CSC324 Principles of Programming Languages (Week 9)

Yilan Gu
yilan@cdf.toronto.edu
http://www.dgp.toronto.edu/~yilan/324f09/

Last Week

- Basic knowledge
- Rules
- Unification of variables and terms

Today

- Operators
- Lists
- Prolog reasoning mechanism
- Recursion in Prolog

Prolog: Operators and Lists

Prolog Operators

- **X = Y**
  - Semantically: unifiable test
  - Succeeds as long as X and Y can be unified
  - X may or may not be instantiated. Y may or may not be instantiated
  - As a side effect, X and Y become bound together (refer to the same object)
  - E.g

```
?- X=joe.                                Yes  X = joe
?- a(b,X,c) = a(b,Y,c).                  Yes  X = Y
?- p1(a, [A, [B,C] ],25) = p1(C, [B, [D, E] ], 25).  Yes A = B = D, C = E = a.
```

- X \= Y
  - Semantically: not-unifiable test
  - Succeeds as long as X and Y cannot be unified
  - Both X and Y must be instantiated but may have uninstantiated elements.
  - No side effects
  - E.g.

```
?- joe \= fred.                  Yes
?- a(b,X,c) \= a(b,Y,c).                 No
```

Prolog Operators

- **== *is already instantiated to*** operator: **X == Y**
  - Semantically: identical test
  - Succeeds as long as X and Y are already instantiated to the same object
  - Any variable inside X and Y must be the same
  - No side effects
  - E.g.

```
?- 4  ==  2 +2                     No
?- a(b,X,c) == a(b,Y,c).           No
?- a(b,X,c) == a(b,X,c).           Yes
```

- **=:= *is already instantiated to*** operator**:** **X =:= Y**
  - Semantically: identical test after evaluating terms
  - E.g.

```
?- 4 =:= 2 + 2                     Yes.
?- a(b,X,c) =:= a(b,Y,c).          Error, a cannot be evaluated
```


Prolog Operators

- **X \== Y**
  - Semantically: not-identical test
  - Succeeds as long as X and Y are not already instantiated to the same object
  - No side effects
  - E.g.

```
?- A \== hello.                    Yes
?- a(b,X,c) \== a(b,Y,c).          Yes
?- 1 +2 \== 3                      Yes
```

- **X =\= Y**
  - Semantically: not-identical test after evaluating terms
  - E.g.

```
?- 4 =\= 2 + 2                     No
```

Prolog Operators

- **Other arithmetic operators :**
  - Add, subtract,multiply, division   + - * /
  - Integer division   //
  - modulus   mod
  - Comparison after evaluation   > => <= <

- **is** operator: **X is Expr**   **is(X,Expr)**
  - Semantically: evaluate second term and test if it is equal to X
  - succeeds a long as X and the arithmetic evaluation of Expr can be unified
  - X may or may not be instantiated
  - Expr must not contain any uninstantiated variables
  - As a side effect, X is instantiated to the arithmetic evaluation of Expr
  - E.g.

```
?-  5 is ( ( 3 * 7 ) + 1 ) //  4          Yes
?-  X is ( ( 3 * 4 ) +10) mod 6           X=4
?- is(2+3,5).                  No
?- is(5,2+3).                  Yes
```


Lists in Prolog

- **A sequence of terms of the form**

  [t_1, t_2, t_3, t_4, ..., t_n]   *where term t_i is the ith element of the list*

- **[ ] is the 'empty list'. It is an atom not a list.**

- **Example:**   [a, b, c, [d, e, [], f]]
  - A list with 4 elements: a, b, c, and a list with 4 elements:d, e, an empty list, and f
  - Prolog supports Scheme-style nested lists

- **Can break apart lists using "|" into [Head | Tail] where Head is the first item as an object and Tail is the rest of the list (as a list)**
  - E.g. ?- [H | T] = [a, b, c]. ; what are H and T?
  - Recall what pairs look like in Scheme.

- **You can also use the same notation "|" to construct lists:**
  - E.g. ?- L = [a | [b, c]]. ; what's L?

