PL Category: Dynamic PLs
Introduction to Lua

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Monday, January 28, 2019

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

Review
Grammars in Practice — Backus-Naur Form

In **Backus-Naur Form** (**BNF**), a notation for writing CFGs:

- Nonterminals are enclosed in angle brackets: (`<for-loop>`). Inside the angle brackets, the name of the nonterminal must begin with a letter and contain only letters, digits, and hyphens (`-`).
- The start symbol can vary.
- Terminals are enclosed in quotes: double (`"x"`) or single (`'"'`).
- Our arrow is replaced by colon-colon-equals: (`::=`).
- The vertical bar (|) is used the same way we use it.
- Epsilon (ε) is not needed.
- Space between nonterminals, terminals, etc., is ignored.

Here is a BNF production for a digit. (Stricly speaking, it is not correct BNF, since a BNF production must lie on a single line.)

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

Review
Grammars in Practice — Extended Backus-Naur Form

Some variations on BNF are referred to as **Extended BNF** (**EBNF**). Most versions of EBNF include the following.

- Nonterminals are not enclosed in angle brackets.
- The "`::=`" is replaced by something shorter, generally "`=`" or "`:`".
- A production ends with a semicolon (`;`) and may use multiple lines.
- Parentheses may be used for grouping.
- Braces `{` … `}` surround optional, repeatable sections.
- Brackets `[` … `]` surround optional sections.

Important!

```
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";
```

Review
Grammars in Practice — Programming-Language Syntax

It is common for **lexical structure** to be specified separately from overall syntax—using a separate grammar, or using some other specification method.

When this is done, a grammar for the overall syntax will have two kinds of terminals:

- Quoted strings indicating exactly what characters must appear.
- Categories of lexemes from the lexical-structure specification.

There must be some way of distinguishing the second kind of terminal from a nonterminal. One common method is to place lexeme-category terminals in ALL UPPER CASE.

```
assign_stmt = IDENTIFIER assign_op expression ";";
assign_op   = "=" | "+=" | "-=" | "*=" | "/=" | "%=";
expression  = …
```

Review
PL Feature: Execution I — Runtime

When a program is **executed**, the computations that it specifies actually occur. The time during which a program is being executed is **runtime**.

An implementation of a PL will include a **runtime system** (often simply **runtime**): code that assists in, or sometimes performs, execution of a program.

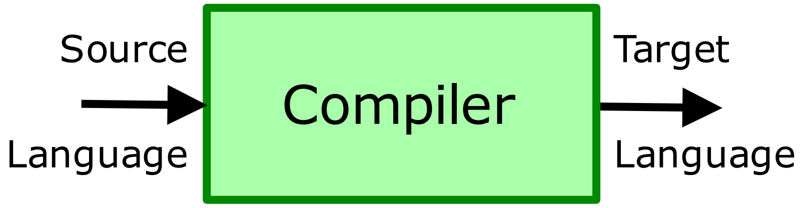
Example services that might be provided by a runtime system:

- **Memory management**.
- Low-level portion of I/O.

Some execution methods never create an executable file or any machine language at all. In such cases, the runtime system will be a separate program that handles all code execution.

Review
PL Feature: Execution I — Compilation [1/2]

A **compiler** takes code in one PL (the **source language**) and transforms it into code in another PL (the **target language**); the compiler is said to **target** this second PL.



