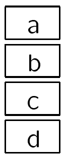
Example

A stack of 4 toy blocks.

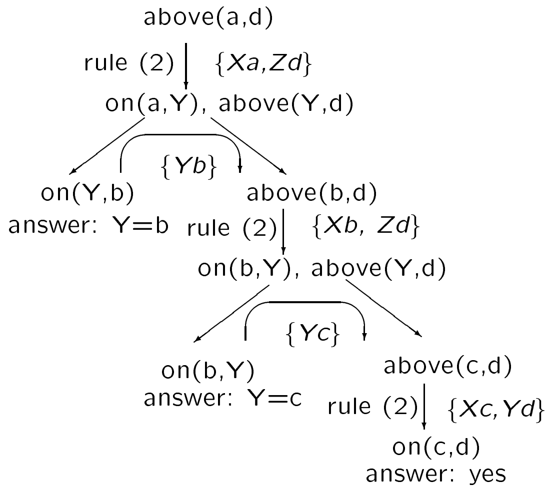


Rules:

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

Use bottom-up inference.

Tree



All leaves are true, so the root is true, i.e., above(a,d) is true.

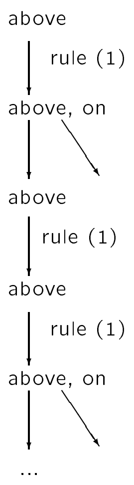
Observation

Changing the order of rules and/or rule premises can cause problems for Prolog. Example:

```
(1)  above(X,Z) :- above(Y,Z), on(X,Y).
(2)  above(X,Y) :- on(X,Y).
```

Because Prolog processes premises from left to right, and rules from first to last, rule (1) causes an infinite loop.

Tree



This is a flow in Prolog.

Declarative Arithmetic

<u>What we would like:</u>

- Given a set of equations with variables, find values of the variables that satisfy the equations.

eg,. query: `X + 3 = 5;`
answer: `X = 2`

query: `X + Y = 1, X - Y = 2;`
answer: `X = 3/2, Y = -1/2`

query: X^2 `= 4;`
answers: `X = 2`
`X = -2`

query: `X + Y = 0, 2X + 2Y = 1;`
answer: `no`
(no solutions since equations are contradictory)

query: `X = 1, X = 2;`
answer: `no`

Declarative Arithmetic (Cont'd)

There are two problems with this deal.
(1) There may be infinitely many answers

eg. query: `X + Y = 0;`
answers: `X = 0, Y = 0`
`X = 1, Y = -1`
`X = 2, Y = -2`
etc.

(2) The solutions may be difficult (or impossible) to compute

eq. query: $XY + XY^2 + Y^2X = 10,$
$(XY)^2 + X^2 + Y^2 = 6.$
answers: ??

There are really problems in <u>numerical analysis</u>, not logic programming.

Dealing with These Problems

Prolog takes a simple, but practical approach (though somewhat procedural and non-logical).

- Require that queries have the form
X_1 is ϕ_1, X_2 is ϕ_2, ... X_n is ϕ_n,
where each ϕ_i is an arithmetic expression and each X_i is a variable or a constant.

This query is interpreted to mean
$(X_1 = \phi_1) \wedge (X_2 = \phi_2) \wedge \ldots \wedge (X_n = \phi_n)$.
This is processed from left to right (as usual):

First X_1 is set to the value of ϕ_1
then X_2 is set to the value of ϕ_2
...
X_n is set to the value of ϕ_n.

Note: once a variable is assigned a value, it is fixed, i.e., it cannot change.

Examples

```
|? - X is 5+7.
     X = 12
|? - X is 5+7, Y is X-2.
     X = 12
     Y = 10
     (* left-to-right evaluation *)
|? - Y is X-2, X is 5+7.
     no
     (* X is unbound here *)
|? - 7 is 4+3.
     yes
|? - 8 is 4+3.
     no
```

A variable can only be given one value.
eq. `|?- X is 4, X is 5.`
(meaning $(X = 4) \wedge (X = 5)$)
`no`
i.e., there is no value of `X` that is <u>both</u> 4 and 5.

Arithmetic in Rule Bodies

```
square(X,Y) :- Y is X*X.
e.g. |? - square(5,Y).
          Y = 25
     |? - square(5,25).
          yes
     |? - square(5,13).
          no
     |? - square(X,25)
```

i.e., subquery is `25 is X*X`. X is unbound. So, error.

Function Symbols in Prolog

In logic, there are two kinds of objects: predicates and functions.

