

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:

$$\sigma = \{ \mathbf{X} \rightarrow \mathbf{a}, \mathbf{Y} \rightarrow \mathbf{f}(\mathbf{a}, \mathbf{b}) \}$$

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

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 $\{\mathbf{X} \rightarrow \mathbf{a}, \mathbf{Y} \rightarrow \mathbf{b}\}$
 - **f** (**X**, **b**) and **g** (**X**, **b**) do not unify
 - **a** (**X**, **X**, **b**) and **a** (**b**, **X**, **X**) unify: a unifier is $\{\mathbf{X} \rightarrow \mathbf{b}\}$
 - **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 $\sigma_1 = \{\mathbf{X} \rightarrow \mathbf{fred}\}$
 - another is $\sigma_2 = \{\mathbf{X} \rightarrow \mathbf{fred}, \mathbf{Y} \rightarrow \mathbf{mary}\}$
- 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 \mathbf{X} and term t unify with $\{\mathbf{X} \rightarrow t\}$:
 - \mathbf{X} and \mathbf{b} unify: an MGU is $\{\mathbf{X} \rightarrow \mathbf{b}\}$
 - \mathbf{X} and $\mathbf{f}(\mathbf{a}, \mathbf{g}(\mathbf{b}, \mathbf{c}))$ unify: an MGU is $\{\mathbf{X} \rightarrow \mathbf{f}(\mathbf{a}, \mathbf{g}(\mathbf{b}, \mathbf{c}))\}$
 - \mathbf{X} and $\mathbf{f}(\mathbf{a}, \mathbf{Y})$ unify: an MGU is $\{\mathbf{X} \rightarrow \mathbf{f}(\mathbf{a}, \mathbf{Y})\}$
- *Unless \mathbf{X} occurs in t :*
 - \mathbf{X} and $\mathbf{f}(\mathbf{a}, \mathbf{X})$ do not unify, in particular not by $\{\mathbf{X} \rightarrow \mathbf{f}(\mathbf{a}, \mathbf{X})\}$

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
 - $\mathbf{p} \text{ :- } \mathbf{q}, \mathbf{r}.$ – To prove a goal, first unify the goal with \mathbf{p} , then prove \mathbf{q} , then prove \mathbf{r}
 - $\mathbf{s}.$ – To prove a goal, unify it with \mathbf{s}
- A proof may involve “calls” to other procedures

Simple Procedural Examples

```
p :- q, r.
```

```
q :- s.
```

```
r :- s.
```

```
s.
```

```
boolean p() {return q() && r();}
```

```
boolean q() {return s();}
```

```
boolean r() {return s();}
```

```
boolean s() {return true;}
```

```
p :- p.
```

```
boolean p() {return p();}
```

