# Prolog's Cut Operator

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science



# Outline







# Objectives

You should be able to...

Prolog's greatest strength is its ability to backtrack to find alternative solutions. If not controlled, it can also be its greatest weakness. In this lecture we will go over the cut operator, which gives a solution to this problem.

- Know what the cut operator is and what it does.
- Know how to use the cut operator to assert failure.
- Know how to use the cut operator to stop recursion.





# Backtracking

```
color(red).
color(blue).
car(honda).
car(ford).
car(toyota).

?- color(A), car(B).
```

```
A = red
B = honda ;
A = red
B = ford ;
A = red
B = toyota ;
A = blue
B = honda ;
A = blue
B = ford ;
A = blue
B = toyota ;
```

Come and see the backtracking inherent in the system!

## The Cut operator

- The Cut operator (`!`) stops backtracking.
- It is considered a goal that always succeeds.

```
?- color(A), !, car(B).

A = red
B = honda ;
A = red
B = ford ;
A = red
B = toyota ;
No
```





## Commitment

Once a cut is activated, the clause we are trying to satisfy is committed to that choice.

```
color(red).
color(green) :- !.
color(blue).

?- color(X).
X = red ;
X = green ;
No
```

Once `X` was set to `green`, the cut operator forces us to stay with `green` or else `color` should fail completely. **Question**: Can `color(blue)` ever be matched?





## Factorial revisited

```
fact(0,1).
fact(N,X) :- M is N-1, fact(M,Y),
             X is Y * N.

?- fact(5,N).

N = 120 ;
ERROR: Out of local stack
```

- What happened here?





## Two fixes

You can add a constraint to the second clause....

```
fact(0,1).
fact(N,X) :- N > 0, M is N-1, fact(M,Y),
             X is Y * N.
```

Or you can add a cut to the first clause.

```
fact(0,1) :- !.
fact(N,X) :- M is N-1, fact(M,Y),
             X is Y * N.
```

Now it will work:

```
?- fact(5,N).
N = 120 ;
No
```

# Efficiency

Suppose you run the campus observatory. You want to allow certain people to use the telescope. They have to be a student, a faculty member, or a member of the astronomy club. And they also need to have been trained on the telescope.

| | |
|---|---|
| students | anna, beth, cindy, david |
| faculty | ernest, frank, gloria |
| astronomy club | anna, frank, harry |
| trained | anna, harry |





# Can frank use the telescope?

```
telescope(X) :- (student(X); faculty(X);
                 club(X)),
                trained(X).
```

- `frank` is a faculty, and also a member of the club.
- But, `frank` doesn't have any training.

What will `telescope(frank).` do?





# Who can use the telescope?

```
telescope(X) :- (student(X); faculty(X);
                 club(X)),
                trained(X).

?- telescope(X).
X = anna ;
X = anna ;
X = harry ;
```

Since `anna` is a student *and* a member of the club, she gets listed twice.





# Now we cut...

```
telescope(X) :- (student(X); faculty(X);
                 club(X)),
                !,
                trained(X).

?- telescope(X).
X = anna ;
No
```

- Oops. Now we've dissed `harry`.
- But at least we don't spend a lot of time when we ask if `frank` can use the telescope....
- Moral: cut will limit your choices to only one answer.

## Inducing failure.

- We also have a predicate called `fail`, which, well, always fails.
- Suppose anna has her telescope privileges revoked...

```
telescope(anna) :- fail.
telescope(X) :- (student(X); faculty(X); club(X)),
                trained(X).

?- telescope(anna).
Yes.
```

This is less than what we hoped for.





## Inducing failure II

```
telescope(anna) :- !, fail.
telescope(X) :- (student(X); faculty(X);
                  club(X)),
                 trained(X).

?- telescope(anna).
No
?- telescope(harry).
Yes
?- telescope(X).
No
```

But cut and fail will work.





## Conclusions

- Cut can stop searches that you already know will be useless.
- Cut can make queries more efficient.
- But, cut can make queries do strange things. Use with care.

Aside: you can define `not` (actually, it's built in) this way:

```
not(X) :- call(X), !, fail.
not(X).
```

This predicate can fix the telescope problems.





## Problems

1. Write a predicate `between(X,Y,Z)` which is true when `Y` is a point between `X` and `Z`.
2. Write a predicate `grandfatherof(X,Y)` which is true when `X` is a grandfather of `Y`. You might want to write another predicate first.
3. Write a predicate `flatten(X,Y)` that is true when `Y` is a flattened version of `X`. E.g.

```
?- flatten([x,3,[a,b],3,
           [[3]]],[x,3,a,b,3,3]).
Yes
```

Hint: there is a predicate called `is_list`, which is true when its argument is a list.

## Problems 1 and 2

```
between(X,Y,Z) :- pathfrom(X,Y),
                  pathfrom(Y,Z).
parentof(X,Y) :- fatherof(X,Y);
                 motherof(X,Y).
grandfatherof(X,Y) :- fatherof(X,Z),
                      parentof(Z,Y).
```





## Almost answer 3 — What goes wrong?

```
myflatten([H|T],X) :- is_list(H),
           append(H,T,R), myflatten(R,X).
myflatten([H|T],[H|X]) :- myflatten(T,X).
myflatten([],[]).

?- myflatten([[2,3],[3,4,[5,6],4],3],X).
X = [2, 3, 3, 4, 5, 6, 4, 3] ;
X = [2, 3, 3, 4, [5, 6], 4, 3] ;
X = [2, 3, [3, 4, [5, 6], 4], 3] ;
X = [[2, 3], 3, 4, 5, 6, 4, 3] ;
X = [[2, 3], 3, 4, [5, 6], 4, 3] ;
X = [[2, 3], [3, 4, [5, 6], 4], 3] ;
No
```





## Answer 3

```
myflatten([H|T],X) :- is_list(H), !,
              append(H,T,R), myflatten(R,X).
myflatten([H|T],[H|X]) :- myflatten(T,X).
myflatten([],[]).
```