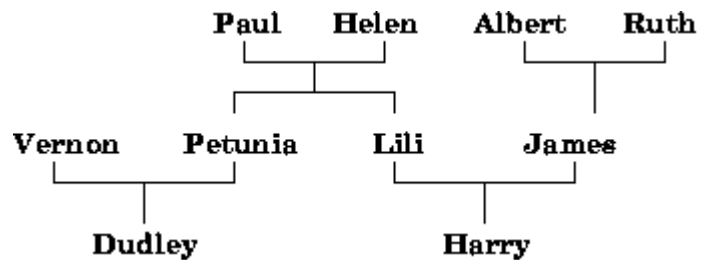# Lab - Recursion in Prolog

Family relationships

Last week we looked at defining some family relationships. Let's go over them again, and we'll use them to illustrate recursion.

Exercise 1: Use the predicates male/1, female/1, and parent_of/2 to represent the following family tree as a Prolog knowledge base.

```
        Paul    Helen    Albert    Ruth
         └───────┤        └─────────┘
             └────────┬────────┘        │
                                        │
Vernon    Petunia      Lili        James
  └──────────┤          └────────────┘
             │               │
          Dudley          Harry
```

Solution: recursion1.pl

Now, formulate rules to capture the following relationships:

father_of(Father,Child) and **mother_of(Mother,Child)**

grandfather_of(Grandfather,Child) and **grandmother_of(Grandmother,Child)**

sister_of(Sister,Person)

aunt_of(Aunt,Person) and **uncle_of(Uncle,Person)**

Try it first, and then look at the hint below.

HINT:

> father_of(Father,Child) and mother_of(Mother,Child):
> Fathers are male parent and mothers are female parents.
>
> grandfather_of(Grandfather,Child) and grandmother_of(Grandmother,Child):
> X is a grandfather of Y if X is a father of Z and Z is a parent of Y. X is a grandmother of Y if X is a mother of Z and Z is a parent of Y.
>
> sister_of(Sister,Person) and brother_of(Brother,Person):
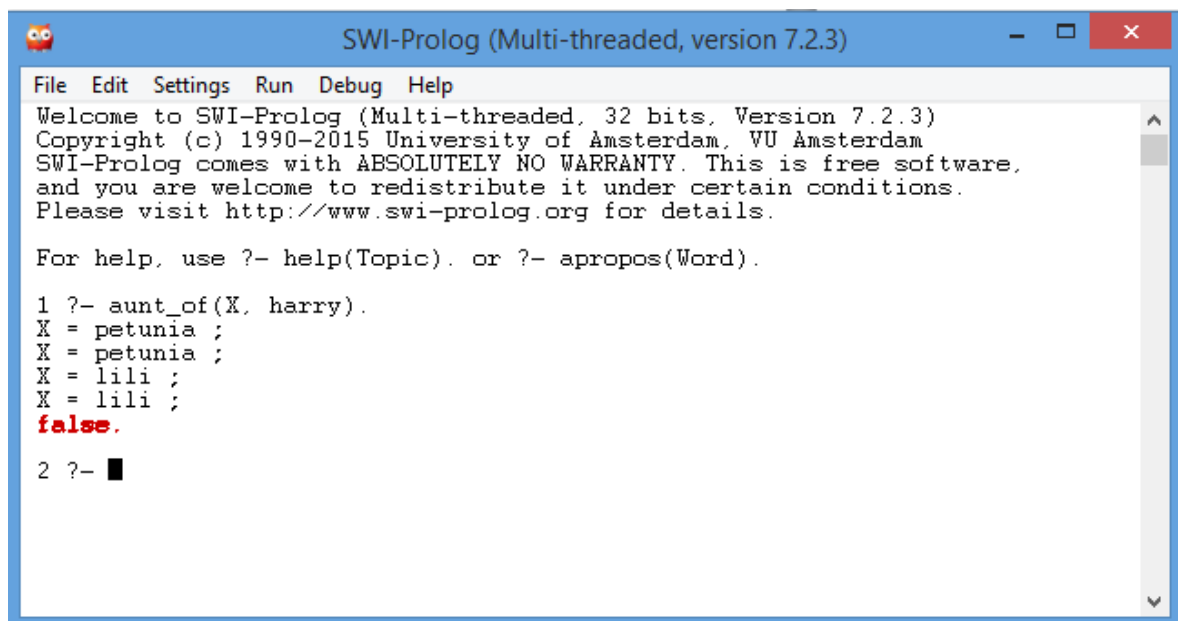> X is a sister of Y if X is female and Z is a parent of both X and Y.
>
> aunt_of(Aunt,Person) and uncle_of(Uncle,Person):
> X is an aunt of Y if X is a sister of Z and Z is a parent of Y.

