Backtracking in Prolog

- ✔ When answering a query, the Prolog algorithm performs depth-first (preorder) traversal of a tree of goal lists
- ✔ The root of the tree contains the goals in the query
- ✔ Each child of a node corresponds to a different rule whose head matches the first goal in the parent goal list
  - ✗ that first goal is replaced by the body of the rule
  - ✗ variable bindings are updated accordingly
- ✔ If the goal list is empty, this is a “success” leaf node
- ✔ If the first goal in the goal list cannot be matched with any rule, this is a “failure” leaf node
- ✔ A “failure” leaf node causes automatic backtracking
- ✔ A “success” leaf node causes backtracking if the user types “;”

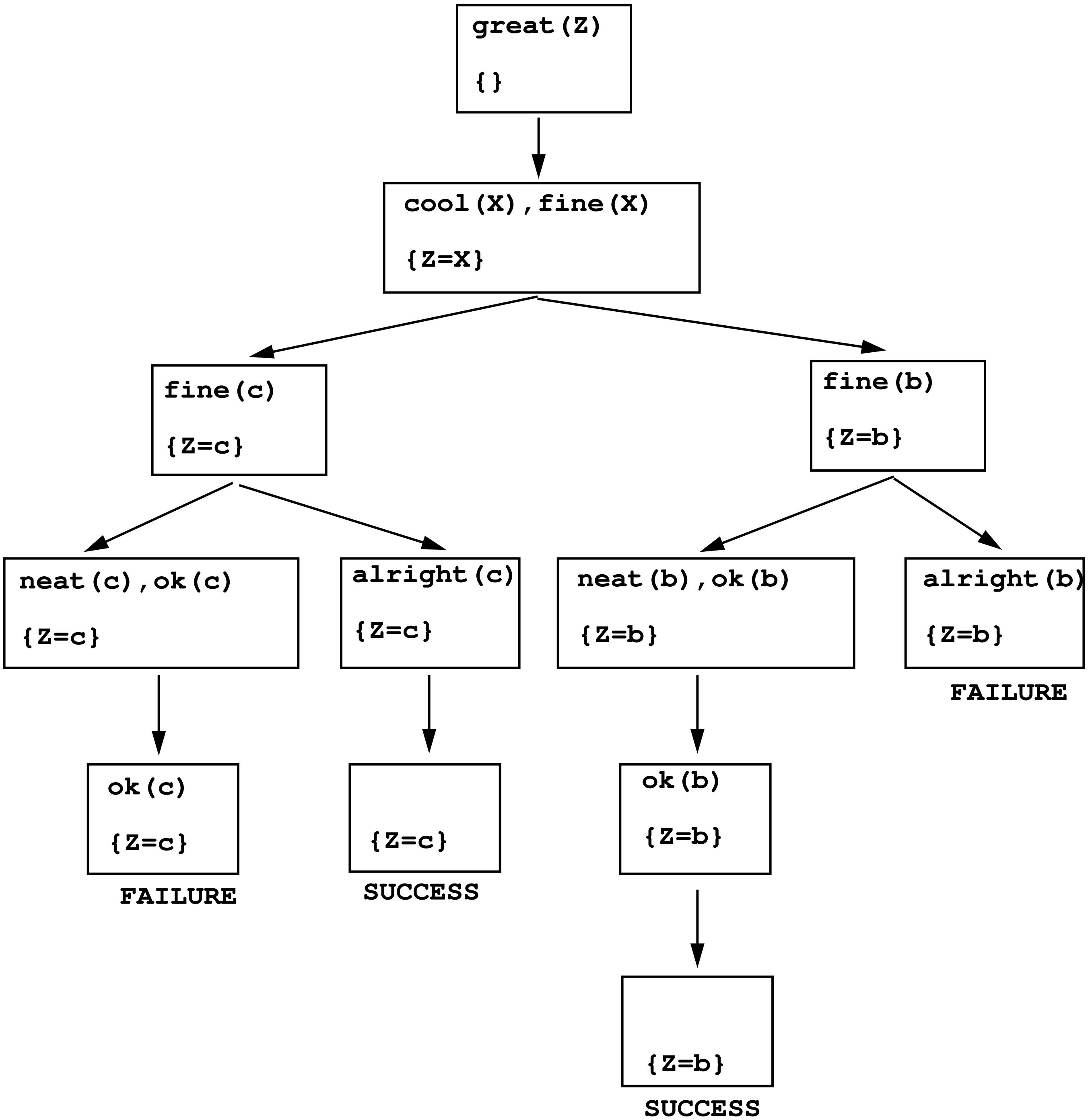
Backtracking in Prolog

- Suppose the following program

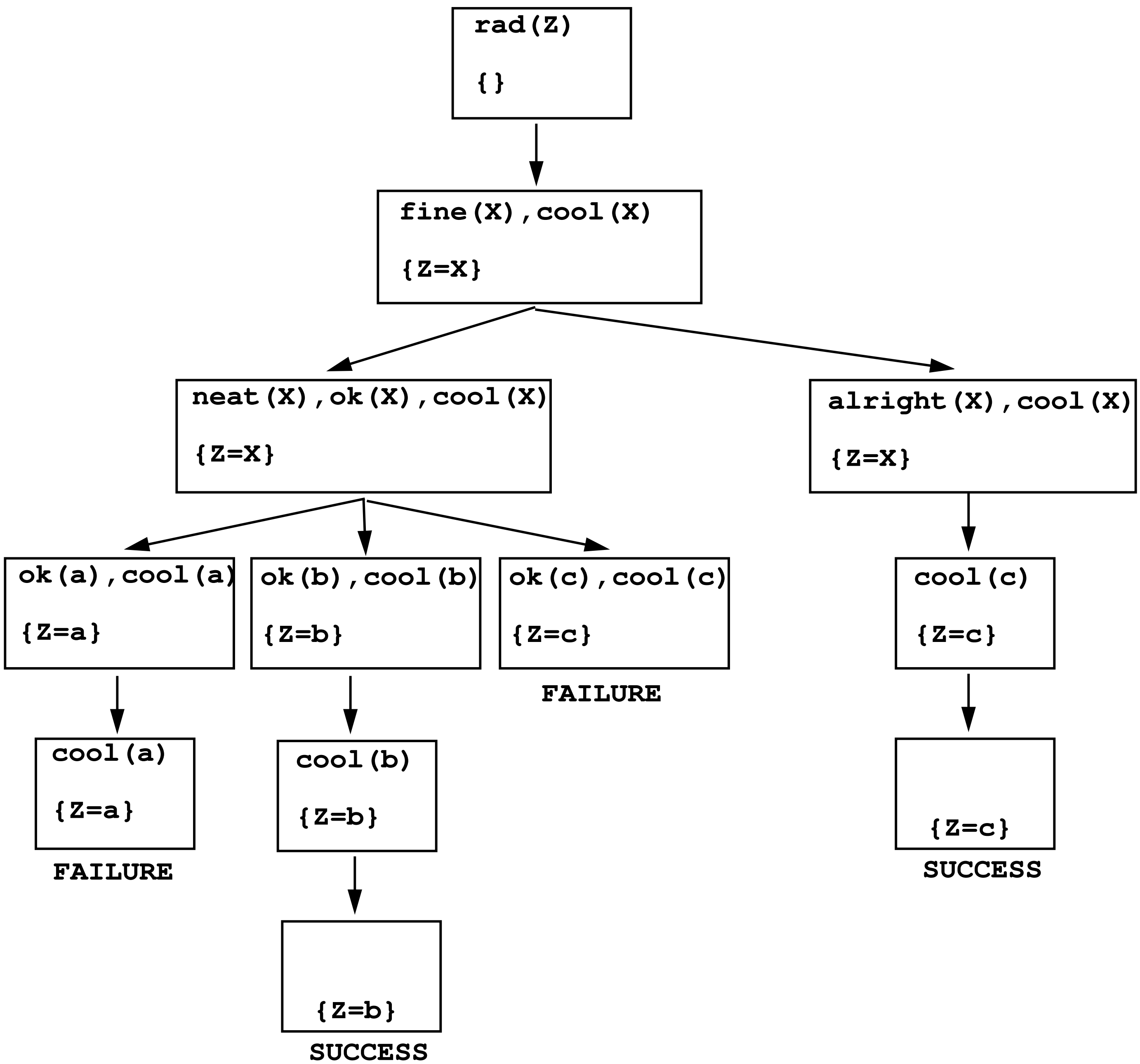
```
cool(c).
cool(b).
neat(a).
neat(b).
neat(c).
alright(c).
ok(a).
ok(b).
```

```
fine(X) :- neat(X),ok(X).

fine(X) :- alright(X).

great(X) :- cool(X),fine(X).

rad(X) :- fine(X),cool(X).
```

