Simple Data structure and computation

Solving equation?

sum(X,Y,Z):- Z is X+Y.

?sum(1,R,7).

is/2: Arguments are not sufficiently instantiated

^ Exception: (8) 8 is 2+_G254 ?

How to make it work.

```
sum(X,Y,Z):- nonvar(X), nonvar(Y), nonvar(Z), !, Z is X+Y.
sum(X,Y,Z):- nonvar(X), nonvar(Y), var(Z), Z is X+Y, !.
sum(X,Y,Z):- var(X), nonvar(Y), nonvar(Z), X is Z-Y, !.
sum(X,Y,Z):- nonvar(X), var(Y), nonvar(Z), Y is Z-X, !.
sum(X,Y,Z):- write('cannot do the operation'), fail.
```

Try it with decimal numbers

```
close_enough(X,X):-!.
close_enough(X,Y):-X<Y, Y-X< 0.0001.
close_enough(X,Y):-Y<X, close_enough(Y,X).
```

```
=:=
=\=
```

Simple data structure example: List

- [a,b,c]
- [a|[b,c]]
- [a,b|[c]]
- [a,b,c|[]]

These all mean the same

head tail

Head is the first element of the list
Tail is the list containing all
Elements except the first

Using List: example

member(X,[X|T]).
member(X,[H|T]):- X\==H, member(X,T).

If we ask ?- member(2,[1,2,3]).

- Prolog will first try to match our question with the first rule, but fails since 2 is not 1
- Then it tries the second rule (substituting X=2, H=1), which then leaves us with the second clause in this second rule, member(2,[2,3]).
- Solving member(2,[2,3]). We use our first rule again. This time, it matches the first rule.
- So the answer is “yes”

```
member(X, [X|T]).
member(X, [H|T]):- X \== H, member(X, T).
```

X H

```
?- member(X,[1,2,3]).
```

2

X=1;

X=2;

X=3;

no.

X \== H

member(2, [2,3]).

Example: testing if elements in the list are sorted

```
sorted([]).  % empty list is already sorted
sorted([X]).  % list with one element is surely sorted
sorted([A,B|T]):- A=<B, sorted([B|T]).
```

If we ask ?- sorted([1,3,2]).

- It will try to use our first and second rules, and fails
- When it tries the third rule, it tries substituting A=1, B=3
  - 1=<3, this first condition matches ok
  - So we are left to do sorted([3,2]).
- Again, it will try to match and only the third rule can match, with new A = 3, and new B = 2
  - 3=<2 is false
  - It will try to go back to use other substitution, but there is no other choice
- The answer is therefore "no"

Example: deleting a first occurrence of an element from a list

```
del(X,[],[]).
del(X,[X|T],T).
del(X,[H|T],[H|T2]) :- X\==H, del(X,T,T2).

?-del(3,[1,2,3],[1|[2]]).
del(3,[2,3],[2|[]]).
del(3,[3],[]).
```

Length of a list- 2 ways to write

```
length([], 0).
length([H|T], N) :- length(T,Nt), N is Nt+1.
```

Or

```
length(L,N):- accumulate(L,0,N).
accumulate([],A,A).
accumulate([H|T],A,N):- A1 is A+1,
   accumulate (T,A1,N).
```

A1 can be found easily, so put it first. Otherwise there will be infinite recursive.

What are the results?

- ?-length([apple, pear], N).
- ?-length(L,3).
- ?-length([alpha],2).

- Homework
  - Write a program that calculates the sum of all integers in a given list.

Ordering Prolog to fail: list example

- Say we want to print all possibilities of 2 lists that can append to form a new list
- Let us have the following definition for append

```
append([],A,A).
append([H|T],A,[H|L]):- append(T,A,L).
```

- And the following definition for printing

  print_partition(L):-append(A,B,L), write(A),
  write(' '), write(B), nl, fail.

  New line

  Force it to fail

- when fail, Prolog will go back and try to substitute other possible values for A and B.
- Therefore we will get the printout of all possible result, and Prolog will say 'no' in the end.

```
?-print_partition([a,b,c]).
[][a,b,c]
[a][b,c]
[a,b][c]
[a,b,c][]
no
```

Example: Inner Product

- Dot product of vector a and b is

$$\sum_{i=1}^{n} a_i \cdot b_i$$

It only works for 2 vectors of the same length.

Let inner(V1,V2,P) be a goal that succeeds for list V1 and V2. The result dot product is P.

Inner Product- two ways

```
inner([],[],0).
inner([A|As],[B|Bs],N):- inner(As,Bs,Ns), N is Ns
    +(A*B).
```

Or

What is the result?

- If the length of the two lists are different?

Example- Maximum of a list

```
max([],A,A).
```

```
maximum(L,M):-max(L,-10000,M).
```

This is not very good, we should use the first value in the list rather than -10000.