- Predicates represent statements about the world:
  `John hates Mary:  hates(john,mary).`
  `John is short:  short(john)`
  (`hates` is a predicate symbol, `short(john)` is an atomic formula)

- Function terms represent objects in the world
  `the mother of Mary:  mother-of(mary)`
  `a rectangle of length 3 and width 4:`
  `rectangle(3,4)`
  (`mother-of(mary)` is a function term, `rectangle` is a function symbol)

Function terms do not have values. In Prolog, the act like data structures:
let `p2(X,Y)` denote a point in 2-dim space
let `p3(X,Y,Z)` denote a point in 3-dim space.
Write a Prolog program, `SQDIST(Point1,Point2,D)`, that returns the square of the distance between two points. The program should work for 2- and 3-dim points.

Want:

```
SQDIST(p2(1,2), p2(3,5), D)
  returns D = (3-1)**2 + (5-2)**2
            = 4+9 = 13
and
SQDIST(p3(1,1,0), p3(22,3), D)
  returns D = (1-2)**2 + (1-2)**2 + (0-3)**2
            = 1+1+9=11
and
SQDIST(p2(0,0), p3(1,1,1), D)
  is undefined
```


The Prolog program:

```
(1) SQDIST(p2(X1,Y1), p2(X2,Y2), D)
      :- XD is X1-X2,
         YD is Y1-Y2,
         D is XD*XD + YD*YD.
(2) SQDIST(p3(X1,Y1,Z1), p3(X2,Y2,Z2), D)
       :- XD is X1-X2,
          YD is Y1 - Y2,
          ZD is Z1 - Z2,
          D is XD * XD + YD*YD + ZD*ZD.
```

Query: `SQDIST(p2(1,2), p2(3,5), D)`
This query unifies with the head of rule (1) with {`X1\1, Y1\2, X2\3, Y2\5`}
so, `XD` is `X1-X2=1-3=-2`
`YD` is `Y1-Y2=2-5=-3`
`D` is $(-2)^2 + (-3)^2 = 13$
So, `D=13` is returned

Note: the query does not unify with the head of rule (2), so only rule (1) is used.

Query: `SQDIST(p3(1,1,0),p3(2,2,3),D).`
The query unifies with the head of rule `(2)`, with {`X1\1, Y1\1, Z1\0, X2\2, Y2\2, Z2\3`}

so, `XD is 1-2 = -1`
`YD is 1-2 = -1`
`ZD is 0-3 = -3`
`D is 1 + 1 + 9 = 11`
So, `D=11` is returned

Note: this query does not unify with the head of rule `(1)`, so only rule `(2)` is used.

Query: `SQDIST(p2(0,0), p3(1,1,1), D).`
The query does not unify with any rule, so Prolog simply returns `no`, i.e., no answers for `D`.

Returning Function Terms as Answers

Eg. Given a point, `p2(X,Y)`, return a new point with double the coordinates.

Query `double(p2(3,4),P)`
Answer `P = p2(6,8).`
Prolog program:

```
double(p2(X1,Y1), p2(X2,Y2))
    :- X2 is 2*X1,
       Y2 is 2*Y1.
```

i.e., if `x2 = 2 * X1` and `Y2 = 2 * Y1`, then the double of `p2(X1,Y1)` is `p2(X2,Y2)`.

An equivalent program using "="

```
double(p2(X1,Y1), P)
    :- X2 is 2*X1, Y2 is 2*Y1,
       P = p2(X2,Y2).
```

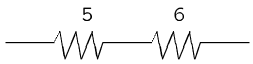
Here, "=" is being used to assign a value to variable `P`. Try to avoid this!!!!! It reflects procedural thinking.

Returning Terms as Answers (Cont'd)

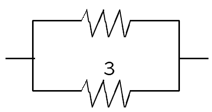
The query `double(p2(3,4),P)` unifies with the head of the rule with
`mgu = {X1\3, Y1\4, P\p2(X2,Y2)}`.
The body of the rule then evaluates:
`X2 is 2*X1`, i.e., 6
`Y2 is 2*Y2`, i.e., 8
The mgu becomes {`X1\3, Y1\4, P\p2(6,8)`}, so the answer is `P = p2(6,8)`.

Recursion with Function Symbols

Example: Electrical circuits

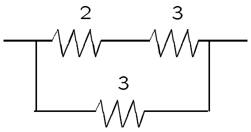


• Two resistors in series, with resistances R_1 and R_2, respectively.
• Total resistance of the circuit is $5 + 6 = 11$.
• Can represent the circuit as a function term: `series(5,6)`.