?- great(Z).


great(Z)
{}
cool(X),fine(X)
{Z=X}
fine(c)
{Z=c}
fine(b)
{Z=b}
neat(c),ok(c)
{Z=c}
alright(c)
{Z=c}
neat(b),ok(b)
{Z=b}
alright(b)
{Z=b}
FAILURE
ok(c)
{Z=c}
FAILURE
{Z=c}
SUCCESS
ok(b)
{Z=b}
{Z=b}
SUCCESS

```
?- rad(Z).
```



rad(Z)
{}
fine(X),cool(X)
{Z=X}
neat(X),ok(X),cool(X)
{Z=X}
alright(X),cool(X)
{Z=X}
ok(a),cool(a)
{Z=a}
ok(b),cool(b)
{Z=b}
ok(c),cool(c)
{Z=c}
FAILURE
cool(c)
{Z=c}
cool(a)
{Z=a}
FAILURE
cool(b)
{Z=b}
{Z=c}
SUCCESS
{Z=b}
SUCCESS

Controlling backtracking

- ✔ Sometimes, for reasons of efficiency or correctness, we want to modify Prolog's ordinary flow of backtracking

- ✔ The *cut* mechanism "!" provides a way of doing this

- ✔ ! is used like a term
  - ✗ encountered as a goal, ! succeeds immediately; however...
  - ✗ ! commits the search to all choices made since the "parent goal" was the first goal in the goal list
  - ✗ the parent goal is the goal that matched the head of the rule containing the !

Controlling backtracking

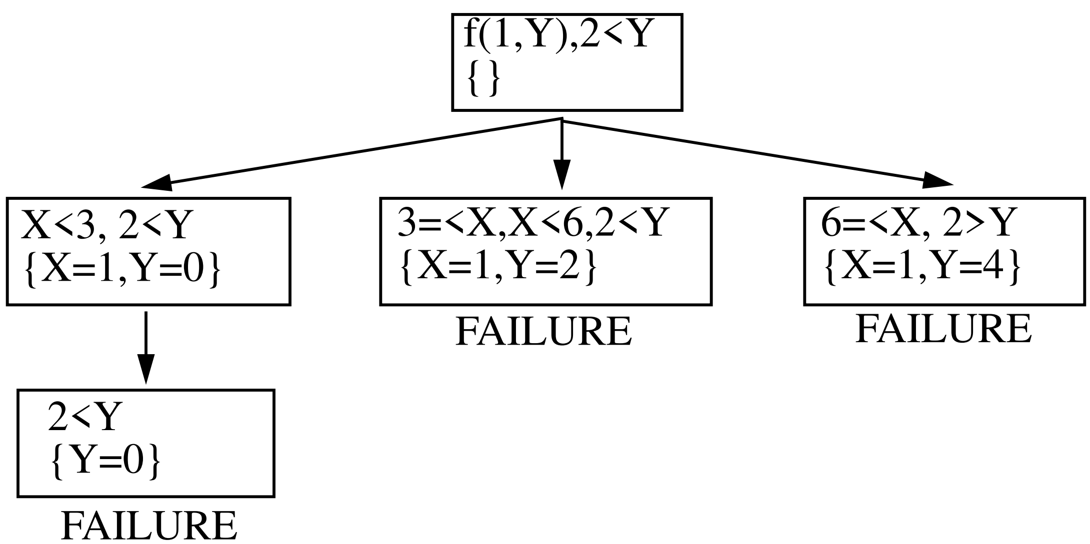
- Consider the program

```
f(X,0) :- X < 3.
f(X,2) :- 3 =< X, X < 6.
f(X,4) :- 6 =< X.
```

