Tutorial SHEET-3

Recursion

1. Recursive Definition

Predicates can be defined recursively. A predicate is recursively defined if one or more rules in its definition refer to itself.

1.1. Example 1

Consider the following knowledge base:

```
is_digesting(X,Y) :- just_ate(X,Y).
is_digesting(X,Y) :-
          just_ate(X,Z),
          is_digesting(Z,Y).
just_ate(mosquito,blood(john)).
just_ate(frog,mosquito).
just_ate(stork,frog).
```

Here ,the definition of the `is_digesting/2` predicate is recursive. Note that `is_digesting` is (at least partially) defined in terms of itself, for the `is_digesting` functor occurs on both the left and right hand sides of the second rule. Crucially, however, there is an **'escape'** from this circularity. This is provided by the `just_ate` predicate, which occurs in both the first and second rules.

The **declarative** meaning (logical meaning of Prolog knowledge base) of this recursive definition is : the first clause (the 'escape' clause/ non-recursive/ **base** clause), simply says that: *if* `X` has just eaten `Y`, *then* `X` is now digesting `Y`. The second clause (**recursive** clause) This says that: *if* `X` has just eaten `Z` and `Z` is digesting `Y`, *then* `X` is digesting `Y`, too.

The **procedural** meaning (how the recursive definition is used) is: for any query asking whether `X` has just eaten `Y`, Prolog first checks whether the base clause applies. If it does not apply, the recursive clause gives Prolog another strategy

for determining whether X is digesting Y: it can try to find some Z such that X has just eaten Z, and Z is digesting Y. That is, this rule lets Prolog break the task apart into two subtasks. Doing so will eventually lead to simple problems which can be solved by simply looking up the answers in the knowledge base.

Consider the query:

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

then Prolog, first, tries to make use of the first rule listed concerning `is_digesting`; that is, the base rule. This tells it that X is digesting Y if X just ate Y, By unifying X with `stork` and Y with `mosquito` it obtains the following goal:

```
just_ate(stork,mosquito).
```

But the knowledge base doesn't contain the information, so this attempt fails. So Prolog next tries to make use of the second rule. By unifying X with `stork` and Y with `mosquito` it obtains the following goals:

```
just_ate(stork,Z),
is_digesting(Z,mosquito).
```

Now, Prolog needs to find a value for Z such that follows the above two facts. And there is such a value for Z, namely `frog`. It is immediate that

```
just_ate(stork,frog).
```

will succeed, for this fact is listed in the knowledge base. Now it has to deduce

```
is_digesting(frog,mosquito).
```

To deduce so, it again tries to use the first clause of `is_digesting/2`
This reduces this goal to deducing

```
just_ate(frog,mosquito).
```

and this is a fact listed in the knowledge base.

Note that it is *crucial to have a base clause/escape clause* otherwise Prolog runs into an **infinite loop**.

