

Prolog – Programming in Logic

COMP2600 — Formal Methods for Software Engineering

Jeremy Dawson

Australian National University
Semester 2, 2011

Prolog's Niche

Suitable application areas will be rich in *facts* and *rules*.
This makes it suitable for many AI problem areas.

The runtime system behaves similarly to a database.

- The database holds the facts and rules.
- Computations are either database updates or queries.
- To satisfy a query, the system searches for relevant facts and rules.
- The search allows *backtracking*.

What is Prolog?

Prolog: *A declarative programming language based on logic with the idea that you specify what is to be done without saying how it is to be computed.*

- The name comes from the phrase **Pro**gramming in **logic**.
- The SWI-Prolog implementation is available on the lab machines.
Use `prolog` to start it up.
- There is also a GNU Prolog implementation `gprolog`.

Facts

Informally:

- The syntax of facts is the same as in the predicate calculus.
- By convention, predicate names start with a lower case letter while variables start with an upper case letter.
- Some examples:

```
fish(perch).           % Perch is a fish.
fish(salmon).          % Salmon is a fish.
red(nikita).           % Nikita is red.
likes(mary, john).     % Mary likes John.
factorial(3,6).        % the factorial of 3 is 6.
```

Prolog is conversational

Here is an illustrative conversation with a Prolog system:

```
?- fish(perch).  
yes  
?- fish(dog).  
no  
?- fish(sardine).  
no
```

- A **sardine** is a fish, but prolog doesn't know that unless it is specified as a fact in the source file.

Search Principles

For queries containing no variables:

- If the query matches a fact in the database, then **SUCCEED**.
- Otherwise **FAIL**.

Multiple possible answers

```
?- fish(X).  
X = perch  
?- ;  
X = salmon  
?- ;  
no
```

- Both **perch** and **salmon** are fish.
- We use `;` to ask for the next answer (otherwise, just RET)
- If we are not happy with any of the available fish, the query fails.

Search Principles

For queries containing a variable:

- Scan the database looking for possible matches.
- If the query matches an item after *instantiating* the variable, then report the binding that gave the match.
- If told to try again, undo the binding and resume scan.
- If no more matches are possible then report failure.

Conjunctive Goals

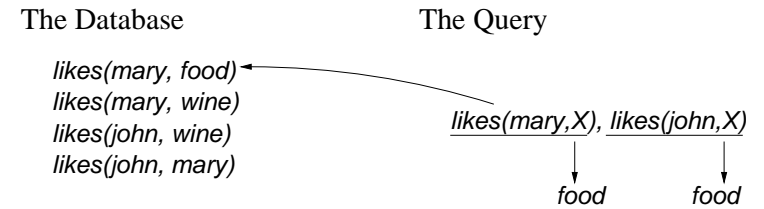
A comma separating two terms indicates a conjunction (and)

Both terms must be satisfied for the expression to be true.

```
?- fish(perch), fish(salmon).
yes
```

```
?- fish(dog), fish(salmon).
no
```

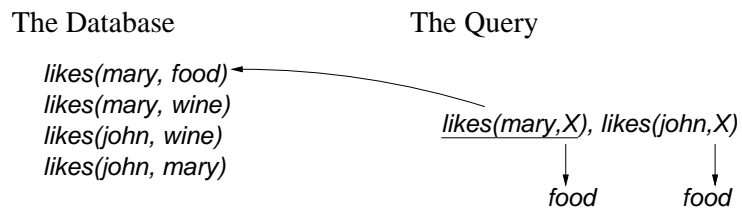
Repeated Variables (ctd.)