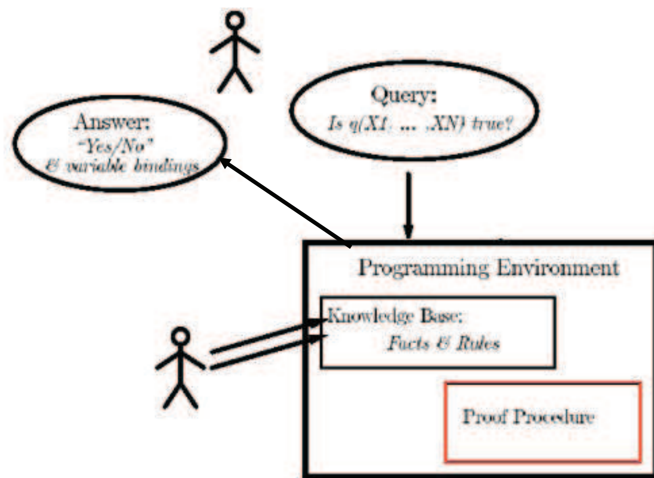
Lists & Unification

- Examples:
  - [A, B | C] = [a, b, c, d, e, f]. A = a B = b C = [c, d, e, f]
  - [A, b | C] = [a, B, c, d, e, f]. A = a B = b C = [c, d, e, f]
  - [X,Y] = [john, skates]. X=john Y=skates
  - [cat] = [H|T]. H = cat T = []
  - [[the,Y]|Z] = [[X,hare],[is,here]]. Y = hare Z = [[is, here]] X = the
  - [H|T] = a(b, c(d)). Error
  - [n(X,Y),a(1)] = [Name,Age]. X = _G17 Y = _G18
    Name = n(_G17, _G18) Age = a(1)
- Will see programming with lists later!

Prolog Reasoning Mechanism

Prolog: proof procedure - revisited

- **Two main processes:**
  - ✓ Unification
  - Top-down reasoning



Prolog: reasoning

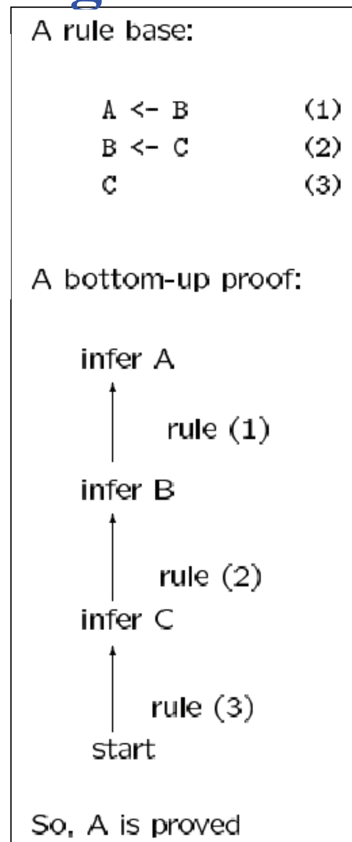
- **Given a set of facts and rules, we need a mechanism to deduce new facts and/or prove that a given rule is true or false or has no answer**

- **There are two techniques to do this:**
  - Bottom-up reasoning
  - Top-down reasoning

Bottom-up Reasoning

- Bottom-up (or forward) reasoning: starting from the given facts, apply rules to infer everything that is true.

  e.g., Suppose the fact B and the rule $A \leftarrow B$ are given. Then infer that A is true.



Example

Rule base:

```
p(X,Y,Z) <- q(X),q(Y),q(Z).
q(a1).
q(a2).
...
q(an).
```

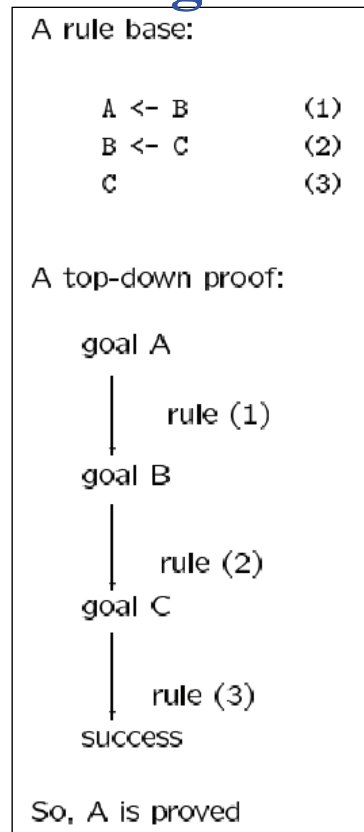
Bottom-up inference derives n^3 facts of the form p(a_i,a_j,a_k):

```
p(a1, a1, a1)
p(a1, a1, a2)
p(a1, a2, a3)
...
```

Prolog: top-down reasoning

- Top-down (or backward) reasoning: starting from the query, apply the rules in reverse, attempting only those lines of inference that are relevant to the query.

  e.g., Suppose the query is A, and the rule $A \leftarrow B$ is given. Then to prove A, try to prove B.