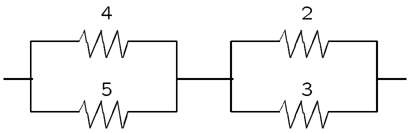


• Two resistors in parallel.
• Total resistance of the circuit is $\frac{2+3}{2\times 3} = 1.2$
• Represent the circuit as a function term: `par(2,3)`.

More Complex Circuits



`par(5, series(2,3))`



`series(par(4,5), par(2,3))`

Circuits (Cont'd)

Problem: Write a Prolog program that computes the total resistance of any circuit. E.g.

Query: `resistance(series(1,2), R)`
Answer: `R = 1+2 = 3`

Query: `resistance(par(2,3), R)`
Answer: `R = (2*3)/(2+3) = 6/5 = 1.2`

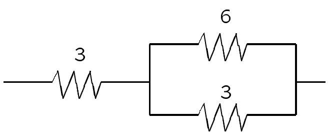
Query: `resistance(series(3,par(2,3)), R)`
Answer: `R = 3 + 1.2 = 4.2`

Query: `resistance(3, R)`
Answer: `R = 3`

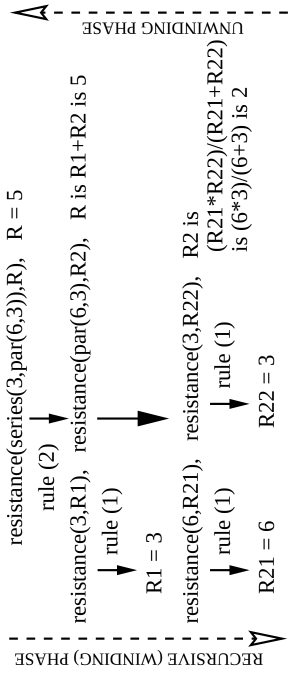
Solution:

```
(1) resistance(R,R) :- number(R).
(2) resistance(series(C1,C2), R)
       :- resistance(C1, R1),
          resistance(C2, R2),
          R is R1 + R2.
(3) resistance(par(C1,C2), R)
       :- resistance(C1,R1),
          resistance(C2,R2),
          R is (R1 * R2) / (R1 + R2).
```


Example



Query: `resistance(series(3,par(6,3)), TR)`



Lists in Prolog

special constant symbol [ ] $\approx$ `nil`
special function symbol $\bullet$ $\approx$ `cons`

Lisp list: `(a b c)`
Prolog list: $\bullet$`(a,` $\bullet$`(b,` $\bullet$`(c, [])))`
$\approx$ `(cons 'a (cons 'b (cons 'c nil)))`

Prolog shorthand:

1. $[t_1, \ldots, t_n] \equiv$ (`list` $'t_1 \cdots 't_n$)

2. $[t_1, \ldots, t_n \mid \texttt{L}] \equiv$ (`cons` $'t_1$ (`cons` $'t_2$ ($\cdots$ (`cons` $'t_n$ `L`) $\cdots$)

Thus `[a,b,c]` $\equiv$ `[a | [b,c]]`
$\equiv$ `[a,b | [c]]`
$\equiv$ `[a,b,c | []]`

List Membership

Prolog uses predicates, not functions. Want to define a two-place predicate

`member(X,Y)`

that holds iff `X` is a member of list `Y`.

Solution:

```
(1) member(X,[X|Y]).
(2) member(X,[Z|Y]) :- member(X,Y).
```

Interpretation:

1. `X` is a member of any list of which it is the first element.
2. If `X` is a member of list `Y`, then `X` is a member of any list that has `Y` as its `cdr`.

An Example

```
| ?- trace, member(c,[a,b,c,d]).
   (2) 1 Call: member(c,[a,b,c,d]) ?
   (3) 2 Call: member(c,[b,c,d]) ?
   (4) 3 Call: member(c,[c,d]) ?
   (4) 3 Exit: member(c,[c,d])
   (3) 2 Exit: member(c,[b,c,d])
   (2) 1 Exit: member(c,[a,b,c,d])
yes
| ?- trace, member(x,[a,b,c]).
   (2) 1 Call: member(x,[a,b,c]) ?
   (3) 2 Call: member(x,[b,c]) ?
   (4) 3 Call: member(x,[c]) ?
   (5) 4 Call: member(x,[]) ?
   (5) 4 Fail: member(x,[])
   (4) 3 Fail: member(x,[c])
   (3) 2 Fail: member(x,[b,c])
   (2) 1 Fail: member(x,[a,b,c])
no
```