1.2. Example 2

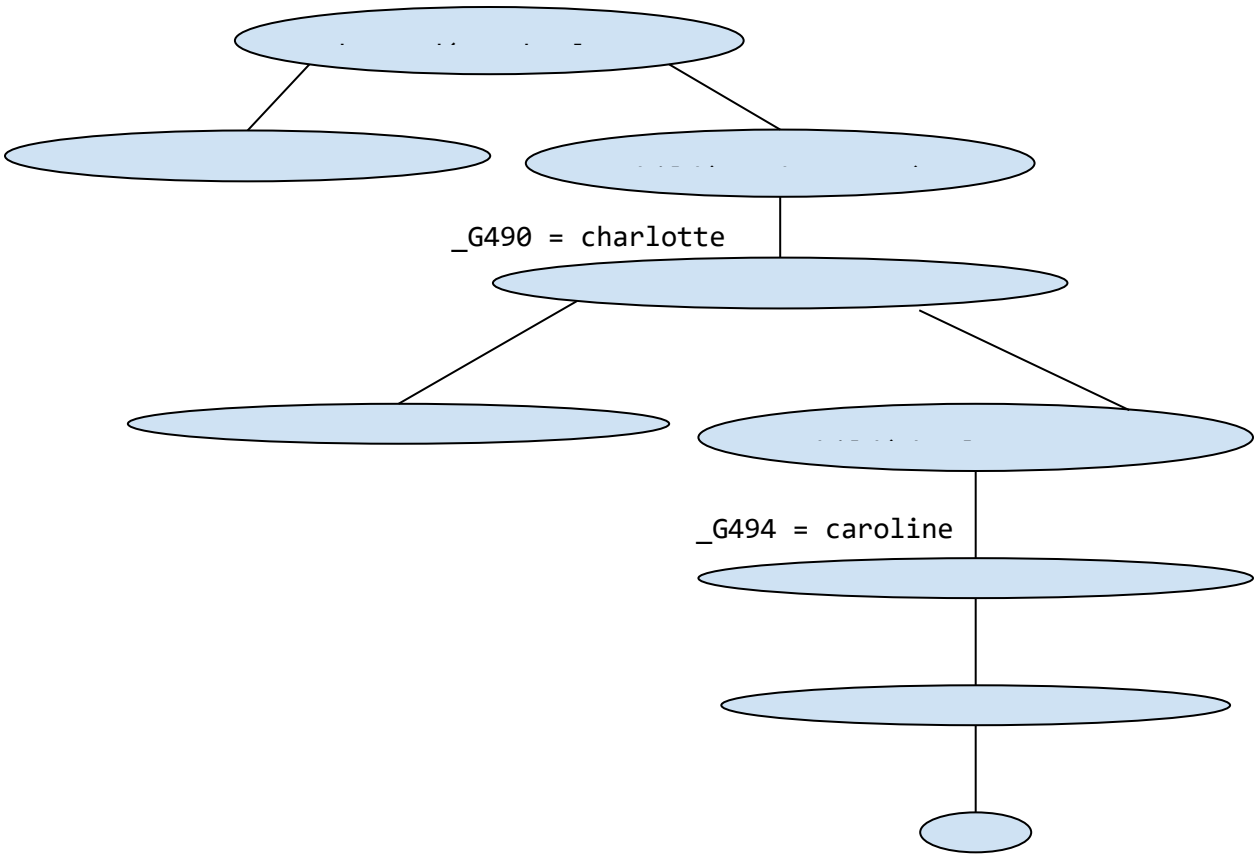
Suppose we have a knowledge base recording facts about the child relation:

```
child(martha,charlotte).
child(charlotte,caroline).
```

```
child(caroline,laura).
child(laura,rose).
```
Suppose we wished to define the descendant relation; that is, the relation of being a child of, or a child of a child of, or a child of a child of a child of, or....
We could add the following two *non-recursive* rules to the knowledge base:
```
descend(X,Y) :- child(X,Y).
descend(X,Y) :- child(X,Z), child(Z,Y).
```

These definitions work up to a point, but they are clearly extremely limited.
The two rules are inadequate. For example, if we pose the queries
```
?- descend(martha,laura).
```
or
```
?- descend(charlotte,rose).
```
we get the answer ‘No!’, which is not what we want. We could add the following rules:
```
descend(X,Y) :- child(X,Z_1), child(Z_1,Z_2), child(Z_2,Y).
descend(X,Y) :-     child(X,Z_1),
                    child(Z_1,Z_2),
                    child(Z_2,Z_3),
                    child(Z_3,Y).
```
But this approach is clumsy and hard to read.
The following recursive rule, however, fixes everything exactly the way we want:
```
descend(X,Y) :- child(X,Y).
descend(X,Y) :- child(X,Z), descend(Z,Y).
```

For example, we pose the query:
```
descend(martha,laura).
```
The search tree for the query would be as follows:



It is obvious from this example that no matter how many generations of children we add, we will always be able to work out the descendant relation. That is, the recursive definition is both general and compact: it contains all the information in the previous rules, and much more besides.

1.3. Example 3

Now we will see examples of *building structures through recursion*.

Nowadays, we usually use decimal notation (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, and so on) to represent numerals.

Here's another way of writing numerals, which is sometimes used in mathematical logic. It makes use of just four symbols: *0, succ, and the left and right brackets*. This style of numeral is defined by the following inductive definition:

1. 0 is a numeral.
2. If X is a numeral, then so is succ(X).

Here *succ* is a short for successor.

Let’s write this as a definition for a Prolog programme:

```
numeral(0).
numeral(succ(X)) :- numeral(X).
```

So if we pose queries like
`numeral(succ(succ(succ(0)))).`
we get the answer ‘yes’. But we can do some more interesting things.

Consider what happens when we pose the following query:
`numeral(X).`
That is, we’re saying ‘Show some numerals’. Then we can have the following dialogue with Prolog:

```
X = 0 ;
X = succ(0) ;
X = succ(succ(0)) ;
X = succ(succ(succ(0))) ;
X = succ(succ(succ(succ(0)))) ;
X = succ(succ(succ(succ(succ(0))))) ;
X = succ(succ(succ(succ(succ(succ(0)))))) ;
X = succ(succ(succ(succ(succ(succ(succ(0))))))) ;
X = succ(succ(succ(succ(succ(succ(succ(succ(0)))))))) ;
X = succ(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))) ;
X = succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))) ;
yes
```

Thus, Prolog is counting by backtracking through the recursive definition, and actually *building numerals* using matching. To properly understand it, try the example with `trace` turned on.