Prolog: top-down reasoning – cont'd

- **Multiple rules and multiple premises:**

  – A fact may be inferred by many rules
    - E.g.
      ```
      E <- B
      E <- C
      E <- D
      ```
  – A rule may have many premises
    - E.g.
      ```
      E <- B /\ C /\ D
      ```

- **In top-down inference, such rules give rise to**
  – Inference trees
  – Backtracking

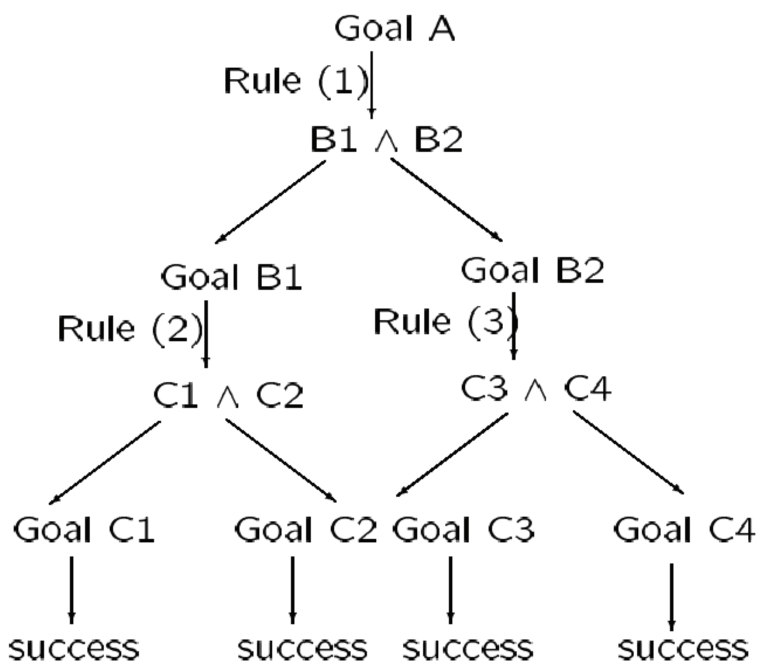
Prolog: top-down reasoning – cont'd

- **Example:** *multiple premises*

Rule base:

```
(1)   A  <- B1 /\ B2
(2)   B1 <- C1 /\ C2
(3)   B2 <- C3 /\ C4
      C1    C2    C3    C4
```

Query: Is A true?



So, goal A is proved. (all paths must succeed)

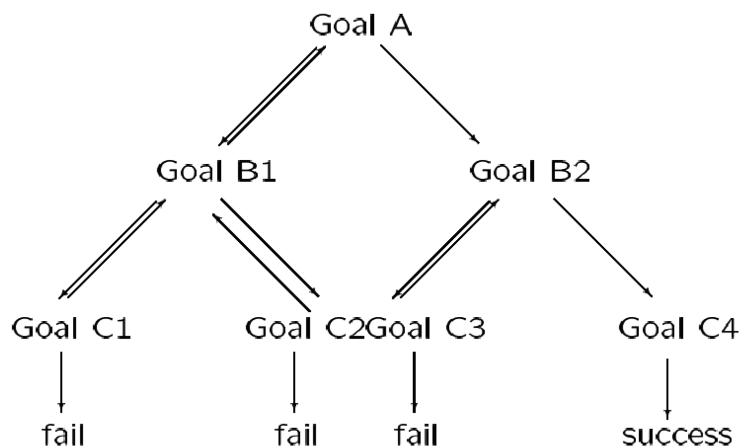
Prolog: top-down reasoning – cont'd

- **Example:** *multiple rules*

Rule base:

```
A <- B1      B1 <- C1      B2 <- C3
A <- B2      B1 <- C2      B2 <- C4
C4
```

Query: Is A true?



