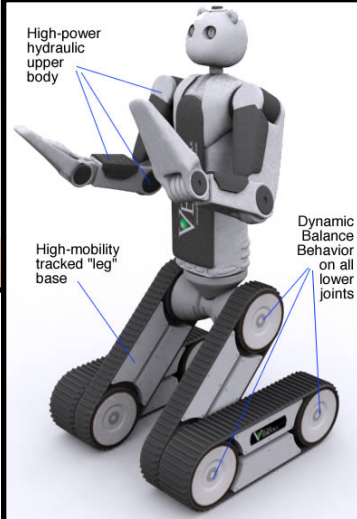
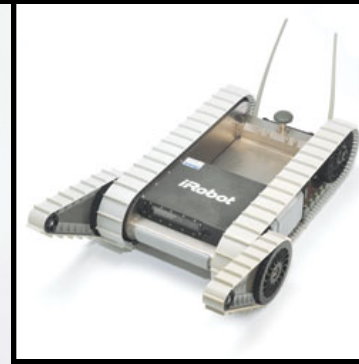
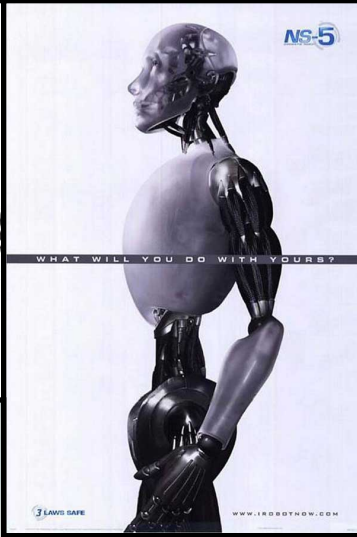


Dr. John D. Kelleher
john.d.kelleher@dit.ie



Introduction to Recursion in Prolog

- Tutorial Goals
 - Introduce recursive definitions in Prolog
 - Show that there can be mismatches between the declarative and procedural meaning of a Prolog program

A predicate is recursively defined if one or more rules in its definition refers to itself

Example 1: Eating

```
justAte(frog,mosquito).
```

```
justAte(stork,frog).
```

```
isDigesting(X,Y):- justAte(X,Y).
```

```
isDigesting(X,Y):- justAte(X,Z), isDigesting(Z,Y).
```

Example 1: Eating (Pictorally)

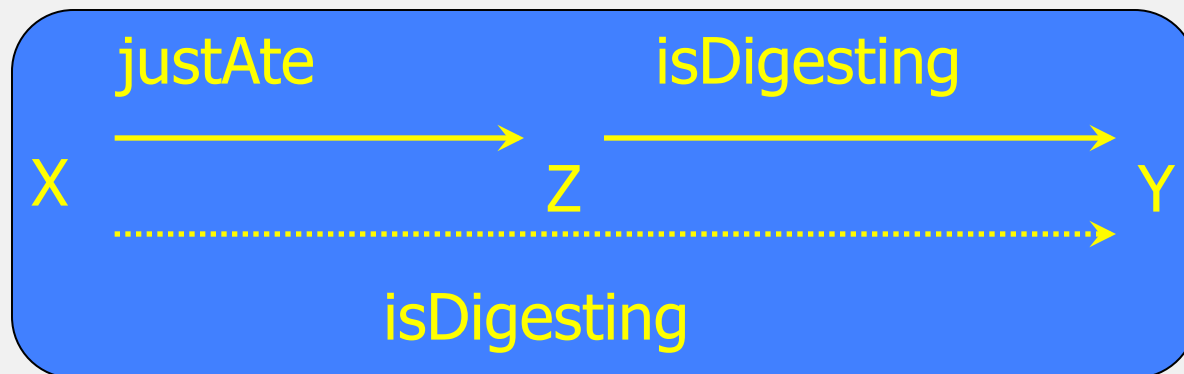
`justAte(frog,mosquito).`

`justAte(stork,frog).`

`isDigesting(X,Y):- justAte(X,Y).`



`isDigesting(X,Y):- justAte(X,Z), isDigesting(Z,Y).`



Example 1: Eating

```
isDigesting(X,Y):- justAte(X,Y).  
isDigesting(X,Y):- justAte(X,Z), isDigesting(Z,Y).  
  
justAte(frog,mosquito).  
justAte(stork,frog).
```