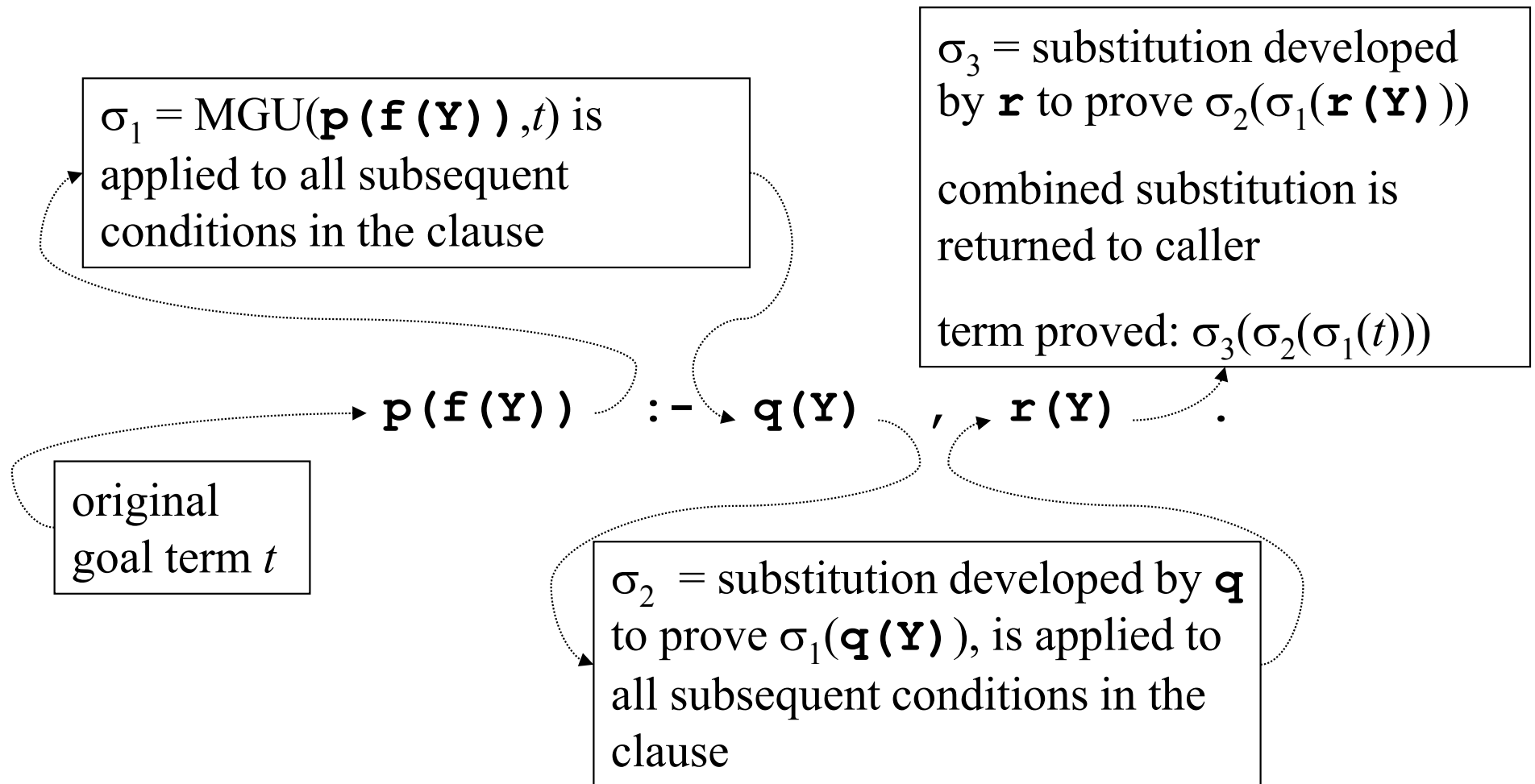
Backtracking

- 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



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:

$p(f(Y)) \text{ :- } q(Y), r(Y).$

The MGU of the heads is $\{X \rightarrow f(Y)\}$, and we get:

$$\begin{aligned} & \text{resolution}([p(f(Y)), q(Y), r(Y)], [p(X), s(X)]) \\ &= [q(Y), r(Y), s(f(Y))] \end{aligned}$$

```
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:

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

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

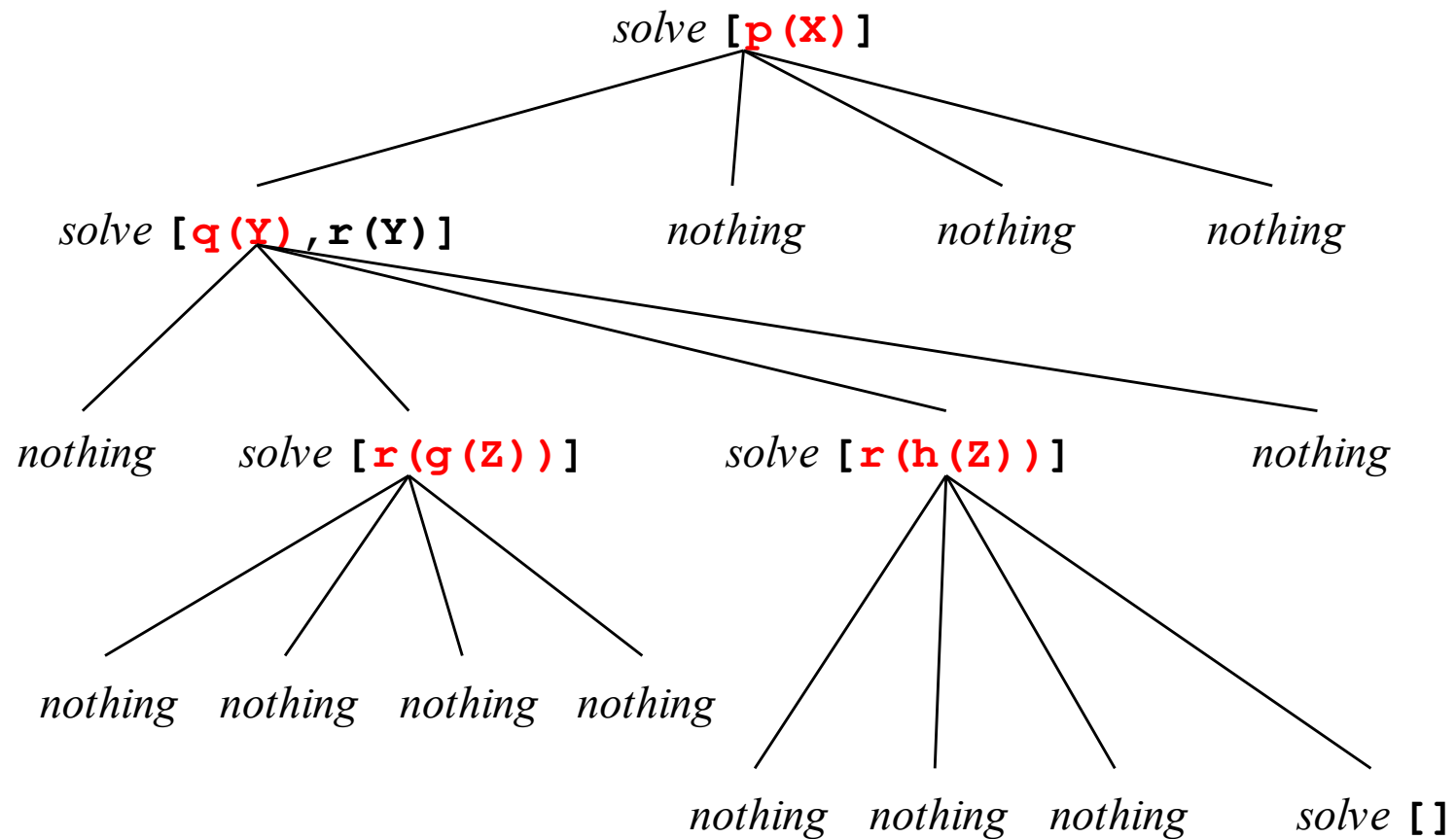
Outline

- 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

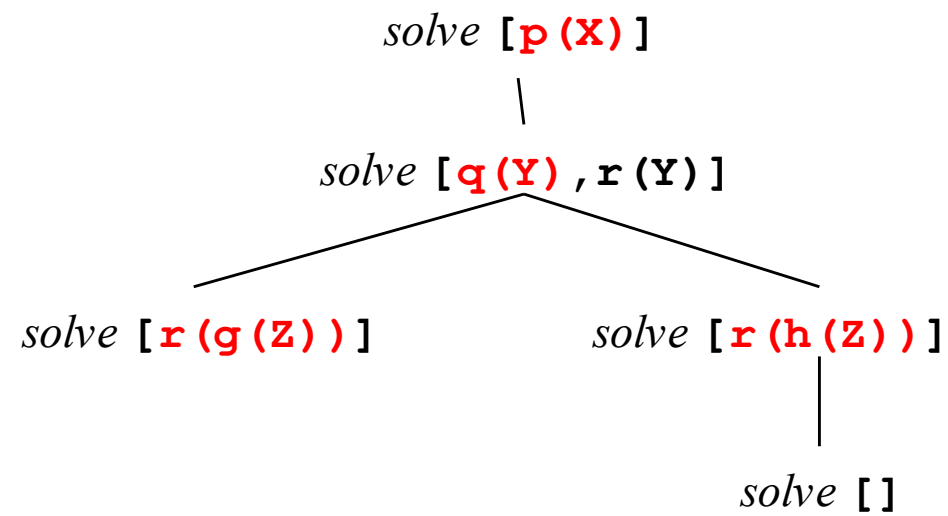
Example



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

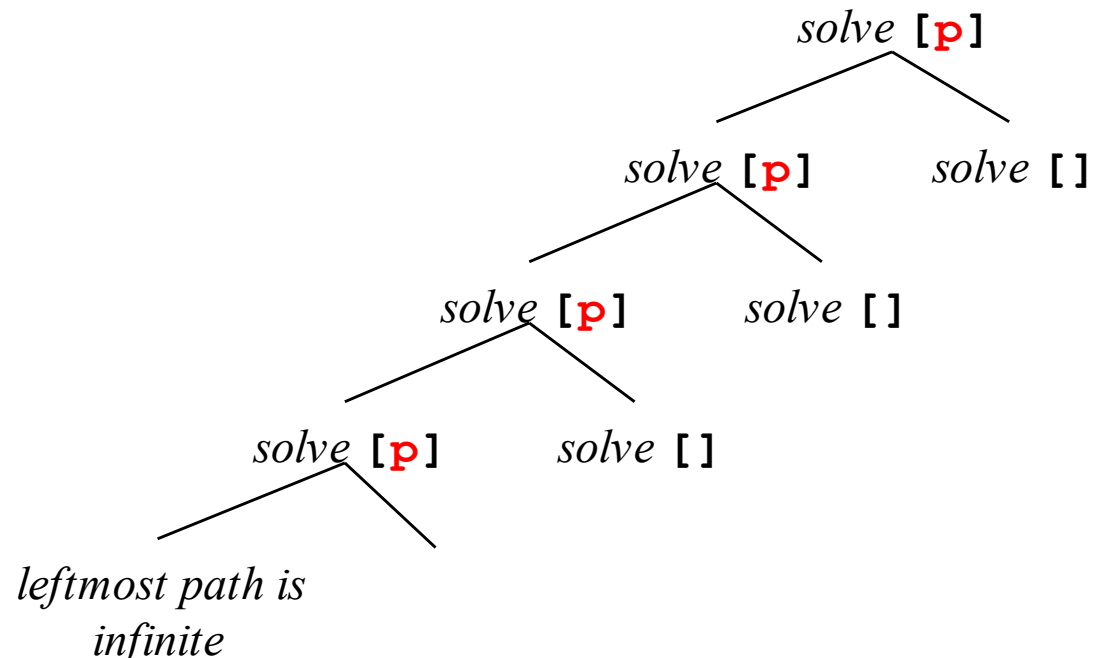


Prolog Semantics

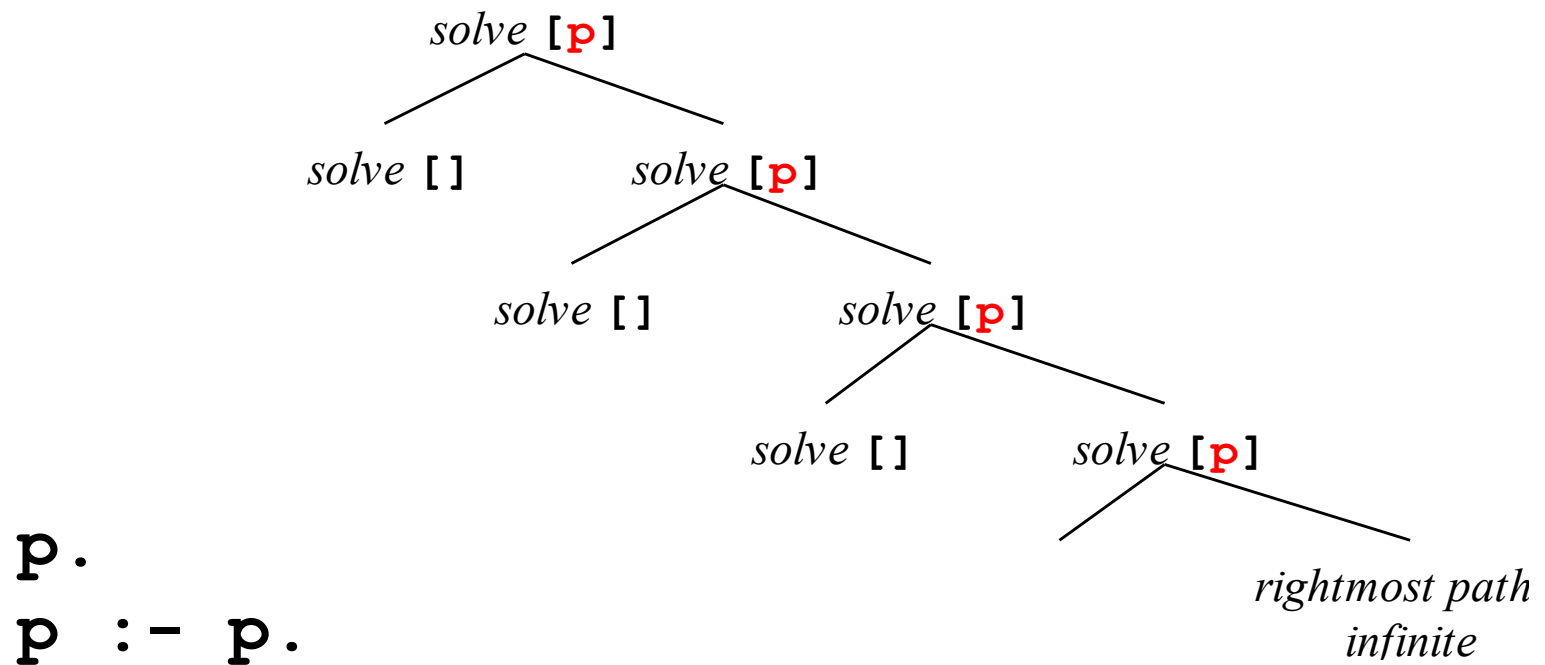
- *Given a program and a query, a Prolog language system must act in the order given by a depth-first, left-to-right traversal of the proof tree*
- It might accomplish that using an interpreter like our **prove**
- Or it might do it by some completely different means

Infinite Proof Tree, Nonterminating Program

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



Infinite Proof Tree, Terminating Program



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**, **\'**, ****

Prolog → configuration. for
easy.

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.
```

