

# **CSE 259 - Logic in Computer Science**

## **Fall 2024**

**Recitation-3**

## **Recursion and Cut**

**Waqar Hassan Khan**



# Recursion - Definition

**Definition:** Recursion is a technique in which one predicate uses itself to find the truth value.

**Example:**

predecessor(X, Z) :- parent(X, Z).

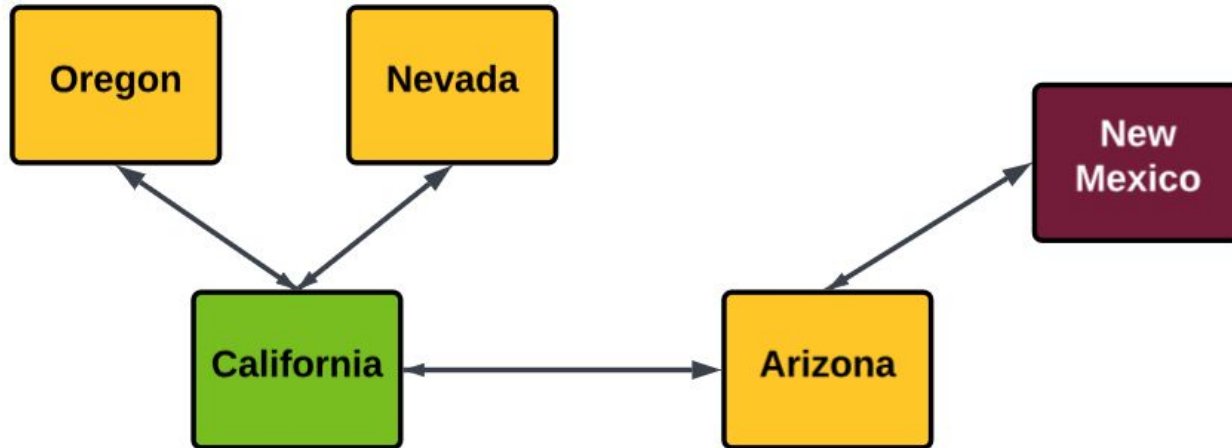
**Base case**

predecessor(X, Z) :- parent(X, Y),predecessor(Y, Z).

# Recursion - Applying recursion

Last time we solved this problem had a **limitation**: Travelling from Oregon to New Mexico was not possible :(

**Solution:** Using recursion



# Recursion - Applying recursion

```
1  % facts
2  next_to(california, arizona).
3  next_to(california, oregon).
4  next_to(california, nevada).
5  next_to(arizona, new_mexico).
6
7  % a rule where we write the logic of when two states are connected
8  connected(A, B) :- next_to(A, B) ; next_to(B, A).
9
10 /*
11  | * recursive rule
12  | */
13 path(A, B) :- connected(A, B).
14 path(A, B) :- connected(A, C), C \== B, path(C, B).
15
16 travel(A, B) :- path(A, B).
```

# Recursion - Applying recursion

**Query:** | ?-travel(oregon, new\_mexico).

path(A, B) :- connected(A, B).

path(A, B) :- connected(A, C), C \== B, path(C, B).

**A**      **B**                      **A**                      **B**                      **B**  
path(oregon, new\_mexico) :- connected(oregon, C), C \== new\_mexico, path(C, new\_mexico)

# Recursion - Applying recursion

**Query:** | ?-travel(oregon, new\_mexico).

path(A, B) :- connected(A, B).

path(A, B) :- connected(A, C), C \== B, path(C, B).

**A**      **B**                      **A**                      **B**                      **B**  
path(oregon, new\_mexico) :- connected(oregon, C), C \== new\_mexico, path(C, new\_mexico)

**A**      **B**                      **A**      **C**      **C**                      **B**                      **C**      **B**  
path(oregon, new\_mexico) :- connected(oregon, california), california \== new\_mexico, path(california, new\_mexico)

# Recursion - Applying recursion

**Query:** | ?-travel(oregon, new\_mexico).

path(A, B) :- connected(A, B).

path(A, B) :- connected(A, C), C \== B, path(C, B).

**A**      **B**                      **A**                      **B**                      **B**  
path(oregon, new\_mexico) :- connected(oregon, C), C \== new\_mexico, path(C, new\_mexico).

**A**      **B**                      **A**      **C**      **C**                      **B**                      **C**      **B**  
path(oregon, new\_mexico) :- connected(oregon, california), california \== new\_mexico, path(california, new\_mexico).

