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

- 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.
- Whitespace between nonterminals, terminals, etc., is ignored.

Here is a BNF production specifying what a digit is.
- It is shown on two lines due to lack of space.

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5"
           | "6" | "7" | "8" | "9"
Some variations on BNF are referred to as **Extended BNF (EBNF)**. Most versions of EBNF include the following differences.

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

```plaintext
phone_number = [ area_code ] digit7;
area_code = "(" digit digit digit digit ")";
digit7 = digit digit digit "-"
          digit digit digit digit;
digit = "0" | "1" | "2" | "3" | "4" | "5"
          | "6" | "7" | "8" | "9";
```
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.
- Classes 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-class terminals in **ALL UPPER CASE**.

```plaintext
assign_stmt = IDENTIFIER assign_op expression ";;";
assign_op   = "=" | "+=" | "-=" | "+=" | "/=" | "/=";
expression   = ...
```
When a program is **executed**, the computations 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 service provided by a runtime system: memory management.

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

A compiler might target **native code** or a **byte code** (or any other PL).

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

Arrangements like the above make it much easier to support new PLs and platforms.
An **interpreter** takes code in some PL and executes it.

Two common misconceptions about interpretation:
- That interpretation is inherent to a PL.
- That compilation and interpretation are completely separate things.
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 being 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).
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 usually allow for scripts. So do many games. Web pages allow for scripts written in the JavaScript PL.

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.
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 shells 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, access to operating-system features, etc.

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 released 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.
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.
  - E.g., new members can be added 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.
Below is our *Hello World* program in C++ again, along with equivalent *complete* programs in five dynamic PLs.

**C++**
```cpp
#include <iostream>
using std::cout;
using std::endl;

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

**Perl**
```perl
print "Hello, world!\n";
```

**Python**
```python
print("Hello, world!")
```

**Ruby**
```ruby
puts "Hello, world!"
```

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

**Falcon**
```falcon
printl("Hello, world!")
```
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.
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 TeX 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.
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. Lua programs do not have direct access to raw memory. The runtime system is aware of the format of data structures, and it does all memory allocation and deallocation. Programmers mostly do not deal with resource-management issues.
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:

```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
```
Lua uses **dynamic typing**: types are determined and checked at runtime. Its 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.”)
Lua has 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. More on these later.
Lua has **first-class functions**. Generally speaking, a type is first-class if its values can be tossed around with the same ease and facility as types like `int` in C++. More formally, a type is **first-class** if its values can be used in each of the following ways.

- New values can be created from existing values at runtime.
- Values can be passed as arguments to functions and returned from functions.
- Values can be stored in containers.

So we can do the above with functions 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, Lua evaluates expressions in a **strict** (or **eager**) manner: an expression is evaluated when it is encountered.
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 also **LuaJIT**, a JIT compiler for Lua (currently version 5.1).
Lua programs may be stored in files to be executed; the standard filename suffix is “.lua”.

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

```lua
> a = 3
> =a
3
> a = a+100
> =a
103
> for i = 1,3 do print(i) end
1
2
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/bin/lua.
When executing a file under most *ix shells, we specify an interpreter by starting the file with “#!”, followed by the path of the interpreter. This is the **sharp-bang** or **shebang** convention.

```bash
#!/usr/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:

```bash
/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 “#!”.
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.

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

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

```
/usr/bin/env lua zzz.lua
```

Then env program does:

```
/usr/bin/lua zzz.lua
```
I have written a simple example Lua program that computes Fibonacci numbers. See fibo.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.

We can define the Fibonacci numbers formally using a recurrence:

\[ F_0 = 0; \quad F_1 = 1; \quad \text{for } n \geq 2, \quad F_n = F_{n-2} + F_{n-1}. \]
The material for this topic is covered mostly in a Lua source file, which is extensively commented.  See progl.lua.

Summary

- Main program is code at global scope.
- String literals use double quotes, single quotes, or double brackets with a balanced number of equals signs separating each pair of brackets: "abc" and [===[abc]=] are the same.
- Only values have types; variables are references to values.
- Arithmetic expressions are as usual.
- **Multiple assignment.**
- Eliminate a variable by setting it to nil.
- 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: Programming I

Functions

Summary

- A function definition begins with the keyword `function`.
- Variables default to global—except parameters, loop counter.
- Newlines are irrelevant.
- Call functions as usual.
- First-class functions.
- Also use keyword `function` to create an unnamed function.
Summary

- Maps/dictionaries, arrays, objects, classes 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 braces for index syntax, as in C++/Java.
- Delete a key by setting 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.
- Make an array by listing values in braces without keys. Indices start at one. `arr = { 7, "abc", fibo, 5.34 }`
- Length of array `arr`: `#arr`