Homework

- Define maximum that initialises the accumulator from the first element of the input list.
- What is the result of the following goals?

  ?-max([3,1,4,1,5,8,2,6],0,N).

  ?-max([2,4,7,7,7,2,1,6],5,N).
- Define minmax which finds both the minimum and maximum values in a list.

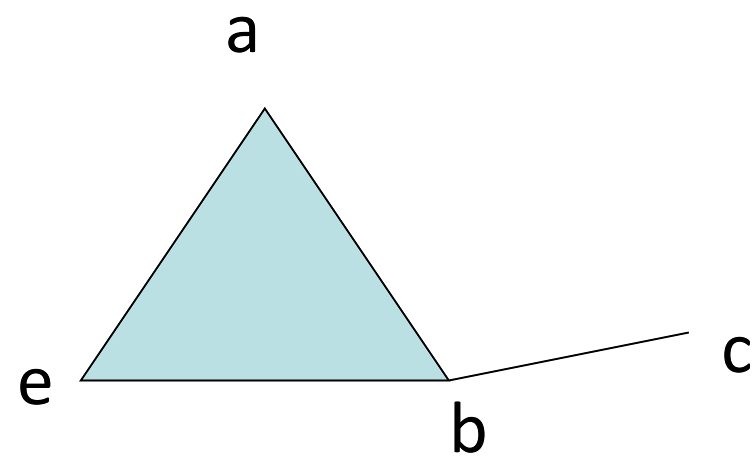
A subgoal should look like
…,minmax(L,MinVal,MaxVal),…

The path problem

- We will look at the definition of path finding problem in a graph.
  - We will see examples on what happens if clauses are switching their orders.
- We will then see how to solve it with the help of list.

- Let’s say we have the following definition

```
edge(a,b).
edge(b,c).
edge(b,e).
edge(a,e).
edge(X,Y):-edge(Y,X).
path(X,X).
path(X,Y):- X\==Y, edge(Z,Y), path(X,Z).
```



We ask ?-path(a,c).

- It will start trying the rule for path. The rule that will match is the second,with X=a and Y = c, now we do the 3 conditions of the rule
  - a\==c. This is ok
  - edge(Z,Y), where Y is c, the fact that will first match is edge(b,c). So Z is substituted by b.
  - path(X,Z). will now be path(a,b)
- So we do path(a,b). Again, it will go down to the rule with 3 conditions (with X2=a, Y2=b)
  - a\==b. This is ok
  - edge(Z2,Y2), where Y2=b. The first match will be edge(a,b). So Z2 is substituted by a.
  - path(X2,Z2). will now be path(a,a). → this matches the first rule of path.
- All the matches are done without any problems, the answer is "yes"
- But if we change the order within our rules, problem can take place.

Bad rule 1

```
edge(a,b).
edge(b,c).
edge(b,e).
edge(a,e).
edge(X,Y):-edge(Y,X).
path(X,X).
path(X,Y):- X\==Y, path(X,Z), edge(Z,Y).
```

Order has changed

- What will happen when we ask ?- path(a,c).

- It will start trying the rule for path. The rule that will match is the second, with X=a and Y = c, now we do the 3 conditions of the rule
  - a\==c. This is ok
  - path(X,Z). This will now be path(a,Z). We will have to use the path rule again at this stage to solve this.
    - The rule that match is path(a,a), so Z = a
  - edge(Z,Y). Now we know that Z=a and Y=c, so this is edge(a,c). False.
    - We have to go back to change the last substitution (Z=a) in the second condition.
  - Back in the second condition. Use path rule again, now we need the rule with 3 conditions.
    - a\==Z, let us substitute Z = b. (b is the second atom we can find)
    - path(a,Z2). Yes, this will need the path rule again.

      path(a,a). Therefore Z2=a.
    - edge(Z2,Z). will then become edge(a,b) → true
  - Do the original third condition again. This is edge(Z,Y). With the substitution, it is now edge(b,c), which is true.
  - So the answer is "yes"
  - Although it can find the answer, arranging the rules this way will cause unnecessary computation.

- Note:
  - It is best to use fact to eliminate unwanted computation as soon as possible.

Bad rule 2

- Say, for the path definition, we have

path(X,Y):- X\==Y, path(X,Z), edge(Z,Y).

path(X,X).

- Now we ask ?-path(a,c). It will have to use the long rule first, with X=a and Y=c
  - a\==c, this is ok
  - path(a,Z), we will have to apply path rule again for this, with X2=a, Y2=Z
    - a\==Z, the program need to choose something, let Z= b
    - path(a,Z2), we will have to apply the path rule again
      - a\==Z2, the program has to choose a substitution, let Z2 = b
      - path(a,Z3).

Infinite execution

The path program path(e,c) can loop forever.