**A**      **B**                      **A**                      **B**                      **B**  
path(california, new\_mexico) :- connected(california, C), C \== new\_mexico, path(C, new\_mexico).

# Recursion - Applying recursion

**Query:** | ?-travel(oregon, new\_mexico).

path(A, B) :- connected(A, B).

path(A, B) :- connected(A, C), C \== B, path(C, B).

**A**      **B**                      **A**                      **B**                      **B**  
path(oregon, new\_mexico) :- connected(oregon, C), C \== new\_mexico, path(C, new\_mexico).

**A**      **B**                      **A**      **C**      **C**                      **B**                      **C**      **B**  
path(oregon, new\_mexico) :- connected(oregon, california), california \== new\_mexico, path(california, new\_mexico).

**A**      **B**                      **A**                      **B**                      **B**  
path(california, new\_mexico) :- connected(california, C), C \== new\_mexico, path(C, new\_mexico).

**A**      **B**                      **A**      **C**      **C**                      **B**                      **A**      **B**  
path(california, new\_mexico) :- connected(california, arizona), arizona \== new\_mexico, path(arizona, new\_mexico).  
path(california, new\_mexico) :- connected(california, oregon), oregon \== new\_mexico, path(oregon, new\_mexico).  
path(california, new\_mexico) :- connected(california, nevada), nevada \== new\_mexico, path(nevada, new\_mexico).



# Recursion - Applying recursion

**Query:** | ?-travel(oregon, new\_mexico).

path(A, B) :- connected(A, B).

path(A, B) :- connected(A, C), C \== B, path(C, B).

**A**      **B**                      **A**                      **B**                      **B**  
path(oregon, new\_mexico) :- connected(oregon, C), C \== new\_mexico, path(C, new\_mexico).

**A**      **B**                      **A**      **C**      **C**                      **B**                      **C**      **B**  
path(oregon, new\_mexico) :- connected(oregon, california), california \== new\_mexico, path(california, new\_mexico).

**A**      **B**                      **A**                      **B**                      **B**  
path(california, new\_mexico) :- connected(california, C), C \== new\_mexico, path(C, new\_mexico).

**A**      **B**                      **A**      **C**      **C**                      **B**                      **A**      **B**  
path(california, new\_mexico) :- connected(california, arizona), arizona \== new\_mexico, path(arizona, new\_mexico).  
path(california, new\_mexico) :- connected(california, oregon), oregon \== new\_mexico, path(oregon, new\_mexico).  
path(california, new\_mexico) :- connected(california, nevada), nevada \== new\_mexico, path(nevada, new\_mexico).

**A**      **B**                      **A**      **B**  
path(arizona, new\_mexico) :- connected(arizona, new\_mexico).

# Recursion - Applying recursion

Query: | ?-travel(oregon, new\_mexico).

 SWI-Prolog (AMD64, Multi-threaded, version 9.2.9)

File Edit Settings Run Debug Help

```
Welcome to SWI-Prolog (threaded, 64 bits, version 9.2.9)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.
```

```
For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).
```

```
?-
% e:/Programming/TA/ASU-CSE-259-Prolog/Recitation-3/travel-from-state
?- travel(oregon, new_mexico).
true.
```

```
?- ■
```

# Cut

- Represented by !
- Always succeeds - in the rules it is considered as true
- Prevents Backtracking

# Cut - example

```
1  % facts
2  red(apple).
3  red(lexus).
4  red(honda).
5  blue(monkey).
6
7  car(honda).
8  car(lexus).
9  bike(monkey).
10
11 fun(X) :- red(X), car(X).
12 fun(X) :- blue(X), bike(X).
13
```

