University of Auckland

07.363

Functional and Logic Programming

Christian Collberg

October 7, 1994

Cut and Negation

Cuts & Negation I

The cut (`!`) is is ued to affect Prolog's backtracking. It can be used to

- reduce the search space (save time).
- tell Prolog that a goal is deterministic (has only one solution) (save space).
- construct a (weak form of) negation.
- construct `if_then_else` and `once` predicates.

Slide 6–1

Cuts & Negation II

- The cut reduces the flexibility of clauses, and destroys their logical structure.
- Use cut as a last resort.
- Reordering clauses can sometimes achieve the desired effect, without the use of the cut.
- If you are convinced that you have to use a cut, try using `if_then_else`, `once`, or `not` instead.

Slide 6–2

The Cut I

The cut succeeds and commits Prolog to all the choices made since the parent goal was called.

Cut does two things:

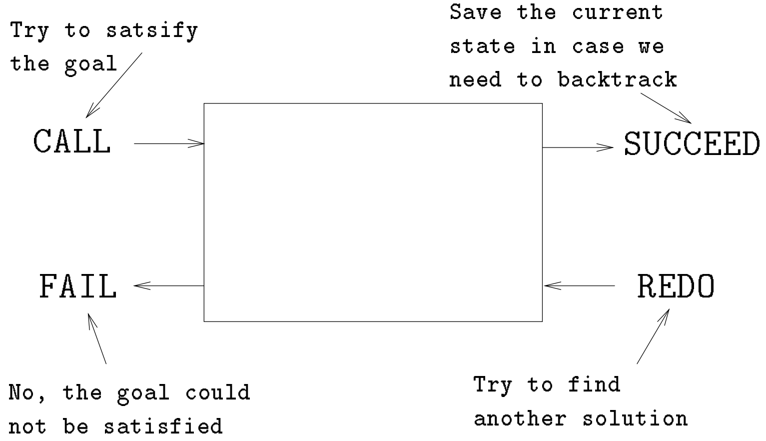
commit: Don't consider any later clauses for this goal.

prune: Throw away alternative solutions to the left of the cut.

Slide 6–3

Slide 6–6

Prolog Execution — The Boxflow Model I



Slide 6–4

The Cut II

prune:
Forget alternative
solutions

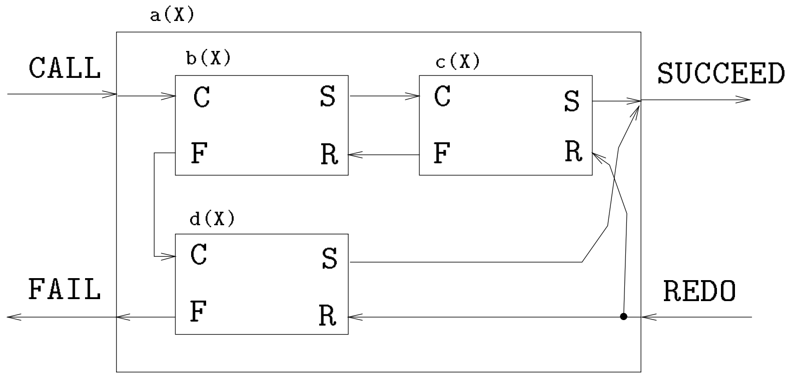
```
p :- a, b, !, c, d.
p :- x, y.
p :- z.
```

commit:
Don't try these

Slide 6–7

Prolog Execution — The Boxflow Model II

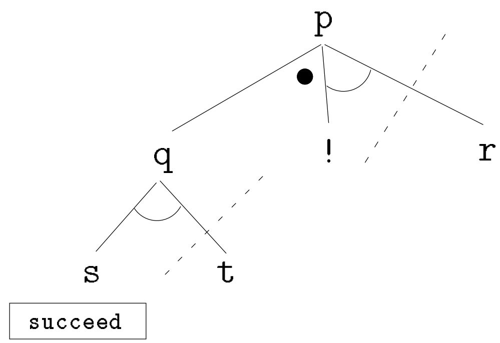
```
a(X) :- b(X), c(X).
a(X) :- d(X).
```



