Prolog

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

You should be able to...

In this lecture, we will introduce Prolog.

- Explain how Prolog uses a unification to drive computation.
- Write some simple programs in Prolog.

Logic

Question: How do you decide truth?

- Start with some *objects*.
  "socrates," "john," "mary"
- Write down some *facts* (true statements) about those objects.
  - Facts express either properies of the object, or
    "socrates is human"
  - relationship to other objects.
    "mary likes john"
- Write down some *rules* (facts that are true if other facts are true).
  "if X is human then X is mortal"
- Facts and rules can become *predicates*.
  "is socrates mortal?"

First-Order Predicate Logic

First-order predicate logic is one system for encoding these kinds of questions.

- Predicate means that we have functions that take objects and return “true” or “false.”
  `human(socrates).`
- Logic means that we have *connectives* like and, or, not, and implication.
- First order means that we have variables (created by “for all” and “there exists”), but that they only work on objects.
  $\forall$ `X . human(X)` $\rightarrow$ `mortal(X).`

History

- Starting point: First-order predicate logic.
- Realization: computers can reason with this kind of logic.
- Impetus was the study of *mechanical theorem proving*
- Developed in 1970 by Alain Colmerauer and Rober Kowalski and others
- Uses: databases, expert systems, AI

The Two Questions

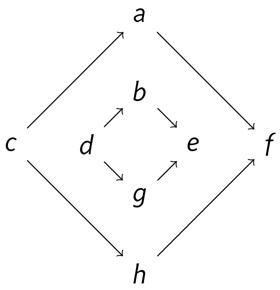
What is the nature of data?

Prolog data consists of *facts* about *objects* and logical *rules*.

What is the nature of a program?

A program in Prolog is a set of facts and rules, followed by a *query*.

The Database




```
human(socrates).
fatherof(socrates,
         jane).
fatherof(zeus,apollo).
```

```
connected(c,a).
connected(c,h).
connected(d,b).
connected(d,g).
connected(a,f).
connected(h,f).
connected(b,e).
connected(g,e).
```

Rules

```
mortal(X) :- human(X).
human(Y) :- fatherof(X,Y), human(X).

pathfrom(X,Y) :- connected(X,Y).
pathfrom(X,Y) :- connected(X,Z),
                 pathfrom(Z,Y).
```

- Capital letters are variables.
  - Appearing left of :- means "for all"
  - Appearing right of :- means "there exists"

$$\forall x.human(x) \rightarrow mortal(x).$$

$$\forall y.(\exists x.fatherof(x,y) \wedge human(x)) \rightarrow human(y)$$

How It Works

Programs are executed by searching the database and attempting to perform unification.

```
?- human(socrates).    -- listed, therefore true
?- mortal(socrates).   -- not listed
```

Relevant rules:

```
human(socrates).
human(Y) :- fatherof(X,Y), human(X).
mortal(X) :- human(X).
```

Socrates is not listed as being mortal, but `mortal(socrates)` unifies with `mortal(X)` if we replace `X` with `socrates`. This gives us a *subgoal*. Replace *X* with `socrates` and try it....

How It Works, Next Step

Replace X with socrates in this rule:

```
mortal(X) :- human(X).
```

to get

```
mortal(socrates) :- human(socrates).
```

Since human(socrates) is in the database, we know that mortal(socrates) is also true.

Another Example

- `?- mortal(jane).`

Another Example

- `?- mortal(jane).`

  `mortal(X) :- human(X).`

Another Example

- ?- **mortal**(jane).

  mortal(jane) :- **human**(jane).

Another Example

- `?- mortal(jane).`

  `mortal(jane) :- human(jane).`

  - `human(jane)`

Another Example

- `?- mortal(jane).`

  `mortal(jane) :- human(jane).`

  - `human(jane)`

    `human(Y) :- fatherof(X,Y), human(X).`

Another Example

- `?- mortal(jane).`

  ```
  mortal(jane) :- human(jane).
  ```

  - `human(jane)`

    ```
    human(jane) :- fatherof(X,jane), human(X).
    ```

Another Example

- `?- mortal(jane).`

  `mortal(jane) :- human(jane).`

  - `human(jane)`

    `human(jane) :- fatherof(X,jane), human(X).`

  - `fatherof(X,jane)`

Another Example

- `?- mortal(jane).`

  `mortal(jane) :- human(jane).`

  - `human(jane)`

    `human(jane) :- fatherof(X,jane), human(X).`
  - `fatherof(X,jane)`
    - `fatherof(socrates,jane)`

Another Example

- `?- mortal(jane).`

  `mortal(jane) :- human(jane).`

  - `human(jane)`

    `human(jane) :- fatherof(socrates,jane), human(socrates).`

  - `fatherof(X,jane)`

    - `fatherof(socrates,jane)`

Another Example

- `?- mortal(jane).`

  `mortal(jane) :- human(jane).`

  - `human(jane)`

    `human(jane) :- fatherof(socrates,jane), human(socrates).`

  - `fatherof(X,jane)`

    - `fatherof(socrates,jane)`

  - `human(socrates)`

You Try ...

