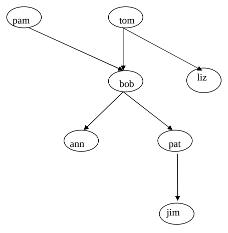
CSI 2165 Winter 2006

Diana Inkpen
SITE
University of Ottawa
Prolog, Part II

Prolog – Part II

- Data Structures.
  – Structures, lists
- Control Structures:
  – Backtracking, recursion, cut and failure.

Definition of predecessor/2



- In Lab 1, we define predecessor/2 by a list of facts on the given objects.
- How to define the *general* (using variables) relationship of predecessor?

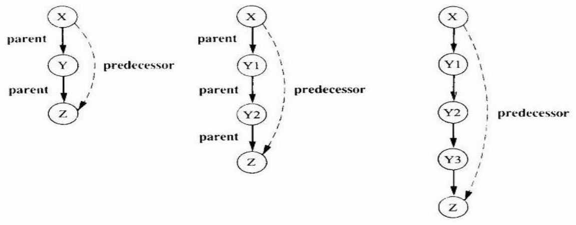
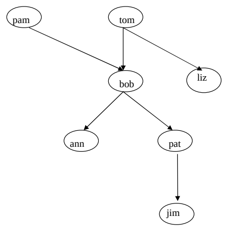


Figure 1.6 Predecessor-successor pairs at various distances.

```
predecessor( X, Z) :-
    parent( X, Y1),
    parent( Y1, Y2),
    parent( Y2, Z).
predecessor( X, Z) :-
    parent( X, Y1),
    parent( Y1, Y2),
    parent( Y2, Y3),
    parent( Y3, Z).
...
```

predecessor/2* can NOT be defined only through *parent/2

Recursive Rule: predecessor/2



For all X and Z,
X is a predecessor of Z if
X is a parent of Z.

For all X and Z,
X is a predecessor of Z if
there is a Y such that
(1) X is a parent of Y and
(2) Y is a predecessor of Z

Recursive Rule: predecessor/2

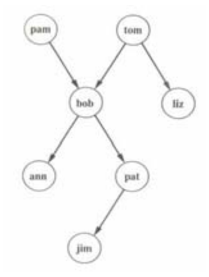
For all X and Z,
X is a predecessor of Z if
X is a parent of Z.

For all X and Z,
X is a predecessor of Z if
there is a Y such that
(1) X is a parent of Y and
(2) Y is a predecessor of Z

```
predecessor( X, Z) :-
    parent( X, Z).

predecessor( X, Z) :-
    parent( X, Y),
    predecessor ( Y, Z).
```

Note: we need two clauses for the full definition!

Let’s do a trace!



```
predecessor( X, Z)  :-
    parent( X, Z).

predecessor( X, Z)  :-
    parent( X, Y),
    predecessor (Y, Z).
```

```
?- predecessor(bob, X).
```

7

Four Versions of predecessor/2

Equivalent in declarative meaning but not procedural meaning! Order matters!

```
% Four versions of the predecessor program
% The original version

pred1( X, Z)  :-
    parent( X, Z).

pred1( X, Z)  :-
    parent( X, Y),
    pred1( Y, Z).
```

```
% Variation b: swap goals in second clause of the original version

pred3( X, Z)  :-
    parent( X, Z).

pred3( X, Z)  :-
    pred3( X, Y),
    parent( Y, Z).
```

```
% Variation a: swap clauses of the original version

pred2( X, Z)  :-
    parent( X, Y),
    pred2( Y, Z).

pred2( X, Z)  :-
    parent( X, Z).
```

```
% Variation c: swap goals and clauses of the original version

pred4( X, Z)  :-
    pred4( X, Y),
    parent( Y, Z).

pred4( X, Z)  :-
    parent( X, Z).
```

8

Arithmetic

Equality!

X =:= Y	X and Y stand for the same number
X =\= Y	X and Y stand for different numbers
X < Y	X is less than Y
X > Y	X is greater than Y
X =< Y	X is less than or equal to Y
X >= Y	X is greater than or equal to Y

9

Built-in Operators for Structures

=..   functor   arg

- Decomposing terms and constructing new terms
- functor/3 :
- arg/3
- =../2 - universal predicate
  - written as infix

10

=..