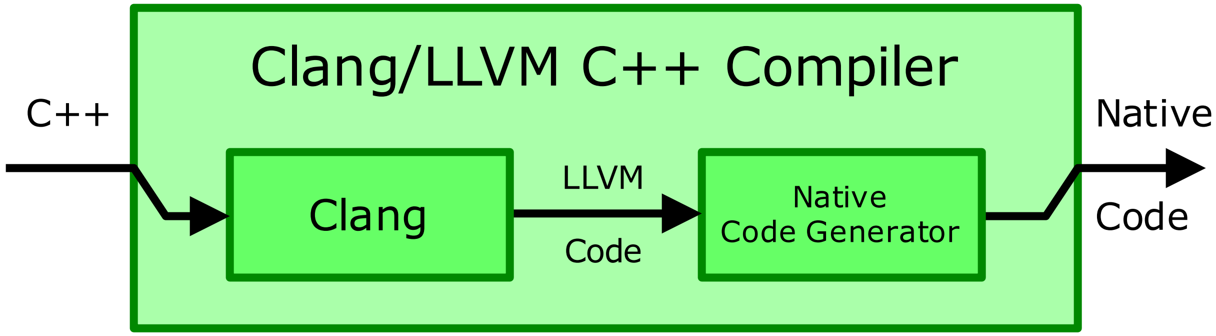
Compilers often target **native code** or a **byte** code.

In practice, we usually only use the term “compiler” when all of the following are true.

- The source and target languages differ significantly.
- The target language is lower-level than the source.
- The transformation is done with execution as the goal.

Review
PL Feature: Execution I — Compilation [2/2]

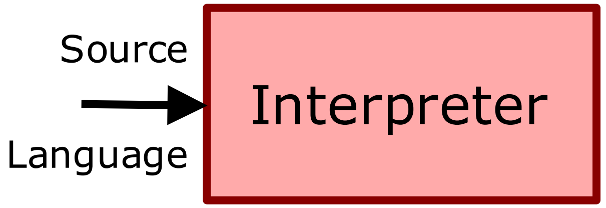
Good compilers proceed in a number of distinct steps. Code is transformed into an **intermediate representation** (**IR**), which is then be transformed into the ultimate target language.



Arrangements like the above have many advantages, including making it easier to support new source languages and platforms.

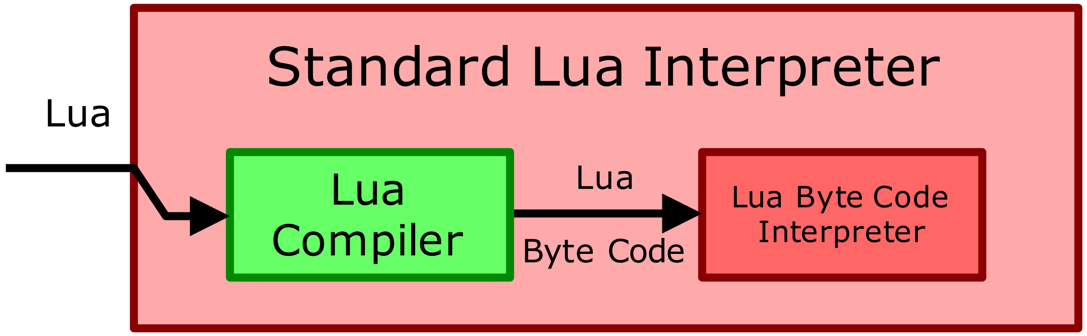
Review
PL Feature: Execution I — Interpretation

An **interpreter** takes code in some PL and executes it.



Two common misconceptions about interpretation:

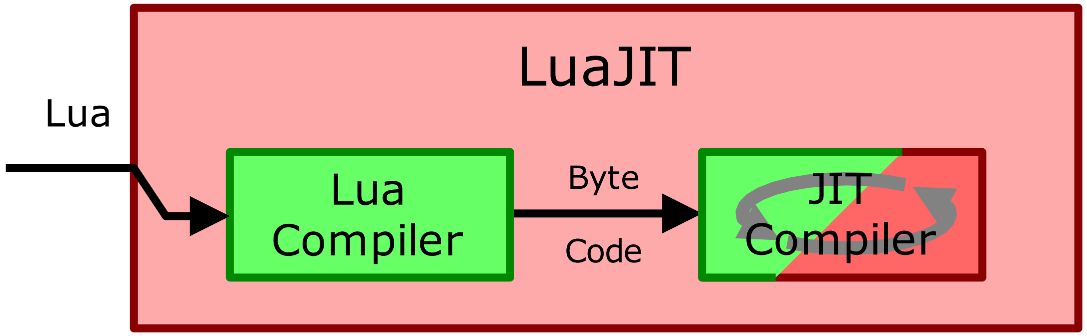
- Interpretation is inherent to a PL.
- Compilation and interpretation are completely separate notions.