The Computation

4. The second goal fails
5. Backtrack – forget previous value of X
6. Attempt to re-satisfy first goal

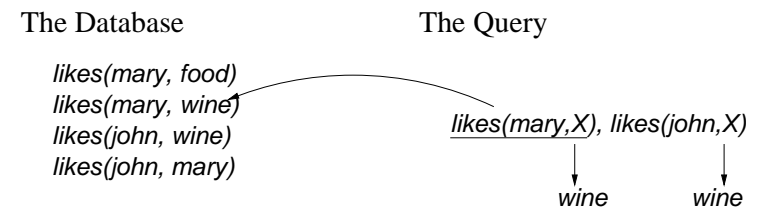
Repeated Variables



The Computation

1. The first goal succeeds
2. X is instantiated to 'food'
3. Attempt to satisfy the 2nd goal

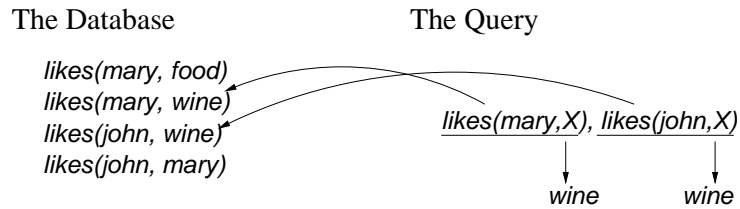
Repeated Variables (ctd.)



The Computation

7. The first goal succeeds again
8. X is instantiated to 'wine'
9. Attempt to satisfy the 2nd goal

Repeated Variables (ctd.)



The Computation

- The second goal succeeds
- The binding is reported indicating success
- Prolog waits for user's reply.

Rules – Intentions

| We write: | If: |
|----------------------------------|--|
| $P(X) \quad :- \quad Q(X)$ | $\forall x. Q(x) \implies P(x)$ |
| $P(X,Y) \quad :- \quad Q(X,Y)$ | $\forall x,y. Q(x,y) \implies P(x,y)$ |
| $P(X) \quad :- \quad Q(X), R(X)$ | $\forall x. (Q(x) \wedge R(x)) \implies P(x)$ |
| $P(X) \quad :- \quad Q(X,Z)$ | $\left\{ \begin{array}{l} \forall x,z. Q(x,z) \implies P(x) \\ \forall x. (\exists z. Q(x,z)) \implies P(x) \end{array} \right.$ |
| $P(X,Y) \quad :- \quad Q(X)$ | $\left\{ \begin{array}{l} \forall x,y. Q(x) \implies P(x,y) \\ \forall x. Q(x) \implies \forall y. P(x,y) \end{array} \right.$ |

Rules

The usual form is

$\langle term \rangle \quad :- \quad \langle term \rangle, \langle term \rangle, \dots \langle term \rangle .$

Examples:

$\overbrace{bird(X)}^{\text{head}} \quad :- \quad \overbrace{animal(X), winged(X)}^{\text{body}} .$
 $evil(X) \quad :- \quad intendsToHaveNuclearWeapons(X) .$

- When a rule's head matches a goal, the body gives a list of sub-goals to be resolved.
- To prove X is a bird, we must show that it is an animal, and is winged.
- A *fact* is a *rule* with an empty body.

Rules – Example 1

Database contains:

$father(john, sue) .$
 $mother(mary, sue) .$
 $parent(X, Y) \quad :- \quad father(X, Y) .$

Conversation is:

$?- \quad parent(X, sue)$
 $X = john$
 $?- \quad ;$
 no

Clearly, also need the rule:

$parent(X, Y) \quad :- \quad mother(X, Y)$

Rules – Example 2

Database from Example 1 with additions:

```
mother(anne, john) .
mother(sonia, mary) .
father(bill, john) .
father(charles, mary) .

grandparent(X, Y) :- parent(X, Z), parent(Z, Y) .
```

Conversation is:

```
?- grandparent(X, sue)
X = bill
?- ;
X = charles
?- ; etc. (Why are the answers returned in this order?)
```

The Prolog definition of append:

```
append([], Xs, Xs) .
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs) .
```

Note that $[X|Xs]$ is equivalent to the Haskell expression $(x:xs)$

A similar definition in Haskell is:

```
append [] xs      = xs
append (x:xs) ys = x : zs
      where zs = append xs ys
```

Example: list concatenation

The expression $\text{append}(Xs, Ys, Zs)$ means that the result of appending Xs and Ys is Zs .

Thus, to concatenate two lists:

```
?- append([a, b, c], [1, 2, 3], Xs) .
Xs = [a, b, c, 1, 2, 3]
```

Compare with Haskell:

```
xs = append ['a', 'b', 'c'] [1, 2, 3]
```

Note that in the Haskell version, 'a' 'b' and 'c' are characters, not atoms.

