

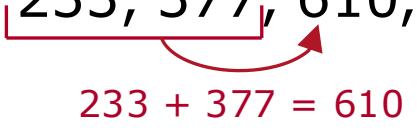
- Try out the Lua interactive environment.
- Write a hello-world program in Lua and execute it in various ways, including `dofile`, using a shebang line, and using an IDE.

Done. See `hello.lua`*.*

Some Programming [2/2]

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.

We can define the Fibonacci numbers formally using a recurrence:

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

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

TO DO

- Run `fibo.lua`.

Done. See `fibo.lua`.