Prolog: backtracking

- **Prolog uses this algorithm for proving a goal by recursively breaking goal down into sub-goals and try to prove these sub-goals until facts are reached.**

- **To satisfy a goal:**
  - Try to unify with conclusion of first rule in database
  - If successful, apply substitution to first premise, try to satisfy resulting sub-goals
  - Then apply both substitutions to next sub-goal (premise), and so on...
  - If not successful, go on to the next rule in database
  - If all rules fail,try again (backtrack) to a previous sub-goal

Prolog Search Tree

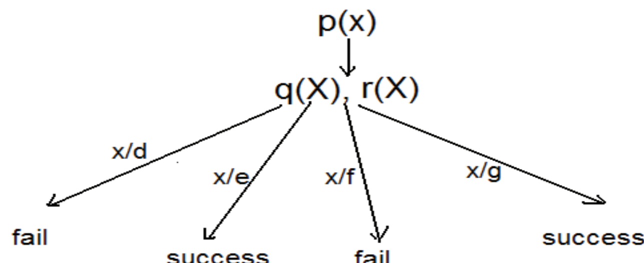
- Encapsulate unification, backward chaining, and backtracking.
  - Internal nodes are ordered list of subgoals
  - Leaves are success nodes or failures, where computation can proceed no further
  - Edges are labeled with variable bindings that occur by unification.
  - Describe all possible computation paths.
  - There can be many success nodes.
  - There can be infinite branches.

Prolog: backtracking example 1

Rule base:

```
p(X) :- q(X),r(X).
q(d).   q(e).   q(f).   q(g).
r(e).   r(g).
```

Query: Find X such that p(X) is true.

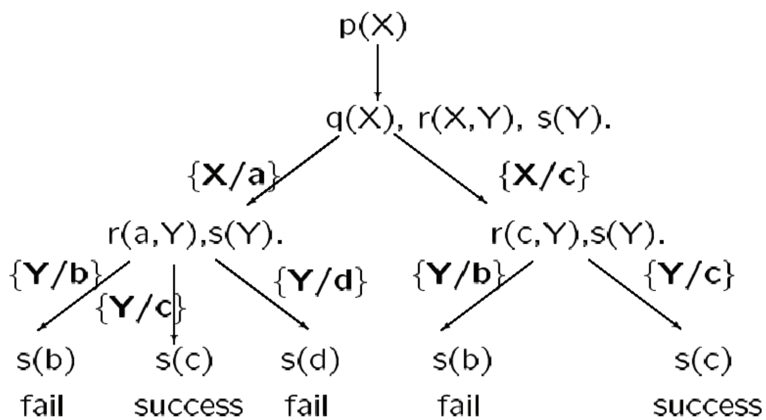


Prolog: backtracking example 2

Rule base:

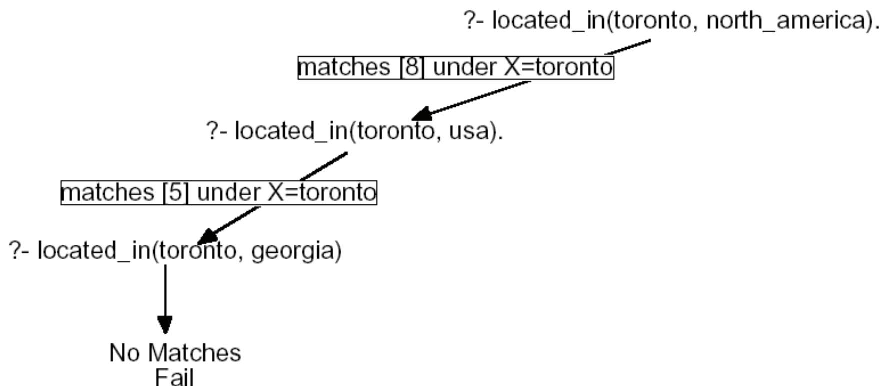
```
p(X) :- q(X),r(X,Y),s(Y).
q(a).    r(a,b).    r(c,b).    s(c).
q(c).    r(a,c).    r(c,c).
         r(a,d).
```

Query: Find X such that p(X) is true.