- Term =.. L.

```
?- f(a, b) =.. L.
L = [f, a, b]

?- T =.. [rectangle, 3, 5].
T = rectangle(3, 5)

?- Z =.. [p, X, f(X, Y)].
Z = p(X, f(X, Y))

?- a =.. L.
L = [a]
```

11

functor/3

- functor(T, F, N)
  - T: term; F: functor; N: arity

```
?- functor(f(a, b, g(Z)), F, N).
Z = _G448, F = f, N = 3.

?- functor([a, b, c], F, N).
F = ‘.’, N = 2

?- functor(a + b, F, N).
F = +, N = 2
```

12

arg/3

- arg(N, T, A)
  - N: integer, which argument; T: term; A: argument

```
?- arg(2, related(john, mother(jane)), X).
X = mother(jane)

?- arg(1, a + (b + c), X).
X = a

?- arg(2, [a, b, c], b).
No
```

13

is_structure/1

- checks that something is a structure
- is_structure(My_Term) :-

  functor(My_Term, _ ,N), N > 0.
  - We are not interested in the name of the structure (its functor), we are interested just in the fact that our argument has a functor and has arguments, which tell us the fact that it is a structure.
- Note: a list is a structure: the functor is '.', and it has 2 arguments - Head and Tail.

14

Operations on Lists

- Membership
- Concatenation
- Adding an item
- Deleting an Item
- Sublist
- Others
  - list/1 - check that something is a list
  - sum/2 - add the elements of a list of numbers
  - list_len/2 - compute the length of a list
  - flatten/2 - flatten a list
  - …

15

list/1

list([ ]).   The empty list is a list.

list([_ | Tail]) :- list(Tail).

- It is a structure that has a head and a tail, and the tail is also a list. We are not interested in the values of the head and the tail, only in the fact that the argument has this particular format.

16

[ ] is special!

- [ ] is a special list.
- [ ] is also an atom!
- [ ] is not a regular list
  - Because we cannot split [ ] into head and tail.
- This is why we use it in the base case of recursion!

```
4 ?- listing.

list([]).
list([A|B]) :-
        list(B).

Yes
5 ?- trace.
Yes
[trace] 5 ?- list([1,2]).
   Call: (7) list([1, 2]) ? creep
   Call: (8) list([2]) ? creep
   Call: (9) list([]) ? creep
   Exit: (9) list([]) ? creep
   Exit: (8) list([2]) ? creep
   Exit: (7) list([1, 2]) ? creep

Yes
```

17

Membership -1

- An element is a member of a list if it is the first element in the list (head), or if it is somewhere in the rest of the list (tail).
- A bit more formal:
  - if X is Head of List, or
  - if X is not the Head of List, but it is somewhere in the Tail of the List:
- Prolog format:

```
member(X,[Head|Tail]) :- X = Head.
member(X,[Head|Tail]) :- X \= Head,
                         member(X,Tail).
```

- We can simplify the above rules:

```
member(X,[X|_]).
member(X,[_|Tail]) :- member(X,Tail).
```

18

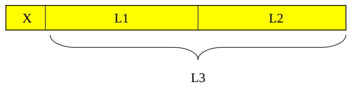
Membership -2

```
?- member(b, [a, b, c]).
Yes

?- member(b, [a, [b, c]]).
No

?- member([b, c], [a, [b, c]]).
Yes
```

- member2/2 - check that an element is a member of a list (can appear on any level)
- member3/2 - check that an element appears in a structure as an argument
- appears/2 - check that an element appears in a structure - can also be a functor name.

Concatenation

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



Decomposition through Concatenation

```
?- append(L1, L2, [a, b, c]).

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

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

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

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

No
```

Adding an Item

```
add(X, L, [X | L]).
```

- This adds an item as the head of the new list.
- What if we want to add it as the last element?
- What if we want to insert it into a given position?

Deleting an Item

```
del(X, [X | Tail], Tail).
del(X, [Y | Tail], [Y | Tail_1]) :-
        del(X, Tail, Tail_1).
```

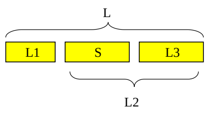
```
?- del(a, [a, b, a, a], L).
L = [b, a, a];
L = [a, b, a];
L = [a, b, a];
No

?- del(a, L, [1, 2, 3]).
L = [a, 1, 2, 3];
L = [1, a, 2, 3];
L = [1, 2, a, 3];
L = [1, 2, 3, a];
No
```

member(X, List) :- del(X, List, _).