Inductive Reasoning

- Identify Base Cases
  Handled directly as facts
  Simple solution, e.g null

- Identify Decomposition
  Reduce General Case to simpler
  sub cases

- Identify Composition
  Compose solution of Subcases to
  solution of original general case

Example - member

- Desired, `member( X , Y )`
  succeeds if X is a member of list Y

- Base case:
  X is at head of list Y
  `member( X , [X,Y] ).`

- Induction case:
  X is member of rest of Y except head
  `member( X, [Z,Y] ) :- member( X, Y).`

- Order of axioms matters.

Example - Append

- Desired: `append(X,Y,Z)`
  Z is result of appending X onto Y

- Base case:
  Append empty list onto Y
  `append( [], Y, Y ).`

- Decomposition:
  Process head of X
  Process rest of X
  Reduce length of X

- Composition:
  Add to head of Z

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

An Example

append([a,b],[c,d],V)

(2) $E_1 = a \quad X_1 = [b]$
$Y_1 = [c,d] V = [a|Z_1]$

append([b],[c,d],Z_1)

(2) $E_2 = b \quad X_2 = []$
$Y_2 = [c,d] Z_1 = [b|Z_2]$

append([ ],[c,d],Z_2)

(1) $Y_2 = [c,d]$
$Z_2 = [c,d]$

success

```
V  = [a |Z1]              = [a | [b |Z2]]
   = [a | [b | [c,d]]]    = [a,b,c,d]
```

More Examples

append([a,b],[c],[a,b,c,d])

$$\Big\downarrow (2) \quad E_1 = a \quad X_1 = [b] \quad Y_1 = [c] \quad Z_1 = [b,c,d]$$

append([b],[c],[b,c,d])

$$\Big\downarrow (2) \quad E_2 = b \quad X_2 = [] \quad Y_2 = [c] \quad Z_2 = [c,d]$$

append([ ],[c],[c,d])

$\Big\downarrow$ No possible unification.

failure

More Examples

append([a,b],W,[a,b,c])

$$\Big\downarrow (2) \quad E_1 = a \quad X_1 = [b] \quad W = Y_1 \quad Z_1 = [a,b,c]$$

append([ ],Y_1,[b,c])

$$\Big\downarrow (1) \quad Y_1 = [c]$$

success

$$\Rightarrow W = Y_1 = [c,d]$$

Similarly,

```
append(D,[c,d],[a,b,c,d])
```

succeeds with `D = [a,b]`.

Example - Reverse a List

- Desired `rev( L , M )`
  where the list M is the result of reversing the order of the list L
- Solution without helper functions
- Base case:
  Reverse empty list
  `rev( [], [] ).`
- Induction case:
  Append head of L to tail of output
  `rev([H,T],M) :- rev(T,Z),append(Z,[H],M).`

Example - Reverse a List with Helper Function

- Use helper function to hold partial result
- Base case:
  Start with empty output list
  `rev2(L1,L2) :- revzap(L1,[],L2).`
- Induction case:
  Accumulate results so far in 2nd arg of revzap

```
revzap([],L,L).
revzap([X,L],L2,L3):-revzap(L,[X,L2],L3).
```

Equality in Lists

The predicate for equality of lists is trivial:

```
equal(X,X).
```

Unification does all the work!

Note that equal(t_1, t_2) succeeds if t_1 and t_2 are merely unifyable:

equal([a,b,c],[a|X]) $\leadsto$ success + X = [b,c]

An Extended Example

We will represent the natural numbers as Prolog terms:

constant	0	
unary function symbol	s	≈ successor

0	represents	0
s(0)	...	1
s(s(0))	...	2
s(s(s(0)))	...	3
	etc.	

[Prolog does have numbers, but we ignore them for now.]

Addition Predicate

Want to define sum:

sum(X,Y,Z) holds iff X + Y = Z.