- Given the connected rules, try to come up with a predicate `exactlybetween(A,B,C)` that is true when `B` is connected to both `A` and `C`.
- Now make a predicate `between(A,B,C)` that is true if there's a path from `A` to `B` to `C`.

```
exactlybetween(A,B,C) :- connected(A,B), connected(B,C).

between(A,B,C) :- pathfrom(A,B), pathfrom(B,C).
```

More Than Just Yes or No ....

- Prolog can also give you a list of elements that make a predicate true. Remember unification.

```
?- fatherof(Who,apollo).
Who = zeus

?- pathfrom(c,X).
X = a ;
X = h ;
X = f ;
X = f ;
No
```

The semicolon is entered by the user — it means to keep searching.

Tracing pathfrom

```
?- pathfrom(c,X).
---> pathfrom(c,Y) :- connected(c,Y).
X = a ;
```

When we hit semicolon, we tell it to keep searching. So we *backtrack* through our database to try again.

```
pathfrom(c,Y) :- connected(c,Y).
---> X = h ;
```

We tell it to try again with this one, too. At this point, we no longer have any rules that say that c is connected to something.

Tracing pathfrom, II

```
pathfrom(c,Y) :- connected(c,Z), pathfrom(Z,Y).
```

We will first find something in the database that says that c is connected to some Z, and then check if there is a path between Z and Y.

We find a and h as last time. When we check a, we check for pathfrom(a,Y), and find that connected(a,f) is in the database. The same thing happens for h, which is why f is reported as an answer twice.

Arithmetic via the is Keyword.

```
fact(0,1).
fact(N,X) :- M is N-1, fact(M,Y), X is Y * N.
?- fact(5,X).
```

- Unify fact(5,X) with fact(N,X).
  `fact(5,X) :- M is 5-1, fact(M,Y), X is Y * 5.`
- Next compute M.
  `fact(5,X) :- 4 is 5-1, fact(4,Y), X is Y * 5.`
- Recursive call sets Y to 24.
  `fact(5,X) :- 4 is 5-1, fact(4,24), X is 24 * 5.`
- Compute X.
  `fact(5,120) :- 4 is 5-1, fact(4,24), 120 is 24 * 5.`

Lists

- Empty list: `[]`
- Singleton list: `[x]`
- List with multiple elements: `[x,y,[a,b],c]`
- Head and tail representation: `[H|T]`

Differences:

- Prolog lists are *not* monotonic!

List Example: mylength

The `length` predicate is built in.

```
mylength([],0).
mylength([H|T],X) :- mylength(T,Y),
                     X is Y + 1.

?- mylength([2,3,4,5],X).
X = 4 ;
No
```

List Example: Sum List

```
sumlist([],0).
sumlist([H|T],X) :- sumlist(T,Y),
                    X is Y + H.

?- sumlist([2,3,4,5],X).
X = 14
```

Try writing list product now!

List Example: Append

```
myappend([],X,X).
myappend([H|T],X,[H|Z]) :- myappend(T,X,Z).
?- myappend([2,3,4],[5,6,7],X).
X = [2, 3, 4, 5, 6, 7] ;
No
?- myappend(X,[2,3],[1,2,3,4]).
No
?- myappend(X,[2,3],[1,2,3]).
X = [1] ;
No
```

List Example: Reverse

Accumulator recursion works in Prolog, too!

```
myreverse(X,Y) :- aux(X,Y,[]).
aux([],Y,Y).
aux([HX|TX],Y,Z) :- aux(TX,Y,[HX|Z]).
?- myreverse([2,3,4],Y).
Y = [4, 3, 2]
```

myreverse([2,3,4],Y) → aux([2,3,4],Y,[]) → aux([3,4],Y,[2]) → aux([4],Y,[3,2]) → aux([],Y,[4,3,2]) → aux([],[4,3,2],[4,3,2]) → myreverse([2,3,4],[4,3,2])