- and the query

```
?- f(1,Y), 2<Y.
no
```

- Prolog tries all 3 rules in attempts to satisfy these goals



- This is inefficient: if the first rule for `f` works, the others must fail: their conditions are mutually exclusive

Controlling backtracking

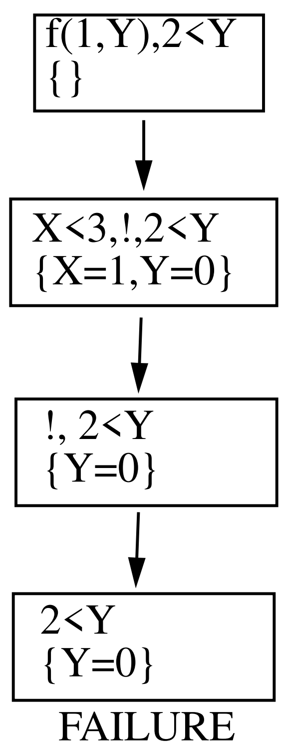
- ! can make this program more efficient. Consider the program

```
f(X,0) :- X < 3, !.
f(X,2) :- 3 =< X, X < 6, !.
f(X,4) :- 6 =< X.
```

- and the query

```
?- f(1,Y), 2<Y.
no
```

- Read this program as "f(X,0) is true if X < 3; else f(X,2) is true if X >= 3 and X < 6; else f(X,4) is true if X >= 6." With the cuts, Prolog will try only the first rule:



Controlling backtracking

- In fact, this program can be made even more efficient:

```
f(X,0) :- X < 3, !.
f(X,2) :- X < 6, !.
f(X,4).
```

- Read this program as "f(X,0) is true if X < 3; else f(X,2) is true if X < 6; else f(X,4) is true."

