A Second Look At Prolog

Outline

- Unification
- Three views of Prolog’s execution model
  - Procedural
  - Implementational
  - Abstract
- The lighter side of Prolog

Substitutions

- A *substitution* is a function that maps variables to terms:

  $\sigma = \{$**`X`**$\rightarrow$**`a`**, **`Y`**$\rightarrow$**`f(a,b)`**$\}$
- This σ maps **`X`** to **`a`** and **`Y`** to **`f(a,b)`**
- The result of applying a substitution to a term is an *instance* of the term
- $\sigma($**`g(X,Y)`**$)$ = **`g(a,f(a,b))`** so **`g(a,f(a,b))`** is an *instance* of **`g(X,Y)`**

Unification

- Two Prolog terms t_1 and t_2 *unify* if there is some substitution σ (their *unifier*) that makes them identical: $\sigma(t_1) = \sigma(t_2)$
  - **`a`** and **`b`** do not unify
  - **`f(X,b)`** and **`f(a,Y)`** unify: a unifier is {**`X`**→**`a`**, **`Y`**→**`b`**}
  - **`f(X,b)`** and **`g(X,b)`** do not unify
  - **`a(X,X,b)`** and **`a(b,X,X)`** unify: a unifier is {**`X`**→**`b`**}
  - **`a(X,X,b)`** and **`a(c,X,X)`** do not unify
  - **`a(X,f)`** and **`a(X,f)`** do unify: a unifier is {}

Multiple Unifiers

- **`parent(X,Y)`** and **`parent(fred,Y)`**:
  - one unifier is $\sigma_1 = \{$**`X`**$\rightarrow$**`fred`**$\}$
  - another is $\sigma_2 = \{$**`X`**$\rightarrow$**`fred`**, **`Y`**$\rightarrow$**`mary`**$\}$
- Prolog chooses unifiers like σ_1 that do just enough substitution to unify, and no more
- That is, it chooses the MGU—the Most General Unifier

MGU

- Term x_1 is *more general than* x_2 if x_2 is an instance of x_1 but x_1 is not an instance of x_2
  - Example: **`parent(fred,Y)`** is more general than **`parent(fred,mary)`**
- A unifier σ_1 of two terms t_1 and t_2 is an MGU if there is no other unifier σ_2 such that $\sigma_2(t_1)$ is more general than $\sigma_1(t_1)$
- MGU is unique up to variable renaming

Unification For Everything

- Parameter passing
  - `reverse([1,2,3],X)`
- Binding
  - `X=0`
- Data construction
  - `X=.(1,[2,3])`
- Data selection
  - `[1,2,3]=.(X,Y)`

The Occurs Check

- Any variable **`X`** and term *t* unify with {**`X`**$\rightarrow$*t*}:
  - **`X`** and **`b`** unify: an MGU is {**`X`**$\rightarrow$**`b`**}
  - **`X`** and **`f(a,g(b,c))`** unify: an MGU is {**`X`**$\rightarrow$**`f(a,g(b,c))`**}
  - **`X`** and **`f(a,Y)`** unify: an MGU is {**`X`**$\rightarrow$**`f(a,Y)`**}
- *Unless* **`X`** *occurs in t:*
  - **`X`** and **`f(a,X)`** do not unify, in particular not by {**`X`**$\rightarrow$**`f(a,X)`**}

Occurs Check Example

```
append([], B, B).
append([Head|TailA], B,
[Head|TailC]) :-
  append(TailA, B, TailC).
```

```
?- append([], X, [a | X]).
X = [a|**].
```

- Most Prologs omit the occurs check
- ISO standard says the result of unification is undefined in cases that should fail the occurs check

Outline

- Unification
- Three views of Prolog's execution model
  - Procedural
  - Implementational
  - Abstract
- The lighter side of Prolog

A Procedural View

- One way to think of it: each clause is a procedure for proving goals
  - `p :- q, r.` – To prove a goal, first unify the goal with **`p`**, then prove **`q`**, then prove **`r`**
  - `s.` – To prove a goal, unify it with **`s`**
- A proof may involve "calls" to other procedures

Simple Procedural Examples

```
p :- q, r.          boolean p() {return q() && r();}
q :- s.             boolean q() {return s();}
r :- s.             boolean r() {return s();}
s.                  boolean s() {return true;}
```

```
p :- p.             boolean p() {return p();}
```

Backtracking

- One complication: backtracking
- Prolog explores all possible targets of each call, until it finds as many successes as the caller requires or runs out of possibilities
- Consider the goal **`p`** here: it succeeds, but only after backtracking

```
1.  p :- q, r.
2.  q :- s.
3.  q.
4.  r.
5.  s :- 0=1.
```