1.4. Example 4

As a final example, let’s see whether we can use the representation of numerals that we introduced in the previous section for doing simple arithmetic. Let’s try to define addition. That is, we want to define a predicate `add/3` which when given two numerals as the first and second argument returns the result of adding them up as its third argument. E.g. Following queries should give results as follows:

```
add(succ(succ(0)),succ(succ(0)),succ(succ(succ(succ(0))))).
yes
?- add(succ(succ(0)),succ(0),Y).
Y = succ(succ(succ(0)))
```

There are two things which are important to notice:

1. Whenever the first argument is 0, the third argument has to be the same as the second argument:
   ?- add(0,succ(succ(0)),Y).
   Y = succ(succ(0))
   ?- add(0,0,Y).
   Y = 0
   This is the case that we want to use for the base clause.
2. Assume that we want to add the two numerals `X` and `Y` (e.g. `succ(succ(succ(0)))` and `succ(succ(0))`) and that `X` is not `0`. Now, if X' is the numeral that has one succ functor less than X (i.e. `succ(succ(0))` in our example) and if we know the result – let's call it Z – of adding X' and Y (namely `succ(succ(succ(succ(0))))`), then it is very easy to compute the result of adding X and Y: we just have to add one succ-functor to Z. This is what we want to express with the recursive clause.

Here is the predicate definition that expresses exactly what we just said:

```
add(0,Y,Y).
add(succ(X),Y,succ(Z)) :- add(X,Y,Z).
```

So, what happens, if we pose the query:

```
add(succ(succ(succ(0))), succ(succ(0)), R).
```

This is a trace for the query:

```
Call: (6) add(succ(succ(succ(0))), succ(succ(0)), R)
Call: (7) add(succ(succ(0)), succ(succ(0)), _G648)
Call: (8) add(succ(0), succ(succ(0)), _G650)
Call: (9) add(0, succ(succ(0)), _G652)
Exit: (9) add(0, succ(succ(0)), succ(succ(0)))
Exit: (8) add(succ(0), succ(succ(0)), succ(succ(succ(0))))
Exit: (7) add(succ(succ(0)), succ(succ(0)),
          succ(succ(succ(succ(0)))))
Exit: (6) add(succ(succ(succ(0))), succ(succ(0)),
          succ(succ(succ(succ(succ(0))))))
```

2. <u>Clause ordering, goal ordering, and termination</u>

Underlying logic programming is a simple (and seductive) vision: the task of the programmer is simply to *describe* problems. The programmer should write down (in the language of logic) a declarative specification (that is: a knowledge base), which describes the situation of interest. The programmer shouldn't have to tell the computer what to do. To get information, he or she simply asks the questions. It's up to the logic programming system to figure out how to get the answer.

But Prolog is *not* a full logic programming language.

Recall Example 2 where we defined the predicate `descend/2`. Let us make some changes in its definition:

```
descend(X,Y) :- descend(Z,Y), child(X,Z).
descend(X,Y) :- child(X,Y).
```

From a declarative perspective, what we have done is very simple: we have merely reversed the order of the two rules, and reversed the order of the two goals in the recursive clause. So, viewed as a purely logical definition, nothing has changed. We have not changed the declarative meaning of the program. But the procedural meaning has changed dramatically. For example, if you pose the query

```
descend(martha,rose).
```

you will get an error message ('out of local stack', or something similar). Prolog is looping.This is because, to satisfy the query `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 new variable `W2`. And of course, this in turn means that its next goal is going to be `descend(W3,rose)`and then `descend(W4,rose)`, and so on.
In short, the two definitions of `descend/2` have the same declarative meaning but different procedural meanings: from a purely logical perspective they are identical, but they behave very differently.

Two more variants for the definition of the predicate are as follows:

```
descend(X,Y) :- child(X,Y).
```

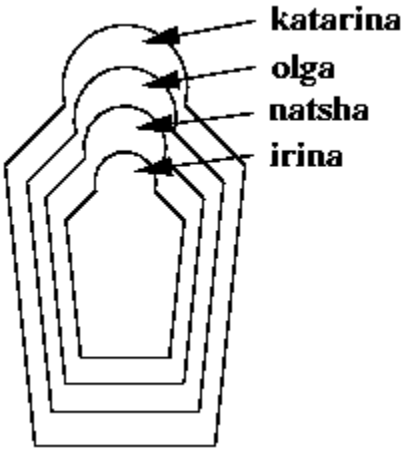
```
descend(X,Y) :- descend(Z,Y), child(X,Z).
```
and
```
descend(X,Y) :- child(X,Z), descend(Z,Y).
descend(X,Y) :- child(X,Y).
```

Can they handle the query `descend(martha,rose)`?
Can they handle queries involving variables? How many steps do they need to find an answer? Are they slower or faster than the original definition?

