Logic Programming: Lecture 2

Madhavan Mukund

Chennai Mathematical Institute

madhavan@cmi.ac.in

PLC, 7 April 2017

Programming with relations

Represent edge relation using the following facts.

```
edge(3,4).
edge(5,4).
edge(5,1).
edge(1,2).
edge(3,5).
edge(2,3).
```

Define `path` using the following rules.

```
path(X,Y) :- X = Y.
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Read the rules read as follows:

Rule 1 For all `X`, `(X,X)` $\in$ `path`.

Rule 2 For all `X`,`Y`, `(X,Y)` $\in$ `path` if there exists `Z` such that `(X,Z)` $\in$ `edge` and `(Z,Y)` $\in$ `path`.

Facts and rules

```
path(X,Y) :- X = Y.
path(X,Y) :- edge(X,Z), path(Z,Y).
```

- Each rule is of the form

  Conclusion if Premise_1 and Premise_2 ... and Premise_n

  - if is written `:-`
  - and is written `,`
  - This type of logical formula is called a Horn Clause

- Quantification of variables
  - Variables in goal are universally quantified
    - `X`, `Y` above
  - Variables in premise are existentially quantified
    - `Z` above

Computing in Prolog

- Ask a query
- Prolog scans facts and rules top-to-bottom
- If the head of a rule matches the query, the body generates subgoals.
  - Matching is unification
- Sub goals are tried depth-first
- If a subgoal fails, backtrack and try another value
- Backtracking is sensitive to order of facts

Unification and pattern matching

- A goal of the form `X = Y` denotes unification.

```
path(X,Y) :- X = Y.
path(X,Y) :- edge(X,Z), path(Z,Y).
```

- Can implicitly represent such goals in the head

```
path(X,X).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

- Unification provides a formal justification for pattern matching in rule definitions
  - Unlike Haskell, a repeated variable in the pattern is meaningful
  - In Haskell, we cannot write

```
path (x,x) = True
```

Reversing the question

- Consider the question

```
?- edge(3,X).
```

- Find all `X` such that `(3,X)` $\in$ `edge`
- Prolog lists out all satisfying values, one by one

```
X=4;
X=5;
X=2;
No.
```

Lists

- Write `[Head | Tail]` for Haskell's `(head:tail)`
  - `[]` denotes the emptylist
  - No types, so lists need not be homogeneous!
- Checking membership in a list

```
member(X,[Y|T]) :- X = Y.
member(X,[Y|T]) :- member(X,T).
```

- Use patterns instead of explicit unification

```
member(X,[X|T]).
member(X,[H|T]) :- member(X,T).
```

- . . . plus anonymous variables.

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

Arithmetic

Computing length of a list

```
length([],0).
length([_|T],N) :- length(T,M), N = M+1.
```

What does the following query yield?

```
?- length([1,2,3,4],N).
```

```
N=0+1+1+1+1
```

- `X = Y` is unification
- `X is Y` captures arithmetic equality

```
length([],0).
length([_|T],N) :- length(T,M), N is M+1.
```

Arithmetic . . .

Another approach

```
length(L,N) :- auxlength(L,0,N).
auxlength([],N,N).
auxlength([H|T],M,N) :- auxlength(T,M1,N), M1 is M+1.
```

`?- length([0,1,2],N)` generates goals

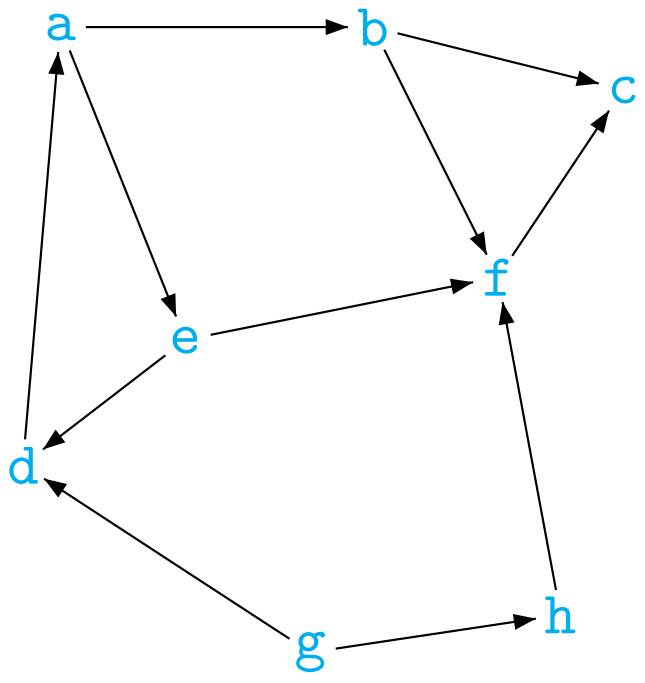
```
auxlength([0,1,2],0,N)
auxlength([1,2],1,N)
auxlength([2],2,N)
auxlength([],3,N)
auxlength([],3,3)
```