- What if I want to delete all repetitions?

sublist/2

```
sublist(S, L) :-
        append(L1, L2, L),
        append(S, L3, L2).
```



```
?- sublist([c, d, e], [a, b, c, d, e, f]).
Yes

?- sublist([c, e], [a, b, c, d, e, f]).
No

?- sublist(S, [a, b, c]).
S = [ ];
S = [a];
S = [a, b];
S = [a, b, c];
S = [ ];
S = [b];
...
```

- Is a more general relation than member/2
  - Consider when S is a single element X.

list_length/2

- The length of an empty list is 0.
- The length of a general list is the length of the Tail of the list, plus 1 (the Head counts for one since it is one element).

```
list_len([], 0).
list_len([_|Tail], ListLen) :-
      list_len(Tail, TailLen),
      ListLen is TailLen + 1.
```

(we are not interested in what the Head of the list is or its value, it just matters to us that there is one)

Another Version - Accumulator

```
listlen(L, N) :- lenacc(L, 0, N).

lenacc([ ], A, A).
lenacc([_|Tail], A, N) :- A1 is A +1, lenacc(Tail, A1, N).
```

- A is called accumulator
- Accumulator does not have to be integer
  - If we are producing a list as a result, accumulator could be a list.
- This is called *tail recursion.*
  - It helps when memory efficiency is a concern.

Traces – See the Difference

```
[trace] 4 ?- list_len([1,2,3], L).
   Call: (7) list_len([1, 2, 3], _G499) ? creep
   Call: (8) list_len([2, 3], _L188) ? creep
   Call: (9) list_len([3], _L207) ? creep
   Call: (10) list_len([], _L226) ? creep
   Exit: (10) list_len([], 0) ? creep
^  Call: (10) _L207 is 0+1 ? creep
^  Exit: (10) 1 is 0+1 ? creep
   Exit: (9) list_len([3], 1) ? creep
^  Call: (9) _L188 is 1+1 ? creep
^  Exit: (9) 2 is 1+1 ? creep
   Exit: (8) list_len([2, 3], 2) ? creep
^  Call: (8) _G499 is 2+1 ? creep
^  Exit: (8) 3 is 2+1 ? creep
   Exit: (7) list_len([1, 2, 3], 3) ? creep

L = 3

Yes
```

```
[trace] 5 ?- listlen([1,2,3], L).
   Call: (7) listlen([1, 2, 3], _G493) ? creep
   Call: (8) lenacc([1, 2, 3], 0, _G493) ? creep
^  Call: (9) _L206 is 0+1 ? creep
^  Exit: (9) 1 is 0+1 ? creep
   Call: (9) lenacc([2, 3], 1, _G493) ? creep
^  Call: (10) _L226 is 1+1 ? creep
^  Exit: (10) 2 is 1+1 ? creep
   Call: (10) lenacc([3], 2, _G493) ? creep
^  Call: (11) _L246 is 2+1 ? creep
^  Exit: (11) 3 is 2+1 ? creep
   Call: (11) lenacc([], 3, _G493) ? creep
   Exit: (11) lenacc([], 3, 3) ? creep
   Exit: (10) lenacc([3], 2, 3) ? creep
   Exit: (9) lenacc([2, 3], 1, 3) ? creep
   Exit: (8) lenacc([1, 2, 3], 0, 3) ? creep
   Exit: (7) listlen([1, 2, 3], 3) ? creep

L = 3

Yes
```

Tail Recursion – Another Example

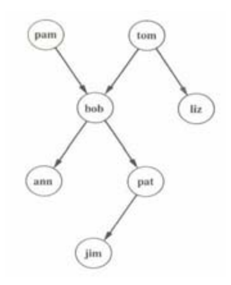
```
sumlist([ ], 0).

sumlist([First|Rest], Sum) :-
      sumlist(Rest, S1),
      Sum is First + S1.
```

```
sumlist(List, Sum) :-
      sumlist(List, 0, Sum).

sumlist([ ], Sum, Sum).

sumlist([F|R], P_sum, T_sum) :-
      P_sum_1 is P_sum + F,
      sumlist(R, P_sum_1, T_sum).
```

Clause Ordering

- In order to satisfy a goal, Prolog will try to match it with each fact and head of rule in the knowledge base, and they are tried in the order in which they were written in the knowledge base (/file).
- In pure logic programming, the order of clauses in a program should not affect the outcome of the program, but ...

Again, the Family Tree



```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```

Four Versions of predecessor/2

Equivalent in declarative meaning but not procedural meaning! Orders matter!

