PROLOG

Controlling Backtracking: The Cut

Artificial Intelligence Programming in Prolog

Lecturer: Tim Smith

Lecture 7

14/10/04

Clearing up equality

- There are various ways to test equality in Prolog.

`X = Y`	succeeds if the terms X and Y unify.
`X is Y`	succeeds if the arithmetic value of expression Y matches the value of term X.
`X =:= Y`	succeeds if the arithmetic value of two expressions X and Y match.
`X =\= Y`	succeeds if the arithmetic value of two expressions X and Y DO NOT match.
`X == Y`	succeeds if the two terms have *literal equality* = are structurally identical and all their components have the same name.
`X \== Y`	succeeds if the two terms are NOT literally identical.
`\+ Goal`	succeeds if Goal does not true

Clearing up equality (2)

```
| ?- 3+4 = 4+3.
no     % treats them as terms

| ?- 3+4 = 3+4.
yes

| ?- X = 4+3.
X = 4+3 ?
yes

| ?- X is 4+3.
X = 7 ?
yes

| ?- 3+4 is 4+3.
no     % left arg. has to be a term
```

```
| ?- 3+4 =:= 4+3.
yes    % calculates both values
| ?- 3+4 =\= 4+3.
no

| ?- 3+4 == 4+3.
no
| ?- 3+4 \== 4+3.
yes
| ?- 3+X = 3+4.
X = 4 ?   yes
| ?- 3+X == 3+4.
no

| ?- \+ 3+4 == 4+3.
yes
```

Processing in Prolog

To `call` the goal G:

1. Find first clause head that matches G:
   1. bind all variables accordingly,
   2. `call` goals in body in order;
   3. if all succeed, G `succeeds` (and `exits`).
2. else try next clause down;
3. if no next clause, `fail` the goal G.

When a goal fails:

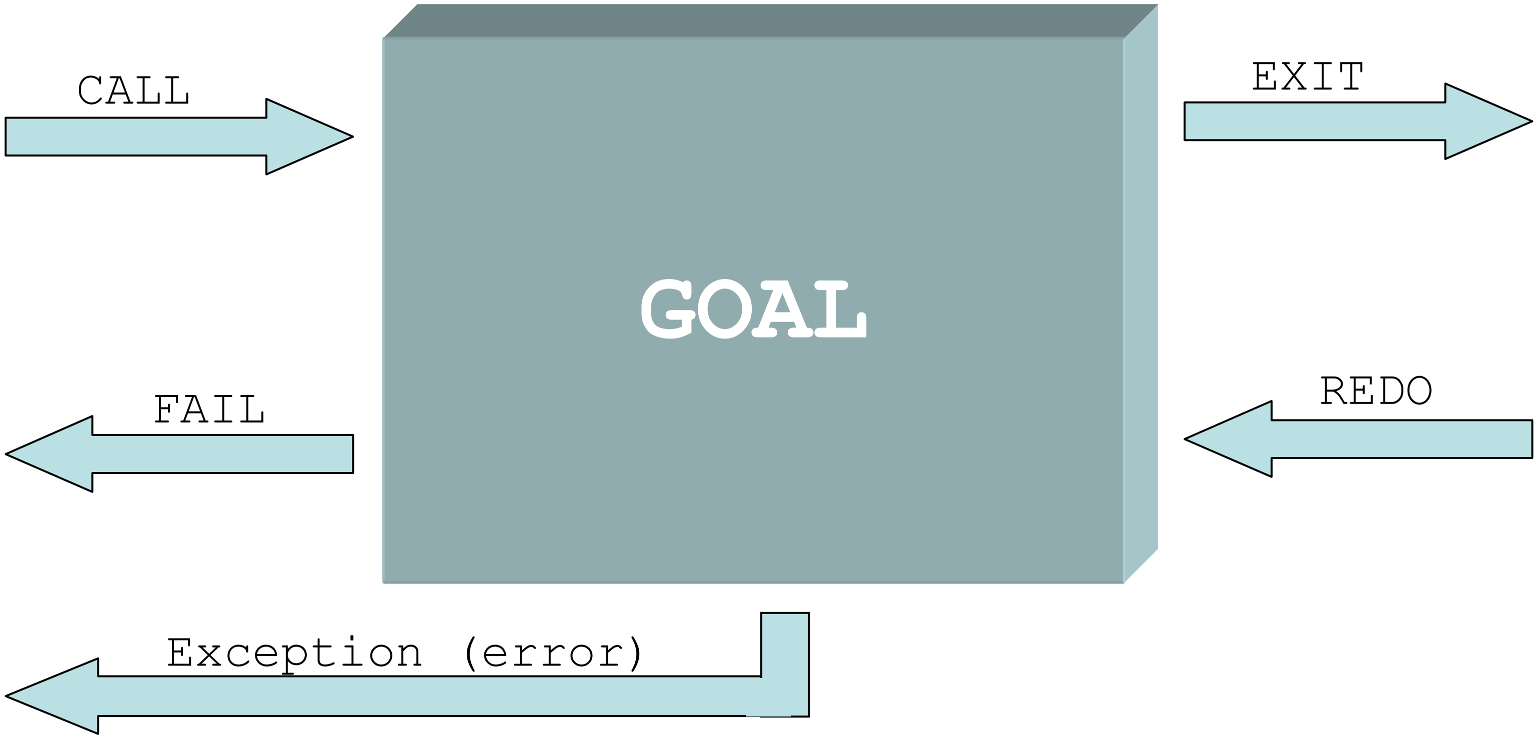
`redo` the most recent successful goal

To `redo` a goal:

1. discard bindings from previous success;
2. try clauses for this goal not so far tried;
3. if none, fail the goal.

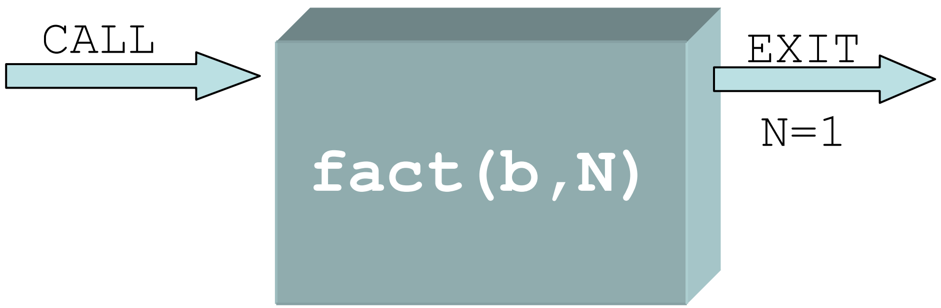
Byrd Box model

- This is the model of execution used by the tracer.
- Originally suggested by Lawrence Byrd.



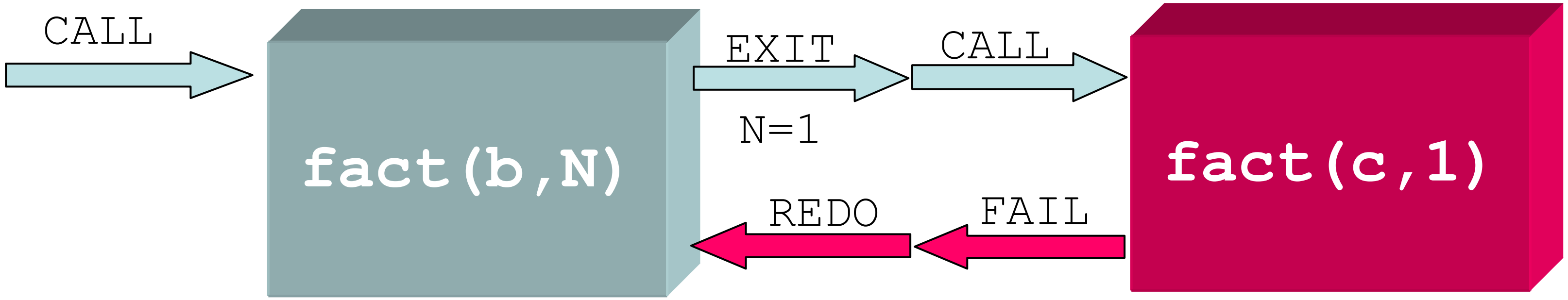
Redo-ing a Goal

```
fact(b,1).
fact(b,2).
a :- fact(b,N), fact(c,N).

|?- a.
```