Second argument to `auxlength` accumulates answer.

Coping with circular definitions

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

path(X,X).
path(X,Y) :- edge(X,Z), path(Z,Y).
```



What does `?- path(a,b)` compute?

Coping with circularity . . .

Instead

```
path(X,X,T).
path(X,Y,T) :- edge(X,Z), legal(Z,T), path(Z,Y,[Z|T]).

legal(Z,[]).
legal(Z,[H|T]) :- Z\==H, legal(Z,T).
```

- `path(X,Y,T)` succeeds if there is a path from `X` to `Y` that does not visit any nodes in `T`
- `T` is an accumulator

Quicksort in Prolog

- How we describe a sorting algorithm in a logic program?

```
% quicksort(Xs, Ys) :- Ys is a sorted permutation of Xs
  quicksort([], []).
  quicksort([X | Xs], Ys) :-
            partition(X, Xs, Littles, Bigs),
            quicksort(Littles, Ls),
            quicksort(Bigs, Bs),
            append(Ls, [X | Bs], Ys).
```

where

```
% partition(X, Xs, Ls, Bs) :-
%       Ls : list of elements of Xs that are < X
%       Bs : list of elements of Xs that are >= X
  partition(_, [], [], []).
  partition(X, [Y | Xs], [Y | Ls], Bs) :-
            X > Y, partition(X, Xs, Ls, Bs).
  partition(X, [Y | Xs], Ls, [Y | Bs]) :-
            X =< Y, partition(X, Xs, Ls, Bs).
```

Quicksort in Prolog

Two issues that arise in `quicksort`.

- Wasteful recomputations in last clause of `partition`

```
...
partition(X, [Y | Xs], [Y | Ls], Bs) :-
            X > Y, partition(X, Xs, Ls, Bs).
partition(X, [Y | Xs], Ls, [Y | Bs]) :-
            X =< Y, partition(X, Xs, Ls, Bs).
```

  - Consider `?- partition(7,[9,8,1,5],Ls,Bs).`

- `append(Ls, [X | Bs], Ys).`
  - As in functional programming, complexity of `append` is proportional to length of `Ls`
  - Can this be avoided?

Backtracking in Prolog

Consider rules

```
G :- P1,P2,P3.
G :- P4,P5,P6.
```

- First try `G`.
  - If `P3` fails, backtrack and retry `P2`.
  - If `P2` fails, backtrack and retry `P1`.
  - If `P1` fails, try second rule.
- Second rule is tried after all possible ways of satisfying first rule fail.

Backtracking in Prolog . . .

Goal `p(X)`, rules of the form `if B then S else T`

```
p(x) :- B,S.
p(X) :- not B, T.
```

- `not B` succeeds if `B` fails.
- Can we avoid recomputing `B`?

Cut

- Special goal `!`, called `cut`

```
p(x) :- B, !, S.
p(x) :- T.
```

- `!` always succeeds
- Discard alternative ways of computing `B`
- Discard second rule `p(x) :- T.`

More generally, if we have

```
p(s1 ) :- A1 .
. . .
p(si ) :- B,!,C.
. . .
p(sk ) :- Ak .
```

`B` is not retried and clauses $i+1$ to k are discarded.

Cut . . .

- Cut is typically used for efficiency, avoid recomputing conditions.