```
% Four versions of the predecessor program
% The original version

pred1( X, Z)  :-
    parent( X, Z).

pred1( X, Z)  :-
    parent( X, Y),
    pred1( Y, Z).
```

```
% Variation b: swap goals in second clause of the original version

pred3( X, Z)  :-
    parent( X, Z).

pred3( X, Z)  :-
    pred3( X, Y),
    parent( Y, Z).
```

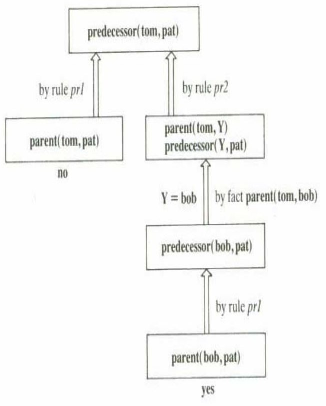
```
% Variation a: swap clauses of the original version

pred2( X, Z)  :-
    parent( X, Y),
    pred2( Y, Z).

pred2( X, Z)  :-
    parent( X, Z).
```

```
% Variation c: swap goals and clauses of the original version

pred4( X, Z)  :-
    pred4( X, Y),
    parent( Y, Z).

pred4( X, Z)  :-
    parent( X, Z).
```

```
predecessor( X, Z)  :-
        parent( X, Z).

predecessor( X, Z)  :-
        parent( X, Y),
         predecessor( Y, Z).
```

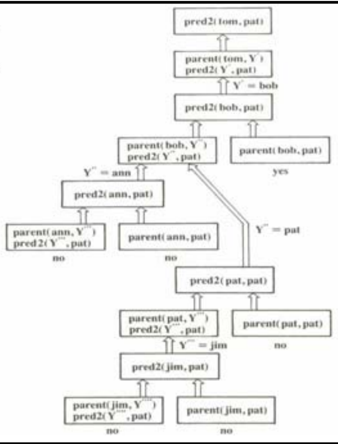
This is the pred1 in previous slide.

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```



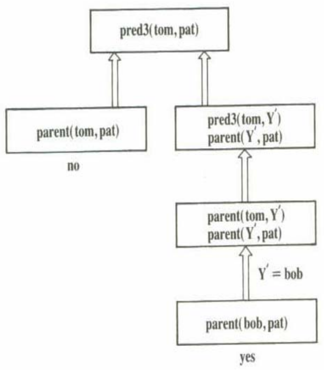
```
pred2( X, Z)  :-
   parent( X, Y),
   pred2( Y, Z).
pred2( X, Z)  :-
   parent( X, Z).
```

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```



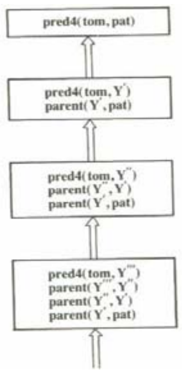
```
pred3( X, Z) :-
   parent( X, Z).
pred3( X, Z) :-
   pred3( X, Y),
   parent( Y, Z).
```

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```



```
pred4( X, Z)  :-
   pred4( X, Y),
   parent( Y, Z).
pred4( X, Z)  :-
   parent( X, Z).
```

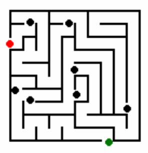
```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```



Lessons Learned

- Order does matter!
  - Sometimes have to consider both declarative and procedural meanings.
- In practice, you can work out a declaratively correct program first, then make it procedurally correct later by rearrange the orders.
- Advice:
  - Always try simpler ideas first!

Backtracking - The labyrinth analogy



In the execution of a Prolog program, every time a certain rule for a predicate is used, a choice is made - from all the rules, this particular one was chosen. There are others to try so if this one doesn't lead to the right answer, we can come back, and try the other possibilities - just like an intersection.

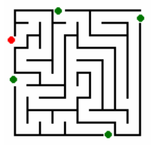
If we arrive at one point from where there is no way to go (a predicate that is false), turn back step by step to a point where a choice was made, and try an alternative.

Backtracking - The labyrinth analogy

What happened in our previous examples when we pressed ';'?

When you press ';'is like you ask the system to find you another proof for the given goal. To do that, it will 'pretend' that the last goal failed, and will try to satisfy it again.

It will stop when it has found another way to prove your goal, or if no such way was found.



There may be only one solution, or there may be many. We can force backtracking, to find all solutions.

