Prolog: Simple Programming

CS 331 Programming Languages
Lecture Slides
Friday, April 11, 2025

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.

There are also logic-programming libraries for PLs that support the necessary constructions well enough (e.g., Python, Lisp-family).

Review
Introduction to Prolog [1/2]

Prolog’s execution involves attempts to prove things true using a strategy involving **unification** in a backtracking search.

Unification: making two constructions the same by binding variables as necessary.

In most situations where we might expect to use a function, we use a **predicate** in Prolog**.** These are different from our previous notion of *predicate* (recall: a function that returns Boolean); a Prolog predicate is something that we might prove to be true.

We can simulate functions using 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? (This is an example of where it is important that `X` be a **free variable**.)

Review
Introduction to Prolog [2/2]

Prolog’s type system is similar to that of Lua and Scheme. However, the notion of *type* is not specified precisely.

The basic kind of entity in Prolog is the **term**. Terms can be divided into 4 categories:

- **Number**. Includes **integer** [`325`] and **float** [`325.7  34.66e+4`].
  - “`5.0`” is a float literal. “`5.`” is an integer literal followed by a dot.
- **Atom**. A name [`div_by_7  @:&  'Hello there!\n'`].
  - Atoms are used as names of predicates and operators, and as something like string literals. An atom can also simply be itself.
- **Variable**. Like a C/C++ identifier starting with upper-case or underscore [`Value_8  _xyz3`].
  - Distinguishing **free** and **bound** variables is important.
- **Compound term**. Something that looks like a function call [`foo(A,3.2,'xy')`] or various kinds of syntactic sugar around that:
  - `X = 3` *is the same as* `=(X, 3)`
  - `[1,2]` *is the same as* `'[|]'(1,'[|]'(2,[]))`

Prolog: Simple Programming

Prolog: Simple Programming
General

Facts and **rules** generally go in a source file. **Queries** are entered interactively.

Facts, rules, and queries all end with a period (`.`).

```
foo7(Ax, BB) :- Ax = 12, BB >= Ax.  % This is a rule
```

Comments:

- Single-line: `%` … *END-OF-LINE*
- Multiline: `/*` … `*/`

For code from this topic, see `simple.pl`*.*

In SWI-Prolog we can enter facts and rules interactively, using a virtual source file named "`user`".

```
?- [user].  % Allow user to enter facts & rules
```

Prolog prompt. Do not type this.

Prolog: Simple Programming
Facts, Queries, Rules, Goals [1/4]

A **fact** is written as an atom or compound term followed by a period. A fact says that something is true.

```
abcd.

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

Here, `abcd` is a zero-argument predicate. SWI-Prolog allows us to include an empty parenthesized list [`abcd().`], but we usually do not.

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.

Prolog: Simple Programming
Facts, Queries, Rules, Goals [2/4]

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. Terms 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 variable might be bound to a new value when execution backtracks back to the term in which it was originally bound.

TO DO

- Write some facts in a source file.
- Do some queries.

Done. See `simple.pl`*.*

Prolog: Simple Programming
Facts, Queries, Rules, Goals [3/4]

A **rule** looks like a fact, then "`:-`", then a query. The term before "`:-`" is the **head** of the rule; afterward is the **body** of the rule.

$$\begin{aligned}\underbrace{\texttt{grandchild(A, B)}}_{\substack{\uparrow\\ \text{Head}}}\ \texttt{:-}\ &\texttt{child(A, X),}\\ &\underbrace{\texttt{child(X, B)}}_{\text{Body}}\texttt{.}\end{aligned}$$

A rule says, essentially, that if the conditions in the body are true, then the head is true.

Once again, in a rule, bound variables remain bound to the same values in all later terms. When execution backtracks to the term in which a variable was originally bound, the variable becomes free—possibly to be bound to a new value.

Prolog: Simple Programming
Facts, Queries, Rules, Goals [4/4]

A query sets up a series of **goals**: prove that each term in the query is true, by unifying it with something known. If it is shown to be true, then it **succeeds**; otherwise, it **fails**.

Unifying with a fact is straightforward.

For a rule, terms that unify with the head are those the rule might be used to prove. The body gives the conditions that must be satisfied, in order to do this proof. Each term in the body of the rule then forms a **subgoal**.

TO DO

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

Done. See `simple.pl`*.*

Prolog: Simple Programming Conventions

Prolog has standard human-readable comments for predicates.

`sq`/2 means `sq` is a predicate with 2 arguments.

Arguments may be marked:

- `+` Input only (cannot be a free variable).
- `–` Output only (must be a free variable).
- `?` Input or output.

These conventions are used in Prolog code comments, language documentation, and error messages.

Example: `sq(+x, ?y)`

TO DO

- Add appropriate comments to the code written so far.

Done. See `simple.pl`*.*

Prolog: Simple Programming
Negation

"`\+`" 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. Eventually, we will.

TO DO

- Try some negations.
- Write some rules involving negations.

Done. See `simple.pl`*.*

Prolog: Simple Programming
Numerical Computation [1/6]

Prolog will do numerical computation, but this is not part of its 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 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.
true.
```