Substitution

- Another complication: substitution
- A hidden flow of information

σ_3 = substitution developed by **`r`** to prove $\sigma_2(\sigma_1($**`r(Y)`**$))$

combined substitution is returned to caller

term proved: $\sigma_3(\sigma_2(\sigma_1(t)))$

σ_1 = MGU(**`p(f(Y))`**,t) is applied to all subsequent conditions in the clause

```
p(f(Y))  :-  q(Y) ,  r(Y)  .
```

original goal term t

σ_2 = substitution developed by **`q`** to prove $\sigma_1($**`q(Y)`**$)$, is applied to all subsequent conditions in the clause

Outline

- Unification
- Three views of Prolog's execution model
  - Procedural
  - Implementational
  - Abstract
- The lighter side of Prolog

Resolution

- The hardwired inference step
- A clause is represented as a list of terms (a list of one term, if it is a fact)
- Resolution step applies one clause, once, to make progress on a list of goal terms

function **`resolution`**(**`clause`**, **`goals`**):
    let **`sub`** = the MGU of head(**`clause`**) and head(**`goals`**)
    return **`sub`**(tail(**`clause`**) concatenated with tail(**`goals`**))

Resolution Example

Given this list of goal terms:

```
[p(X),s(X)]
```

And this rule to apply:

```
p(f(Y)) :- q(Y), r(Y).
```

The MGU of the heads is {**X**→**f(Y)**}, and we get:

```
resolution([p(f(Y)),q(Y),r(Y)], [p(X),s(X)])
      = [q(Y),r(Y),s(f(Y))]
```

function **resolution**(**clause**, **goals**):
  let **sub** = the MGU of head(**clause**) and head(**goals**)
  return **sub**(tail(**clause**) concatenated with tail(**goals**))

A Prolog Interpreter

```
function solve(goals)
    if goals is empty then succeed()
    else for each clause c in the program, in order
        if head(c) does not unify with head(goals) then do nothing
        else solve(resolution(c, goals))
```

Program:

```
1.   p(f(Y)) :-
         q(Y),r(Y).
2.   q(g(Z)).
3.   q(h(Z)).
4.   r(h(a)).
```

A partial trace for query **p(X)**:

```
solve([p(X)])
 1. solve([q(Y),r(Y)])
          …
 2. nothing
 3. nothing
 4. nothing
```

- **solve** tries each of the four clauses in turn
  - The first works, so it calls itself recursively on the result of the resolution step (not shown yet)
  - The other three do not work: heads do not unify with the first goal term

Program:

```
1.   p(f(Y)) :-
         q(Y),r(Y).
2.   q(g(Z)).
3.   q(h(Z)).
4.   r(h(a)).
```

A partial trace for query **p(X)**, expanded:

```
solve([p(X)])
 1.  solve([q(Y),r(Y)])
       1.  nothing
       2.  solve([r(g(Z))])
                 …
       3.  solve([r(h(Z))])
                 …
       4.  nothing
 2.  nothing
 3.  nothing
 4.  nothing
```

Program:

```
1.   p(f(Y)) :-
         q(Y),r(Y).
2.   q(g(Z)).
3.   q(h(Z)).
4.   r(h(a)).
```

A complete trace for query **`p(X)`**:

```
solve([p(X)])
 1. solve([q(Y),r(Y)])
      1. nothing
      2. solve([r(g(Z))])
           1. nothing
           2. nothing
           3. nothing
           4. nothing
      3. solve([r(h(Z))])
           1. nothing
           2. nothing
           3. nothing
           4. solve([]) — success!
      4. nothing
 2. nothing
 3. nothing
 4. nothing
```

Collecting The Substitutions

function **`resolution`**(**`clause`**, **`goals`**, **`query`**):
    let **`sub`** = the MGU of head(**`clause`**) and head(**`goals`**)
    return (**`sub`**(tail(**`clause`**) concatenated with tail(**`goals`**)), **`sub`**(**`query`**))

function **`solve`**(**`goals`**, **`query`**)
    if **`goals`** is empty then **`succeed`**(**`query`**)
    else for each clause **`c`** in the program, in order
        if head(**`c`**) does not unify with head(**`goals`**) then do nothing
        else **`solve`**(**`resolution`**(**`c`**, **`goals`**, **`query`**))

- Modified to pass original query along and apply all substitutions to it
- Proved instance is passed to **`succeed`**

Program:

```
1.   p(f(Y)) :-
         q(Y),r(Y).
2.   q(g(Z)).
3.   q(h(Z)).
4.   r(h(a)).
```

A complete trace for query **p(X)**:

```
solve([p(X)],p(X))
 1. solve([q(Y),r(Y)],p(f(Y)))
     1. nothing
     2. solve([r(g(Z))],p(f(g(Z))))
         1. nothing
         2. nothing
         3. nothing
         4. nothing
     3. solve([r(h(Z))],p(f(h(Z))))
         1. nothing
         2. nothing
         3. nothing
         4. solve([],p(f(h(a))))
     4. nothing
 2. nothing
 3. nothing
 4. nothing
```