Backtracking – Example 1

```
likes(john,apples).
likes(john,csi2165)
likes(john,mary).
? - likes(john,X).
```

- First fact is found that matches the goal.
  - → Unify likes(john,X) with likes(john,apples).
  - → *X* is instantiated to apples
  - → The goal has succeeded, the system prints out X = apples
- We press ';'
  - → the last goal failed (the matching on the first fact has failed),
  - → the variable becomes **uninstantiated**, and next facts are tried.
- Second fact is found that matches the goal.
  - → Unify likes(john,X) with likes(john,csi2165C).
  - → X is instantiated to csi2165
  - → The goal has succeeded, the system prints out X = csi2165
-

Backtracking – Example 2

```
member(X,[X|_]).
member(X,[_|Tail]) :-
        member(X,Tail).
?- member(X,[a,b,c]).
```

- Find match - match with the head of the first rule
  - → unify member(X,[a,b,c]) with member(X,[X|_])
  - → instantiate X to a
  - → (rule has no body and since matching has succeeded) goal has succeeded
  - → display X = a
- We press ';'
- Find another match - match goal with head of second rule
  - → unify member(X,[a,b,c]) with member(X,[_|Tail])
  - → instantiate X to X, instantiate Tail to [b,c]
  - → replace head of rule with the body, and try to satisfy the body of the rule
- New goal (subgoal) member(X,[b,c])
- Find match - match with head of the first rule
  - → unify member(X,[b,c]) with member(X,[X|_])
  - → instantiate X to b
  - → (rule has no body and since matching has succeeded) goal has succeeded.
  - → display X = b

Backtracking – Example 3

- What happens when a goal fails?

```
halve(L,FirstHalf,SecondHalf) :-
        conc(FirstHalf,SecondHalf,L),
        length(FirstHalf,HalfLength),
        length(SecondHalf,HalfLength).
```

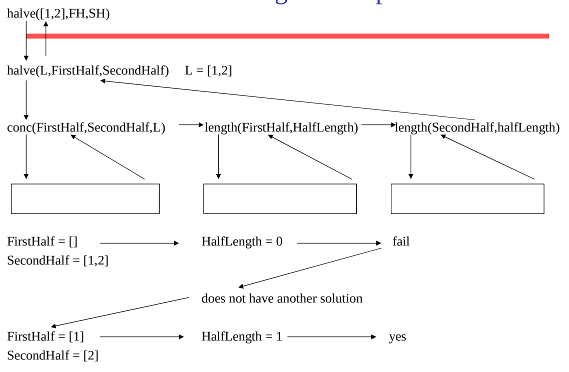
conc/3 is append/3 **in many books**

In order for Prolog to satisfy the goal:	halve(L, FirstHalf, SecondHalf),
it must satisfy the goal:	1. conc(FirstHalf, SecongHalf, L)
then it must satisfy the goal:	2. length(FirstHalf, HalfLength)
then it must satisfy the goal:	3. length(SecondHalf, HalfLength)

1, 2, 3 are called ***subgoals*** of the ***parent goal*** halve(L, FirstHalf, SecondHalf)

- whenever a subgoal is satisfied, Prolog marks the subgoal if there is *another way* it can be satisfied
- if a subsequent subgoal *fails* (cannot be satisfied in any way), Prolog *backtracks* to the to the most recent marked subgoal
- if no previous subgoals can be redone, the parent goal fails.

Backtracking – Example 3


halve([1,2],FH,SH)
halve(L,FirstHalf,SecondHalf)   L = [1,2]
conc(FirstHalf,SecondHalf,L)
length(FirstHalf,HalfLength)
length(SecondHalf,halfLength)
FirstHalf = []
SecondHalf = [1,2]
HalfLength = 0
fail
does not have another solution
FirstHalf = [1]
SecondHalf = [2]
HalfLength = 1
yes

List Processing – Program Patterns

1. Test for Existence
2. Test All Elements
3. Return a Result – having processed one element
4. Return a Result – having processed all elements

1. Test for Existence

- To determine that some collection of objects has at least one object with a desired property

```
list_existence_test(Info,[Head|Tail]):-
                element_has_property(Info,Head).
list_existence_test(Info,[Head|Tail]):-
                list_existence_test(Info,Tail).
```

1. Test for Existence - Examples

```
member(Element,[Head|Tail]):-
                Element = Head.
member(Element,[Head|Tail]):-
                member(Element,Tail).
```

```
nested_list([Head|Tail]):-
                sublist(Head).
nested_list([Head|Tail]):-
                nested_list(Tail).
sublist([]).
sublist([Head|Tail]).
```

2. Test All Elements

- We require that the elements of a list all satisfy some property.

```
test_all_have_property(Info,[]).
test_all_have_property(Info,[Head|Tail]):-
                element_has_property(Info,Head),
                test_all_have_property(Info,Tail).
```