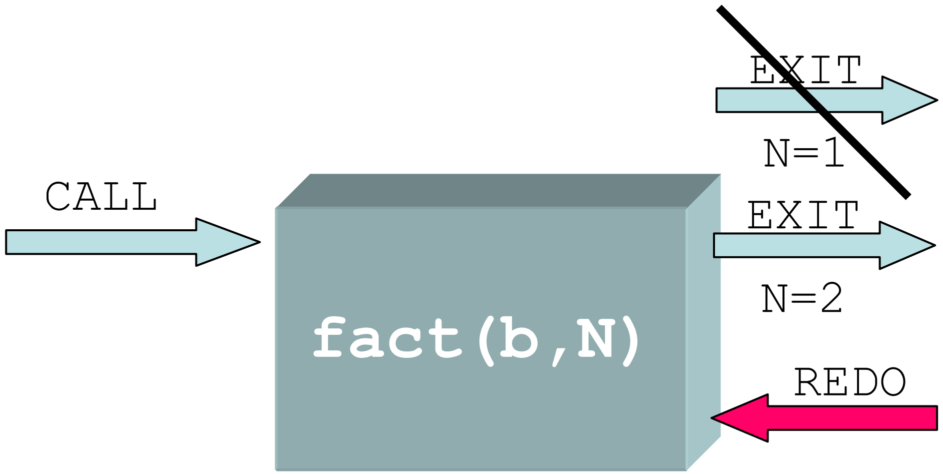
Redo-ing a Goal (2)

```
fact(b,1).
fact(b,2).
a :- fact(b,N), fact(c,N).

|?- a.
```



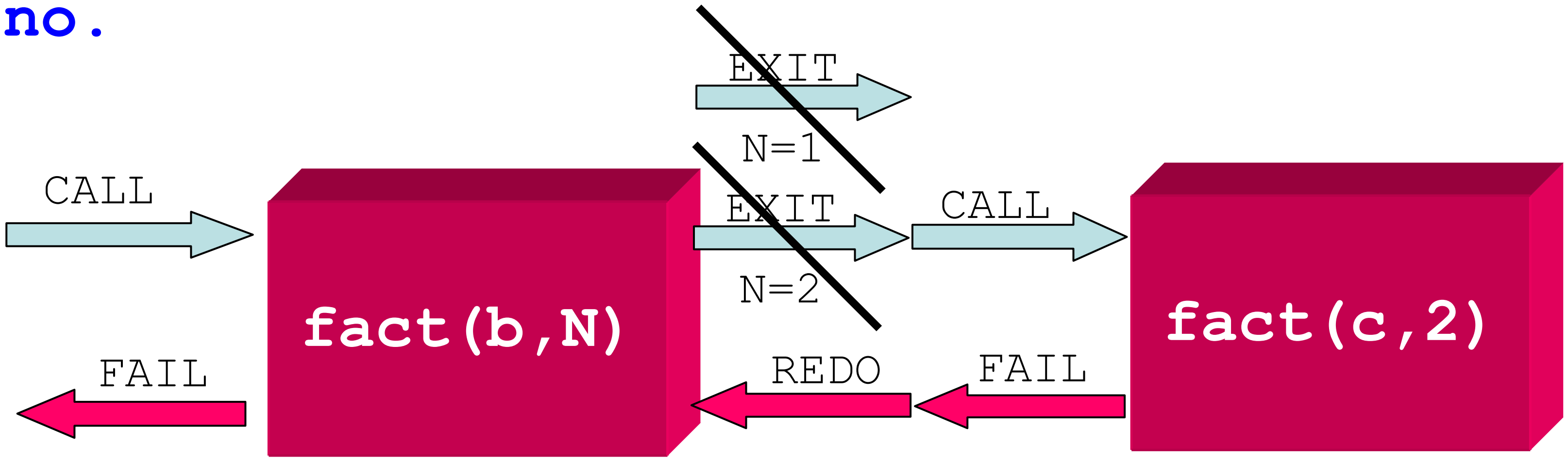
Redo-ing a Goal (3)

```
fact(b,1).
fact(b,2).
a :- fact(b,N), fact(c,N).

|?- a.
```



