

# Programming Language Concepts: Lecture 23

Madhavan Mukund

Chennai Mathematical Institute

`madhavan@cmi.ac.in`

`http://www.cmi.ac.in/~madhavan/courses/pl2009`

PLC 2009, Lecture 23, 20 April 2009

# Quicksort in Prolog

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

# 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).
```

# 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`.

# 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).`

# 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.



# 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.

# 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.`

# Cut

- ▶ Special goal `!`, called `cut`

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

`p(x) :- T.`

- ▶ `!` always succeeds

# Cut

- ▶ Special goal `!`, called `cut`

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

`p(x) :- T.`

- ▶ `!` always succeeds
- ▶ Discard alternative ways of computing `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.`

# Cut

- ▶ Special goal  $!$ , called **cut**

$p(x) \text{ :- } B, !, S.$

$p(x) \text{ :- } T.$

- ▶  $!$  always succeeds
- ▶ Discard alternative ways of computing  $B$
- ▶ Discard second rule  $p(x) \text{ :- } T.$

More generally, if we have

$p(s_1) \text{ :- } A_1 .$

$\vdots$

$p(s_i) \text{ :- } B, !, C.$

$\vdots$

$p(s_k) \text{ :- } A_k .$

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



# Cut ...

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

# 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.

# Control structures

- ▶ `call(X)` invokes `X` as a goal.

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

# 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),!.
```

# 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).
```

# 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



# 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).`

# 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

# 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

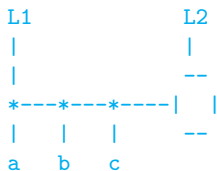
- ▶ `member(X, Ls), not(X = 1).` ✓
- ▶ `not(X = 1), member(X, Ls).` ✗

- ▶ Should only use `not` when term is already instantiated

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

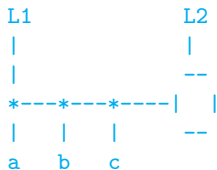
# Difference lists

- Represent a list in terms of **front** and **back**



# Difference lists

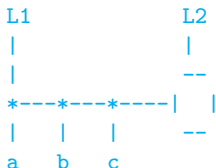
- Represent a list in terms of **front** and **back**



- Unify **L1** with  $[a,b,c|Z]$  and **L2** with **Z**

# Difference lists

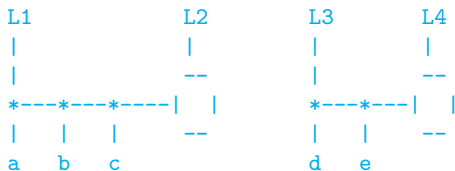
- Represent a list in terms of **front** and **back**



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

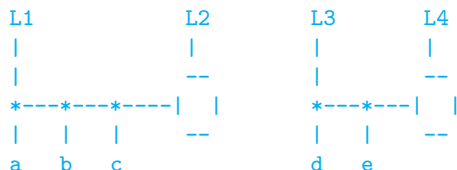
# Difference lists ...

- Suppose we want to append L1 and L3



# Difference lists ...

- Suppose we want to append L1 and L3

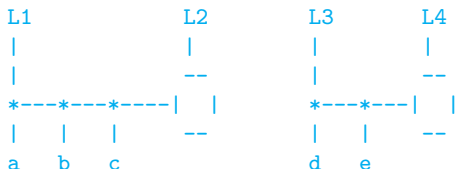


- `app(L1,L2,L3,L4,X,Y)` succeeds when difference lists  $(L1,L2)$  and  $(L3,L4)$  combine to form difference list  $(X,Y)$



# 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,L4,L1,L4) .`
- ▶ Normally, difference lists are denoted `L1-L2`.
- ▶ If `X` is a difference list, unify with `Y-[]` to rectify it

# Flatten

- ▶ Flatten an arbitrarily nested list into an linear list

# Flatten

- Flatten an arbitrarily nested list into an linear list

```
flatten(X,Y) :- flatpair(X,Y-[]).
```

```
flatpair([],L-L).
```

```
flatpair([H,T],L1-L3) :- flatpair(H,L1-L2), flatpair(T,L2-L3).
```

```
flatpair(X,[X|Z]-Z).
```