Prolog Interpreters

- The interpreter just shown is how early Prolog implementations worked
- All Prolog implementations must do things in that order, but most now accomplish it by a completely different (compiled) technique

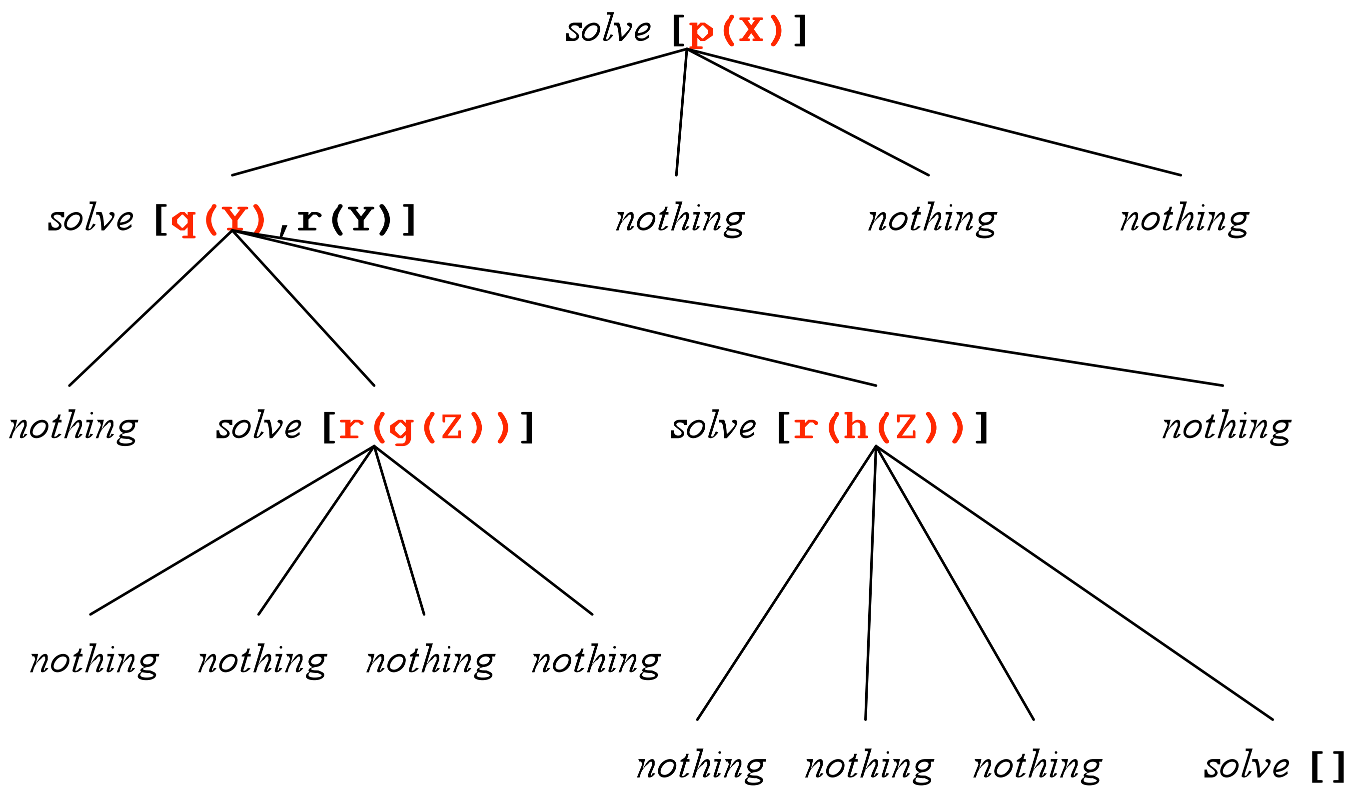
Outline

- Unification
- Three views of Prolog's execution model
  - Procedural
  - Implementational
  - Abstract
- The lighter side of Prolog

Proof Trees

- We want to talk about the order of operations, without pinning down the implementation technique
- Proof trees capture the order of traces of **`prove`**, without the code:
  - Root is original query
  - Nodes are lists of goal terms, with one child for each clause in the program