Slide 6–5

The Cut III

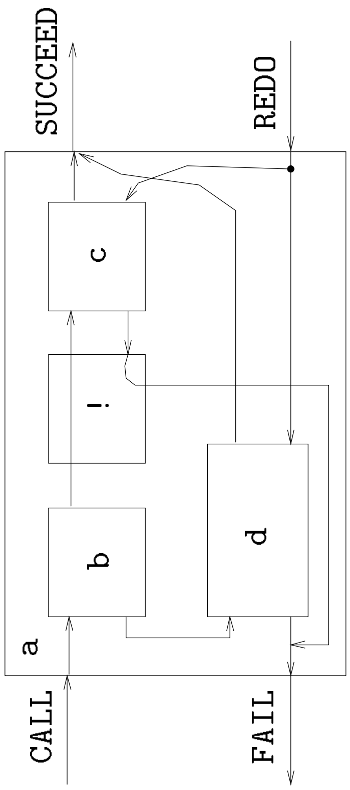
```
p :- q, !.
p :- r.
q :- s.
q :- t.
s.
```



Prolog Execution — The Cut

```
a :- b, !, c.
a :- d.
```

```
if b then c
     else d
```



Slide 6–8

Classifying Cuts

grue No effect on logic, improves efficiency.

green Prune away

- irrelevant proofs
- proofs which are bound to fail

blue Prune away

- proofs a smart Prolog implementation would not try, but a dumb one might.

red Remove unwanted logical solutions.

Slide 6–9

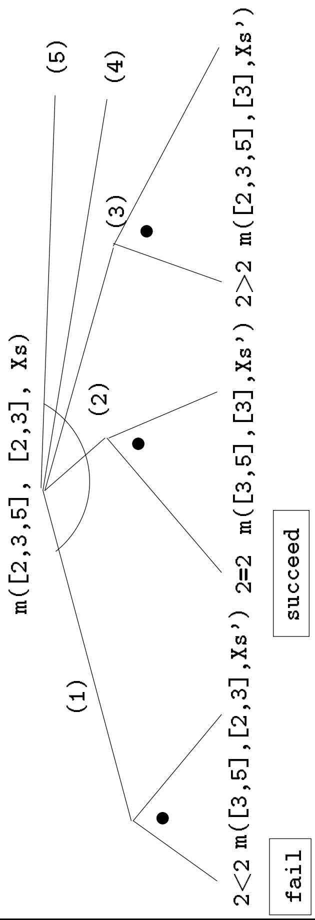
Green Cuts – Merge I

Produce an ordered list of integers from two ordered lists of integers.

```
merge([X|Xs], [Y|Ys], [X|Zs]) :-
   X < Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X,Y|Zs]) :-
   X = Y, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :-
   X > Y, merge([X|Xs], Ys, Zs).

merge(Xs, [], Xs).
merge([], Ys, Ys).

?- merge([1,4], [3,7], L).
   L = [1,3,4,7]
```

Slide 6–10

Green Cuts – Merge II



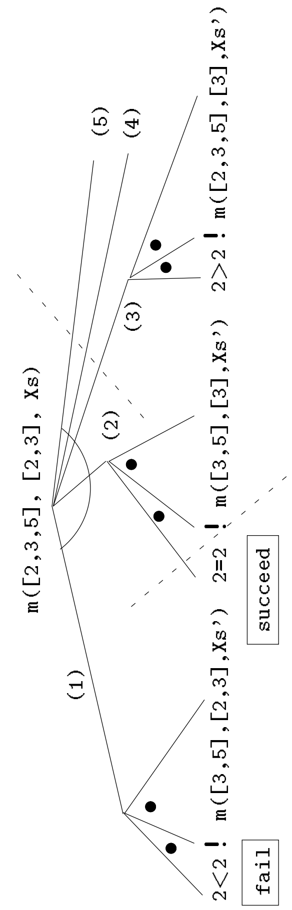
Slide 6–11

Green Cuts – Merge III

```
merge([X|Xs], [Y|Ys], [X|Zs]) :-
   X < Y, !,
   merge(Xs, [Y|Ys], Zs).

merge([X|Xs], [Y|Ys], [X,Y|Zs]) :-
   X = Y, !,
   merge(Xs, Ys, Zs).

merge([X|Xs], [Y|Ys], [Y|Zs]) :-
   X > Y, !,
   merge([X|Xs], Ys, Zs).

merge(Xs, [], Xs) :- !.
merge([], Ys, Ys) :- !.
```