Redo-ing a Goal (4)

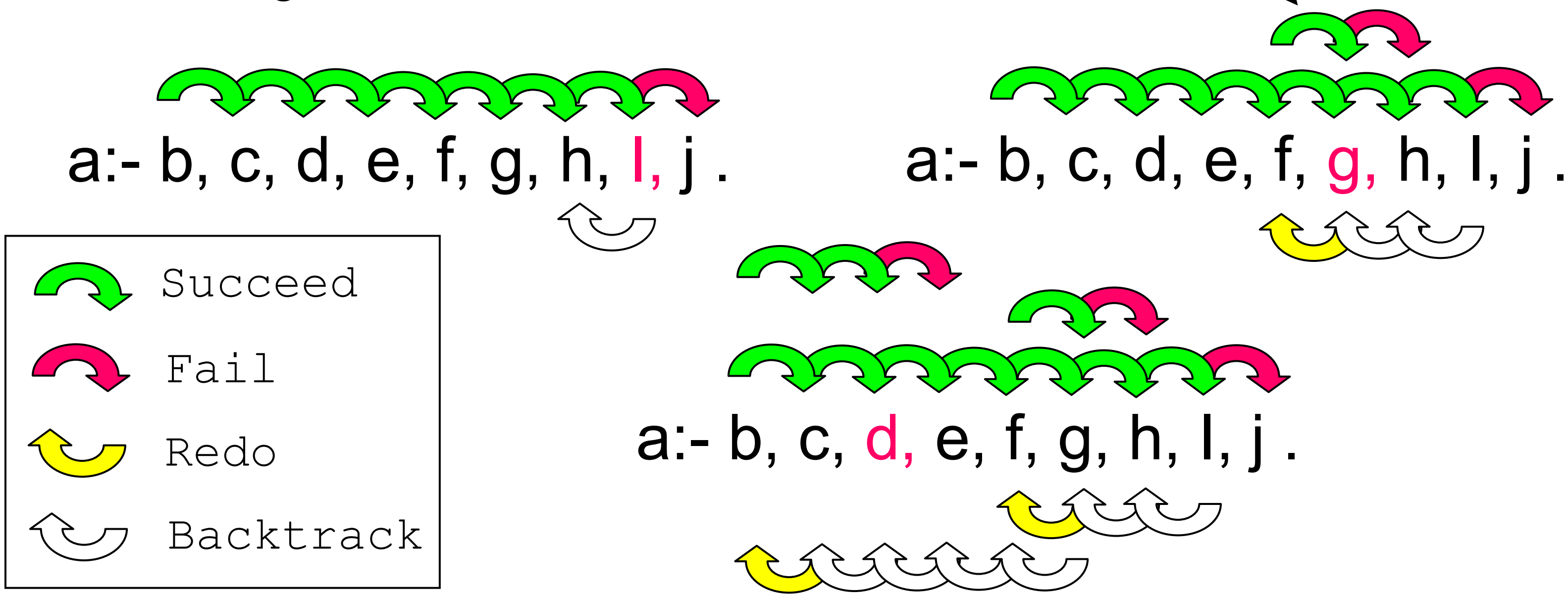
```
fact(b,1).
fact(b,2).
a :- fact(b,N), fact(c,N).

|?- a.
no.
```