Thus the declarative and procedural meanings of a Prolog program can differ, when writing Prolog programs you need to bear both aspects in mind. Try drawing the search trees of all the variants.

3. Exercises

1. Do you know these wooden Russian dolls, where smaller ones are contained in bigger ones? Here is a schematic picture of such dolls.



   First, write a knowledge base using the predicate `directlyIn/2` which encodes which doll is directly contained in which other doll. Then, define a (recursive) predicate `in/2`, that tells us which doll is (directly or indirectly) contained in which other doll. E.g. the query `in(katarina,natasha)` should evaluate to true, while `in(olga, katarina)` should fail.

2. Define a predicate `greater_than/2` that takes two numerals in the notation that we introduced above (i.e. 0, succ(0), succ(succ(0)) ...) as

arguments and decides whether the first one is greater than the second one. E.g:

```
?- greater_than(succ(succ(succ(0))),succ(0)).
yes
?- greater_than(succ(succ(0)),succ(succ(succ(0)))).
no
```

3. Binary trees are trees where all internal nodes have exactly two children. The smallest binary trees consist of only one leaf node. We will represent leaf nodes as `leaf(Label)`. For instance, `leaf(3)` and `leaf(7)` are leaf nodes, and therefore small binary trees. Given two binary trees `B1` and `B2` we can combine them into one binary tree using the predicate `tree`: `tree(B1,B2)`. So, from the leaves `leaf(1)` and `leaf(2)` we can build the binary tree `tree(leaf(1), leaf(2))`.
   And from the binary trees `tree(leaf(1), leaf(2))` and `leaf(4)` we can build the binary tree `tree(tree(leaf(1), leaf(2)), leaf(4))`.
   Now, define a predicate `swap/2`, which produces a mirror image of the binary tree that is its first argument. For example:

```
?- swap(tree(tree(leaf(1), leaf(2)), leaf(4)),T).
T = tree(leaf(4), tree(leaf(2), leaf(1))).
yes
```

4. We saw the predicate

```
descend(X,Y) :- child(X,Y).
descend(X,Y) :- child(X,Z), descend(Z,Y).
```

   Could we have formulated this predicate as follows?

```
descend(X,Y) :- child(X,Y).
descend(X,Y) :- descend(X,Z), descend(Z,Y).
```

   Compare the declarative and the procedural meaning of this predicate definition.
   Hint: What happens when you ask the query `descend(rose,martha)`?

5. We have the following knowledge base:

```
directTrain(forbach,saarbruecken).
directTrain(freyming,forbach).
directTrain(fahlquemont,stAvold).
directTrain(stAvold,forbach).
directTrain(saarbruecken,dudweiler).
```

```
directTrain(metz,fahlquemont).
directTrain(nancy,metz).
```
That is, this knowledge base holds facts about towns it is possible to travel between by taking a direct train. But of course, we can travel further by 'chaining together' direct train journeys. Write a recursive predicate `travelBetween/2` that tells us when we can travel by train between two towns. For example, when given the query
```
travelBetween(nancy,saarbruecken).
```
it should reply 'yes'.
It is, furthermore, plausible to assume that whenever it is possible to take a direct train from A to B, it is also possible to take a direct train from B to A. Can you encode this in Prolog? You program should e.g. answer 'yes' to the following query:
```
travelBetween(saarbruecken,nancy).
```
Do you see any problems you program may run into?
Hint: Try using a third intermediate predicate which has the recursive definition of `travelBetween/2` *in the first part.*

6. We are given the following knowledge base of travel information:
```
byCar(auckland,hamilton).
byCar(hamilton,raglan).
byCar(valmont,saarbruecken).
byCar(valmont,metz).
byTrain(metz,frankfurt).
byTrain(saarbruecken,frankfurt).
byTrain(metz,paris).
byTrain(saarbruecken,paris).
byPlane(frankfurt,bangkok).
byPlane(frankfurt,singapore).
byPlane(paris,losAngeles).
byPlane(bangkok,auckland).
byPlane(losAngeles,auckland).
```
Write a predicate `travel/2` which determines whether it is possible to travel from one place to another by 'chaining together' car, train, and plane journeys. For example, your program should answer 'yes' to the query
```
travel(valmont,raglan).
```