Solution: recursion2.pl

To test your knowledgebase ask all kinds of queries. For example:

1. *Does Harry have an aunt? Who?*

2. *Who are the grandparents of Harry?*

3. *Who are the grandchildren of Paul and Helen?*

4. *Does James have a sister?*

The image below shows the state of the terminal on my machine after I loaded the family relationships knowledgebase and entered the query aunt_of(X, harry).

Notice, that Prolog finds each binding twice. Can you figure out why?

Answer:

The **sister_of** rule is defined as:

```
sister_of(X,Y) :- parent_of(Z,X),
         parent_of(Z,Y),
         female(X).
```

As you can see, Prolog thinks that Petunia is the sister of herself (this may be more clear if you evaluate the sister_of(X,Y) predicate)

This may seem strange however, according to our rule about sisters Prolog's answer is perfectly logical. Our rule about sisters does no mention that X and Y must not be the same if X is to be the sister of Y. As this is not required Prolog (rightfully) assumes

that X and Y can be the same, and will as a consequence find that any female who has a parent is a sister of herself. To correct our rules about sisters we have to add that X and Y must be different.

We can do this by:
(a) defining a predicate that succeeds if its two arguments are different:
          different(X,X) :- !,fail.
          different(X,Y) :- true.

According to this rule, if X and Y match then **different(X,Y)** fails, otherwise **different(X,Y)** succeeds. This rule uses the **cut** (!) and **fail** predicates that are built into Prolog. We will see how these work exactly later on, but for now keep in mind that cut (!) cuts off backtracking after encountering the cut operator. And,

(b) updating the **sister_of/2** rule to use this new predicate.

          sister_of(X,Y) :- parent_of(Z,X), parent_of(Z,Y), female(X), different(X,Y).

Recursive predicate definitions

Recursion in any language is the ability for a unit of code to call itself, repeatedly, if necessary. Recursion is often a very powerful and convenient way of representing certain programming constructs.

In Prolog, recursion occurs when a predicate contains a goal that refers to itself.

As we have seen in earlier chapters, every time a rule is called, Prolog uses the body of the rule to create a new query with new variables. Since the query is a new copy each time, it makes no difference whether a rule calls another rule or itself.

A recursive definition (in any language, not just Prolog) always has at least two parts, a boundary condition and a recursive case. The boundary condition defines a simple case that we know to be true. The recursive case simplifies the problem by first removing a layer of complexity, and then calling itself. At each level, the boundary condition is checked. If it is reached the recursion ends. If not, the recursion continues.

In Prolog a recursive rule is defined using two separate predicates. One predicate covers the trivial or boundary case. The other predicate covers the general case where the solution is constructed from solutions of (simpler) version of the original problem itself.

The basic structure of a recursive rule in Prolog is:

          f(x):-boundarycase.

          f(x):-g(part_of_x),f(rest_of_x).

Note: it is very important when defining a recursive rule in Prolog that the boundary case be listed in the knowledge base above the recursive case. Can you figure out why?

Recursion Exercise: ancestor/2

Exercise 2:
Extend the family tree that you defined in that exercise by adding great-grandparents for Harry.

Now, define a predicate ancestor_of/2. ancestor_of(X,Y) should be true if X is an ancestor of Y. Your predicate definition should work correctly even if you add Harry's great-great-grandparents, his great-great-great-grandparents, his great-great-... to the knowledge base.