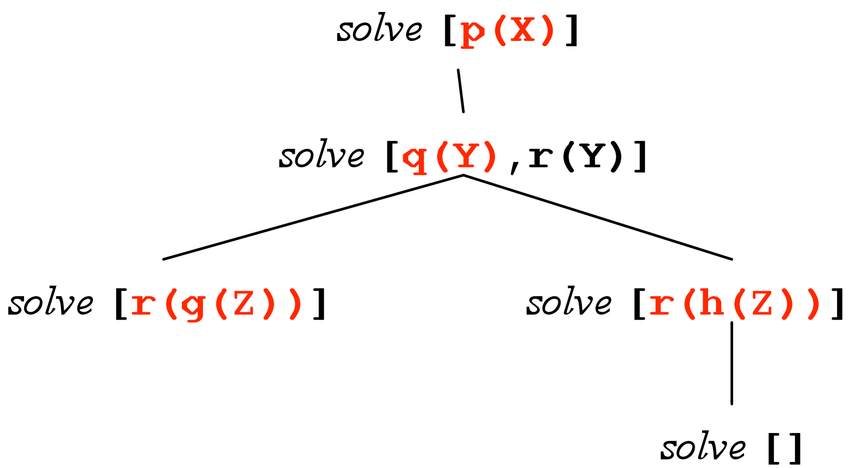
Example


solve [p(X)]
solve [q(Y),r(Y)]
nothing
nothing
nothing
nothing
solve [r(g(Z))]
solve [r(h(Z))]
nothing
nothing
nothing
nothing
nothing
nothing
nothing
nothing
solve []

Simplifying

- Children of a node represent clauses
- They appear in the order they occur in the program
- Once this is understood, we can eliminate the *nothing* nodes, which represent clauses that do not apply to the first goal in the list

Example

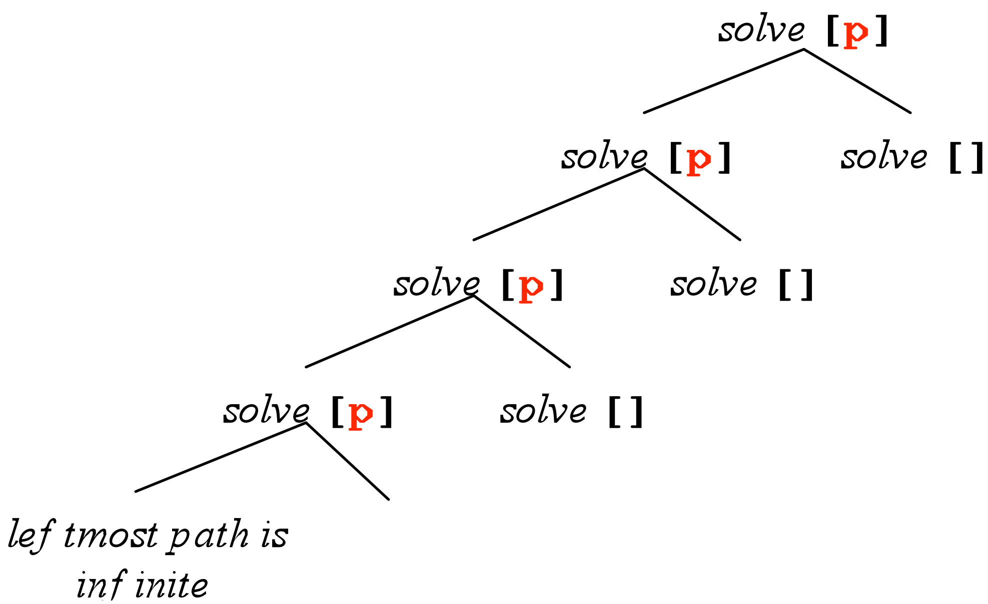


Prolog Semantics

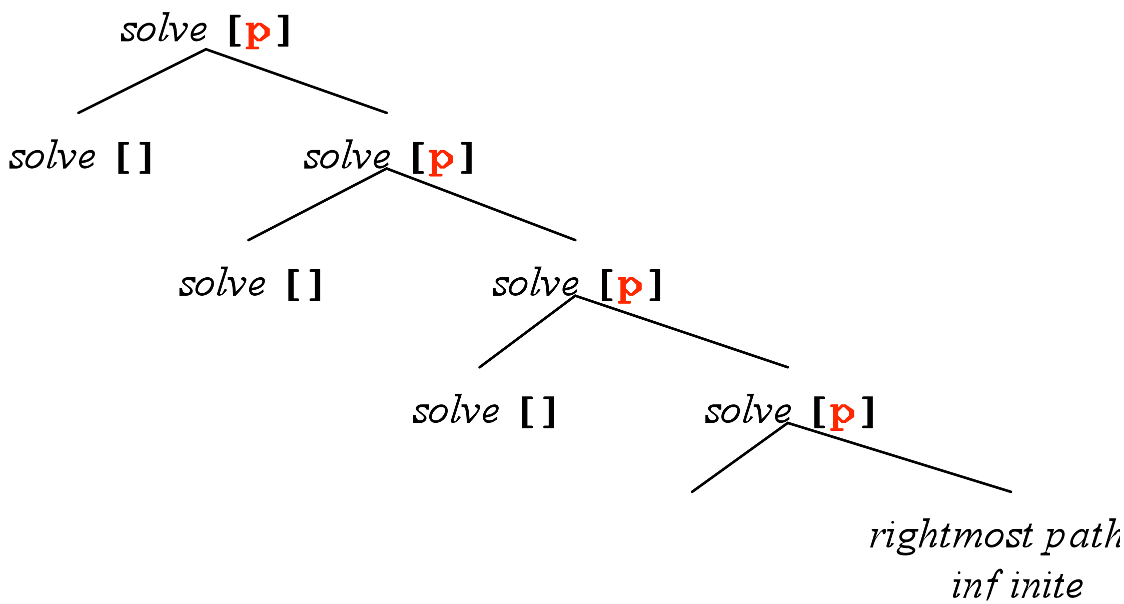
- *Given a program and a query, a Prolog language system must act in the order given by a depth-first, left-to-right traversal of the proof tree*
- It might accomplish that using an interpreter like our `prove`
- Or it might do it by some completely different means

