Prolog

September 6, 2004

Basics

A Prolog program is a set of *rules* which are built from *atoms* of a first order logic (FOL) theory. Formally, a rule is of the form

$$h{:-}l_1, \ldots, l_n$$

where h is an atom and each l_i is an atom or an expression of the form $\backslash{+}a$ (or $not\ a$) for some atom a.

Eclipse Prolog — a Prolog interpreter — will be used during the course of the class. Given a Prolog program, we can *compile* it and *ask* questions which will be answered with yes/no or one (or many) variable assignment. The intuitive is that each Prolog program represents a knowledge base (KB) and each query posed to the KB represents a theorem which needs to be evaluated against the KB.

Knowledge Representation in Prolog

To represent our knowledge about a domain of interest, we first need to identify its objects, the properties of them, and the relationships between the objects. For example, to describe the *family relations*, we need the following

- The individuals such as *tom, marry, ...*
- The properties of each individual such as *gender, age, ...*
- The relationships between individuals such as *father, mother, son, daughter, ...*

For example, let us consider a family of four individuals: John, Marry, Tom, and Celine. John is the father of Tom and Celine and Marry is the mother of Tom and Celine. Tom is a boy and Celine is a girl. Tom is older than Celine.

Given the above information can we answer the following questions:

- Is John a male?
- What is the gender of Celine?

- Are Celine and Tom siblings?
- Who is a brother of Celine?
- Who is a brother of Tom?
- etc.

To represent the fact that John is the father of Tom and Celine and Marry is the mother of Tom and Celine in Prolog, we write

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

Here, *father* and *mother* are predicate symbols and *john, marry, tom,* and *celine* denote the four individuals of the family. These are *constants* – in Prolog, a string starts with a lower case letter denotes a constant. We edit a file called `p1.pl` which consists of the above facts and compile the file using Eclipse Prolog.

Now that we have the program. We can ask questions about the individuals and their relationships. To ask, whether *john* is a father of *tom*, we write in the query entry of Eclipse

$$father(john, tom).$$

Try the above and see what happens. Here is what you will see in the **Results** screen:

```
?- father(john, tom).
Yes (0.00s cpu)
```

To ask who is the father of *tom*, we write

$$father(X, tom).$$

Here, X is a *variable*. In Prolog, a string beginning with a upper case letter denotes a variable. Instead of the 'yes/no' answer we get the following:

```
?- father(X, tom).
X = john
Yes (0.00s cpu)
```

Notice the difference between answer for queries with variables and answer for queries without variable. Queries with variables are answered with an assignment for variable ($X = john$, meaning that $john$ is an answer for the query $father(X, tom)$. For queries without variables, the answer is 'yes/no'.

How do we represent the other facts?

$$male(tom).$$
$$female(celine).$$
$$older(tom, celine).$$

We add the above to the program `p1.pl` and call the new program `p2.pl`. We compile the file and ask questions. You can try with your questions to see how Prolog answer them.

Let us figure out what happend if we ask the program the following questions:

- Is John a male? `?male(john).` NO
- What is the gender of Celine? ?
- Are Celine and Tom siblings? `?sibling(celine, tom).` NO
- Who is a brother of Celine? `?brother(X, celine).` NO
- Who is a brother of Tom? `?brother(X, tom).` NO

What is wrong? The problem is that we have in our mind a lot of information, such as John – being a father – is normaly a male; or Celine and Tom – having the same parent – are siblings. However, this information is not represented in our program. For this reasons, the answer will be NO.

We can, of course, represent the fact that $John$ is a male and $Marry$ is a female using the facts:

$$male(john).$$
$$female(marry).$$

Asking now whether John is a male, we will get the correct answer. Observe that to record the gender of some one, denoted by x, we need either $male(x)$ or $female(x)$. This solution will not be good if we have a large number of individuals. **Can we do it better than that?** A reasonable assumption would be that every individual is either a male or a female. So, we can choose to represent the 'female' individuals and write some rules to deduce that someone who is not a female is a male. This can be written as follows:

$$male(X) :- \ not \ female(X).$$

The not in the above rule is called the *negation-as-failure* operator. The rule is read as if *female(X)* cannot be proven then *male(X)* is true.