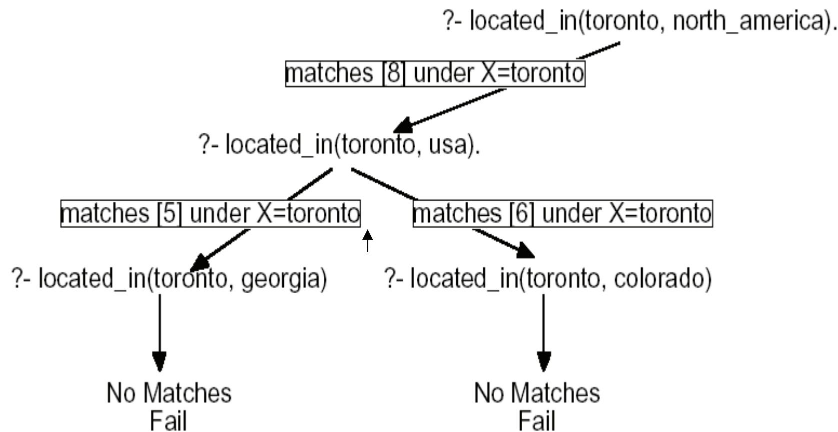
Prolog: backtracking example 3

```
[1] located_in(atlanta, georgia).
[2] located_in(denver, colorado).
[3] located_in(boulder, colorado).
[4] located_in(toronto, ontario).
[5] located_in(X, usa) :- located_in(X, georgia).
[6] located_in(X, usa) :- located_in(X, colorado).
[7] located_in(X, canada) :- located_in(X, ontario).
[8] located_in(X, north_america) :- located_in(X, usa).
[9] located_in(X, north_america) :- located_in(X, canada).

?- located_in(toronto, north_america).
```



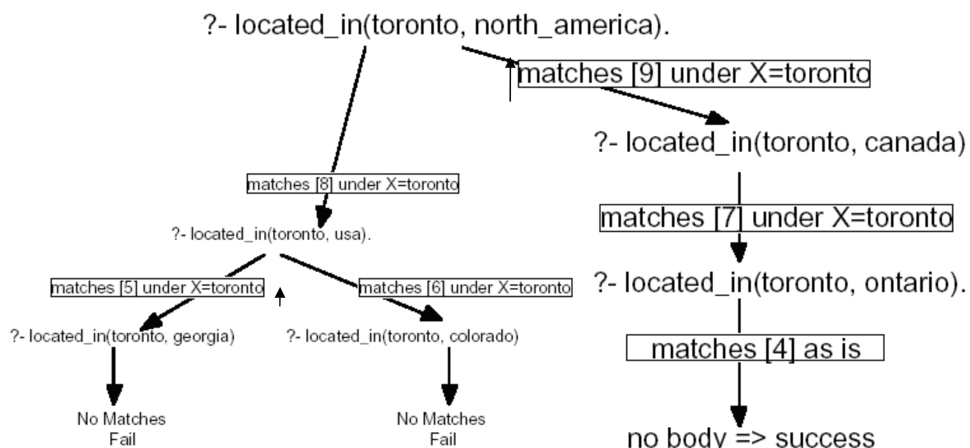
Prolog: backtracking example 3 – cont'd

```
[1] located_in(atlanta, georgia).
[2] located_in(denver, colorado).
[3] located_in(boulder, colorado).
[4] located_in(toronto, ontario).
[5] located_in(X, usa) :- located_in(X, georgia).
[6] located_in(X, usa) :- located_in(X, colorado).
[7] located_in(X, canada) :- located_in(X, ontario).
[8] located_in(X, north_america) :- located_in(X, usa).
[9] located_in(X, north_america) :- located_in(X, canada).

?- located_in(toronto, north_america).
```



Prolog: backtracking example 3 – cont'd

```
[1] located_in(atlanta, georgia).
[2] located_in(denver, colorado).
[3] located_in(boulder, colorado).
[4] located_in(toronto, ontario).
[5] located_in(X, usa) :- located_in(X, georgia).
[6] located_in(X, usa) :- located_in(X, colorado).
[7] located_in(X, canada) :- located_in(X, ontario).
[8] located_in(X, north_america) :- located_in(X, usa).
[9] located_in(X, north_america) :- located_in(X, canada).

?- located_in(toronto, north_america).
```



Top-down vs. Bottom-up Reasoning

- **Prolog uses top-down inference, although some other logic programming systems use bottom-up inference (e.g. Coral)**

- **Each has its own advantages and disadvantages:**
  - Bottom-up may generate many irrelevant facts
  - Top-down may explore many lines of reasoning that fail.

