Prolog

Programming in Logic

Arasu Arun
for CS3100 by Prof. Raghavendra Rao BV

Contents

Prolog program = Facts database

A Prolog program is actually

a collection of “facts” in a “database”.

1. Declare the first values (base cases)

“The first Fibonacci number is 1”

“The second Fibonacci number is 1”

```
% Prolog program: [db.pl]
fib(1,1). % 1st fib number is 1
fib(2,1). % 2nd fib number is 1
```

Think of it as declaring a predicate (“fib”) and its valid tuples (1,1) and (2,1).

2. Declare the recursive rule [1/2]

Imagine you are writing a mathematical statement.

```
fib(N,X) :-         % the Nth fib number is X IF
    fib(N-1,Y),     % (N-1)th fib number is Y AND
    fib(N-2,Z),     % (N-2)nd fib number is Z AND
    X is Y + Z.      % X is Y + Z.


% But, this code is syntactically wrong
% We cannot use expressions like (N-1),(N-2) for predicate
arguments.
```

2. Declare the recursive rule [2/2]

Thinking of it in terms of predicates and tuples:

```
fib(N,X) :-       % (N,X) is a valid tuple IF
    N1 is N-1,
    N2 is N-2,
    fib(N1,Y),    % (N1,Y) is a valid tuple AND
    fib(N2,Z),    % (N2,Z) is a valid tuple AND
    X is Y + Z.    % X = Y + Z.
```

Overall Code / “Rules”:

```
% This is a comment.
fib(1,1).
fib(2,1).
fib(N,X) :-
        N1 is N-1,
        N2 is N-2,
        fib(N1,Y),
        fib(N2,Z),
        X is Y+Z.
% End of code.
```

How do you call the function?

Query “`fib(5,X)`”

- Translates to asking Prolog: what value of X makes that predicate True?

```
?- fib(1,X).
X = 1 .

?- fib(3,X).
X = 2 .

?- fib(5,X).
X = 5 .

?- fib(6,X).
X = 8 .

?- fib(20,X).
X = 6765 .
```

Imperative

- Writing steps

```
int fib(int n){
    if (n <= 2)
        return 1;
    else
    return fib(n-1)+fib(n-2);
}
```

Declarative

- Logic formulas

```
fib(1,1).
fib(2,1).
fib(N,X) :-
    ...
    fib(N-1,Y),
    fib(N-2,Z),
    ...
```

Notice that there was no “structure” in the code

- No “function”
- No “if else” statements like in C
- No prescribed sequence of steps!
- No loops

Matching and Backtracking (~ depth first search)

Prolog tries to match a query to
a rule in its database 1 by 1.

Matching and Backtracking (~ depth first search)

Prolog tries to match a query to a rule in its database 1 by 1.

Procedurally assigns values to variables.

```
% Query = fib(5,X).

fib(1,1). % no... 5 isn't 1
```

Matching and Backtracking (~ depth first search)

Prolog tries to match a query to a rule in its database 1 by 1.

Procedurally assigns values to variables.

```
% Query = fib(5,X).

fib(1,1). % no… 5 isn’t 1

fib(2,1). % no… 5 isn’t 2
```

Matching and Backtracking (~ depth first search)

Prolog tries to match a query to a rule in its database 1 by 1.

Procedurally assigns values to variables.

```
% Query = fib(5,X).

fib(1,1). % no… 5 isn’t 1

fib(2,1). % no… 5 isn’t 2

fib(N,X) :- % yes: N=5, X still free
```

Matching and Backtracking (~ depth first search)

Prolog tries to match a query to a rule in its database 1 by 1.

Procedurally assigns values to variables.

```
% Query = fib(5,X).

fib(1,1). % no… 5 isn’t 1

fib(2,1). % no… 5 isn’t 2

fib(N,X) :- % yes: N=5, X still free

        N1 is N-1, % yes: let N1=4

        N2 is N-2, % yes: let N2=3
```

Matching and Backtracking (~ depth first search)

Prolog tries to match a query to a rule in its database 1 by 1.

Procedurally assigns values to variables.

```
% Query = fib(5,X).
fib(1,1). % no… 5 isn’t 1
fib(2,1). % no… 5 isn’t 2
fib(N,X) :- % yes: N=5, X still free
        N1 is N-1, % yes: let N1=4
        N2 is N-2, % yes: let N2=3
        fib(N1,Y), % recurse: try Y=3
```

Matching and Backtracking (~ depth first search)

Prolog tries to match a query to a rule in its database 1 by 1.