Prolog's Persistence

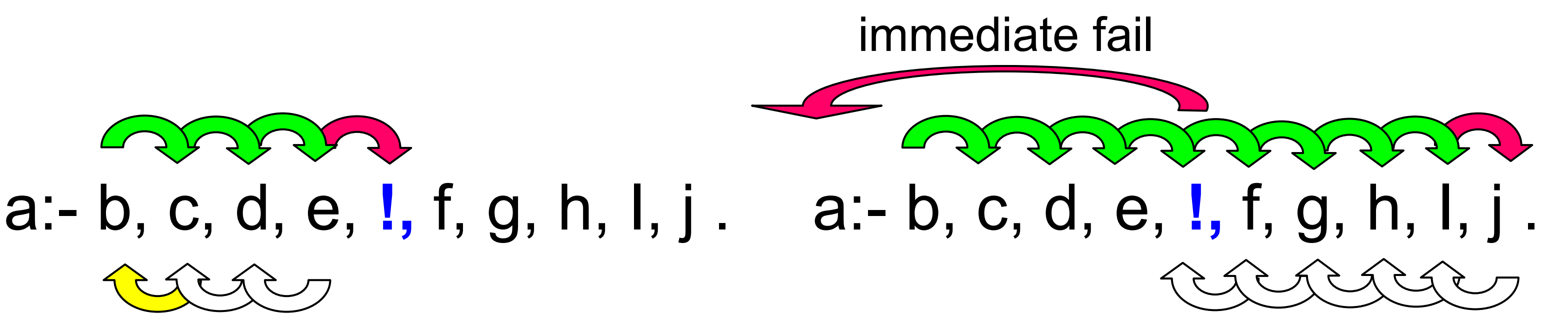
- When a sub-goal fails, Prolog will backtrack to the most recent successful goal and try to find another match.
- Once there are no more matches for this sub-goal it will backtrack again; retrying every sub-goal before failing the parent goal.
- A `call` can match any clause head.
- A `redo` ignores old matches.

A new instantiation



Cut !

- If we want to restrict backtracking we can control which sub-goals can be redone using the cut = **!**.
- We use it as a goal within the body of clause.
- It succeeds when `called`, but `fails` the parent goal (the goal that matched the head of the clause containing the cut) when an attempt is made to `redo` it on backtracking.
- It commits to the choices made so far in the predicate.
  - unlimited backtracking can occur before and after the cut but no backtracking can go through it.



Failing the parent goal

```
a:- b, c, d, e, !, f, g, h, I, j .
a:- k.
a:- m .
```

Treated as if don't exist

```
a:- b, c, d, e, !, f, g, h, I, j .
a:- k.
a:- m .
```

This clause and these choices committed to

- The cut succeeds when it is `called` and commits the system to all choices made between the time the parent goal was invoked and the cut.
- This includes committing to the clause containing the cut.
  = the goal can only succeed if this clause succeeds.
- When an attempt is made to backtrack through the cut
  - the clause is immediately failed, and
  - no alternative clauses are tried.

Mutually Exclusive Clauses

- We should only use a cut if the clauses are mutually exclusive (if one succeeds the others won't).
- If the clauses are mutually exclusive then we don't want Prolog to try the other clauses when the first fails

  = redundant processing.

- By including a cut in the body of a clause we are committing to that clause.
  - Placing a cut at the start of the body commits to the clause as soon as head unification succeeds.

    ```
    a(1,X):- !, b(X), c(X).
    ```

  - Placing a cut somewhere within the body (even at the end) states that we cannot commit to the clause until certain sub-goals have been satisfied.

    ```
    a(_,X):- b(X), c(X), !.
    ```

Mutually Exclusive Clauses (2)

```
f(X,0):- X < 3.
f(X,1):- 3 =< X, X < 6.
f(X,2):- 6 =< X.
```

```
|?- trace, f(2,N).
   1      1 Call: f(2,_487) ?
   2      2 Call: 2<3 ?
   2      2 Exit: 2<3 ? ?
   1      1 Exit: f(2,0) ?
N = 0 ? ;
   1      1 Redo: f(2,0) ?
   3      2 Call: 3=<2 ?
   3      2 Fail: 3=<2 ?
   4      2 Call: 6=<2 ?
   4      2 Fail: 6=<2 ?
   1      1 Fail: f(2,_487) ?
no
```

Green Cuts !

```
f(X,0):- X < 3, !.
f(X,1):- 3 =< X, X < 6, !.
f(X,2):- 6 =< X.
```

If you reach this point don't bother trying any other clause.

```
|?- trace, f(2,N).
   1      1 Call: f(2,_487) ?
   2      2 Call: 2<3 ?
   2      2 Exit: 2<3 ? ?
   1      1 Exit: f(2,0) ?
N = 0 ? ;
no
```

- Notice that the answer is still the same, with or without the cut.
  - This is because the cut does not alter the logical behaviour of the program.
  - It only alters the procedural behaviour: specifying which goals get checked when.
- This is called a *green cut*. It is the correct usage of a cut.
- Be careful to ensure that your clauses are actually mutually exclusive when using green cuts!

