SEARCHING and BACKTRACKING

When Prolog is trying to solve a goal

$G(X_1 , X_2 , \ldots , X_n)$

it starts with the first rule (or fact) in its database whose left side MATCHES the goal.

$$G(Y_1 , Y_2 , \ldots , Y_n) \ :- \ S_1(Y_1 , Y_2 , \ldots , Y_n) ,$$
$$S_2(Y_1 , Y_2 , \ldots , Y_n) ,$$
$$S_3(Y_1 , Y_2 , \ldots , Y_n) , \ldots$$
$$S_k(Y_1 , Y_2 , \ldots , Y_n)$$

It proceeds, left-to-right, trying to solve the first subgoal.

If a subgoal succeeds, it goes on to the next.

If a subgoal fails, it backs up to try for another solution to the previous subgoal.

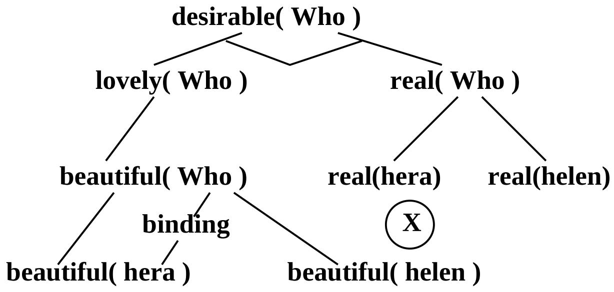
If the first subgoal fails, it goes on to the next rule in the database whose left side matches the goal.

EXAMPLE

```
real( helen ) .
beautiful(hera ) .
beautiful( helen ) .
beautiful( aphrodite ) .

lovely( X )  : -  beautiful( X ) .
desirable( X )  : -  lovely( X ) ,  real( X ) .

| ? -  desirable( Who ) .
```



Who = helen

Birdwatching Example


```
weather( sunday ,  fair ).
weather( monday , overcast ) .
weather( tuesday ,  fair ) .
weather( wednesday , fair ) .
weather( thursday ,  overcast ) .
weather( friday , rainy ) .
weather( saturday , overcast ) .

active( birds , sunday ) .
active( birds , tuesday ) .
active( birds , thursday ) .

observed( rarebird , wednesday ) .
observed( rarebird , friday ) .

happy( birders , Day )   : -   weather( Day , fair ) ,
                               active( birds , Day ) .
happy( birders , Day )   : -   observed( rarebird , Day ) .

|? -   happy( birders , When ) .
```

B E S S Y


```
owned( bessy ,  Person )    : -
                bought( bessy , Person , manufacturer ) .

owned( bessy , Person )     : -
                bought( bessy , Person , Seller ) ,
                owned( bessy , Seller ) .

bought( bessy , fred , manufacturer ) .
bought( bessy , ben , carl ) .
bought( bessy , carl , fred ) .

|? -    owned( bessy , Who ) .
```

P.21

EQUALITY and INEQUALITY

Prolog provides built-in predicates for testing equality and inequality.

But you write them in INFIX!

a = a .
yes

a = b .
no

1 = 2 .
no

1 < 2 .
yes

1 = < 2 .
yes

1 \ = = 2 .
yes

NOTE!

between(X , A , B) :- X > = A , X = < B .

P.22

SIMPLE ARITHMETIC and IS

Predicates are only true or false.

We need functions to do arithmetic.

But if a function returns a value, can we assign that value to a variable?

Yes, using the "is" operator.

```
twice( Y , X )  : -   Y   is   2 * X .
```

```
|?-   X  is   4 / 2 .
      X = 2

|?-   X  is   5 / 2 .
      X = 2.5

|?-   twice( 4 , 2 ) .
      Yes

|?-   twice( Z , 4 ) .
      Z = 8

|?-   twice( 4 , Z ) .
      EXCEPTION
```

SWI -- PROLOG (VERSION 2.9.5)

To enter it: prolog

To exit it: halt or CNTL D

To enter a mode for interactively typing in facts and rules :

consult(user) .

To leave user consult mode :

CNTL D

To query it :

type your query .

To ask for another solution:

;

To go on to the next query :

CR

EXAMPLE


```
prolog

|?-   consult( user ) .

|   baby( mike ) .
|   baby( nick ) .
|   father( mike ,  bob ) .
|   father( nick ,  mauro ) .

|   CNTL D

|?-   baby( nick ) .
      yes

|?-   baby( X ) .
      X = mike  ;  X = nick  ;  no

|?-   father( mike , Y ) .
      Y = bob   CR

|?-   father( X , Y ) .
      X = mike , Y = bob ;  X = nick , Y = mauro    CR

      halt .
```

To read rules and facts from a file:

```
consult( ' < f name > ' ) .
```

What do you put at the end of the file?

```
end_of_file .
```

To interrupt:

CNTL-C and h for help

To get online help:

help.	online help
help(< topic >)	general topic, ref. not known
help(i - j)	displays section i . j of manual

A RECURSIVE PROLOG PROGRAM FOR FACTORIAL

```
factorial( 0 , 1 ) .

factorial( N , ANS )  : -

        N1  is  N - 1 ,
        factorial( N1 ,  ANS2 ) ,
        ANS  is  N * ANS2 .
```

```
                                         bound to ANS_1
| ? -   factorial( 3 , X ) .  <--
                                                 bound to ANS_2
        N1_1  <--  2
        factorial( 2 , ANS2_1 )  <--
                                                   bound to ANS_3
          N1_2  <--  1
          factorial( 1 , ANS2_2 )  <--
               N1_3  <--  0
               factorial( 0 , ANS2_3 )
               factorial( 0 ,  1 )
               ANS_3  <--  1 * 1 = 1
          ANS_2  <--  2 * 1 = 2
        ANS_1  <--  3 * 2 = 6
```

X = 6

What happens if you now type ';' ?