A Prolog numerical expression is actually a compound term. The infix-operator notation is syntactic sugar over Prolog's usual function-call-like notation.

Prolog: Simple Programming
Numerical Computation [2/6]

The value of a variable can be an unevaluated expression.

```
?- X = 3+5, Y = 9+sqrt(X).
X = 3+5
Y = 9+sqrt(3+5).
```

"`is`" will evaluate such expressions.

```
?- X = 3+5, Y = 9+sqrt(X), Z is Y-7.
X = 3+5
Y = 9+sqrt(3+5)
Z = 4.82842712474619898.
```

Prolog: Simple Programming
Numerical Computation [3/6]

`+`, `-`, `*`, and parentheses are as usual. `/` is floating-point division. $\underbrace{\texttt{//}\text{ is integer division.}}\ \underbrace{\texttt{**}\text{ is exponentiation.}}$

Like Python

```
?- X is 9/4.
X = 2.25.
?- X is 9//4.
X = 2.
```

Find remainders with “`mod`”. This can be used as an infix operator.

```
?- X is mod(11, 4).
X = 3.
?- Y is 11 mod 4.
Y = 3.
```

Prolog: Simple Programming
Numerical Computation [4/6]

The following use function-call notation: `sqrt exp log sin cos tan asin acos atan ceiling floor` (and others).

```
?- X is sqrt(5).
X = 2.23606797749979
```

2-argument functions include `min`, `max`.

```
?- X is max(5, 8).
X = 8
```

Prolog: Simple Programming
Numerical Computation [5/6]

Operators `=` `\=` mean unifiable and not unifiable, respectively.

Numeric comparisons are usable with function-call notation or as infix operators. Both operands can be expressions, which are evaluated. They cannot contain free variables.

Numeric equality & inequality: `=:=` `=\=`

Numeric ordered comparison operators are as usual, except that they must not look like arrows; so `<=` becomes `=<`.

```
?- 3+5 =< 9+1.
true.
```

Operators `==` `\==` exist, but do something different. They test (in)equality of ASTs, doing no unification or evaluation. Avoid using these, unless you are sure they do what you want!

Prolog: Simple Programming
Numerical Computation [6/6]

TO DO

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

Done. See `simple.pl`*.*

Comments

- We can have multiple facts and rules involving the same predicate.
- As in Haskell, and in Scheme macros, we can do a form of pattern matching in Prolog. A query can only be unified with a fact, or the head of a rule, if it matches in some sense.
- However, unlike the other kinds of pattern matching we have seen, Prolog does not simply go with the first pattern that works; it *tries them all*. This is why we need the various comparisons (e.g., "`A > 0`") in the body of the rules for predicate `gcd`.