Slide 6–12

Green Cuts – Merge IV



Slide 6–13

Red Cuts – Abs I

```
abs1(X, X) :- X >= 0.
abs1(X, Y) :- Y is -X.
?- abs1(-6, X).
      X = 6 ;
?- abs1(6, X).
      X = 6 ;
      X = -6 ;

abs2(X, X) :- X >= 0, !.
abs2(X, Y) :- Y is -X.
?- abs2(-6, X).
      X = 6 ;
?- abs2(6, X).
      X = 6 ;
```

Slide 6–14

Red Cuts – Abs II

```
abs3(X, X) :- X >= 0.
abs3(X, Y) :- X < 0,
              Y is -X.

?- abs3(-6, X).
      X = 6 ;
      no
?- abs3(6, X).
      X = 6 ;
      no
```

Slide 6–15

Red Cuts – Intersection I

Find the intersection of two lists A & B, i.e. all elements of A which are also in B.

```
intersect([H|T], L, [H|U]) :-
   member(H, L),
   intersect(T, L, U).
intersect([_|T], L, U) :-
   intersect(T, L, U).
intersect(_,_,[]).
```

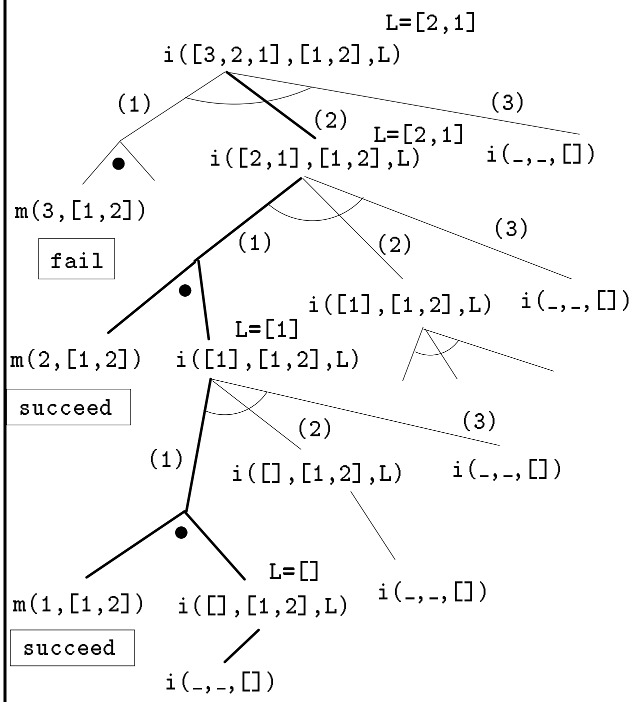
Slide 6–16

Red Cuts – Intersection II

```
?- intersect([3,2,1],[1,2], L).
     L = [2,1] ;
     L = [2] ;
     L = [2] ;
     L = [1] ;
     L = [] ;
     L = [] ;
     L = [] ;
     L = [] ;
     no
```