2. Test All Elements - Example

- all_digits/1 tests that a list consists digits only.

```
all_digits([]).
all_digits([Head|Tail]):-
                member(Head,[0,1,2,3,4,5,6,7,8,9]),
                all_digits(Tail).
plus definition of member/2.
```

3. Return a Result (one element)

- This requires an extra argument to be carried around --- termed the *result* argument.
- work through a list until an element satisfies some condition whereupon we stop and return some result.

```
return_after_event(Info,[H|T],Result):-
                property(Info,H),
                result(Info,H,T,Result).
return_after_event(Info,[Head|Tail],Ans):-
                return_after_event(Info,Tail,Ans).
```

3. Return a Result (one element)- Example

- predicate everything_after_a/2 that takes a list and returns that part of the list after any occurrence of the element **a**

```
everything_after_a([Head|Tail],Result):-
                    Head = a,
                    Result = Tail.
everything_after_a([Head|Tail],Ans):-
                    everything_after_a(Tail,Ans).
```

4. Return a Result (all elements)

- common task: taking a list of elements and transforming each element into a new element (this can be seen as a mapping)

```
process_all(Info,[],[]).
process_all(Info,[H1|T1],[H2|T2]):-
                process_one(Info,H1,H2),
                process_all(Info,T1,T2).
```

where **process_one/1** takes **Info** and **H1** as input and outputs **H2**

The second clause can be rewritten to:

```
process_all(Info,[H1|T1],Ans):-
                process_one(Info,H1,H2),
                process_all(Info,T1,T2),
                Ans = [H2|T2].
```

4. Return a Result (all elements) - Example

- **triple/2** takes a list of integers and triples each of them

```
triple([],[]).
triple([H1|T1],[H2|T2]):-
                H2 is 3*H1,
                triple(T1,T2).
```

Recipes for processing of nested lists

(Nested list = a list that contains other lists as elements)

1. Test for Existence

- To determine that at least one object in a nested list has a desired property:
  - If head of list has property, success and stop.
  - If head of list is not a list, recursive test on tail of list.
  - If head of list is a list, recursive test of head of list to go into nested levels and recursive test on tail of list. (use OR so that the recursive call of tail is executed only if the element was not found in head)

1. Test for Existence - Example

```
nested_member(X,[X|_]).
nested_member(X,[H|T]):- H\=X,not(is_list(H)),
                         nested_member(X,T).
nested_member(X,[H|T]):- H\=X, is_list(H),
   (nested_member(X,H); nested_member(X,T)).


?- nested_member(b, [a, [b, [c]], [d]).
yes
```

2. Test All Elements

- Test if all the elements of a nested list all satisfy some property.
  - If empty list, success and stop.
  - If head of list is not a list, test head of lists and if success, recursive test on tail of list.
  - If head of list is a list, recursive test of head of list to go into nested levels and (if success) recursive test on tail of list.

2. Test All Elements - Example

```
alldigits([]).
alldigits([H|T]):- not(is_list(H)),
  member(H,[0,1,2,3,4,5,6,7,8,9]),
  alldigits(T).
alldigits([H|T]):- is_list(H),
  alldigits(H), alldigits(T).



?- alldigits([1,[2,[3]],4]).
Yes
?- alldigits([1,[2,[3, a]],4]).
No
```

3. Return a Result (one element)

- Same as test for one element, but requires an extra argument to be carried around - the *result*.
- Work through a list until an element satisfies the condition whereupon we stop and return some result.
- If the head is a list, look in the head too.

4. Return a Result (all elements)

- Common task: taking a list of elements and transforming each element into a new element (this can be seen as a mapping).
- Same as test for all elements, but requires an extra argument - the result.
- For nested lists, if the head of the list is a list, recursive call on it too.

4. Return a Result (all elements) - Example

```
addone([],[]).
addone([H|T],[H1|T1]):- not(is_list(H)),
              H1 is H+1, addone(T,T1).
addone([H|T],[H1|T1]):- is_list(H),
              addone(H,H1), addone(T,T1).



?- addone([1,[2,[3]],4], R).
R = [2, [3, [4]], 5]
```

Four Versions of predecessor/2

Equivalent in declarative meaning but not procedural meaning! Orders matter!

```
% Four versions of the predecessor program
% The original version

pred1( X, Z)  :-
    parent( X, Z).

pred1( X, Z)  :-
    parent( X, Y),
    pred1( Y, Z).
```

