Prolog – Three

August 25, 2003

The negation-as-failure operator \+ is a convenient way for us to write rule under the closed-word-assumption which states that an atom is *false* if it cannot be proven to be *true*. In using \+ we have to pay attention that no cycle between atoms can occur as in the following case:

```
male(X) :- person(X), \+ female(X).
female(X) :- person(X), \+ male(X).
```

The intuition of the above rules is clear. Unlike an ealier program, where we have only the first rule. If we have a person John and do not specify the gender of John, asking `?male(john).` will lead to an error. (Try it!)

Prolog allows us to define recursive function in a very straightforward way. In Prolog, it is very easy to define the relationship $ancestor$ (Try to write the definition for this in first order logic!). Intuitively, we can define the ancestor in two steps:

1. If X is a parent of Y then X is an ancestor of Y;
2. If X is an ancestor of Y and Y is an ancestor of Z, then X is an ancestor of Z.

This can be translated into the following Prolog rules:

```
ancestor(X,Y):- parent(X,Y).
ancestor(X,Z):- ancestor(X,Y), ancestor(Y,Z).
```

The first rule is the *base rule* and the second rule represents the *second rule*. It represents the idea of a inductive definition: the base case and the recursive case.

Add the above rules to the information about John's family, we get the program `p5.pl` as follows.

```
father(john, tom).
father(john, celine).
mother(marry, tom).
mother(marry, celine).

parent(X,Y) :- father(X, Y).
parent(X,Y) :- mother(X, Y).

ancestor(X,Y):- parent(X,Y).
ancestor(X,Z):- ancestor(X,Y), ancestor(Y,Z).
```

Let us compile the program and run the query `?- ancestor(marry,tom).` in Eclipse. The program responses with 'More' – meaning that the answer to the query is 'yes' and there might be another answer – click on 'More', we get an error "Overflow ...". This means that we probably get into an infinite loop!! More seriously, if we exchange the place of the last two rules and creates the program `p6.pl` as follows:

```
father(john, tom).
father(john, celine).
mother(marry, tom).
mother(marry, celine).

parent(X,Y) :- father(X, Y).
parent(X,Y) :- mother(X, Y).

ancestor(X,Z):- ancestor(X,Y), ancestor(Y,Z).
ancestor(X,Y):- parent(X,Y).
```

the same query will not be answered correctly. Why?

This is a well-known problem in Prolog. The main reason is that Prolog uses a fixed order (**top-to-bottom**) in selecting rules in query answering. As a rule of thumb, the basic rule of a recursive definition always needs to be placed before the general rule.

To avoid the infinite loop that might appear as in `p5.pl`, we need stop the application of the general rule when the answer is already found. Two ways: (i) use the cut operator which is denoted by !; (ii) strengthen the definition such that it is not applicable only when the base case cannot be applicable.

```
ancestor(X,Y):- parent(X,Y),!.
ancestor(X,Z):- ancestor(X,Y), ancestor(Y,Z).
```

This solution is shown in `p7.pl`. The cut operator ! causes Prolog to commit all the choices made since the parent goal was invoked and discard all the other alternatives. Here, it causes Prolog to commit to the choice made when $ancestor(marry, tom)$ is invoked and discard other possibilities that might exist if the second rule can be selected.

The other solution is to change the second rule as follows

```
ancestor(X,Y):- parent(X,Y).
ancestor(X,Z):- parent(X,Y), ancestor(Y,Z).
```

The program `p8.pl` contains this solution.

We will continue in this class with recursive definition. We will move to another important type of data in Prolog called *list*. In Prolog, a list is a special type of term. It is a recursive data structure consisting of pairs (whose tails are lists). A list is either the atom [] called *nil*, or a pair of the form $[H|T]$ whose tail is a list. The notation: $[a, b, c]$ is a shorthand for the list $[a|[b|[c|[]]]]$.

There are several useful functions that operate on lists. For example, we can compute the number of elements in a list using the following rules:

```
list_length([], 0).
list_length([H|T], N1):- list_length(T, N), N1 is N+1.
```

Notice that we write "N1 is N+1" instead of using N1=N+1. This is because N1=N+1 will mean that N1 is a term that can be unified with the term N+1. 'N1 is N+1" on the other hand asks Prolog to evaluate the term N+1 and assigned its value to N1. (See using compile scratch-pad).

Your job in this class is to define the following predicates:

- $list\_member(X, Y)$ which is true if and only if X is a member of the list Y.
- $list\_append(X, Y, Z)$ which is true if and only if Z is a list constructed by appending the elements of Y to X.
- $list\_delete(X, Y, Z)$ which is true if and only if Z is a list consists of elements in X which does not appear in Y.
- $list\_intersection(X, Y, Z)$ which is true if and only if Z is a list consists of elements appearing in both X and Y.
- $list\_no\_duplicate(X, Y)$ which is true if and only if Y is a list consists of elements in X without duplicates.