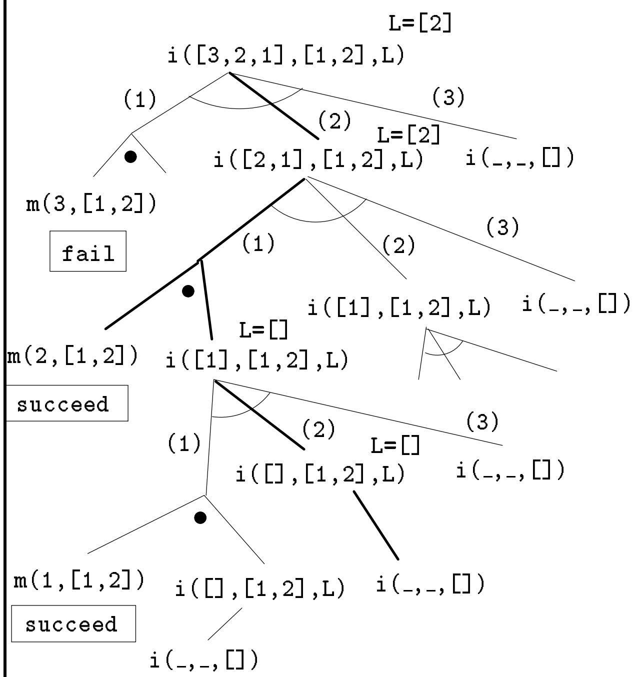
Slide 6–17

Red Cuts – Intersection III



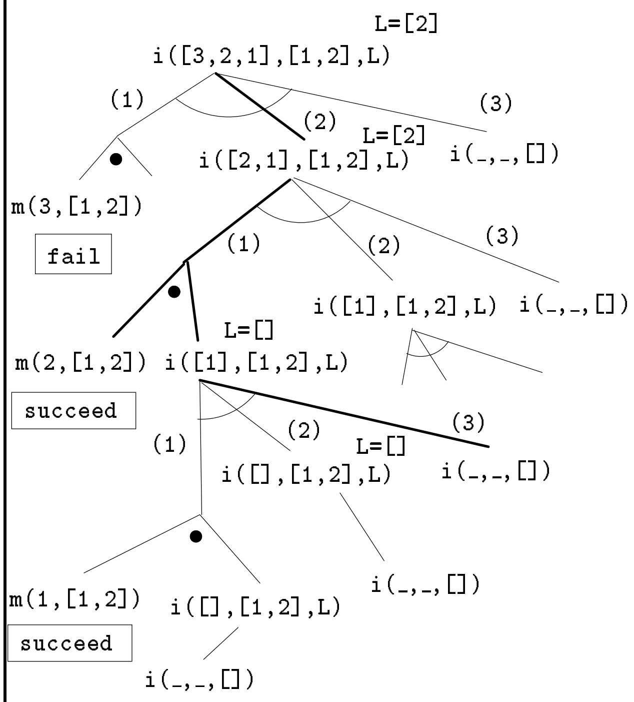
Slide 6–18

Red Cuts – Intersection IV



Slide 6–19

Red Cuts – Intersection V



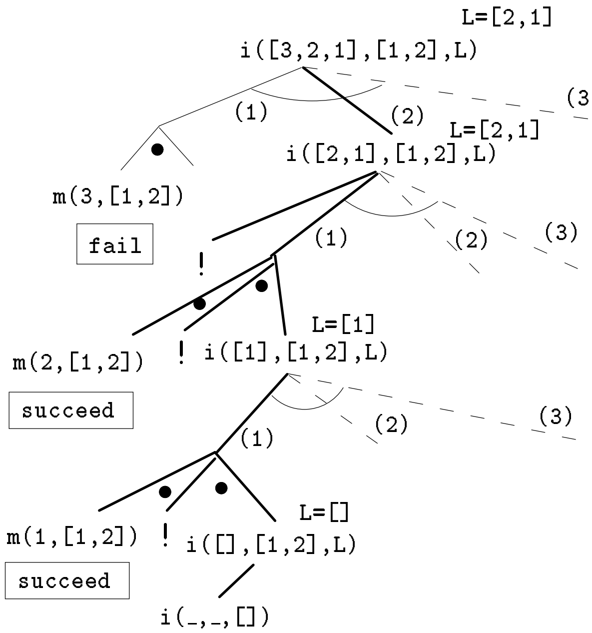
Slide 6–20

Red Cuts – Intersection VI

```
intersect([H|T], L, [H|U]) :-
   member(H, L),
   intersect(T, L, U).
intersect([_|T], L, U) :-
   intersect(T, L, U).
intersect(_,_,[]).

intersect1([H|T], L, [H|U]) :-
   member(H, L), !,
   intersect1(T, L, U).
intersect1([_|T], L, U) :-
   !, intersect1(T, L, U).
intersect1(_,_,[]).
```