Infinite Proof Tree, Nonterminating Program

```
p :- p.
p.
```



Infinite Proof Tree, Terminating Program

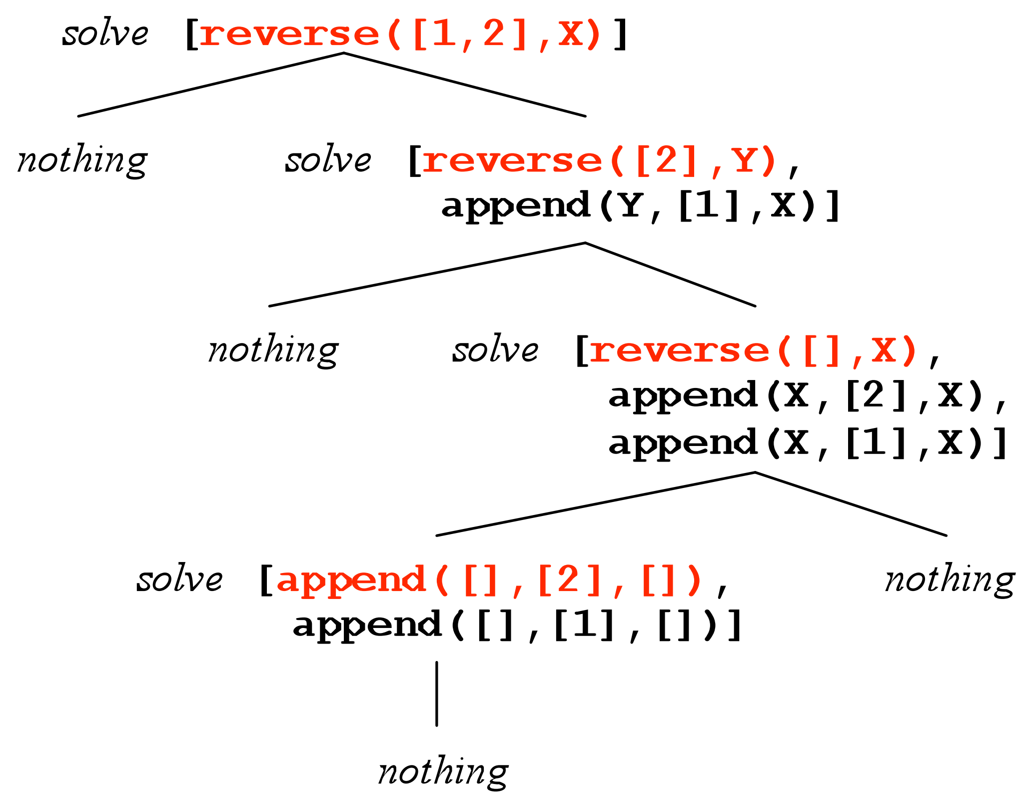



```
p.
p :- p.
```

A Problem

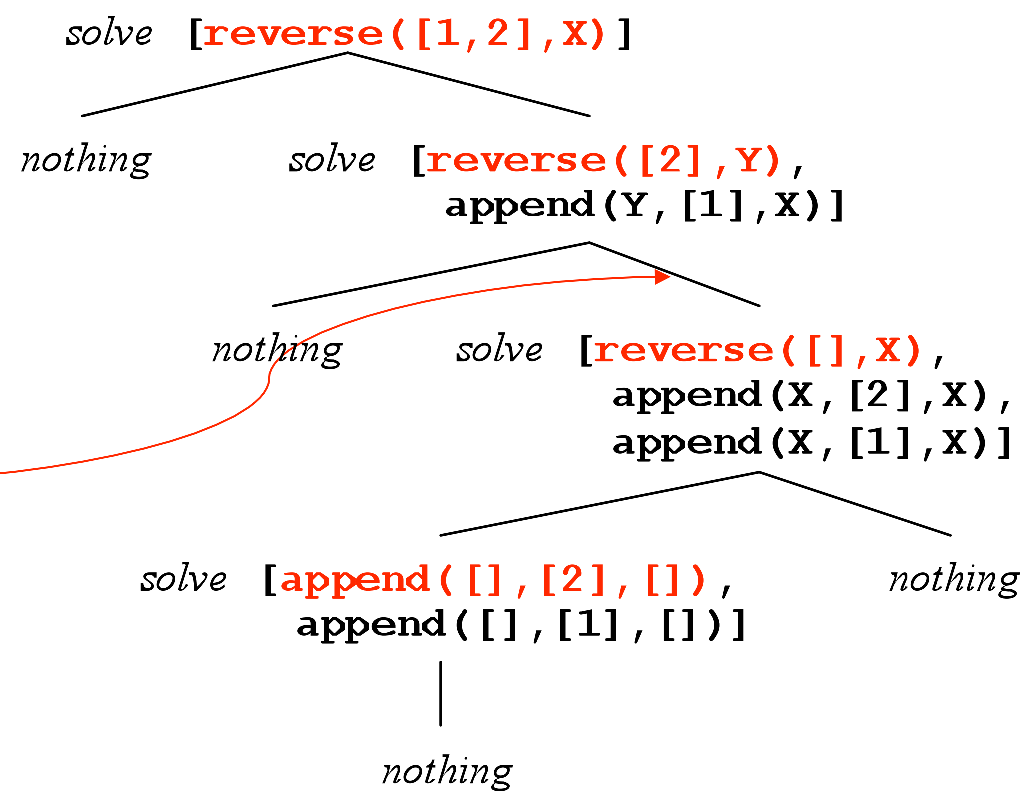
- All three of the models of Prolog execution we have seen are flawed
- They work on the examples we chose
- On other examples they would not agree with common sense, or with the actual behavior of a Prolog language system
- For instance, **`reverse([1,2],X)`**

A Problem




```
reverse([],[]).
reverse([Head|Tail],X) :-
