Recall: When a program is **executed**, the computations it specifies actually occur.

In all cases, something **drives** the execution; there is some **task** the computer is attempting to perform. And there is some **strategy** for carrying out the execution. Both of these can vary significantly from one PL to another.
What drives the execution of a **Lua** program? Answer: the code at global scope in a source file is executed.

Strategy: The computer is given a sequence of commands. It carries out each, in order, performing the action that each specifies. If a command specifies that another function is to be called, then this becomes a **subtask** to be completed as part of the main task.

Answers are similar for C++, Java, and Forth — **imperative** PLs.
Scheme code does not consist of commands to be carried out, but rather expressions to be evaluated. Scheme is a more declarative PL than those mentioned earlier. We do not tell the computer what to do, but what the values of things are.

What drives Scheme execution is a procedure call.

Strategy: Evaluate the expression that forms the body of this procedure and return its value. If this expression involves calls to other procedures, then those evaluations are carried out as subtasks. In addition, if the evaluation of any of these expressions produces side effects, then these are performed.
Like Scheme, **Haskell** code does not consist of commands to be carried out, but rather **expressions** to be **evaluated**. Haskell is very much a **declarative** PL.

What drives Haskell execution is a function call — to `Main.main` or some other function.

**Strategy:** As with Scheme, evaluate the expression that forms the body of this function. If this expression involves calls to other procedures, then those evaluations are carried out as subtasks. Haskell is a **pure** PL — it has no side effects — so Haskell execution consists **entirely** of evaluation.
**Prolog** is another declarative language. Instead of telling it values, we tell it **what is true**. Given a **query** (essentially a question—yes/no or “What ... ?”) Prolog attempts to unify the query with something in its knowledge base (program).

Prolog execution is driven by a query.

The execution strategy involves **unification**.

To **unify** two constructions means to make them the same by setting values of variables as necessary. Unification, in full generality, includes expression evaluation as a special case. But unification can also do things that evaluation cannot do.

Can we unify \([A, 6]\) and \([4, B]\)? Yes. Set A to 4, B to 6.
We can start programming in Prolog by creating a source file (SOMETHING.pro) holding facts. Here are some facts.

```prolog
is_even(2). % This is a comment
is_child_of(alice, bill). % Alice is a child of Bill
```

Note the period at the end of each fact, above.

Execute Prolog code with a query at the “?-” prompt. Examples:

?- is_even(2).
?- is_even(N).
?- is_child_of(alice, X).
?- if_child_of(alice, bob).
At the interactive prompt, consult (load & compile) file \texttt{abc.pro} with "['abc.pro']." or "[abc].".

\[\texttt{?- [abc].}\]

**TO DO**

- Write some facts in a file.
- Consult the file, and do some queries.

\textit{Done. See simple.pro.}
In Prolog, **facts** and **rules** go in the source file. **Queries** are entered interactively.

Comments: single-line `% ...` end-of-line. Multiline: `/* ... */`

Prolog source code is free-form. Indentation is not significant. Between lexemes, space is mostly optional, and blanks are treated the same as newlines.

Facts, rules, and queries all end with a period ( . ).
The primary building block of Prolog code is the **term**. There are a number of kinds of terms, including **atoms**, **numeric literals**, **variables**, and **compound terms**.

A Prolog **atom** is one of the following:

- A nonempty sequence of letters, digits, and/or underscores, beginning with a lower-case letter.
- The above with a backslash (\) prepended.
- A sequence of one or more of the following 17 special characters: #\$&*_-./:<=>?@\^~
- Arbitrary characters enclosed in single quotes (’ … ’)
  - The usual backslash escapes are available: \’ \n \\n  - If the string inside the quotes forms a legal atom beginning with a lower-case letter, then having the quotes is the same as leaving them off. So `abc_d` and `'abc_d'` are the same.
Prolog numeric literals include integer and floating-point literals.

Integer literals are as usual: a nonempty sequence of digits with an optional minus sign prepended.

```
123   -42
```

Floating-point literals must contain a dot followed by a digit. An optional exponent at the end is $e$ or $E$ followed by an integer.

```
-3.0   5.2e-14
```
A Prolog **variable** is a nonempty sequence of letters, digits, and/or underscores, beginning with an *upper-case letter or underscore*.

A variable with a value is said to be **bound**; other variables are **free**. Prolog makes a strong distinction between these two. In some situations an argument must be a free variable; in other situations an argument cannot be a free variable, but may be a bound variable.
A **compound term** has an atom followed by a parenthesized, comma-separated list. No space is permitted before the opening parenthesis.

```
foo('ab cd', 14.2, QQQ)
```
A **fact** is an atom or compound term followed by a period. A fact says that something is true.

```
abcd.
defg(a, X, 28).
```

A fact is generally legal as a **query** also. A query asks whether something is true, and, if so, what variable values make it true.

A more complex query can be formed as a comma-separated list. This asks whether all items are true, and, if so, what variable values make them all true. Items are dealt with in the order given, backtracking and retrying as necessary.

```
eats(cat, X), eats(X, grain).
```

Any variable that is bound in a term remains bound to the same value in all later terms.
A rule looks like a fact, then “:-”, then something like a query. Before the “:-” is the head; after is the body.

\[
is_{\text{grandchild}}(A, B) \leftarrow \text{is}_{\text{child}}(A, X), \text{is}_{\text{child}}(X, B).
\]

Once again, in a rule, bound variables remain bound to the same values in all later terms.
A query sets up a **goal**: show that the query is true, by unifying is with something known. If it is shown to be true, then it **succeeds**; otherwise, it **fails**.

If goal processing involves attempted unification with the head of a rule, then the body of the rule forms a **subgoal**.

**TO DO**
- Add rules to the source file.
- Do some more queries.

*Done. See simple.pro.*
Prolog will do numerical computation, but this is not part of normal unification. For example, the query “\(X = 3+5\).” will give the result “\(X = 3+5\)”, not “\(X = 8\)”. To evaluate a numerical expression, use “is”. The expression is the second argument, and it must not contain any free variables.

?- is(X, 3+5).
\(X = 8\)

“is” can be used as an infix operator.

?- X is 3+5.
\(X = 8\).
?- 8 is 3+5.
yes
Arithmetic operators +, -, * are as usual. Division is / for floating-point division and // for integer division. ** does floating-point exponentiation.

?- X is 2 ** 3.
X = 8.0

“mod” is an infix operator

?- X is 11 mod 4.
X = 3
The following use function-call notation: \textit{sqrt exp log sin cos tan acos atan ceiling floor inc dec} (and others).

?- X is sqrt(5).
\begin{verbatim}
X = 2.2360679774997898
\end{verbatim}

2-parameter functions include \textit{min, max}.

?- X is max(5, 8).
\begin{verbatim}
X = 8
\end{verbatim}
Numeric comparison operators are 2-parameter predicates, usable either with function-call notation or as infix operators. Both operands can be expressions.

Ordered comparison operators are as usual, except that they shouldn’t look like arrows (so $\leq$ becomes $\leq\leq$).

?- 3+5 $\leq\leq$ 9+1.

yes

Numeric equality & inequality: $=:=$ $\not=\not=$

Operators $==$ $\not==$ exist, but do something different. Avoid these, unless you are sure they do what you want!

Lastly, operators $=$ $\not=$ mean unifiable and not unifiable, respectively.
TO DO

Write a Prolog predicate that computes the greatest common divisor (GCD) of two integers, using the Euclidean Algorithm.

Done. See simple.pro.