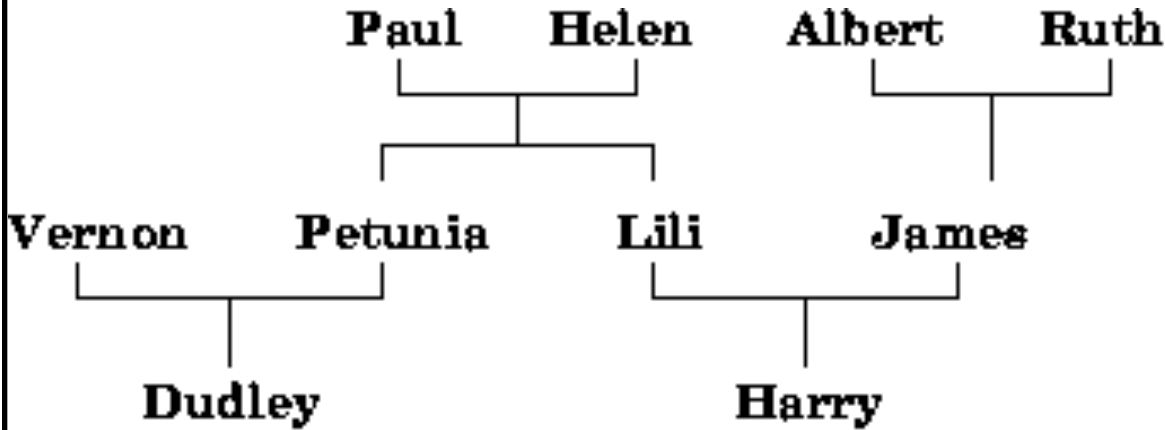
```
?- isDigesting(stork,mosquito).
```

Example 1: Eating

```
isDigesting(X,Y):- justAte(X,Y).  
isDigesting(X,Y):- justAte(X,Z), isDigesting(Z,Y).
```

```
justAte(frog,mosquito).  
justAte(stork,frog).
```

```
?- isDigesting(stork,mosquito).  
true.  
?-
```



```
parent_of(paul,petunia).  
parent_of(helen,petunia).  
parent_of(paul,lili).  
parent_of(helen,lili).  
parent_of(albert,james).  
parent_of(ruth,james).  
parent_of(petunia,dudley).  
parent_of(vernion,dudley).  
parent_of(lili,harry).  
parent_of(james,harry).
```

Task: Define a predicate **ancestor_of(X,Y)** which is true if **X** is an ancestor of **Y**

grandparent_of(X,Y) :- parent_of(X,Z), parent_of(Z,Y).

greatgrandparent_of(X,Y) :- parent_of(X,Z), parent_of(Z,A), parent_of(A,Y).

greatgreatgrandparent_of(X,Y) :- parent_of(X,Z), parent_of(Z,A),
parent_of(A,B), parent_of(B,Y).

Doesn't work for ancestor_of; don't know "how many parents we have to go back".

ancestor_of(X,Y) :- parent_of(X,Y).

People are ancestors of their children,

ancestor_of(X,Y) :- parent_of(X,Z), ancestor_of(Z,Y).

and they are ancestors of anybody that their children may be an ancestor of (i.e., of all the descendants of their children).

ancestor_of(X,Y)

recursion

People are ancestors of their children,

ancestor_of(X,Y) :- parent_of(X,Z), ancestor_of(Z,Y).

and they are ancestors of anybody that their children may be an ancestor of (i.e., of all the descendants of their children).

Roughly speaking, a predicate is recursively defined if one or more rules in its definition refers to itself.

In C we deal with situations which require iteration by means of constructs like *while*, *do*, *for* and so on.

Prolog does not use these imperative-style constructs: instead, when we need to iterate, we use *recursion*.

Basically recursion involves **defining something in terms of itself**.