Slide 6–21

Red Cuts – Intersection VI



Slide 6–22

Blue Cuts I

First clause indexing will select the right clause in **constant** time:

```
clause(x(5), ...)  :- ...
clause(y(5), ...)  :- ...
clause(x(5, f), ...)  :- ...
?- clause(x(C, f),...).
```

First clause indexing will select the right clause in **linear** time:

```
clause(W, x(5), ...)  :- ...
clause(W, y(5), ...)  :- ...
clause(W, x(5, f), ...)  :- ...
?- clause(a, x(C, f),...).
```

Slide 6–23

Blue Cuts II

```
capital(britain, london).
capital(sweden, stockholm).
capital(nz, wellington).
?- capital(sweden, X).
      X = stockholm
?- capital(X, stockholm).
      X = sweden

capital1(britain, london) :- !.
capital1(sweden, stockholm) :- !.
capital1(nz, wellington) :- !.
?- capital1(sweden, X).
      X = stockholm
?- capital1(X, stockholm).
      X = sweden
```

Slide 6–24

Red Cuts – Once I

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

?- member(1,[1,1]), write('x'), fail.
      xx

mem1(H,[H|_]) :- !.
mem1(I, [_|T]) :- mem1(I, T).

?- mem1(1, [1,1]), write('x'), fail.
      x

once(G) :- call(G), !.
one_mem(X, L) :- once(mem(X, L)).

?- one_mem(1,[1,1]), write('x'),fail.
      x
```

Slide 6–25

Red Cuts – Once II

Red cuts prune away logical solutions. A clause with a red cut has no logical reading.

```
?- member(X, [1,2]).
   X = 1 ;
   X = 2 ;
   no

?- one_mem(X, [1,2]).
   X = 1 ;
no
```

Slide 6–26

Red Cuts – Abs III

```
abs2(X, X) :- X >= 0, !.
abs2(X, Y) :- Y is -X.

if_then_else(P,Q,R):-call(P),!,Q.
if_then_else(P,Q,R):-R.

abs4(X, Y) :- if_then_else(X >= 0,
                   Y=X, Y is -X).

?- abs4(-6, X).
      X = 6 ;
      no
?- abs4(6, X).
      X = 6 ;
      no
```

Slide 6–27

IF–THEN–ELSE I

```
intersect([H|T], L, [H|U]) :-
   member(H, L), !,
   intersect(T, L, U).
intersect([_|T], L, U) :-
   !, intersect(T, L, U).
intersect(_,_,[]).
```

IF `H` $\in$ `L` **THEN**
   `compute the inters.  of T and L,`
   `let H be in the resulting list.`
ELSEIF `the list \= []` **THEN**
   `let the resulting list be the`
   `intersection of T and L.`
ELSE
   `let the resulting list be [].`
ENDIF

Slide 6–28

IF–THEN–ELSE II

```
if_then_else(P,Q,R) :- call(P), !, Q.
if_then_else(P,Q,R) :- R.

intersect2([X|T], L, W) :-
   if_then_else(member(X, L),
      (intersect2(T, L, U), W=[X|U]),
      if_then_else(T \= [],
         intersect2(T, L, W),
         W = [])).
```

Slide 6–29

Open vs. Closed World I

How should we handle *negative information*?

Open World Assumption:

If a clause `P` is not currently asserted then `P` is neither true nor false.

Closed World Assumption:

If a clause `P` is not currently asserted then the negation of `P` is currently asserted.

Slide 6–30

Open vs. Closed World II