  reverse(Tail,Y),
  append(Y,[Head],X).
```

The Error



This step is wrong: we substituted **X** for **Y**, but there is already a different **X** elsewhere in the goal list.

```
reverse([],[]).
reverse([Head|Tail],X) :-
  reverse(Tail,Y),
  append(Y,[Head],X).
```

Variable Renaming

- To avoid capture, use fresh variable names for each clause, every time you apply it
- The first application of **`reverse`** might be:

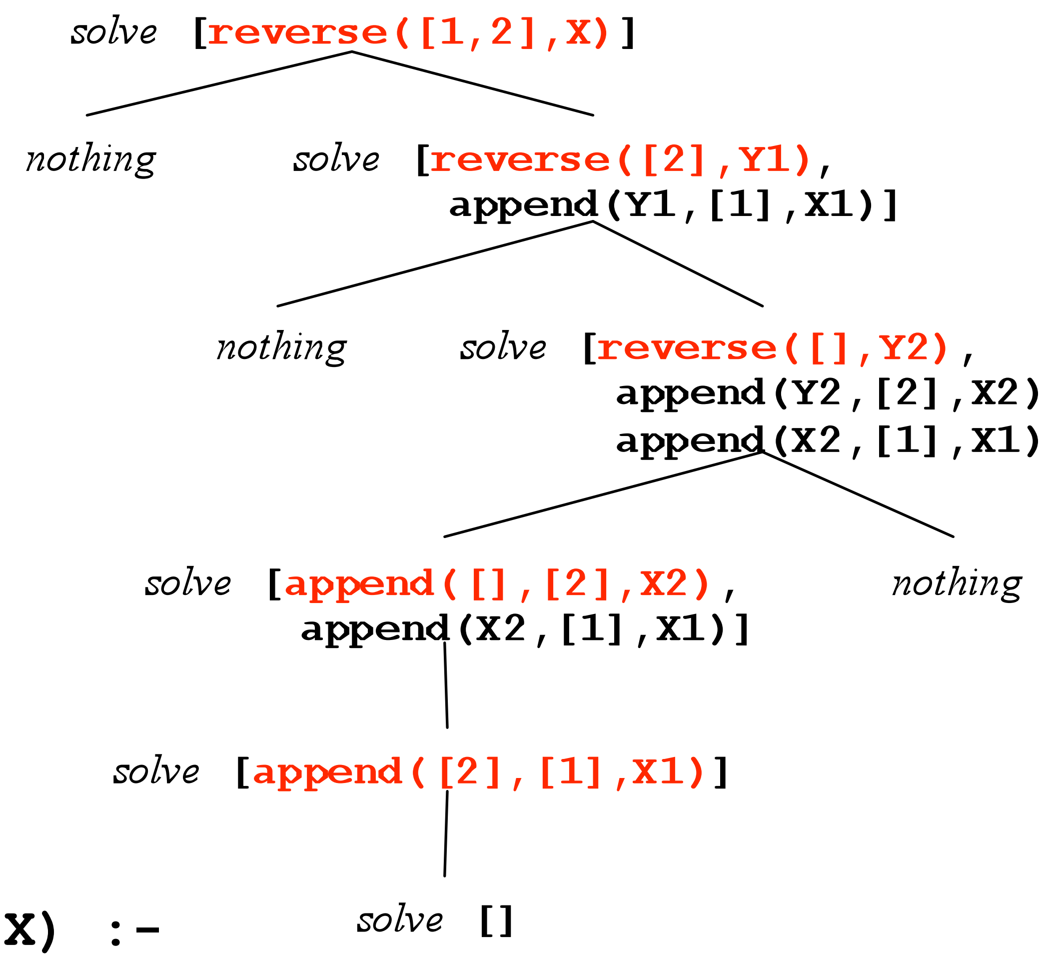
```
reverse([Head1|Tail1],X1) :-
  reverse(Tail1,Y1),
  append(Y1,[Head1],X1).
```