```
% partition(X, Xs, Ls, Bs) :-
%         Ls : list of elements of Xs that are < X
%         Bs : list of elements of Xs that are >= X
  partition(_, [], [], []).
  partition(X, [Y | Xs], [Y | Ls], Bs) :-
              X > Y, !, partition(X, Xs, Ls, Bs).
  partition(X, [Y | Xs], Ls, [Y | Bs]) :-
              partition(X, Xs, Ls, Bs).
```

Control structures

- `call(X)` invokes `X` as a goal.

```
once(G) :- call(G),!.
```

```
for(0,G) :- !.
for(N,G) :- N > 0, call(G), M is N-1, for(M,G),!.
```

```
if_then_else(B, S, T) :- call(B),!,call(S).
if_then_else(B, S, T) :- call(T).
```

Use with care. Destroys declarative structure!

Control structures

- Goal `fail` always fails

```
not(G) :- call(G),!,fail
not(_).
```

- Use `not` with care
- To generate all members of a list that are not `1`
  - `member(X, Ls), not(X = 1).` ✓
  - `not(X = 1), member(X, Ls).` ×
- Should only use `not` when term is already instantiated

```
?- not(X = 1).
no
```

Quicksort in Prolog

Two issues that arise in `quicksort`.

- Wasteful recomputations in last clause of `partition`

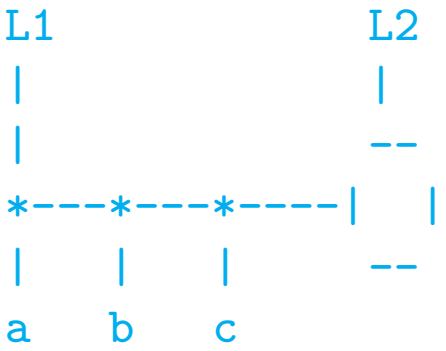
```
...
partition(X, [Y | Xs], [Y | Ls], Bs) :-
            X > Y, partition(X, Xs, Ls, Bs).
partition(X, [Y | Xs], Ls, [Y | Bs]) :-
            X =< Y, partition(X, Xs, Ls, Bs).
```

  - Consider `?- partition(7,[9,8,1,5],Ls,Bs).`

- `append(Ls, [X | Bs], Ys).`
  - As in functional programming, complexity of `append` is proportional to length of `Ls`
  - Can this be avoided?

Difference lists

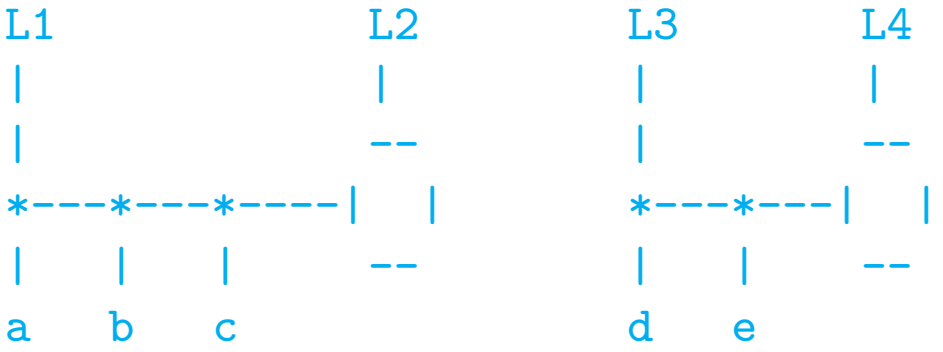
- Represent a list in terms of front and back



- Unify `L1` with `[a,b,c|Z]` and `L2` with `Z`
- `L2` points to a "hole" that can be instantiated by another term

Difference lists . . .

- Suppose we want to append `L1` and `L3`



- `app(L1,L2,L3,L4,X,Y)` succeeds when difference lists `(L1,L2)` and `(L3,L4)` combine to form difference list `(X,Y)`
- Single goal

```
app(L1,L2,L2,L3,L1,L3).
```

- Normally, difference lists are denoted `L1-L2`.

```
app(L1-L2,L2-L3,L1-L3).
```

- If `X` is a difference list, unify with `Y-[]` to rectify it