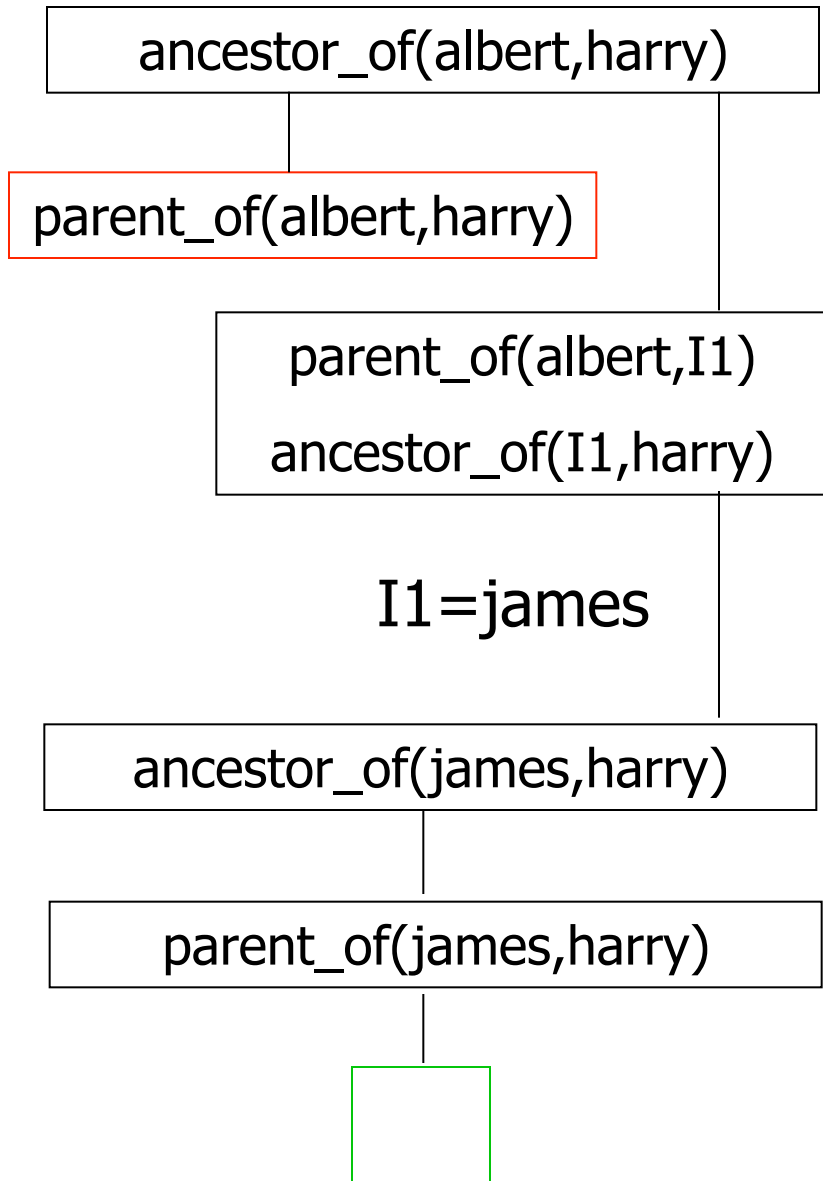
The key to ensuring that this makes sense is that you always define something in terms of a *smaller* copy of itself.

When you do recursion you must have three things:

1. Some set (or "data structure") over which you are doing the recursion: i.e., our KB.
2. A **base case** definition, usually dealing with an empty structure
3. A **recursive case** definition, explaining how to work out a non-trivial case in terms of some smaller version of itself.

Ancestors (cont.)

```
parent_of(paul,petunia).  
parent_of(helen,petunia).  
parent_of(paul,lili).  
parent_of(helen,lili).  
parent_of(albert,james).  
parent_of(ruth,james).  
parent_of(petunia,dudley).  
parent_of(vernion,dudley).  
parent_of(lili,harry).  
parent_of(james,harry).  
ancestor_of(X,Y) :- parent_of(X,Y).  
ancestor_of(X,Y) :- parent_of(X,Z),  
                        ancestor_of(Z,Y).
```



Ensuring Recursion Terminates

Every recursive predicate should always have at least two clauses:

- a base clause (the clause that stops the recursion at some point)
- and one that contains the recursion.

If you don't do this Prolog can spiral off into an unending sequence of useless computations.

Ensuring Recursion Terminates: Example

Here's an extremely simple example of a recursive rule definition:

$p:-p$

Declaratively this states that 'if property p holds, then property p holds'. Seems reasonable!

From a procedural perspective this is a very dangerous rule.

Ensuring Recursion Terminates: Example

What happens when we pose the following query:

`?-p`

Prolog asks itself: 'how do I prove p?' And it realises, 'Hey, I've got a rule for that! To prove p I just need to prove p!'. So it asks itself (again): 'how do I prove p?' And it realises, 'Hey,'

If you make this query, Prolog won't answer you: it will head off, looping desparately away in an undending search. That is, it won't terminate!

Another recursive definition

p:- p.

?-

Another recursive definition

$p:- p.$

$?- p.$

Another recursive definition

p:- p.

?- p.
ERROR: out of memory

Prolog has a very specific way of working out the answer to queries: it searches the knowledge base from **top to bottom** , **clauses from left to right**, and **uses backtracking to recover from bad choices**.