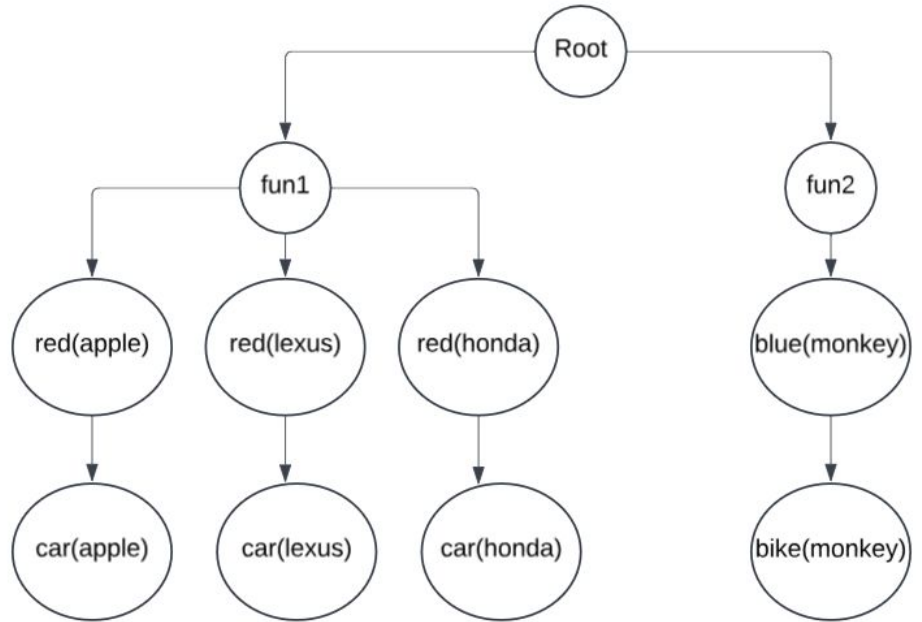
```
.
% e:/Programming/TA/ASU-1
?- fun(X).
X = lexis .

?- fun(X).
X = lexis ;
X = honda ;
X = monkey.

?-
```

# Cut - example

```
1  % facts
2  red(apple).
3  red(lexus).
4  red(honda).
5  blue(monkey).
6
7  car(honda).
8  car(lexus).
9  bike(monkey).
10
11 fun(X) :- red(X), car(X).
12 fun(X) :- blue(X), bike(X).
13
```



# Cut - example

```
1  % facts
2  red(apple).
3  red(lexus).
4  red(honda).
5  blue(monkey).
6
7  car(honda).
8  car(lexus).
9  bike(monkey).
10
11 fun(X) :- red(X), car(X), !.
12 fun(X) :- blue(X), bike(X).
13
```

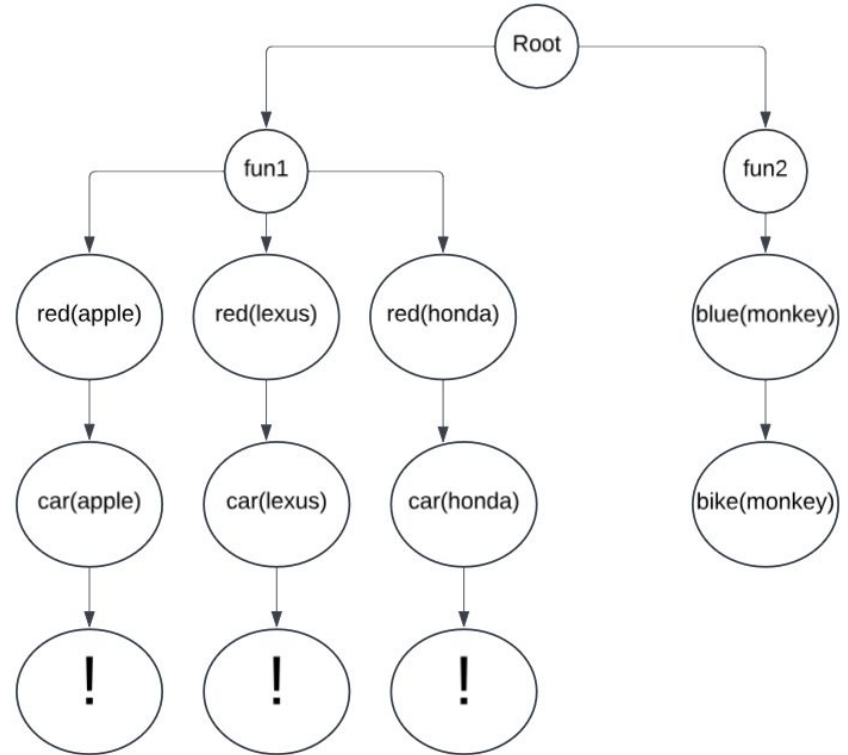
- First, X is bound to apple. apple is red, so red(X) is true. Then car(X) is checked and it is false. So, we get back to red(X).
- Now X is bound to lexus. lexus is red and a car. So we reach the cut operator. And then stop backtracking

```
% e: /Programming/
?- fun(X).
X = lexus.

?-
```