Review
PL Feature: Execution I — JIT Compilation

In **Just-In-Time** (**JIT**) compilation, code is compiled at runtime.

A typical strategy is to do static compilation of source code into a byte code. Then execution begins, with the byte code being JIT compiled to native code.



Because the compilation is done at runtime, information that is only available at runtime can be used—for example, which parts of the code execution spends the most time in.

PL Category: Dynamic PLs Background [1/3]

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 computer programs called **scripts**. The PL in which these scripts are written is the package’s **scripting language**.

Full-featured word processors and spreadsheets often allow for scripts. So do many games. Web pages allow for scripts written in the JavaScript programming language.

The first scripts were aimed at automating tasks that had previously been done by typing in commands at a command prompt. These were **shell scripts**.

To improve on some of the more annoying aspects of early scripting languages, various small, high-level text-processing PLs appeared. One of the foremost was **AWK**, developed at Bell Labs in the 1970s.

PL Category: Dynamic PLs Background [2/3]

In 1987 the quality of PLs available for scripting tasks rose significantly with the release of **Perl**, a PL designed by Larry Wall and based on AWK, various shell scripting languages, and other text-processing tools. While aimed at solving the same kinds of problems as these tools, Perl differed from them in that it was a full-featured programming language, with sophisticated data structures and access to operating-system features.

Perl was soon used for 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 developed in the next few years: **Python** in 1991, **Lua** in 1993, and **Ruby** and **JavaScript** in 1995.

These PLs continue to be used today. They are called **dynamic programming languages**. They are heavily used in web programming, and increasingly in scientific computing.

PL Category: Dynamic PLs Background [3/3]

To Clarify

- I refer to the programming-language category as **dynamic programming languages**.
- “**Scripting language**” is 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 type checking.
- Little text overhead in code.
- Just about everything is modifiable at runtime.
  - We might be able to add new members to classes at runtime.
- 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 object-oriented programming.
- Implementations are mostly interpreters, with compilation to a byte code as an initial step. Compilation to native code is uncommon.

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

Below is the *Hello World* program in C++ again, along with equivalent *complete* programs in five dynamic PLs.

C++

```
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    cout << "Hello, world!"
         << endl;
    return 0;
}
```

Perl

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

Python

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

Lua

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

Ruby

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

Falcon

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

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. The effort 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). *Sol* means sun in Portuguese; *lua* means moon.

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

Introduction to Lua
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 for various software products, with entire Lua source tree being included in the source tree of the relevant product. It has become the standard scripting language for a number of games—most notably *World of Warcraft*. Lua scripts can also be executed as part of Wikipedia pages and within the LaTeX typesetting system.

I estimate that, today, Lua is the fifth most popular dynamic PL, after JavaScript, Python, Ruby, and Perl. Lua gets less publicity than the others, because it is generally included as part of some other software package. Lua is heavily used, but few large projects are written *entirely* in Lua.

Introduction to 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 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 (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 is aware of the format of data structures; it 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.

Introduction to Lua
Characteristics — Code Structure

Lua uses much less **punctuation** than C++ & Java; the braces and semicolons that litter C++ code are absent in Lua, and fewer parentheses are required. Where C++ uses braces to delimit a **block**, Lua 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:

```lua
function fibo(n)
  local a, b = 0, 1  -- Curr & next Fibonacci number
  for i = 1, n do
    a, b = b, a+b    -- Advance a, b as much as needed
  end
  return a           -- a is now our answer
end
```

Introduction to Lua
Characteristics — Type System [1/4]