A way to prevent loops is to keep a trail of what we have visited so far. And then visit only nodes that are not on the trail.

path(X,Y,T) is true if there is a legal path from X to Y, without passing through any nodes list in T.

searching a cyclic graph(cont)

edge

```
path(X,X,T).
path(X,Y,T):-a(X,Z), legal(Z,T),
   path(Z,Y,[Z|T]).
```

A trail is a kind of accumulator

```
legal(Z,[]).
legal(Z,[H|T]):-Z\==H, legal(Z,T).
```

homework

a(g,h).
a(d,a).
a(g,d).
a(e,d).
a(h,f).
a(e,f).
a(a,e).
a(a,b).
a(b,f).
a(b,c).
a(f,c).

- Using facts on the left:
- What are the answers from:

  ?-path(g,c,[])

  ?-path(g,c,[f])

  ?-path(a,X,[f,d])

Mapping

- Mapping the input list to the output list = produce an output list whose elements are transformations of corresponding elements of the input list.
- Full map example
  - [1,2,3,4] -> [2,4,6,8]
- Partial map example
  - [57,-2,34,-21] -> [34]   only positive even elements

- Multiple maps
  - [57,-2,34,-21] -> [57,34] and [-2, -21]
  - Can be disjoint or non disjoint
- Sequential map
  - For ordered data, the state variable determining a particular output value depends only on input values previous in the sequence
  - [a,a,f,3,3,3,w,f,f,f,f,3,3] ->[2*a,1*f,3*3,1*w,4*f,2*3]
  - Can create input list from output list

- Scattered map
  - The output value can depend on any of the input values.
  - [a,a,f,3,3,3,w,f,f,f,f,3,3] ->[2*a,5*f,5*3,1*w]
  - It’s a frequency map
  - Does not preserve order information from the input list.

Full map example

- Maps a list of integers to their squares.

sqlist([],[]).

- Map each integer to a compound term s(X,Y), where Y is the square of X.

sqterm([],[]).

The general scheme for full map

```
fullmap([],[]).
fullmap([X|T],[Y|L]):- transform(X,Y),
    fullmap(T,L).
```

homework

- See

```
envelope([],[]).
envelope([X|T],[container(X)|L]):-
    envelope(T,L).
```

What does the goal envelope([apple, peach,cat,37, john], X) do?

Multiple Choice example

- Try to do the square map again, but this time let any non integer map to itself.

```
squint([],[]).
squint([X|T],[Y|L]):- integer(X), Y is X*X,
    squint(T,L).
```

The problem is   if integer(X) fails, the whole thing will fail.

So we need another clause:

```
squint([],[]).
squint([X|T],[Y|L]):- integer(X), Y is X*X,
    squint(T,L).
squint([X|T],[X|L]):-squint(T,L).
```

But there is still a problem

- The third clause can be chosen to match any input if we fail the second clause. But the third clause should not be allowed if X is integer.
  - squint([2],[2]) returns true?????
- We need a way to commit to the first rule and only come to the second rule if necessary. -> next chapter
- Otherwise, need a “NOT” in order to make the cases mutually exclusive.

homework

- Find all solutions to

?-squint([1,3,w,5,goat],X).

Determine which clause choices were made to give each solution. Draw picture or tree to show.

Partial map example

- Map to even integers

```
evens([],[]).
evens([X|T],[X|L]):- 0 is X mod 2, evens(T,L).
evens([X|T],L):- 1 is X mod 2, evens(T,L).
```

homework

given

prohibit(bother).

prohibit(blast).

prohibit(drat).

Define censor(X,Y) whichs maps the input list of words to the output list of words. No prohibited words appear.

Removing duplicate from a list: example

setify([],[]).

Again, the third clause can be used to do the matching at any time, producing wrong answers. We need some commitment notation or mutually exclusive condition.

homework

- why

?- setify([a,a,b,c,b],X). succeeds

?- setify([a,a,b,c,b],[a,c,b]). succeeds

But ?- setify([a,a,b,c,b],[a,b,c]). does not succeed. Show how their executions go.

Path problem again

- We can have all the nodes in the list first.
- As a node is visited, it is struck off the list.
- The reduced list is then given to the recursive call.

reduce(L,X,M) succeeds for list L, term X, and output list M. M contains elements of L except the first occurrence of X.

```
reduce([X|T], X, T).
reduce([H|T], X, [H|L]):- H\==X,
    reduce(T,X,L).

path(X,X,L).
path(X,Y,L):-a(X,Z), reduce(L,Z,L1),
    path(Z,Y,L1).
```

Used-> ?-path(a,b,[a,b,c,d,e,f,g,h])

Multiple disjoint partial maps

goal herd(L,S,G) succeeds if S is a list of all sheep in L and G is a list of all goats in L.

herd([],[],[]).