PL Category: Logic PLs
Introduction to Prolog

CS 331 Programming Languages

Lecture Slides

Wednesday, April 8, 2026

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

Unit Overview
The Prolog Programming Language

Topics

- ✓ PL feature: execution model
- PL category: logic PLs
- Introduction to Prolog
- Prolog: simple programming
- Prolog: lists
- Prolog: flow of control
- Prolog: interaction

Review

Review
PL Feature: Execution Model

There is always something that drives the execution of a program.

- There is some **task** the computer is attempting to perform.
- There is some **strategy** for carrying out the execution.

In Lua and Swift, the task is executing the code at global scope. The strategy is to carry out the commands in this scope. If functions are called, these become **subtasks**.

To **unify** two constructions means to make them the same by **binding** variables as necessary. For example, we can unify [A, 6] and [4, B] by setting A to 4, and B to 6.

In the **Prolog** programming language (covered next), execution is driven by the task of answering some **query**. The strategy involves unification.

PL Category: Logic PLs

PL Category: Logic PLs
Background [1/4]

In the late 1960s, Stanford researcher Cordell Green proposed representing a computer program in terms of logical statements.

This programming paradigm is known as **logic programming**.

In logic programming, a computer program can be thought of as a **knowledge base**. It typically contains two kinds of knowledge.

- **Facts.** Statements that are known to be true.
- **Rules.** Ways to find other true statements from those known.

Execution is then driven by a **query**: essentially a question. The computer attempts to answer the question using facts and rules.

Logic programming generally has no notion of **falsehood**. There are statements the computer can prove to be true, and others that it cannot. But some of those others might be true. Logic programming is thus not so much about **truth** as **provability**.

PL Category: Logic PLs
Background [2/4]

Here are some facts.

- Gil is a child of Ernest.
- Glenn is a child of Gil.

Recall: a variable is **free** if it is not bound to a value.

Here is a rule. (*A*, *B*, and *C* are free variables.)

- If *A* is a child of *B*, and *B* is a child of C, then *A* is a grandchild of *C*.

Here are some queries.

1. Is Glenn a grandchild of Ernest?
2. Is Ernest a grandchild of Glenn?
3. Is Glenn a grandchild of Bob?

The answer to query #1 is "yes". This is provable.

It is not provable what the answers to queries #2 and #3 are.

These are real people, and the facts and the rule are actually true.

In terms of *truth*, query #2 gets "no", while query #3 gets "yes".

So truth and provability are different.

PL Category: Logic PLs
Background [3/4]

In the 1970s, programming languages based on logic-programming concepts began to appear. These are **logic programming languages**. Foremost among those was—and still is—**Prolog**. Others include **Mercury** and **Gödel**.

Designing a practical PL based *solely* on logic has turned out to be somewhere in the range from difficult to impossible. Logic PLs nearly always "cheat" by including constructions that are not logical in nature.

Some people would say that the effort to create a true logic-based programming language has failed. I would say that the effort was worthwhile, but what resulted was not what was originally planned. *I will have more to say about this later.*

PL Category: Logic PLs
Background [4/4]

Logic programming may be offered as a library—if a PL supports the necessary constructions well.

Mainstream PLs with sufficient support have been rare until recently. But there are now a number of actively maintained logic-programming libraries for various mainstream PLs.

In particular, several logic-programming packages for the Python programming language appear to be well spoken of (but I have no experience with any of them). One also sees logic-programming libraries for various Lisp-family PLs.

PL Category: Logic PLs
Typical Characteristics

In a typical logic programming language:

- A program consists of facts and rules—or something very similar.
- Execution begins with a query, which establishes a **goal**: proving the query true—or determining how to make it true. During execution, various **subgoals** may be established.
- Execution is primarily interactive, typically based on a source file.
- Execution is about finding what can be proven—*not* what is true.
  - So **negation** says something fails to be provable, not that it is false.
- **Unification** is the primary proof tool.
- Typing is dynamic and implicit.

