1 Dealing with NOT

In one of the examples we looked at :

```
nice :-
  sunny,
  not_windy.
```

- This is *really* cheating
- How do we represent the **not** case
- Use the *built in* procedure *not* with one argument (not/1)
- So the goal, *not(windy)* succeeds if the goal *windy* fails
- It is possible that the goal is *composed* of *sub-goals*, for instance :
  not((windy, cold;hot)).
  although the arity is still 1 : goals are combined using *and*, *or* primitives
- *Not* : implements negation as the failure of proving that the goal is true
- Prolog assumes that everything defined in a program, or what can be derived from it is true : else : everything false - this is the *closed world assumption.* Hence,
  not(clever(user))
  it answers *yes* - even though there is no rule about the user

If I have the simple database of facts :

```
tel(james, 465676).
tel(mark, 332312).
tel(helen, 766786).
```

Then we can pose questions (queries) to Prolog in a number of different ways :

```
tel(mark, N)
N = 332312
yes

tel(helen, 766786)
yes

tel(Name, 766786)
Name = helen
yes

tel(Name,Num)
Name=james
Num=465676 ;
Name=mark
Num= 332312;
...
no
```

Hence, the idea of obtaining information, using the *reverse* way. For instance, in arithmetic :

```
X is 2 + 18 / 5
X = 5.6
cannot say
5.6 is A + B / C

however
X = 2 + 18 / 5
and
A + B / C = 2 + 5 / 6
A = 2
B = 5
C = 6
```

Similarly, the use of the predicate *not* also prevents using procedures in different ways. We can, for instance, use the procedure

```
has_no_children(X) :-
  not( child(_,X)).
```

only to check whether a certain person has no children, but cannot use it to find these people. An additional condition has to be introduced into the rule, to force X to be instantiated before executing the *not(child(_,X))* goal. Hence :

```
has_no_children(X) :-
  person(X),
  not( child( _,X)).
```

- In this case, the order of the rules is also important

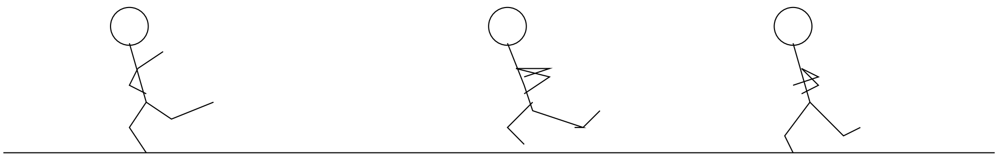

Figure 1: A track event

- Want to define a relation *better*, for which, *better(X, Y)* is true if X finished the race before Y
- X is also better than Y, if X reached the winning post before Z, and Z reached the winning post before Y
- Could be numerous others between X and Y
- Need a more general model
- Need to consider two main cases

```
better(X,Y) :-
  before(X,Y).

better(X,Y) :-
  before(X,Z),
  better(Z,Y).

before(tom, tim).
before(tim, jack).
before(jack, jim).
before( jim,john).
```

- *Recursive* procedure - with a *base case* and a *general definition*

2 Recursive Arithmetic

Consider the definition of *check numbers* from a previous lecture. Another important definition, is of the *factorial* function :

```
0! = 1
n! = n*(n-1)!

factorial(0,1).

factorial(N,F) :-
   N > 0,
   N1 is N-1,
   factorial(N1, F1),
   F is N * F1.
```

- Again, the *boundary* condition is given first, followed by the definition of the rule
- The first check is on the input arguments, and validates the calling arguments
- the repeated call to factorial is then made with a slightly smaller parameter

Possible to extend the definition of our racers with the use of a third argument, which encapsulates time - therefore

```
better(X,Y,T) :-
  before(X,Y,T).

better(X,Y,T) :-
 before(X, Z, T1),
 better(Z, Y, T2),
 T is T1 + T2.
```

- Ordering in the 3rd goal above is important, because T cannot be evaluated before T1 and T2 have been instantiated
- We can also perform other arithmetic operations to achieve more statistics about the race, done in the same way as the second definition of better

3 Back to Lists

- Lists are a recursive data structures
- Since in a list, the composition is of a head and a tail, where the tail itself could be a list - hence the concept of recursion comes into the definition
- *Hence* : Most operations on lists are defined recursively
- These include definitions already considered : *member* to list manipulation operations like *concatenating lists*,*deleting elements from lists* etc.