LISTS & TREES

Lists are the same concept as in Lisp.

The syntax is different in this Prolog.

[ ]   is the empty list.

[a , b , c , d]   corresponds to the Lisp list (a b c d) .

Dotted pairs (trees) are also equivalent to those in Lisp.

. (a , [ ])   is the Lisp (a • NIL) or (a) .

. (a , . (b , . (c , [ ])))   is the Lisp list (a b c) or
Prolog list [a , b , c ,]

. (. (a , b) , . (c , d))   is the Lisp ((a • b) • (c • d))
or
(cons (cons a b) (cons c d))

Working with cars and cdrs is different!

A list with   a car of X and
a cdr of Y

is represented by the notation

[x | y] .

When this pattern is matched against a list,
X matches the car
Y matches the cdr .

Example

```
        ok( [ a | _ ] ) * .
        ok( [ _ | A ] ) : -  ok( A ) .
| ?-  ok( [ a , b , c , d ] ) .
      yes
| ? -  ok( [ b , c , d , e ] ) .
      no
|?-   ok( [ b, c, a ] ) .
      Yes
```

* _ is a wild card variable.

LIST OPERATIONS IN PROLOG

Member

```
member2 ( X , [ X | _ ] ) .
member2 ( X , [ _ | Rest ] ) :- member 2( X, Rest )
.
| ? -   member ( cheese , [ milk , bread , cheese , eggs ] ) .
        yes
| ? -   member ( cheese , [ milk , [ bread , cheese ] , eggs ] ).
        no
```

Length

```
length1( [ ] , 0 ) .
length1( [ _ | Rest ] , L ) : -
        length1( Rest , L 1 ) ,
        L is L1+ 1 .

| ? -  length1 ( [ a , b , c , d ] , X ) .
```

APPEND


```
/* append3( List1, List2, Result ) */
append3( [ ] , Alist , Alist ) .
append3( [ First | Rest1 ] , Alist , [ First | Rest2 ] ) : -
        append3( Rest1 , Alist , Rest2 ) .

| ? -  append3( [ a , b , c , d ] , [ e , f , g , h ] , X ] .
       X = [ a , b , c , d , e , f , g , h ]

| ? -  trace.
       append3( [ a , b ] , [ c , d ] , R ) .

      0  Call: append3( [ a , b ] , [ c , d ] , __ 410 )
      1  Call: append3( [ b ] , [ c , d ] , __ 534 )
      2  Call: append3( [ ] , [ c , d ] , __ 566 )
      2  Exit: append3( [ ] , [ c , d ] ,  [ c , d ] )
      1  Exit: append3( [ b ] , [ c , d ] , [ b , c , d ] )
      0  Exit: append3( [ a , b ] , [ c , d ] , [ a , b , c , d ] )

R = [ a , b , c , d ]
```

Reverse (using append)

```
/* reverse3( Listin, Listout ) */

reverse3( [ ] , [ ] ) .

reverse3( [ First | Rest ] , Ans ) : -
    reverse3( Rest , Temp ) ,
    append3( Temp , [First] , Ans ) .
```

```
| ? - reverse3( [ a , b , c , d ] , X ) .
  X = [ d , c , b , a ]
```

How efficient is this procedure?

reverse3([a, b, c], X) caused
4 calls to reverse3 .
6 calls to append3 .

reverse (using an accumulator)

```
/* rev2 ( Listin, Accum, Listout ) */

rev2( [ ] , X , X ) .

rev2( [ X | Y ] , Inter , Ans ) : -
    rev2( Y , [ X | Inter ] , Ans ) .

|?- trace

| ? - rev2( [ a , b , c ] , [ ] , X ) .

0 rev 2 ( [ a , b , c ] , [ ] , __ 406 )
1 rev 2 ( [ b , c ] , [ a ] , __ 406 )
2 rev 2 ( [ c ] , [ b , a ] , __ 406 )
3 rev 2 ( [ ] , [ c , b , a ] , __ 406 )
3 rev 2 ( [ ] , [ c , b , a ] , [ c , b , a ] )
2 rev 2 ( [ c ] , [ b , a ] , [ c , b , a ] )
1 rev 2 ( [ b , c ] , [ a ] , [ c , b , a ] )
0 rev 2 ( [ a , b , c ] , [ ] , [ c , b , a ] )

X = [ c , b , a ]
```

Now it is 0(n) for a list of length n.

COUNTING THE NUMBER OF OCCURRENCES OF A PARTICULAR ITEM IN A LIST

```
/* count ( Item, List, Ans ) */
count( _ , [ ] , 0 ) .

count( X , [ X | T ] , N )  : -  count( X , T , J ) ,
                                 N  is  J + 1 .

count( X , [ _ | T ] , N )  : -  count( X , T , N ) .

| ? -  count ( a , [ a , b , a , e , r , a , a , e , q , a ] , X ) .
```


FLATTEN

```
/* FLATTEN ( Listin, Listout ) */
flatten3( [ ] , [ ] ) .
flatten3(  [ X | Xs ] , Y )  : -
        flatten3( X , XF ) ,
        flatten3( Xs , XsF ) .
        append3( XF , XsF , Y ) .
flatten3( X , [ X ] ) .

| ? -   flatten3( [ [ a , b ] , c , [ d , [ e , f ] ] ] , Q1 ) .

| ? -   flatten3( [ [ a , b ] , [ ] , c ] , Q2 ) .
```