Introduction to Prolog

Introduction to Prolog
History [1/2]

In the early 1970s, a group at the U. of Aix-Marseille (France), led by Alain Colmeraur, began an effort to produce a logic programming language, starting from Cordell Green's ideas.

Their PL, first released in 1972, was called **Prolog**, short for "*PROgrammation en LOGique*" (French: "programming in logic").

Prolog was, and continues to be, the most important logic PL.

There was a flurry of interest in Prolog in the 1970s. This has mostly faded, but an active—although relatively small—community remains.

Introduction to Prolog
History [2/2]

In 1987, a University of Amsterdam group released **SWI-Prolog**. (SWI = *Sociaal-Wetenschappelijke Informatica*, Dutch: informatics for social science). This Prolog implementation is available free on all major platforms and actively maintained.

In 1995, an ISO standard for Prolog was released. In 2000, corrections and a module extension were published. SWI-Prolog now mostly follows the ISO standard.

In 1996, work began on a free ISO Prolog implementation under the auspices of the GNU Project: **gprolog**.

Proprietary varieties of Prolog also exist. The most successful is **Visual Prolog**. Unlike most versions of Prolog, Visual Prolog has static typing and support for object-oriented programming.

Characteristics — General

Prolog is a logic programming language.

Prolog execution is driven by a query, which establishes a goal.

The primary execution strategy involves unification.

Prolog attempts to unify using **backtracking search**.

Prolog has some support for reflection—which we will not look at closely.

From here on, "Prolog" means Prolog as implemented in SWI-Prolog.

Characteristics — Functions & Predicates

Prolog does not really use *functions*, as we generally think of them—except that there are the usual numeric functions involving arithmetic, exponentiation, and trigonometry.

Rather, Prolog has **predicates**. We have defined a predicate to be a function that returns true or false. But a Prolog predicate is slightly different: it is something that we might prove to be true.

In the earlier facts-rules-queries example, we might use predicates `is_a_child_of` and `is_a_grandchild_of`.

We can simulate functions with Prolog predicates.

Say we have a two-argument predicate `isSquared`, where `isSquared(3, 9)` is true, while `isSquared(3, 5)` is not. To find the square of `37`, use a query: which values of `X` make `isSquared(37, X)` true?

Introduction to Prolog
Characteristics — Syntax

Like Swift, C, C++, Java, Lua, and Scheme—but not Haskell or Python—Prolog syntax is **free-form**: indentation is not significant, and newlines are mostly treated like blanks—with some exceptions.

Prolog uses rather unusual punctuation. Here is a Prolog rule.

```
foo7(Ax, BB) :- Ax = 12, BB >= Ax.
```

We know this is a rule
because it contains ":-".

We continue looking at
Prolog syntax in the
context of its type system.

Prolog's type system is similar to that of Lua: dynamic, implicit, duck typing, new types cannot be created.

But Prolog does not have a clear, standardized notion of what exactly a *type* is. Type-like properties are checked in different ways in different contexts.

The basic kind of entity in Prolog is the **term**. Our rule again:

```
foo7(Ax, BB) :- Ax = 12, BB >= Ax.
```

`foo7`, `Ax`, and `BB` are terms. So are `=` and `12`. And so are "`foo7(Ax, BB)`", "`Ax = 12`" and "`BB >= Ax`".

Terms can be divided into 4 categories (call these "types", if you like): **number**, **atom**, **variable**, **compound term**.

```
foo7(Ax, BB) :- Ax = 12, BB >= Ax.
```

A **number** is what you expect. There are two varieties, which we might call **subtypes**: **integer** and **float**. The kinds of values these can hold are also what you would expect.

Integer literals are as usual: `724`

Float literals are *mostly* as usual: `325.0` `12.34e+07`

But "`325.`" is not a float; it is an integer literal followed by a dot.

```
foo7(Ax, BB) :- Ax = 12, BB >= Ax.
```

An **atom** is a name. Atoms are written in one of three ways.