Membership again :

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

- X is a member of a list, if X is the head of the list
- X is a member of the tail
- Boundary condition is placed first - and satisfied when X is the first element (head) of the list
- The *recursive call* in the second clause searches for an element in the tail of the list that is shorter than the original list. The *recursive call* successively reduces the size of the list by comparing the head of the list with X, then removing the head, and using the remainder, and taking the first element of the remainder as the new head, comparing with X and so on. This process stops when we are left with tail being the empty list

Concatenating Lists (combining two lists together into one) :

```
conc( [], L, L).

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

- L3 is obtained by concatenating L1 and L2 together.
- It works by first stating the *base* case : i.e. joining an empty list to any list is the list itself
- A list obtained by concatenating a list L2 with a list whose head is X and tail L1, is a list whose head is X and tail composed of concatenating L1 and L2.
- Hence, the new formed list, L3, will also have X as its head

Issuing the command :

```
conc( [1,2], [3,4], [1,2,3,4])
```

will return yes

Similarly, a query :

```
conc( [1,2], [3,4], L)
```

will return L = [1,2,3,4]

4 Using Conc for more list functions

We can write a procedure for deleting elements from within a list

```
delete( X, List, Result) :-
 conc( L1, [X| L2], List),
 conc( L1, L2, Result).
```

- To delete X from list, X should be found as the head of the second (or right) part of the List
- This partitions the list into L1 and L2
- Result is then obtained by removing X, and concatenating the remained of L2 to L1

Sample query :

```
delete(2, [1,2,3,4], Result)
Result = [1,3,4]
```

Similarly, one can insert elements into the list :

```
insert(X, List, Result) :-
  delete(X, Result, List).
```

How does this work ? Try developing the definition in terms of the conc definition earlier

It is also possible to look for an arbitrary *sublist* within the list, this can again be achieved by using the *conc* definition earlier. Hence :

```
sublist(List, Sublist) :-
 conc(_, L1, List),
 conc( Sublist, _, L1).
```

A query of the form :

```
sublist([1,2,3,4,5,6],[3,4,5])
yes
```

- Sublist works by assigning L1 to be Sublist and *something else*, note the use of the *anonymous variable)*
- *List is then constructed by combining L1 with some something else as the left part*
- *Sublist is therefore the left part of some right segment of List*

5 Lists - Use

- Prolog works by *Matching* and *Searching*. This works well with processes like backtracking and recursion that we have looked at
- A data structure within Prolog which can best exploit these is *Lists*
- Lists can be used for the representation of many things : from matrices to compound structures we encountered earlier
- Learn to get used to thinking in *definitions* and *Lists* to use Prolog efficiently

6 Searching Graphs

Consider the following :

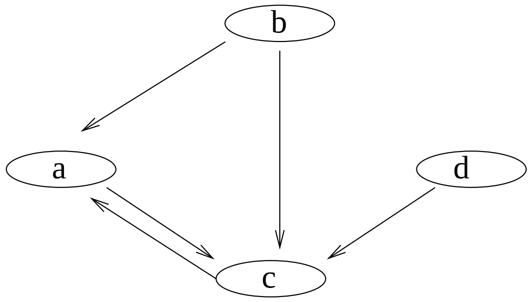

Figure 2: A graph

- The graph is obtained by linking *nodes* via *edges*
- A *directed* graph is one where the edges also have direction (usually called arcs)

The graph above can be described, by simply stating the interconnections :

```
arc(a,c).
arc(a,b).
arc(c,a).
arc(c,b).
arc(c,d).
```

Also, we can define a simple procedure which evaluates to true if a path between two given nodes can be found :

```
path(X,Y) :-
 arc(X,Y).

path(X,Y) :-
 arc(X,Z),
 path(X,Z).
```

- What does this definition tell us?
- First rule says that a path between X and Y exists, if an arc can be found between X and Y in the database
- Second rule says that a path exists between X and Y, if a *series* of arcs (intermediate nodes Z) can be obtained from X to Y
- *BUT*

If issued the query :

```
path(a,d)
```

- Prolog goes into an infinite loop, because of the presence of the cycle between a and c, hence we have to change our definition to overcome this problem
- So how do we do this ?

```
path(X,Y) :-
 path(X,Y, []).

path(X,Y,_) :-
 arc(X,Y).

path(X,Y,Trail) :-
 arc(X,Z),
 not(member(Z,Trail)),
 path(Z,Y,[X|Trail]).
```

what does this do ?

- In the first case X is directly connected to Y by a *single* arc - so the *Trail* list is empty

- This is defined by the first two rules. The use of the anonymous variable suggests that whatever the Trail, if a direct arc exists - thats fine
- Third rule uses an indirection, by considering an intermediate node, labelled Z - and tries to determine if Z has been examined already or not
- Notice the use of *member* definition along with a list definition. The last *path* statement is used to say that X has already been examined
- This prevents the occurrence of cycles and overcomes the problems with the first definition

More complex arithmetic manipulation of lists via recursion

```
squares([],[]).

squares([ N| Tail],[S | Stail]) :-
 S is N*N,
 squares(Tail, Stail).
```

- A procedure for performing a transformation on all elements of a list
- Gives the square of numbers in a given list
- First rule : square of an empty list is an empty list
- Second rule : Input composed of two lists, with each list composed of a head (N for first list and S for second list), and tail (Tail for first list, and Stail for second list)
- Take the first element of first list and square it

- First element of second list becomes that result
- Remove the first element from both lists (as that has now been squared), and perform the same on the remainder of the list i.e. the tail
- Repeat the process until an empty list is obtained