✓

```
% Variation b: swap goals in second clause of the original version

pred3( X, Z)  :-
    parent( X, Z).

pred3( X, Z)  :-
    pred3( X, Y),
    parent( Y, Z).
```

✓✗

```
% Variation a: swap clauses of the original version

pred2( X, Z)  :-
    parent( X, Y),
    pred2( Y, Z).

pred2( X, Z)  :-
    parent( X, Z).
```

✓✗

```
% Variation c: swap goals and clauses of the original version

pred4( X, Z)  :-
    pred4( X, Y),
    parent( Y, Z).

pred4( X, Z)  :-
    parent( X, Z).
```

✗

Clause Ordering – Another Example

- Delete all occurrences of an element from a list:

```
delete_all_1(E, [], []).

delete_all_1(E, [E | Tail], List) :-
      delete_all_1(E, Tail, List).

delete_all_1(E, [NotE | Tail], [NotE | NewTail]) :-
    delete_all_1(E, Tail, NewTail).

? - delete_all_1(b, [a, b, c, d, b, e, b, f], X).
X = [a, c, d, e, f]
```

Clause Ordering – Another Example

- Another definition (we switch the order of the second and third rules in the previous definition):

```
delete_all_2(E, [], []).

delete_all_2(E, [NotE | Tail], [NotE | NewTail]) :-
    delete_all_2(E, Tail, NewTail).

delete_all_2(E, [E | Tail], List) :-
      delete_all_2(E, Tail, List).

? - delete_all_2(b, [a, b, c, d, b, e, b, f], X).
X = [a, b, c, d, b, e, b, f]
```

Clause Ordering – Another Example

- Yet another definition (same order, but that this time it works):

```
delete_all_3(E, [], []).
delete_all_3(E, [NotE | Tail], [NotE | NewTail]) :-
     E \== NotE,
     delete_all_3(E, Tail, NewTail).
delete_all_3(E, [E | Tail], List):-
      delete_all_3(E, Tail, List).
? - delete_all_3(b, [a, b, c, d, b, e, b, f], X).
X = [a, c, d, e, f]
```

Try the 3 versions in SWI-Prolog by hitting ";"

- In general, the clauses of a predicate should be **exclusive**; *i.e.*, only one clause should be satisfiable for a given input, to make order irrelevant.

Failure

- When Prolog cannot satisfy a goal we say it fails.
- We can force some predicate to fail
- '*fail*' - built in Prolog predicate - *it always fails*
  - opposite one if *'true'* – *it always succeeds.*
- Example:

```
? - fail.
? - number(5).
? - number(5), fail.
```

Why is such a predicate interesting?

Using 'fail' to Force Backtracking

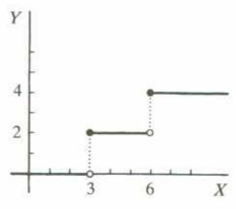
- How can we obtain all the solutions to a problem?
  - We can press ';'
  - We can use 'fail'
- Example:
  - write everything that John likes:

```
all_he_likes(Somebody) :-
          likes(Somebody,X), write(X), fail.
?- all_he_likes(john).
```

  - write all the elements of a list one by one:

```
write_member(List) :- member(X, List), write(X),
   write(' '), fail.
?- write_member([aa, bb, cc]).
```

Controlling Backtracking - CUT

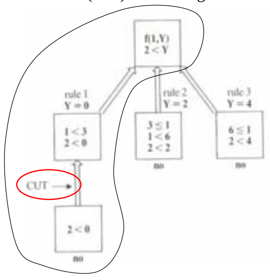


```
f( X, 0)  :-  X < 3.
f( X, 2)  :-  3 =< X, X < 6.
f( X, 4)  :-  6 =< X.
```

?- f(1, Y), 2 < Y.

Controlling Backtracking - CUT

- The three rules are mutually exclusive.
- There is no point in trying others as soon as one of them succeed.
- CUT (“ ! “) tells Prolog not to backtrack over this point!



```
f( X, 0)  :-  X < 3, !.
f( X, 2)  :-  3 =< X, X < 6, !.
f( X, 4)  :-  6 =< X.
```

```
?- f(1, Y), 2 < Y.
```

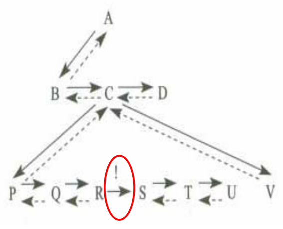
A third version:

```
f(X, 0) :- X < 3, !.
f(X, 2) :- X < 6, !.
f(X, 4).
```

CUT - Example

•Solid arrows indicate the sequence of calls.
•The dashed arrows indicate backtracking.
•Notice that there is “one way traffic” between R and S.

```
C  :-  P, Q, R, !, S, T, U.
C  :-  V.
A  :-  B, C, D.
?-  A.
```



CUT - Example

```
member(X,[X|_]).
member(X,[_|Tail]) :- member(X,Tail).

member(X,[X|_]) :- !.
member(X,[_|Tail]) :- member(X,Tail).
```

How do these two definitions of the member predicate behave differently?