- And the next might be:

```
reverse([Head2|Tail2],X2) :-
  reverse(Tail2,Y2),
  append(Y2,[Head2],X2).
```

- And so on…

Correct




```
reverse([],[]).
reverse([Head|Tail],X) :-
  reverse(Tail,Y),
  append(Y,[Head],X).
```

Rename Everywhere

- This renaming step is required for all three of our models of Prolog execution
- Every time a clause is used, it must have a fresh set of variable names
- This implements clause scope as required: the scope of a definition of a variable is the clause containing it

Outline

- Unification
- Three views of Prolog’s execution model
  - Procedural
  - Implementational
  - Abstract
- The lighter side of Prolog

Quoted Atoms As Strings

- Any string of characters enclosed in single quotes is a term
- In fact, Prolog treats it as an atom:
  - **`'abc'`** is the same atom as **`abc`**
  - **`'hello world'`** and **`'Hello world'`** are atoms too
- Quoted strings can use **`\n`**, **`\t`**, **`\'`**, **`\\`**

Input and Output

```
?- write('Hello world').
Hello world
true.

?- read(X).
|: hello.
X = hello.
```

- Simple term input and output.
- Also the predicate **`nl`**: equivalent to **`write('\n')`**

Debugging With **write**

```
?- p.
[] [1, 2]
[1] [2]
[1, 2] []
false.
```

```
p :-
  append(X,Y,[1,2]),
  write(X), write(' '), write(Y), write('\n'),
  X=Y.
```

The `assert` Predicate

```
?- parent(joe,mary).
false.

?- assert(parent(joe,mary)).
true.

?- parent(joe,mary).
true.
```

- Adds a fact to the database (at the end)

The **`retract`** Predicate

```
?- parent(joe,mary).
true.

?- retract(parent(joe,mary)).
true.

?- parent(joe,mary).
false.
```

- Removes the first clause in the database that unifies with the parameter
- Also **`retractall`** to remove all matches

Dangerous Curves Ahead

- A very dirty trick: self-modifying code
- Not safe, not declarative, not efficient—but can be tempting, as the final example shows
- Best to use them only for facts, only for predicates not otherwise defined by the program, and only where the clause order is not important
- Note: if a predicate was compiled by **`consult`**, SWI-Prolog will not permit its definition to be changed by **`assert`** or **`retract`**

The Cut

- Written `!`, pronounced “cut”
- Logically simple: a goal that always succeeds (sort of like **`true`**)
- Procedurally tricky: when it succeeds, it usually also eliminates some backtracking
- We’ll use it in only one simple way: as the final condition in a rule

What Cut Does There

```
p :- q1, q2, …, qj, !.
```

- If `q1` through `qj` succeed, the cut does too
- It tells Prolog there's no going back:
  - No backtracking to look for other solutions for `q1` through `qj`
  - And, no backtracking to try other clauses for the goal `p` that succeeded this way
- In effect: the first solution found for a given goal using this rule will be the last solution found for that goal

No Cut, Normal Backtracking

```
p :- member(X,[a,b,c]), write(X).
p :- write(d).
```

```
?- p.
a
true ;
b
true ;
c
true ;
d
true.
```

Cut Discards Backtracking

```
p :- member(X,[a,b,c]), write(X), !.
p :- write(d).
```

```
?- p.
a
true.
```

- Because of the cut, it stops after finding the first solution

Cut With Care

- Uses of cut are non-declarative, and can be extremely subtle and error prone
  - Some cuts improve efficiency, saving time and space on backtracking where you know there are no more solutions anyway (“green cuts”)
  - Others (like the previous example) change the solutions that are found (“red cuts”)
- Useful and sometimes necessary, but use with caution