Solution:

```
sum(0,Y,Y).                        (1)
sum(s(X),Y,s(Z)) :- sum(X,Y,Z)     (2)
```


An Example

sum(s(s(0)),s(s(s(0))),Total)

(2) $Total = s(Z)$, $X = s(0)$, $Y = s(s(s(0)))$

$\downarrow$

sum(s(0),s(s(s(0))),Z)

(2) $Z = s(Z_1)$, $X_1 = 0$, $Y_1 = s(s(s(0)))$

$\downarrow$

sum(0,s(s(s(0))),Z_1)

(1) $Z_1 = s(s(s(0)))$

$\downarrow$

success

$\Rightarrow Total = s(s(s(s(s(0)))))$

Another Example

Notation: $\overline{n} \equiv \underbrace{s(s(\cdots s(}_{n \text{ times}}0)\cdots)$

---

sum(s(s(0)),s(s(s(0))),s(s(s(s(0)))))

$\Big\downarrow$ (2) $X = \overline{1}$, $Y = \overline{3}$, $Z = \overline{3}$

sum(s(0),s(s(s(0))),s(s(s(0))))

$\Big\downarrow$ (2) $X_1 = 0$, $Y_1 = \overline{3}$, $Z_1 = \overline{2}$

sum(0,s(s(s(0))),s(s(0)))

$\Big\downarrow$ No unification possible.

failure

$\Rightarrow$ `sum`$(\overline{2},\overline{3},\overline{4})$ fails

Addition as Subtraction

sum$(\overline{1}, W, \overline{3})$

$\Big\downarrow$ (2) $X = 0$, $Y = W$, $Z = \overline{2}$

sum$(\overline{0}, Y, \overline{2})$

$\Big\downarrow$ (1) $Y = \overline{2}$

success $\Rightarrow$ `W` = $\overline{2}$

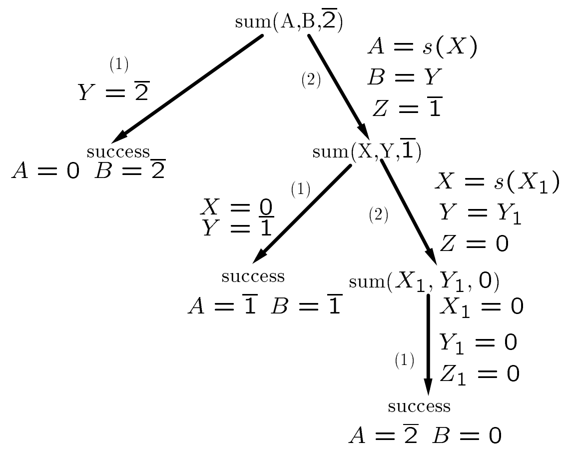
Similarly, `sum(D,`$\overline{3},\overline{8}$`)` $\rightsquigarrow$ `D` = $\overline{5}$

$\Rightarrow$ can define subtraction via one axiom:

```
diff(X,Y,Z) :- sum(Y,Z,X).
```

`diff(X,Y,Z)` holds iff $X - Y = Z$.

Creative Use of Variables



Similarly, `sum(A,B,`$\overline{5}$`)` succeeds with

$A = 0$ $B = \overline{5}$
$A = \overline{1}$ $B = \overline{4}$
$A = \overline{2}$ $B = \overline{3}$
etc.

A Bigger Example: Sorting

Declarative approach:

```
sort(X,Y) :- permute(X,Y), ordered(Y).
```

i.e. `Y` is the sorted version of `X` if `Y` is a permutation of `X` and `Y` is ordered.

`permute` and `ordered` are not difficult to define, but `ordered` will depend of the type list element.

However, this is a terrible way to sort.

A list with n elements has $n!$ permutations, and each of them will be tried until one is found to be ordered.

Sorting Lists

- Common Operation for Organizing Data
- Could use builtin list sorting primitives in Prolog. We'll look at how to do it directly.
- Assume that there is an ordering relation:
  `gt(X,Y)`
  for whatever data type we want to sort.
  Also assume non-empty lists.
- Look at several alternatives:
  bubble sort
  insertion sort
  quick sort

Basic Bubble Sort

Algorithm: swap adjacent elements in the list until all items are in order

```
bubblesort( List, Sorted ) :-
    swap( List, List1 ),
    bubblesort( List1, Sorted ).

bubblesort( Sorted, Sorted ).

swap( [] , [] ).
swap( [ X, Y | Rest ], [ Y, X | Rest ]) :-
    gt( X, Y ).
swap( [ Z | Rest ] , [ Z | Rest1 ] ) :-
    swap( Rest, Rest1 ).
```