Procedurally assigns values to variables.

```
% Query = fib(5,X).
fib(1,1). % no... 5 isn't 1
fib(2,1). % no... 5 isn't 2
fib(N,X) :- % yes: N=5, X still free
        N1 is N-1, % yes: let N1=4
        N2 is N-2, % yes: let N2=3
        fib(N1,Y), % recurse: try Y=3
        fib(N2,Z), % recurse: try Z=2
```

Matching and Backtracking (~ depth first search)

Prolog tries to match a query to a rule in its database 1 by 1.

Procedurally assigns values to variables.

```
% Query = fib(5,X).
fib(1,1). % no… 5 isn’t 1
fib(2,1). % no… 5 isn’t 2
fib(N,X) :- % yes: N=5, X still free
        N1 is N-1, % yes: let N1=4
        N2 is N-2, % yes: let N2=3
        fib(N1,Y), % recurse: try Y=3
        fib(N2,Z), % recurse: try Z=2
        X is Y+Z.  % let Z=5
% End of code.
```

Matching and Backtracking (~ depth first search)

Prolog tries to match a query to a rule in its database 1 by 1.

Procedurally assigns values to variables.

If any subclause fails, it "**backtracks**" to the previous subclause and tries a different variable assignment.

```
% Query = fib(5,X).
fib(1,1). % no… 5 isn’t 1
fib(2,1). % no… 5 isn’t 2
fib(N,X) :- % yes: N=5, X still free
        N1 is N-1, % yes: let N1=4
        N2 is N-2, % yes: let N2=3
        fib(N1,Y), % recurse: try Y=3
        fib(N2,Z), % recurse: try Z=2
        X is Y+Z.  % let Z=5
% End of code.
```

Lists: [square brackets]

Access is only from the front.

Access lists as **[H,T]** where Prolog will **match**:

- H to the first element
- T to the remaining list (could be empty)

You can access a fixed number of elements at the front:

- [A,B,C | T]

```
?- all_same([4,4,4,4,4]).
true .

?- all_same([1,2,3,4]).
false.
```

```
all_same([]).          % empty list
all_same([A]).         % singleton
all_same([A,B|T]) :- % [A,B,...]
    A = B,
    all_same([B|T]). % recurse
```

Permutations - 1 (removing element from list)

```
remove(H,[H|T],T).          % removing H from [H|T] gives T
```

Permutations - 2 (removing element from list)

```
remove(H,[H|T],T).              % removing H from [H|T] gives T
remove(X,[H|T],[H|S]) :-  % removing X from [H|T] gives [H|S]
    remove(X,T,S).              % where S is T having X removed
```

Permutations - 3 (base cases)

```
remove(H,[H|T],T).            % removing H from [H|T] gives T
remove(X,[H|T],[H|S]) :-      % removing X from [H|T] gives [H|S]
    remove(X,T,S).            % where S is T having X removed

perm([],[]).                % empty
perm([X],[X]).              % singleton
```

Permutations - 4 (recurse)

```
remove(H,[H|T],T).            % removing H from [H|T] gives T
remove(X,[H|T],[H|S]) :-      % removing X from [H|T] gives [H|S]
    remove(X,T,S).            % where S is T having X removed

perm([],[]).             % empty
perm([X],[X]).           % singleton
perm([H|T],P) :-         % P is a permutation of [H|T] if
    remove(H,P,Q),       % removing H from P gives Q
    ...

(removing H from a permutation P gives us Q... )
```

Permutations - 5 (recurse)

```
remove(H,[H|T],T).           % removing H from [H|T] gives T
remove(X,[H|T],[H|S]) :-     % removing X from [H|T] gives [H|S]
    remove(X,T,S).           % where S is T having X removed

perm([],[]).               % empty
perm([X],[X]).             % singleton
perm([H|T],P) :-           % P is a permutation of [H|T] if
    remove(H,P,Q),         % removing H from P gives Q
    perm(T,Q).              % Q is a permutation of T

(removing H from a permutation P gives us Q...
 then Q must be a permutation of T).
```

The last output being `false` is useful for backtracking.

Notice that we didn't even need **loops**!

That's elegantly handled by backtracking behind the scenes.

Many programs can be simplified so well using Prolog.

```
?- perm([1,2,3],X).
X = [1, 2, 3] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [1, 3, 2] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;
false.
```

Prolog in real life

http://what-when-how.com/information-science-and-technology/using-prolog-for-developing-real-world-artificial-intelligence-applications-information-science/

http://www.drdobbs.com/parallel/the-practical-application-of-prolog/184405220

Lots of other verification portals, proof checking, etc…