Inverse Functions

But you can also find the inverse of a function:

```
?- append(Xs, Ys, [a, b, c]) .
Xs = []
Ys = [a, b, c]
```

We are asking what possible lists could we append together to get the result $[a, b, c]$. The above is just one possible answer...

```
?- ;
Xs = [a]
Ys = [b, c]

?- ;
Xs = [a, b]
Ys = [c] .... etc
```

Parsing example, from slides 18-19 of PDA lecture

Right factored Grammar:

```
 $E \rightarrow T \mid T + E$   
 $T \rightarrow F \mid F * T$   
 $F \rightarrow \mathbf{id} \mid (E)$ 
```

PDA transitions:

```
 $\delta(q_1, +, +) \mapsto q_1/\epsilon$   
 $\delta(q_1, *, *) \mapsto q_1/\epsilon$   
...  
 $\delta(q_1, \epsilon, E) \mapsto q_1/T$   
 $\delta(q_1, \epsilon, E) \mapsto q_1/E + T$   
 $\delta(q_1, \epsilon, T) \mapsto q_1/F$   
 $\delta(q_1, \epsilon, T) \mapsto q_1/T * F$   
...  
 $\delta(q_0, \epsilon, Z) \mapsto q_1/EZ$   
 $\delta(q_1, \epsilon, Z) \mapsto q_2/\epsilon$ 
```

This is non-deterministic as there are multiple valid transitions for a given PDA state.

Practical Considerations — Ordering

- The order of clauses for a predicate matters
- The order of subgoals within the rhs of a clause matters
- A predicate may work well when one argument is variable and another is a particular value, or vice versa

Example: delay in discovering that $1 \neq 5$, try this with tracing turned on

```
sillyappend([], Ys, Ys).
```

```
sillyappend([X|Xs], Ys, [Z|Zs]) :- sillyappend(Xs, Ys, Zs), X = Z.
```

```
?- sillyappend([1,2,3,4], Ys, [5,6,7,8]).
```

Pop matching terminals from the stack:

```
match(q1, [T|Input], [T|Stack])  
:- match(q1, Input, Stack)
```

Push productions onto the stack:

```
match(q1, Input, [NonTerm| Stack])  
:- prod(NonTerm, Rhs)  
  , append(Rhs, Stack, NewStack)  
  , match(q1, Input, NewStack)
```

Push the first non-terminal:

```
match(q0, Input, [z]) :- match(q1, Input, [start, z])
```

Accept the matching string:

```
match(q1, Input, [z]) :- match(q2, Input, [])  
match(q2, [], []).
```

Ordering of Clauses and Goals

```
sillierappend([], Ys, Ys).
```

```
sillierappend(XXs, Ys, ZZs) :-
```

```
  sillierappend(Xs, Ys, Zs), XXs = [X|Xs], ZZs = [X|Zs].
```

first goal succeeds once, then runs forever

second goal runs forever

```
?- sillierappend([1,2,3,4], Ys, [1,2,3,4,5,6,7,8]).
```

```
?- sillierappend([1,2,3,4], Ys, [5,6,7,8]).
```

Both examples semantically identical to standard append

A Textbook List

- Bratko, *Prolog Programming for A.I.*, 1990, Addison-Wesley.
- Clarke and McCabe, *micro-PROLOG: Programming in Logic*, 1984, Prentice-Hall.
- Clocksin and Mellish, *Programming in Prolog*, 1981, Springer.
- Gazdar and Mellish, *Natural Language Processing in Prolog*, 1990, Addison-Wesley.
- Ginanesini, Kanoui, Pasero, Canegham, *PROLOG*, 1986, Addison-Wesley.
- Lassez (ed), *Logic Programming, Vols. I & II*, 1987, MIT.
- Stirling, *The Practice of Prolog*, 1990, MIT.
- Stirling and Shapiro, *The Art of Prolog*, 1986, MIT.

Web Resources

In an online search for “Prolog”, many of the early matches are for on-line tutorials, courses, books, implementations and specialist Prolog web-sites. In particular, see:

- ["http://gnu-prolog.inria.fr/"](http://gnu-prolog.inria.fr/) The GNU Prolog web site
- ["http://www.swi-prolog.org/"](http://www.swi-prolog.org/) SWI-Prolog's Home
- ["http://www.sics.se/sicstus/"](http://www.sics.se/sicstus/) SICStus Prolog Homepage
- ["http://www.visual-prolog.com/"](http://www.visual-prolog.com/) Visual Prolog Home Page
- ["http://www.dobrev.com/"](http://www.dobrev.com/) Strawberry Prolog