# Cut - example

```
1  % facts
2  red(apple).
3  red(lexus).
4  red(honda).
5  blue(monkey).
6
7  car(honda).
8  car(lexus).
9  bike(monkey).
10
11 fun(X) :- red(X), car(X), !.
12 fun(X) :- blue(X), bike(X).
13
```



# Cut - example - order matters!

```
1 % facts
2 red(apple).
3 red(honda).
4 red(lexus).
5 blue(monkey).
6
7 car(honda).
8 car(lexus).
9 bike(monkey).
10
11 fun(X) :- red(X), car(X), !.
12 fun(X) :- blue(X), bike(X).
13
```

```
% e:/Programming/TA/4
?- fun(X).
X = honda.

?-
```



# Cut - example - order matters!

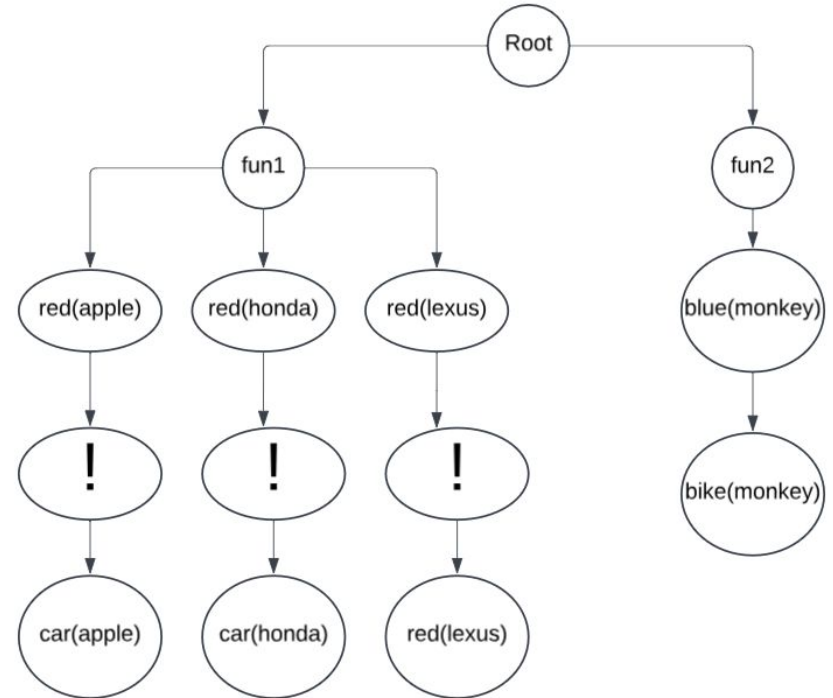
```
1  % facts
2  red(apple).
3  red(honda).
4  red(lexus).
5  blue(monkey).
6
7  car(honda).
8  car(lexus).
9  bike(monkey).
10
11 fun(X) :- red(X), !, car(X).
12 fun(X) :- blue(X), bike(X).
13
```

- X is bound to apple: apple is red. Then ! is true. The apple is not a car. Now, we need to get back to red(X) for more solution. But we can't get past the ! operator. So, not backtracking for more solution

```
% e:/Programming
?- fun(X).
false.
```

# Cut - example - order matters!

```
1  % facts
2  red(apple).
3  red(honda).
4  red(lexus).
5  blue(monkey).
6
7  car(honda).
8  car(lexus).
9  bike(monkey).
10
11 fun(X) :- red(X), !, car(X).
12 fun(X) :- blue(X), bike(X).
13
```



# Cut - example - order matters!

```
1  % facts
2  red(apple).
3  red(honda).
4  red(lexus).
5  blue(monkey).
6
7  car(honda).
8  car(lexus).
9  bike(monkey).
10
11 fun(X) :- !, red(X), car(X).
12 fun(X) :- blue(X), bike(X).
13
```

- ! is true
- X is bound to apple, X is red, X is not a car. So, we get back to red(X). The cut was before red(X) so no problem getting back.
- honda and lexus is found.
- Finally we get back to the cut. We stop backtracking which is why fun(X) :- blue(X), bike(X) is not checked.

```
% e:/Programmin
?- fun(X).
X = honda ;
X = lexus.

?-
```

# Cut - example - order matters!

```
1  % facts
2  red(apple).
3  red(honda).
4  red(lexus).
5  blue(monkey).
6
7  car(honda).
8  car(lexus).
9  bike(monkey).
10
11 fun(X) :- !, red(X), car(X).
12 fun(X) :- blue(X), bike(X).
13
```