- **Top-down and bottom-up inference are logically equivalent**
  - i.e. they both prove the same set of facts.

- **However, only top-down inference simulates program execution**
  - i.e. execution is inherently top down, since it proceeds from the main procedure downwards, to subroutines, to sub-subroutines, etc...

Logic Programming vs. Prolog

```
cousin(X,Y) :- parent(W,X), sister(W,Z), parent(Z,Y).\\
cousin(X,Y) :- parent(W,X), brother(W,Z), parent(Z,Y).

?- cousin(X,jane).          % a query
```

- Rule and Goal Ordering:
  - There are two rules for cousin
  - Which rule do we try first?
  - Each rule for cousin has several subgoals
  - Which subgoal do we try first?

Logic Programming vs. Prolog

- Logic Programming:
  - Nondeterministic
  - Arbitrarily choose rule to expand first
  - Arbitrarily choose subgoal to to explore first
  - Results don't depend on rule and subgoal ordering
- Prolog: Deterministic
  - Expand first rule first
  - Explore first subgoal first
  - Results may depend on rule and subgoal ordering

Recursion in Prolog

Prolog: recursion

- **Recursively defined predicate:** *if a predicate symbol occurs both in in the head and and body of a rule, then the rule is recursive.*
  - E.g. a(X) :- b(X,Y), a(Y).

This predicate acts like a recursive subroutine.

- **Mutually recursive predicates:** *recursion might be indirect, involving several rules.*
  - E.g. a(X) :- b(X,Y), c(Y).
    c(Y) :- d(Y,Z), a(Z).

The predicates a and c are said to be mutually recursive.

- **Non-linear recursion:**
  - E.g. a(X) :- b(X,Y), a(Y), c(Y,Z), a(Z).

    This generates what we call a recursive proof tree.

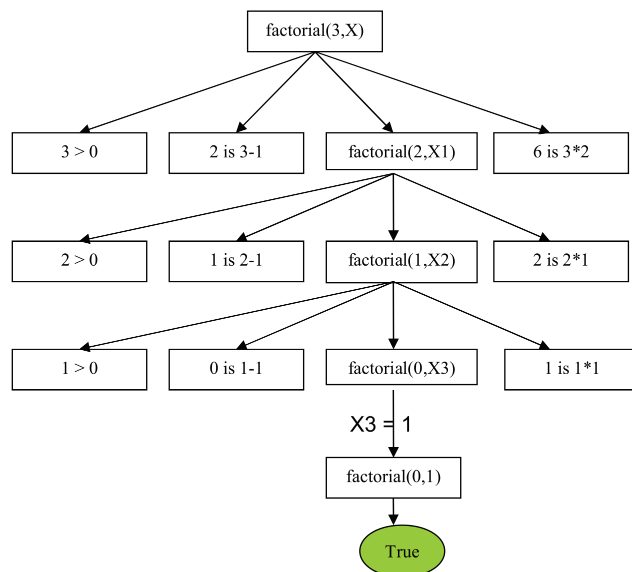
Prolog: recursion – examples

- **Factorial:** $n! \equiv n(n-1)\cdots 2\cdot 1.$
  - Declarative Semantics:

Factorial is 1 if n = 0, else Factorial is n * factorial (n-1)

  - Prolog:

```
factorial(0,1).
factorial(Y,X) :- Y>0,  Y1 is Y-1, factorial(Y1,X1), X is Y*X1.
```



Prolog: recursion – examples

- **Appending lists:**
  - Declarative Semantics:

  Appending an empty list to a non-empty list is the non-empty list
  else work on one list by removing its elements and adding it to the other list.
  - Prolog

```
append([],X,X).
append([H | X], Y, [H | Z]) :- append(X,Y,Z).
```

- **Member of a list:**
  - Declarative Semantics:

  X is a member of a list if X is equal to the first element, or a member of any sublist of that list
  - Prolog:

```
member(X,[X|T]).
member(X,[Y|T]):-member(X,T).
```


Prolog: recursion – examples

- **Blocks:**
  - Declarative Semantics:

a
b
c
d

Block X is above block Y if X is placed on top of Y, or X is placed on top of some block Z that is above Y.
  - Prolog:

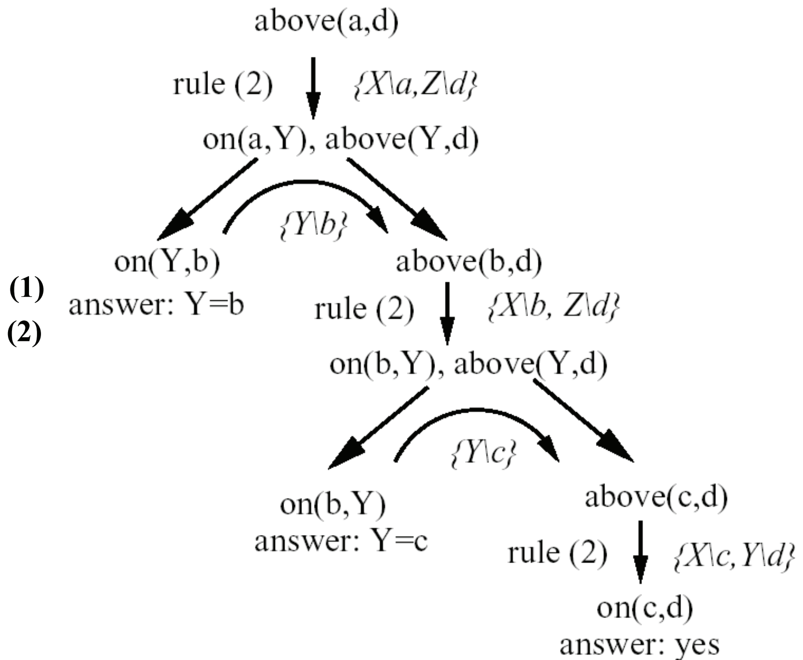
```
% first attempt
above(X,Y) :- on(X,Y).                      (1)
above(X,Z) :- above(X,Y), above(Y,Z).  (2)
on(a,b).        (3)
on(b,c).        (4)
on(c,d).        (5)

?- above(a,b).  ;yes
?- above(a,d).  ;yes
?- above(b,b).  ;Infinite recursion! trace it to see why.
?- above(c,a).   ;Infinite recursion! trace it to see why.
```

Prolog: recursion – examples

- **Blocks:**
  - Declarative Semantics:
    Block X is above block Y if X is placed on top of Y, or X is placed on top of some block Z that is above Y.
  - Prolog:

a
b
c
d

```
% Second attempt
above(X,Y) :- on(X,Y).                (1)
above(X,Z) :- on(X,Y), above(Y,Z).    (2)
on(a,b).     (3)
on(b,c).     (4)
on(c,d).     (5)

?- above(a,d).
Yes
```



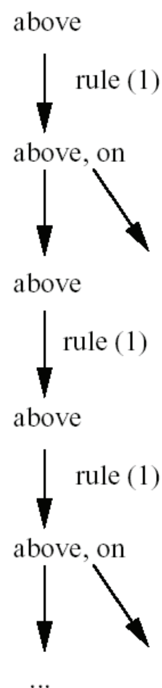
Prolog: recursion – examples

- **Note that sometimes changing the order of rules and/or rule premises can cause problems for Prolog**

- **Example:**

```
above(X,Z) :- on(X,Y), above(Y,Z).  (1)
above(X,Y) :- on(X,Y).              (2)
on(a,b).                            (3)
on(b,c).                            (4)
on(c,d).                            (5)

?- above(a,d).
```

a
b
c
d



Prolog: recursion – examples

- **Infinite recursion:**
  - E.g. **p :- p.**

    %declaratively perfectly correct, but procedurally causes infinite loop

- **What to do about infinite recursion?**
  - Rewrite the rules and facts *(most widely used technique)*
  - Define a second non-recursive version *(similar to a base case)*
  - Use ! to stop the unification (*more about this later*).

Exercises in Class

- Length of a list
- Swap first two element in a list.