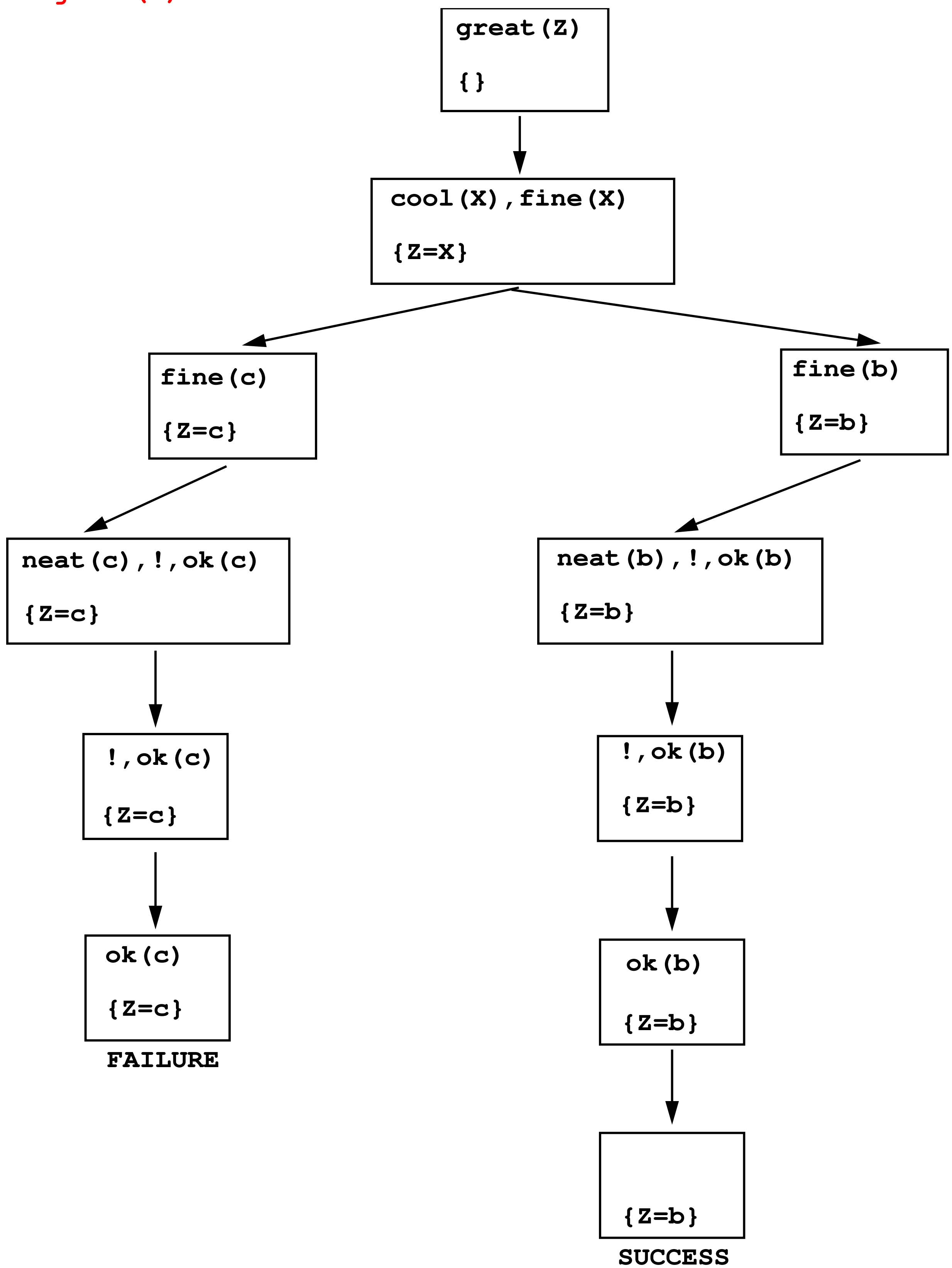
- Note that now the program will not work correctly without the cuts!

Using !