Red Cuts !

```
f(X,0):- X < 3, !.
f(X,1):- 3 =< X, X < 6, !.
f(X,2):- 6 =< X.
```

Redundant?

```
| ?- f(7,N).
   1      1 Call: f(7,_475) ?
   2      2 Call: 7<3 ?
   2      2 Fail: 7<3 ?
   3      2 Call: 3=<7 ?
   3      2 Exit: 3=<7 ?
   4      2 Call: 7<6 ?
   4      2 Fail: 7<6 ?
   5      2 Call: 6=<7 ?
   5      2 Exit: 6=<7 ?
   1      1 Exit: f(7,2) ?
N = 2 ?
yes
```

- Because the clauses are mutually exclusive and ordered we know that once the clause above fails certain conditions must hold.
- We might want to make our code more efficient by removing superfluous tests.

Red Cuts !

```
f(X,0):- X < 3, !.
f(X,1):- X < 6, !.
f(X,2).
```

```
f(X,0):- X < 3.
f(X,1):- X < 6.
f(X,2).
```

```
| ?- f(7,N).
   1      1 Call: f(7,_475) ?
   2      2 Call: 7<3 ?
   2      2 Fail: 7<3 ?
   3      2 Call: 7<6 ?
   3      2 Fail: 7<6 ?
   1      1 Exit: f(7,2) ?
N = 2 ?
yes
```

```
| ?- f(1,Y).
  1      1 Call: f(1,_475) ?
  2      2 Call: 1<3 ?
  2      2 Exit: 1<3 ? ?
  1      1 Exit: f(1,0) ?
  Y = 0 ? ;
  1      1 Redo: f(1,0) ?
  3      2 Call: 1<6 ?
  3      2 Exit: 1<6 ? ?
  1      1 Exit: f(1,1) ?
  Y = 1 ? ;
  1      1 Redo: f(1,1) ?
  1      1 Exit: f(1,2) ?
  Y = 2 ?
  yes
```

Using the cut

- *Red cuts* change the logical behaviour of a predicate.
- TRY NOT TO USE RED CUTS!
- Red cuts make your code hard to read and are dependent on the specific ordering of clauses (which may change once you start writing to the database).
- If you want to improve the efficiency of a program use *green cuts* to control backtracking.
- Do not use cuts in place of tests.

To ensure a logic friendly cut either:

```
p(X):- test1(X), !, call1(X).
p(X):- test2(X), !, call2(X).
p(X):- testN(X), !, callN(X).
```

`testI` predicates are mutually exclusive.

```
p(1,X):- !, call1(X).
p(2,X):- !, call2(X).
p(3,X):- !, callN(X).
```

The mutually exclusive tests are in the head of the clause.

PROLOG

Cut - fail

- As well as specifying conditions under which a goal can succeed sometimes we also want to specify when it should fail.
- We can use the built-in predicate `fail` in combination with a cut to achieve this: " `!, fail.` "

  = if you reach this point, fail regardless of other clauses.
- e.g. If we want to represent the fact that '*Mary likes all animals except snakes*'.

```
likes(mary,X):-
    snake(X), !, fail.

likes(mary,X):-
    \+ snake(X),
    animal(X).
```

We need to combine a cut with the fail to stop the redundant call to the second clause on backtracking.

Cut – fail: why?

- However, using a cut-fail can make your code hard to follow.
- It is generally clearer and easier to define the conditions under which a fact is true rather than when it is false.

```
likes(mary,X):-
    \+ snake(X),
    animal(X).
```

} This is sufficient to represent the fact.

- However, sometimes it can be much simpler to specify when something is false rather than true so cut-fail can make your code more efficient.
- As with all cuts; be careful how you use it.

PROLOG

Summary

- Clearing up equality: `=, is, =:=, =\=, ==, \==, \+`
- REDO vs. CALL
- Controlling backtracking: the cut `!`
  - Efficiency: avoids needless REDO-ing which cannot succeed.
  - Simpler programs: conditions for choosing clauses can be simpler.
  - Robust predicates: definitions behave properly when forced to REDO.
- Green cut = cut doesn't change the predicate logic = **good**
- Red cut = without the cut the logic is different = **bad**
- Cut – fail: when it is easier to prove something is false than true.