add this fact.

```
?- 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.
```

remove

```
?- 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**

An Adventure Game

□ Prolog comments

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

`/*`

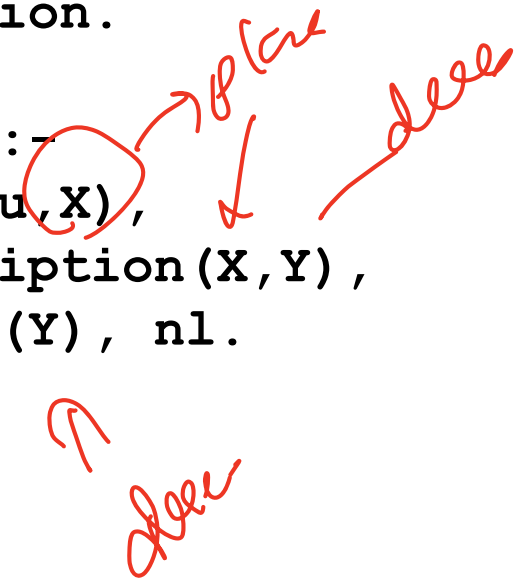
This is a little adventure game. There are three entities: you, a treasure, and an ogre. There are six places: a valley, a path, a cliff, a fork, a maze, and a mountaintop. Your goal is to get the treasure without being killed first.

`*/`




```
/*  
    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,  
    !.
```

*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.*

```
/*
```

```
    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.
```

```
/*  
    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.