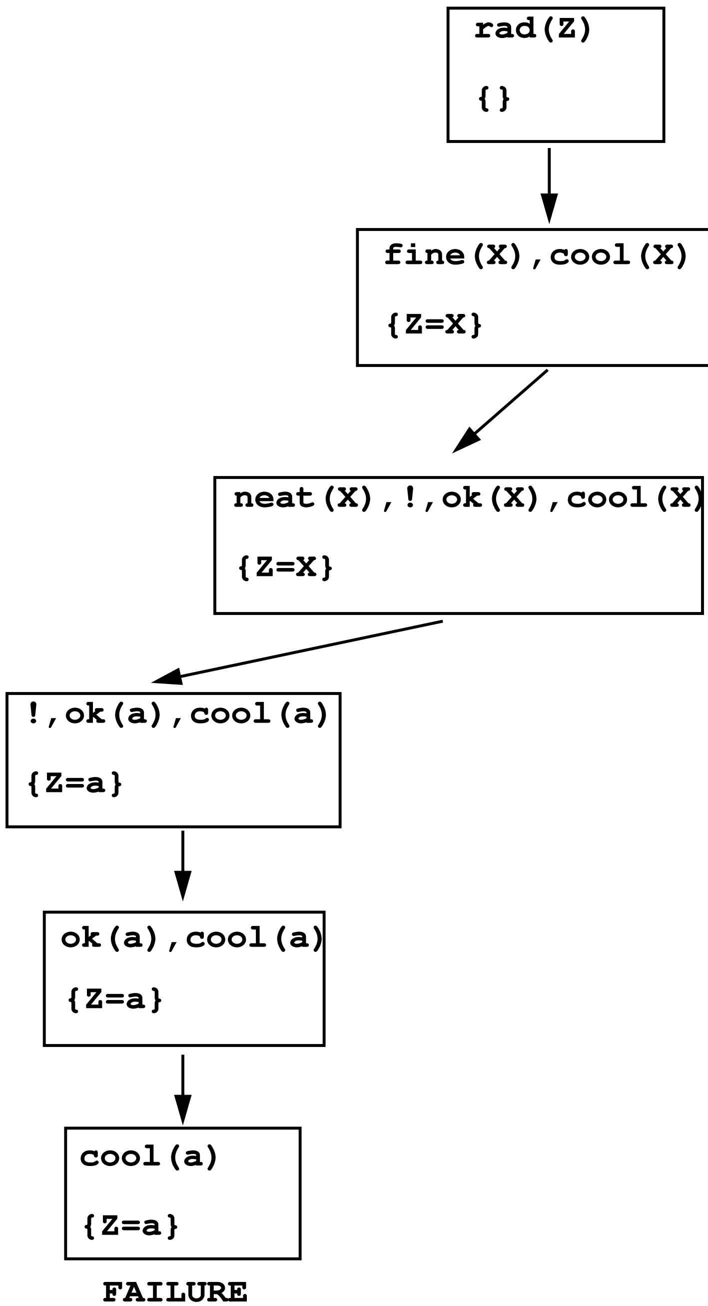
- Suppose the following program

```
cool(c).
cool(b).
neat(a).
neat(b).
neat(c).
alright(c).
ok(a).
ok(b).



fine(X) :- neat(X),!,ok(X).

fine(X) :- alright(X).

great(X) :- cool(X),fine(X).

rad(X) :- fine(X),cool(X).
```

?- great(Z).


great(Z)
{}
cool(X),fine(X)
{Z=X}
fine(c)
{Z=c}
fine(b)
{Z=b}
neat(c),!,ok(c)
{Z=c}
neat(b),!,ok(b)
{Z=b}
!,ok(c)
{Z=c}
!,ok(b)
{Z=b}
ok(c)
{Z=c}
FAILURE
ok(b)
{Z=b}
{Z=b}
SUCCESS

```
?- rad(Z).
```


rad(Z)
{}
fine(X),cool(X)
{Z=X}
neat(X),!,ok(X),cool(X)
{Z=X}
!,ok(a),cool(a)
{Z=a}
ok(a),cool(a)
{Z=a}
cool(a)
{Z=a}
FAILURE

Examples using cut

- ✔ The program for the `max` relation

```
max(X,Y,X) := X >= Y.
max(X,Y,Y) := Y > X.
```

can be made more efficient:

```
max(X,Y,X) := X >= Y, !.
max(X,Y,Y).
```

“if X>=Y then max(X,Y,X) is true, else max(X,Y,Y) is true.”

Examples using cut

- ✔ The following definition of `del(X,L1,L2)` will delete any occurrence of `X` from `L1`

```
del(X,[X|Tail],Tail).
del(X,[Y|Tail1],Tail2) :- del(X,Tail1,Tail2).

?- del(a,[a,b,a,a],L).
L = [b,a,a];
L = [a,b,a];
L = [a,b,a];
no
```

- ✔ We can define a version of `del` using cut that will delete only the first occurrence

```
del(X,[X|Tail],Tail) :- !.
del(X,[Y|Tail1],Tail2) :- del(X,Tail1,Tail2).

?- del(a,[a,b,a,a],L).
L = [b,a,a];
no
```

Examples using cut

- ✔ Defining the `add` relation: `add(X,L1,L2)` is true if `L1 == L2` and `X` is a member of `L1`; else L2 is equal to L1 with X inserted.