An Adventure Game

- Prolog comments
  - `/*` to `*/`, like Java
  - Also, `%` to end of line

```
/*
  This is a little adventure game.  There are three
  entities: you, a treasure, and an ogre.  There are
  six places: a valley, a path, a cliff, a fork, a maze,
  and a mountaintop.  Your goal is to get the treasure
  without being killed first.
*/
```

```
/*
  First, text descriptions of all the places in
  the game.
*/
description(valley,
  'You are in a pleasant valley, with a trail ahead.').
description(path,
  'You are on a path, with ravines on both sides.').
description(cliff,
  'You are teetering on the edge of a cliff.').
description(fork,
  'You are at a fork in the path.').
description(maze(_),
  'You are in a maze of twisty trails, all alike.').
description(mountaintop,
  'You are on the mountaintop.').
```

```
/*
  report prints the description of your current
  location.
*/
report :-
  at(you,X),
  description(X,Y),
  write(Y), nl.
```

```
?- assert(at(you,cliff)).
true.

?- report.
You are teetering on the edge of a cliff.
true.

?- retract(at(you,cliff)).
true.

?- assert(at(you,valley)).
true.

?- report.
You are in a pleasant valley, with a trail ahead.
true.
```

```
/*
  These connect predicates establish the map.
  The meaning of connect(X,Dir,Y) is that if you
  are at X and you move in direction Dir, you
  get to Y.  Recognized directions are
  forward, right and left.
*/
connect(valley,forward,path).
connect(path,right,cliff).
connect(path,left,cliff).
connect(path,forward,fork).
connect(fork,left,maze(0)).
connect(fork,right,mountaintop).
connect(maze(0),left,maze(1)).
connect(maze(0),right,maze(3)).
connect(maze(1),left,maze(0)).
connect(maze(1),right,maze(2)).
connect(maze(2),left,fork).
connect(maze(2),right,maze(0)).
connect(maze(3),left,maze(0)).
connect(maze(3),right,maze(3)).
```

```
/*
  move(Dir) moves you in direction Dir, then
  prints the description of your new location.
*/
move(Dir) :-
  at(you,Loc),
  connect(Loc,Dir,Next),
  retract(at(you,Loc)),
  assert(at(you,Next)),
  report,
  !.
/*
  But if the argument was not a legal direction,
  print an error message and don't move.
*/
move(_) :-
  write('That is not a legal move.\n'),
  report.
```

Note the final cut: the second clause for **move** *will be used only if the first one fails, which happens only if* **Dir** *was not a legal move.*

```
/*
  Shorthand for moves.
*/
forward :- move(forward).
left :- move(left).
right :- move(right).
```

```
?- assert(at(you,valley)).
true.

?- forward.
You are on a path, with ravines on both sides.
true.

?- forward.
You are at a fork in the path.
true.

?- forward.
That is not a legal move.
You are at a fork in the path.
true.
```

```
/*
  If you and the ogre are at the same place, it
  kills you.
*/
ogre :-
  at(ogre,Loc),
  at(you,Loc),
  write('An ogre sucks your brain out through\n'),
  write('your eyesockets, and you die.\n'),
  retract(at(you,Loc)),
  assert(at(you,done)),
  !.
/*
  But if you and the ogre are not in the same place,
  nothing happens.
*/
ogre.
```

Note again the final cut in the first clause, producing an "otherwise" behavior: **`ogre`** *always succeeds, by killing you if it can, or otherwise by doing nothing.*

```
/*
  If you and the treasure are at the same place, you
  win.
*/
treasure :-
  at(treasure,Loc),
  at(you,Loc),
  write('There is a treasure here.\n'),
  write('Congratulations, you win!\n'),
  retract(at(you,Loc)),
  assert(at(you,done)),
  !.
/*
  But if you and the treasure are not in the same
  place, nothing happens.
*/
treasure.
```

```
/*
  If you are at the cliff, you fall off and die.
*/
cliff :-
  at(you,cliff),
  write('You fall off and die.\n'),
  retract(at(you,cliff)),
  assert(at(you,done)),
  !.
/*
  But if you are not at the cliff nothing happens.
*/
cliff.
```

```
/*
  Main loop.  Stop if player won or lost.
*/
main :-
  at(you,done),
  write('Thanks for playing.\n'),
  !.
/*
  Main loop.  Not done, so get a move from the user
  and make it.  Then run all our special behaviors.
  Then repeat.
*/
main :-
  write('\nNext move -- '),
  read(Move),
  call(Move),
  ogre,
  treasure,
  cliff,
  main.
```

The predefined predicate **`call(X)`** *tries to prove* **`X`** *as a goal term.*

```
/*
  This is the starting point for the game.  We
  assert the initial conditions, print an initial
  report, then start the main loop.
*/
go :-
  retractall(at(_,_)), % clean up from previous runs
  assert(at(you,valley)),
  assert(at(ogre,maze(3))),
  assert(at(treasure,mountaintop)),
  write('This is an adventure game. \n'),
  write('Legal moves are left, right or forward.\n'),
  write('End each move with a period.\n\n'),
  report,
  main.
```

```
?- go.
This is an adventure game.
Legal moves are left, right or forward.
End each move with a period.

You are in a pleasant valley, with a trail ahead.

Next move -- forward.
You are on a path, with ravines on both sides.

Next move -- forward.
You are at a fork in the path.

Next move -- right.
You are on the mountaintop.
There is a treasure here.
Congratulations, you win!
Thanks for playing.
true.
```