```
striker(dahlin).
striker(thern).
striker(andersson).
```

Open World Assumption:

Dahlin, Thern, and Andersson are strikers, but there may be others we don't know about.

Closed World Assumption:

`X` is a striker *if and only if* `X` is one of Dahlin, Thern, and Andersson.

Slide 6–31

Negation in Prolog

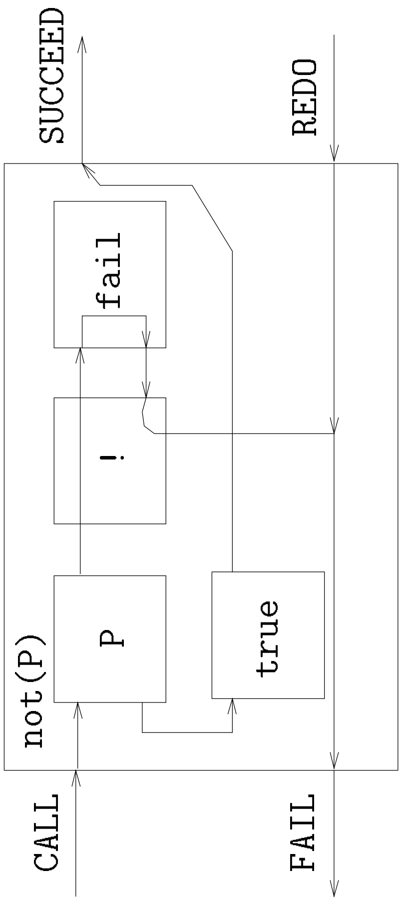
- Prolog makes the closed world assumption.
- Anything that I do not know and cannot deduce is not true.
- Prolog's version of negation is *negation as failure.*
- `not(G)` means that `G` *is not satisfiable as a Prolog goal.*

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

?- not(member(5, [1,3,5])).
no
?- not(member(5, [1,3,4])).
yes
```

Slide 6–32

Prolog Execution – Not

`not(P) :- P, !, fail; true.`



Slide 6–33

Negation Example – Disjoint

Do the lists `X` & `Y` **not** have any elements in common?

```
disjoint(X, Y) :-
      not(member(Z, X),
         member(Z, Y)).

?- disjoint([1,2],[3,2,4]).
      no

?- disjoint([1,2],[3,7,4]).
      yes
```

Slide 6–34

Prolog Negation Problems I

```
man(john).  man(adam).
woman(sue). woman(eve).
married(adam, eve).

married(X) :- married(X, _).
married(X) :- married(_, X).
human(X) :- man(X).
human(X) :- woman(X).

% Who is not married?
?- not married(X).
   false

% Who is not dead?
?- not dead(X).
   true
```

Slide 6–35

Prolog Negation Problems II

```
man(john).  man(adam).
woman(sue). woman(eve).
married(adam, eve).
married(X) :- married(X, _).
married(X) :- married(_, X).
human(X) :- man(X).
human(X) :- woman(X).

% Who is not married?
?- human(X), not married(X).
   X = john ;
   X = sue

% Who is not dead?
?- man(X), not dead(X).
   X = john ;
   X = adam ;
```

Slide 6–36

Prolog Negation Problems II

- If `G` terminates then so does `not G`.
- If `G` does not terminate then `not G` may or may not terminate.

```
married(abraham, sarah).

married(X, Y) :- married(Y, X).

?- not married(abraham,sarah).
   false
?- not married(sarah,abraham).
```
non-termination

Slide 6–37

Open World Assumption I

We can program the *open world assumption*:

- A query is either *true, false,* or *unknown.*
- A false facts `F` has to be stated explicitly, using `false(F)`.
- If we can't prove that a statement is *true* or *false*, it's *unknown.*

```
% Philip is Charles' father.
father(philip, charles).

% Charles has no children.
false(father(charles, X)).
```

Slide 6–38

Open World Assumption II

```
prove(P) :- call(P), write('** true'), nl, !.

prove(P) :- false(P), write('** false'), nl, !.

prove(P) :-
  not(P), not(false(P)),
  write('*** unknown'), nl, !.
```

Slide 6–39

Open World Assumption III

```
father(philip, charles).
false(father(charles, X)).

% Is Philip the father of ann?
?- prove(father(philip, ann)).
   ** unknown

% Does Philip have any children?
?- prove(father(philip, X)).
   ** true
   X = charles

% Is Charles the father of Mary?
?- prove(father(charles, mary)).
   ** false
```