Without cut:

```
add(X,L,L) :- member(X,L).
add(X,L,[X|L]).

?- add(a,[a,b],L).
L = [a,b];
L = [a,a,b];
no
```

With cut:

```
add(X,L,L) :- member(X,L),!.
add(X,L,[X|L]).


?- add(a,[a,b],L).
L = [a,b];
no
```

Examples using cut

- ✔ Consider the program

```
p(1).
p(2) :- !.
p(3).
```

What are the answers to these queries:

```
?- p(X).
```

```
X = 1;
X = 2;
no
```

```
?- p(X),p(Y).
```

```
X = 1
Y = 1;
X = 1
Y = 2;
X = 2
Y = 1;
X = 2
Y = 2;
no
```

```
?- p(X),!,p(Y).
```

```
X = 1
Y = 1;
X = 1
Y = 2;
no
```

Using !

- Some uses of ! do not change the declarative semantics of the program: the only change is an improvement in efficiency
  - these are called “green cuts”
- Some uses of ! do change the declarative semantics of the program, moving it away from “pure Prolog”
  - these are called “red cuts”

Negation as failure

- ✔ Represent “Pat likes any ice cream except chocolate” in Prolog
  - ✗ If X is ice cream and it’s chocolate, Pat does not like it
  - ✗ Otherwise, if it’s ice cream, Pat likes it

```
likes(pat,X) :- ice_cream(X),chocolate(X),!,fail.
likes(pat,X) :- ice_cream(X).
```

`fail` is a goal that always fails

...why doesn’t the following work:

```
likes(pat,X) :- ice_cream(X),chocolate(X),fail,!.
likes(pat,X) :- ice_cream(X).
```

Negation as failure

✔ Another way to represent "Pat likes any ice cream except chocolate" in Prolog

✗ If X is ice cream and it's not chocolate, Pat likes it

```
likes(pat,X) :- ice_cream(X), not chocolate(X).
```

Negation as failure

- ✔ A goal of the form

  not *goal*

  succeeds if

  goal

  fails; otherwise, it fails

- ✔ This is *negation as failure*

- ✔ **not** is defined as

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

Negation as failure

- Negation as failure is not the usual interpretation of negation in logic

- In logic, if it is impossible to prove a sentence `P` from a set of sentences `S`, that does not mean it is possible to prove the sentence `~P` from the set of sentences `S`

- Standard logics have the property of *monotonicity:*
  - Suppose `S1` and `S2` are two sets of sentences, such that every sentence in `S1` is also in `S2.` Then, if a sentence `P` is provable from `S1`, it is also provable from `S2`
  - That is, adding more to your axioms can never reduce what you can prove from the axioms

- Prolog uses the "closed world assumption": Assume that all relevant facts are represented in the program
  - This gives nonmonotonic inference: adding more facts to your program can prevent proving some terms that you could prove before

Monotonicity and nonmonotonicity

- In ordinary logic, if your axioms are

```
man(socrates).

mortal(X) :- man(X).
```

you cannot prove `mortal(john)`
and you cannot prove `not mortal(john)`

- In Prolog, if your program is

```
man(socrates).

mortal(X) :- man(X).
```

you cannot prove `mortal(john)`
but because of that, you can prove `not mortal(john)`

(closed world assumption: Assume that if `mortal(john)` were true, it would be represented in the program)

- Now if you add `man(john)`,
  - in logic, you can prove `mortal(john)`
  - in Prolog, you can no longer prove `not mortal(john)`