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

Lua's typing is largely **implicit**: types do not need to be explicitly stated. Types are applied only to values. 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
```

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 [2/4]

From the Lua Tutorial reading:

> This is called *dynamic typing*. This means that you don't have to specify what type a variable is.

I like this tutorial, but **the second sentence above is false**.

Lua *does* use dynamic typing. This means that types are determined and checked at runtime. Lua’s typing is also **implicit**, which means that you do not have to specify types—of variables, for example.

But dynamic typing and implicit typing are not the same thing.

Introduction to Lua
Characteristics — Type System [3/4]

Lua's type system includes only eight types:

- `number` — a floating-point number. Since version 5.3, Lua makes guarantees that some operations will always 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, array, object, and the equivalent of a C++/Java class. Tables are also used to support operator overloading.
- `function`
- `nil` — a "nothing" type.
- `userdata` — an opaque blob that Lua code cannot access. These are used when Lua code is an intermediary, passing data between two functions written in some other PL.
- `thread` — a thread of execution.

Introduction to Lua
Characteristics — Type System [4/4]

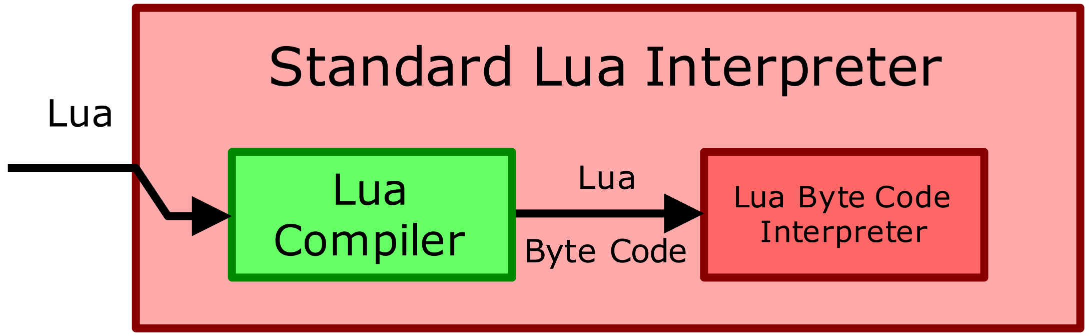
Lua has **first-class functions**. A type is **first-class** if its values can be created, stored, operated on, and passed/returned with the same ease and facility as types like `int` in C++. So in Lua, a function is an ordinary value. Examples of types that are *not* first-class are functions and built-in arrays in C.

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, Lua evaluates expressions in an **eager** manner: an expression is evaluated when it is encountered. (The opposite is **lazy** evaluation, in which an expression is only evaluated when its value is needed; we will see this later in the semester.)

Introduction to Lua
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 is other Lua implementations, including **LuaJIT**, a JIT compiler for Lua (5.2, currently).

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 = 1,3 do print(i) end
1
2
3
```

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

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

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

```
> dofile("zzz.lua")
```

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

On Unix-derived operating systems (I will say "***ix**"), there is a standard convention for specifying an interpreter for a program.

On the *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`.

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

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

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

I begin `zzz.lua` as above, and I set the execute permission for the file. When I execute this file, the shell 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 "`#!`".

Introduction to Lua
Build & Execution — The Shebang Convention [3/3]

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

To solve this, there is a program called "`env`". Its job is to know where the interpreters are. File `env` should always be 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
```

Introduction to Lua
Some Programming [1/2]

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

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \underbrace{34, 55}_{34 + 55 = 89}, 89, 144, 233, 377, 610, \ldots$$

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 simple example Lua program that computes and prints Fibonacci numbers.

See `fibo.lua`.

Introduction to Lua
Some Programming [2/2]

From the interactive Lua prompt, we can write multi-line constructions, define functions, and call them.

Below, we create a **table** and look at its values:

```
> t = {}  -- Empty table
> t[2] = "dog"
> t["cat"] = 7
> for k,v in pairs(t) do
>>   io.write(k.." "..v.."\n")
>> end
```