Which one if better? – It depends!

CUT - Example

- Example: Adam and Eve had no parents. Everybody else has two.
```
number_of_parents(adam, 0).
number_of_parents(eve, 0).
number_of_parents(X, 2) :- X \= adam, X \= eve.
```
- Alternative definition:
```
number_of_parents(adam, N) :- !, N = 0.
number_of_parents(eve, N) :- !, N = 0.
number_of_parents(X, 2).
```
- Alternative definition:
```
number_of_parents(adam, 0) :- !.
number_of_parents(eve, 0) :- !.
number_of_parents(X, 2).

? - number_of_parents(eve, X).      ? - number_of_parents( eve, 2).
 X=0.                               Yes
```

Negation as Failure

- How do you say “Mary likes all animals but snakes” in Prolog?
  - If X is a snake then “Mary likes X” is not true.
  - Otherwise if X is an animal then Mary likes X.

```
likes(mary, X) : -
       snake(X), !, fail.
likes(mary, X) :-
       animal(X).
```

Negation as Failure - Example

- We want a predicate consonant/1 which tells us whether the argument is a consonant or not.
- There are around 20 consonants, but only 5 vowels. Everything that is not a vowel is a consonant (we assume)

```
vowel(a).            vowel(o).
vowel(e).            vowel(u).
vowel(i).
```

We could write the predicate consonant/1:
- if the argument is a vowel, fail.
- everything else is a consonant.

```
consonant(X) :- vowel(X), fail.  ┐
consonant(_).                    ┘ ?
```

Negation as Failure - Example

- When the argument is a vowel, then we want the predicate to fail without trying any other way to satisfy the goal - **cut the backtracking**
- **Cut** - **'!'** - built in predicate that always succeeds
- **'!'** has a very important side effect - cuts the backtracking
- Use "!, fail" pair to indicate negation!

```
consonant(X) :- vowel(X),fail.  } ⇒
consonant(_).

consonant(X) :- vowel(X),!,fail.
consonant(_).
```

Closed World Assumption

- Closed World Assumption means that Prolog's world is closed: everything in the universe that is true is provable from the facts and rules in the knowledge base. Everything else is false.
- If Prolog is unable to prove `vowel(X)` for some given `X`, it fails. All objects X for which Prolog cannot prove `vowel(X)` are assumed to not be vowels. Sometimes is easier, in order to prove that something is false, that its opposite is true and then negate it.
- The '`yes`' answer to the query `?- not(human(mary))` should not be interpreted as 'May is not a human'.
  – Prolog really says: there is not enough information to prove that Mary is human.

Common Uses of CUT

- In places where we want to tell the Prolog system that it has found the right rule for a particular goal.
- In places where we want to tell the system to fail a particular goal immediately without trying for alternative solutions.
- In places where we want to terminate the generation of alternative solutions through backtracking (you don't want the alternative solutions if there are any).

CUT – Pros and Cons

- **Advantages**
  – Improve efficiency by cutting backtracking explicitly.
  – the program may occupy less memory space if backtracking points do not have to be recorded for later examination.
  – We can specify mutually exclusive rules.
- **Disadvantages**
  – the program is less readable.
  – you should be careful using it, the program may not behave they way you want it to.
  – Destroying the correspondence between declarative and procedural meanings.

not

- A unary predicate `not` is useful.
  – `not(Goal)` is true if Goal is not true.

```
not(P) :- P, !, fail.
not(P) :- true.
```

- SWI-Prolog implements not/1.
- Now

```
likes(mary, X) :- animal(X), not(snake(X)).
```

- `not` should be used with care.
- `not` does not correspond to negation in mathematical logic.
- Alternatively, we use `\+`.

not - Example

- A letter is a consonant if it is not a vowel:

```
vowel(a).
vowel(e).
vowel(i).
vowel(o).
vowel(u).

consonant(X) :- not(vowel(X)).
```