Ask queries to test your definition. For example:

1. *Is Albert an ancestor of Harry?*
2. *Who are the ancestors of Harry?*
3. *Is Dudley an ancestor of Harry?*
4. *Who are the descendants of james?*

HINT:

The predicate grandparent_of/2 could be defined as follows:

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

The predicate greatgrandparent_of/2 could be defined as follows:
```
greatgrandparent_of(X,Y) :- parent_of(X,Z),
                parent_of(Z,A),
                parent_of(A,Y).
```

And the predicate greatgreatgrandparent_of/2 could be defined analogously by simply adding one more parent_of step to the definition of greatgrandparent/2. However, ancestor_of/2 cannot be defined in this way, as we don't know how many 'parents we have to go back' to find an ancestor. Anybody who is an *arbitrary number* of parent relations back is an ancestor.

So, how can we express what ancestor_of means? Well, obviously if I am the parent of somebody, I also belong to his ancestors. Furthermore, if I am the parent of somebody and that somebody is an ancestor of a third person, I am also an ancestor of this third person.

Solution: recursion3.pl

If Prolog does not come back with an answer, it has probably gone off into an endless loop. Stop it by pressing Ctrl-c. Then type a to abort.

Exercise 3: Recursion Exercise - Finding paths through a maze

The following image shows the topology of a maze:
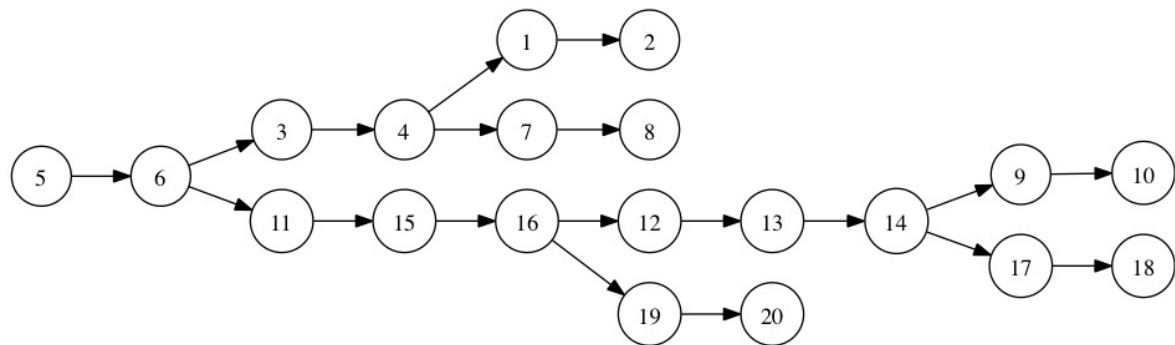
The following knowledge base describes the topology of this maze. The facts determine which points are connected, i.e., from which point you can get to which other point in one step.

Furthermore, imagine that all paths are one-way streets, so that you can only walk them in one direction. So, you can get from point 1 to point 2, but not the other way round.

connected(1,2).

connected(3,4).

connected(5,6).

connected(7,8).

connected(9,10).

connected(12,13).

connected(13,14).

connected(15,16).

connected(17,18).

connected(19,20).

connected(4,1).

connected(6,3).

connected(4,7).

connected(6,11).

connected(14,9).

```
connected(11,15).

connected(16,12).

connected(14,17).

connected(16,19).
```

Download the maze knowledgebase: **kbmase.pl**

Write a predicate **path/2** that tells you from which point in the maze you can get to which other point when chaining together connections given in the above knowledge base. Try it out before looking at the hint!

Hint: Note that the rule that the predicate path/2 has to express is very similar to the rule expressed by the predicate ancestor_of/2: in the first case, you can get from point X to point Y by chaining together an arbitrary number of connected/2 relations, and in the second case, X is an ancestor of Y if a chain of an arbitrary number of parent_of/2 relations connects them.

So, path(X,Y) should succeed if X is directly connected to Y. It should also succeed if X is directly connected to some Z from where in turn there is a path to Y.

Now ask some queries.

Can you get from point 5 to point 10? Which other point can you get to when starting in point 1? And which points can be reached from point 13?

Clause ordering and recursion

Clause ordering is very important with recursion! Look at those two examples below.

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

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

From a declarative perspective the difference between these to 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.

But from the procedural perspective the meaning has changed considerably!
For example, using the KB if you pose the query:
     ?-descend(A, B).

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

What causes the loop?

Well, to satisfy descend(harry,helen). Prolog uses the first rule. This means that its next goal will be to satisfy the query descend(W1,helen)
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,helen) for some variable W2 ...