- A sequence of letters, digits, and/or underscores, beginning with a lower-case letter. Example: `foo7`
- A sequence of one or more of the following 17 special characters: `#$&*+-./:<=>?@\^~` Examples: `=` `>=` `@:^$`
- An arbitrary sequence of characters enclosed in single quotes. The usual backslash escapes work. Example: `'hello there!\n'`

If an atom of the 1st or 2nd kind has no backslash, then single-quoting it gives an alternate name: `foo7`, `'foo7'` are the same.

Atoms play several roles in Prolog.

- They are the names of predicates and operators (`foo7` `=` `>=`).
- They function as string literals (`'yo!'`).
- An atom can also simply be itself (`ernest`).

```
foo7(Ax, BB) :- Ax = 12, BB >= Ax.
```

A **variable** is a placeholder than can be bound to a value (term). It is written as a sequence of letters, digits, and/or underscores, beginning with an upper-case letter or underscore. Examples:

```
Ax   BB   _xyz_3
```

Recall that, when we bind a variable to a value, it becomes a **bound** variable. Other variables are **free**.

In Prolog, the distinction between free and bound variables is important. A bound variable is treated like a fixed value. When a variable is passed to a predicate, the predicate can check whether it is free and alter its behavior based on this.

When Prolog tries to unify two terms, it does a **backtracking search**. When it backtracks, it can undo the binding of a bound variable, making the variable free again.

```
foo7(Ax, BB) :- Ax = 12, BB >= Ax.
```

A **compound term** is, roughly, something that looks like a function call or nontrivial expression. Above, "`foo7(Ax, BB)`", "`Ax = 12`", and "`BB >= Ax`" are all compound terms. The infix-operator syntax is actually optional; we can rewrite the latter two as "`=(Ax, 12)`" and "`>=(BB, Ax)`".

More formally, a compound term is a **functor** (typically a predicate) along with its arguments. Above, `foo7` is a functor (it is a predicate); its arguments are `Ax` and `BB`.

Lists, like `[1,2,3]`, are actually compound terms. We can write them using the functor `'[|]'`, which does the *cons* operation. In the function-call-like syntax, the list `[1,2,3]` would be written as follows: `'[|]'(1,'[|]'(2,'[|]'(3,[])))`

Characteristics — Logical Statements [1/2]

Prolog programs consist of facts and rules.

A Prolog **fact** says that something is true. Here is a fact:

```
abc(ZZ, 24).
```

Anything that can be unified with the above is considered proven.

Characteristics — Logical Statements [2/2]

A Prolog **rule** says that something is true if other things can be proven. Here again is our rule:

```
foo7(Ax, BB) :- Ax = 12, BB >= Ax.
```

The conclusion of a rule comes first. Roughly, the above says that `foo7(Ax, BB)` is true if each of `Ax = 12` and `BB >= Ax` is true.

Prolog can use the above rule to attempt to prove anything can be unified with `foo7(Ax, BB)`. After unification—so variables might be bound now—if each of the terms on the right can be proven, in order, then whatever was unified with `foo7(Ax, BB)` is considered proven.

Introduction to Prolog
Build & Execution

SWI-Prolog prefers Prolog source filenames to use the "`.pl`" suffix. We will use it—even though it more commonly marks Perl files.

Prolog can be compiled to an executable, but interactive use is common. We will do the latter exclusively:

- Facts and rules will usually be placed in a source file.
- Queries will be entered interactively.

In the SWI-Prolog interactive environment, load source file `abc.pl` using any of the following. "`?-`" is the prompt. Single quotes are *required* if special characters other than underscore are used.

```
?- [abc].
?- ['abc'].
?- ['abc.pl'].
```

Introduction to Prolog
Some Programming [1/2]

TO DO

- Try out interactive Prolog using SWI-Prolog.
- Write a hello-world program in Prolog and execute it.

Done. See `hello.pl`.

Introduction to Prolog
Some Programming [2/2]

I have written a Prolog program that computes and prints Fibonacci numbers: `fibo.pl`.

TO DO

- Run `fibo.pl`.

See `fibo.pl`.