Prolog: Flow of Control

CS 331 Programming Languages
Lecture Slides
Wednesday, April 15, 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 Category: Logic PLs

In **logic programming**, a computer program is 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**.

Logic programming languages are PLs based on the ideas underlying logic programming. **Prolog** is the most important logic programming language.

Prolog: Simple Programming [1/2]

Fact (this is true; usually in source file):

```
abc.
def(a, 28).
```

Rule (this is true if these are true; usually in source file):

```
ghi(X, Y) :- X =< 3, Y is X+5.
```

Query (is this true? OR how to make it true? interactive):

```
?- ghi(3, 8).
?- ghi(3, Y), Y > 9.
```

A query sets up one or more **goals**. Rule bodies set up **subgoals.**

Review
Prolog: Simple Programming [2/2]

"\+" is a 1-argument predicate that works as a prefix operator. It succeeds if its argument fails, so it means negation.

```
?- \+ 3 = 5.
true.
```

We can write "\+" in Prolog.
Shortly, we will.

Prolog: Lists

A few useful things:

`_` (underscore)

Dummy variable. Unifies with anything, and indicates that its value will not be used. (Prolog calls unused variables "**singleton variables**" and warns you when it finds one. Use "`_`" to avoid such warnings.)

`call`/≥1

Arguments are either a predicate & its arguments or a single compound term. Calls the predicate/term and succeeds if it succeeds.

Prolog: Flow of Control

Prolog: Flow of Control
Preliminaries

A few more useful things:

For code from this topic, see `flow.pl`.

`write`/1

Argument is any term; input-only. Always succeeds. As a side effect, writes its argument to standard output. An atom is written as a string.

`nl`/0

Always succeeds. Writes a newline to standard output.

`true`/0

Succeeds.

`fail`/0

Does not succeed. Therefore, always backtracks.

Prolog: Flow of Control
Basic Repetition [1/2]

We have already seen encapsulated loops that iterate over lists:

`map`, `filter`, `zip`.

We can also repeat an operation directly, using recursion.

TO DO

- Write a Prolog predicate `print_squares`/2, so that, for example, `print_squares(4, 10)` prints 4^2, 5^2, …, 10^2 in some nice format. Use recursion.

Done. See `flow.pl`.

Prolog: Flow of Control
Basic Repetition [2/2]

`print_squares` does repetition of a kind that we might write with a for-loop in Swift or C++. But it is very special purpose. What about writing a more general for-loop?

TO DO

- Write a Prolog predicate `myFor` that encapsulates the general idea of a counted for-loop.
- Rewrite `print_squares` using this for-loop predicate.

Done. See `flow.pl`.

The functionality of `myFor` is already available in SWI-Prolog, in the form of `between`/3.

Prolog: Flow of Control
Cut [1/6]

In order to be truly full-featured programming languages, logic PLs will "cheat" by including features that are not logical in nature.

Prolog's cheat is the **cut**: "!".

- This is written as "!", but read as "cut".
- It takes no arguments.
- It always succeeds.
- Once it succeeds, backtracking past the *cut* is not allowed, for the current goal.
- Included in this: use of another fact or rule for the current goal is not allowed.

Cut turns out to be very versatile—surprisingly versatile, I think. We look at a few ways it can be used.

Prolog: Flow of Control
Cut [2/6]

Cut can be used as the rough equivalent of a Swift or C++ `break`.

TO DO

- Write a Prolog predicate `print_near_sqrt`/1, which takes a positive number *x* and prints the largest integer whose square is at most *x*.

Done. See `flow.pl`.

Prolog: Flow of Control
Cut [3/6]

Cut can do something like if-then-else. Consider the Swift below.

```
func test_big(_ n: Int) {
    if n > 20 {
        print("\(n) IS A BIG NUMBER!")
    } else {
        print("\(n) is not a big number.")
    }
}
```

TO DO

- Write a Prolog predicate that does essentially the same thing as `test_big`, above.

Done. See `flow.pl`.

Prolog: Flow of Control
Cut [4/6]

More generally, *cut* can make it easy to ensure that only one fact/rule is used for any particular goal.

TO DO

- Rewrite our `gcd` predicate (from *Prolog: Simple Programming*) using a cut.

Done. See `flow.pl`.

Note that this new version avoids printing the annoying extra "`false`" that our original `gcd` predicate prints when it succeeds. So another thing *cut* can do is prevent spurious "`false`" output.

Prolog: Flow of Control
Cut [5/6]

Cut allows us to write negation.

TO DO

- Write a predicate `not`, which takes a zero-argument predicate or a compound term, and succeeds if it fails.

Done. See `flow.pl`.

As we have seen, the functionality of `not` is already available in SWI-Prolog, in the form of `\+`/1.

Prolog: Flow of Control
Cut [6/6]

A thought about *cut*: I have seen the necessity of the *cut* described as a failure of the idea of logic programming: to make Prolog work, a nonlogical construction was needed.

But I have another point of view. Prolog is a *stream processor*.

Terms can create new streams.

```
…, between(1, 10, X), …
```

Terms can filter streams.

```
…, X < 5, …
```

And a *cut* truncates a stream. When thought of in that way, it does not seem out of place (I think).

Prolog: Flow of Control
Other Repetition [1/2]

Our `myFor` predicate repeats by succeeding and then calling itself recursively. On the way, it does some checking, so that it does not repeat forever.

What if we do not do any checking?

TO DO

- Write a predicate `myRepeat` that repeats forever.
- Try using `myRepeat`.

Done. See `flow.pl`*.*

The functionality of `myRepeat` is already available in SWI-Prolog, in the form of `repeat`/0.

Prolog: Flow of Control
Other Repetition [2/2]

Predicate `repeat` does not alter variables. So it is largely useless unless we have functionality that is **nondeterministic**: it can give different results for the same input.

One form of nondeterminism is pseudorandom number generation. SWI-Prolog's facilities for this include *function* `random`, which takes an integer *n* and returns a number in the range 0 .. *n* – 1.

Note that `random` is a *function!*

TO DO

- Using `repeat` and `random`, write a predicate that outputs random numbers until some condition becomes true. Make this predicate *end* when the condition is met.

Done. See `flow.pl`*.*

Another kind of nondeterministic functionality involves reading console input from the user. *We discuss this next time.*