Insertion Sort

- To sort a non-empty list `L = [X|T]`
  - Sort the tail T of L
  - Insert the head X into the sorted tail such that the resulting list is sorted

```
insertsort( [], [] ).
insertsort([ X | Tail ], Sorted ) :-
    insertsort( Tail, SortedTail ),
    insert( X, SortedTail, Sorted ).
insert( X, [ Y, Sorted ], [ Y | Sorted1 ] ) :-
    gt( X, Y ) , ! ,
    insert( X, Sorted, Sorted1 ).
insert( X, Sorted, [ X | Sorted ] ).
```


Quicksort

To sort a non-empty list L:

- Select some element X from L as a *pivot*
- Split the rest of L into two lists (Small and Big) such that all elements of L greater than X are in Big and all the rest are in Small
- Sort Small to obtain SortedSmall
  Sort Big to obtain SortedBig
- The whole sorted list is the concatenation of
  SortedSmall and `[ X | SortedBig ]`

Quicksort Algorithm

```
quicksort([] , [] ).
quicksort([X | Tail ], Sorted) :-
    split(X, Tail, Small, Big),
    quicksort(Small, SortedSmall),
    quicksort(Big, SortedBig),
    conc(SortedSmall,[ X | SortedBig],Sorted).

split(X, [], [], []).
split(X, [ Y | Tail ], [ Y | Small ], Big):-
    gt( X, Y ), ! ,
    split( X, Tail, Small, Big ).
split(X, [ Y | Tail ], Small, [ Y | Big ]) :-
    split( X, Tail, Small, Big ).

conc( [], L, L ).
conc( [ X | L1 ], L2, [ X | L3 ] ):-
    conc( L1, L2, L3 ).
```


More Efficient List Concatenation

- Naive list concatenation:

```
conc( [], L, L ).
conc( [ X | L1 ], L2, [ X | L3 ] ):-
    conc( L1, L2, L3 ).
```

- This is very inefficient when first list is long.
  Recurs down first list until it is empty.
  Could do better if we could efficiently find end of first list.

- Change representation of lists to represent a list as a pair of lists.
  A *Difference List* represents a list L and a difference of two lists L1 and L2, written as L1-L2

Difference List Example

```
Let   L = [ a, b, c ]

Then L can be represented as L1-L2 using any of

      L1 = [ a, b, c, d, e ]
      L2 = [ d, e ]

      L1 = [ a, b, c ]
      L2 = [ ]

      L1 = [ a, b, c, d, e | T ]
      L2 = [ d, e | T ]

Where T is any list.
Note that L2 is the end of the list L.
```


Concatenation of Difference Lists

```
Difference list concatenation:

concat( A1-Z1, Z1-Z2, A1-Z2 ).

Example:
Let
    M = [ a, b, c ]
    M = [ a, b, c | T1 ] - T1
    N = [ d, e ]
    N = [ d, e | T2 ] - T2
Then
   concat( M, N )
is
   concat([a, b, c | T1]-T1,[d, e, T2]-T2,L)
Resulting in
   T1 = [ d, e, | T2 ]
   L  = [ a, b, c, d, e | T2 ] - T2
```

More Efficient Quicksort Using Difference Lists

```
quicksort( List, Sorted ) :-
   quicksort2(List,Sorted-[]).

quicksort2([],Z-Z).

quicksort2([ X | Tail], A1-Z2) :-
    split(X, Tail, Small, Big ),
    quicksort2(Small, A1-[X | A2]),
    quicksort2(Big, A2-Z2 ).
```


Prolog - Review

- Overview
- Predicate Calculus
- Substitution and Unification
- Introduction to Prolog
- Prolog Inference Rules
- Programming in Prolog
  - Recursion
  - List Processing
  - Arithmetic
  - Higher-order programming
- Conclusion

Prolog vs. "Standard" Procedures

1. Success vs. failure

```
p(X) :- <body 1>.
p(X) :- <body 2>.
```

Procedural: Try `<body 1>`.
If that fails, try`<body 2>`.

2. Automatic backtracking

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

Call `p(a)`. Assign $Y = b$ then fails.
Reassign $Y = c$ then succeeds.
Can tell Prolog to find another proof.
Reassign $Y = d$ then succeeds.
Try again $\leadsto$ failure.

3. Non-determinism (in principle)
We can execute a goal in more than one way.

```
p(X) :- <body 1>.
p(X) :- <body 2>.
```

Which body will be executed?
(Prolog's use of DFS makes this deterministic.)

4. Variable use

- variables can be given values before proc. call or by proc. call
- variables need not have or be given values at all, but can simply be constrained with respect to other variables via unification.
- can defer decisions about particular values that variables must have as long as possible
$\longrightarrow$ major strength of Prolog.