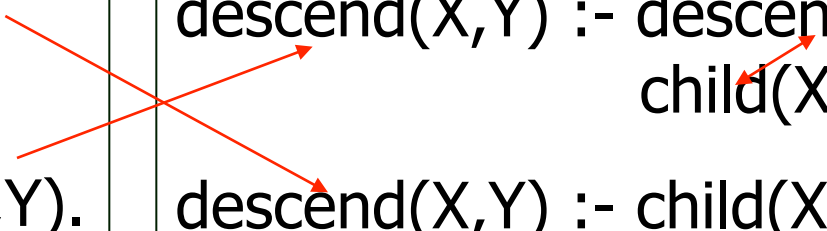
These procedural aspects have an important influence on what actually happens when you make a query.

Be careful, **it is easy to define two KBs with the same declarative meaning but different procedural meanings**.

Clause ordering, goal ordering and termination

```
child(martha, charlotte).  
child(charlotte, caroline).  
child(caroline, laura).  
descend(X,Y) :- child(X,Y).  
descend(X,Y) :- child(X,Z),  
                  descend(Z,Y).
```

```
child(martha, charlotte).  
child(charlotte, caroline).  
child(caroline, laura).  
descend(X,Y) :- descend(Z,Y),  
                  child(X,Z).  
descend(X,Y) :- child(X,Y).
```

A diagram consisting of two red arrows. One arrow originates from the recursive goal 'descend(Z,Y)' in the second rule of the left column and points to the recursive goal 'descend(Z,Y)' in the second rule of the right column. The other arrow originates from the base goal 'child(X,Z)' in the second rule of the left column and points to the base goal 'child(X,Z)' in the second rule of the right column.

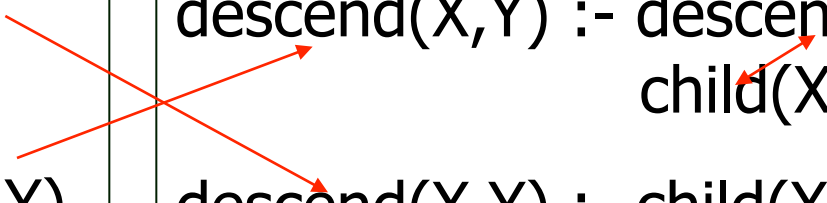
From a declarative perspective the difference between these two KBs is very simple: we have merely reversed the order of two rules, and reversed the order of the two goals in the recursive clause.

So from a purely logical perspective nothing has changed.

Clause ordering, goal ordering and termination

```
child(martha, charlotte).  
child(charlotte, caroline).  
child(caroline, laura).  
descend(X,Y) :- child(X,Y).  
descend(X,Y) :- child(X,Z),  
                  descend(Z,Y).
```

```
child(martha, charlotte).  
child(charlotte, caroline).  
child(caroline, laura).  
descend(X,Y) :- descend(Z,Y),  
                  child(X,Z).  
descend(X,Y) :- child(X,Y).
```

A diagram with two red arrows. One arrow starts from the second clause of the left box, 'descend(X,Y) :- child(X,Z), descend(Z,Y).', and points to the first clause of the right box, 'descend(X,Y) :- descend(Z,Y), child(X,Z).'. The other arrow starts from the first clause of the left box, 'descend(X,Y) :- child(X,Y).', and points to the third clause of the right box, 'descend(X,Y) :- child(X,Y).'. This illustrates how the order of clauses in the right-hand knowledge base is different from the left-hand one.

But from the procedural perspective the meaning has changed considerably!
For example, using the KB on the right if you pose the query:

?-descend(martha, rose).

You will get an error message ('out of local stack', or something similar.
Prolog is looping!

Clause ordering, goal ordering and termination

What causes the loop?

Well, to satisfy `descend(martha,rose)`. Prolog uses the first rule. This means that its next goal will be to satisfy the query:

`descend(W1,rose)`

for some new variable W1. But to satisfy this new goal, Prolog again has to use the first rule, and this means that its next goal is going to be

`descend(W2,rose)`

for some variable W2 ...

```
child(martha, charlotte).  
child(charlotte, caroline).  
child(caroline, laura).  
descend(X,Y) :- descend(Z,Y),  
                child(X,Z).  
descend(X,Y) :- child(X,Y).
```


Because the declarative and procedural meanings of a Prolog program can differ, when writing Prolog programs you need to bear both aspects in mind.

Often you can get the overall idea of how to write the program by thinking declaratively, that is, by thinking simply in terms of describing the problem accurately.

However, you also need to think about how Prolog will actually evaluate queries. Are the rule orderings sensible? How will the program actually run?

Summary of this lecture

- In this lecture we introduced recursive predicates
- We also looked at the differences between the declarative and the procedural meaning of Prolog programs