Suppose that we have the program `p3.pl` as follows:

```
father(john, tom).
father(john, celine).
mother(marry, tom).
mother(marry, celine).
female(celine).
female(marry).
male(X) :- not female(X).
older(tom, celine).
```

Now, if we ask our program about the gender of every individual, we will get the correct answer. As you can see, a Prolog program can be extended incrementally by adding new atoms (facts) and rules. Now, suppose that we wanted to add to `p3.pl` also some information about the addresses of the four individuals. John and Marry are in Las Cruces whereas their two children are studying in California, San Francisco, to be precise. We could do so by adding the facts:

```
livein(john, las_cruces).
livein(marry, las_cruces).
livein(tom, san_francisco).
livein(celine, san_francisco).
```

As you can see, constants can also contain the underscore symbol. Let `p4.pl` be the program consisting of `p3.pl` and the above four atoms. Let's compile it and ask queries about the gender of the individuals in the story, we get the same result. We also can ask queries about who is living where. The program will also return the correct answers. Now, let us ask the query

```
? male(las_cruces).
```

We get the answer 'YES'. This is certainly not what we wanted but why it is so? The culprit is the rule that defines whoever is not a female is a male! Of course, the error is ours. For Prolog, a string like 'john' is not much different than the string 'las_cruces' except that the former is shorter and they contain different characters. How will you solve this problem? This is the first homework for this class. The second homework asks you to extend the program so that we can answer the questions about relationship between members of an extended family.

Homework 1 (Question 1): Extending the program `p4.pl` with necessary rules and facts so that the following queries will be answered correctly:

- Questions about gender of people?
- Questions about who is living where?
- Questions about the relationships between Tom and Celine.

Homework 1 (Question 2): Let the program resulting from Question 1 be `p5.pl`. Extending the program with the following facts:

- John has a brother Paul.
- Marry has a brother Ringo and a sister Ono.
- Paul is living in San Francisco.
- Everyone lives in San Francisco or Las Cruces.

Define the relationship *uncle* and *aunt*. Your program should be able to answer queries about the relationship between the new individuals and the previously mentioned ones. It should also giving correct answer about the living place of each individual. Furthermore, the answers should not be changed if we add some facts about tables and chairs in the class.

Negative Information

The negation-as-failure operator $\backslash+$ 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 $\backslash+$ 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.

In this class, we will practice writting some recursive definitions which – hopefully – will provide us a better understanding of Prolog. In particular, we will see some examples that explain why our program – although looking perfectly fine – can get into an infinite loop. We will take some simple problems that are well-known to all of us and write programs to solve them.

Example 1

Defining non-negative integers.

Isn't it trivial to define the set of non-negative integers? Well, the set of non-negative integers is $\{0, 1, 2, \ldots, n, \ldots\}$. We can come up with something like:

```
% num(N-) iff N is a non-negative integer

num(0).
num(N1):- num(N), N1 is N+1.
```

Notice that the line begginning with % is a comment. The comment here states that we define a predicate $num(-)$, where the minus $-$ stands for output, which is true iff its output is a non-negative number. Compiling the program and post the goal `?num(X).` to Prolog, it will correctly generate all the number 0,1,2,....

However, when we ask the query "`?num(3).`", the program will correctly answer with 'yes' and then says that there is more solutions. Ask for more and we get into an infinite loop. Why? The reason is that in the second rule, we ask Prolog to 'guess' a number and then check whether that number plus 1 equals the original number.

To fix the program of getting into the loop, we could rewrite the second rule as follows:

```
num(0).
num(N1):- N1 > 0, N is N1-1, num(N).
```

Asking now `?num(3).` and the system correctly get out of the loop. However, if we ask for all X by posting the goal `?num(X).` we will get into trouble. In this case, we define a predicate which is correct when the input is given. The comment should be changed to

% num(N+) iff the input is a non-negative integer

The above two programs exhibit a property of Prolog whose root is the answering mechanism implemented in many PROLOG systems. It says that the system is sound but incomplete. While the first progam is able to generate the set of non-negative integers, the second one provides a "better answer" to the query whether a number is a non-negative number. This suggests that we should use the first program only in situation when we would like to generate all the numbers. The second program is more appropriate when we want to check for a number being non-negative. It is important to realize when/where/how to write proram that terminates when you *need* it to.

Checking for prime numbers

We will now write a program that determine whether a number N is a prime number or not. The most primitive algorithm can be described by the formula

$$(\forall M,\ M > 1,\ M < N, N \text{ is not divisible by } M) \rightarrow N \text{ is a prime number }.$$

So, we can implement this step-by-step as follows.

First, we define the predicate $divisible$ as follows.

```
% divisible(X+,Y+) is true iff X is divisible by Y

divisible(0,Y).
divisible(X,Y):- X > 0, X1 is X-Y, divisible(X1,Y).
```

Try out this program and notice the difference when you run $divisible(1,3),\ divisible(100,4)$, and $divisible(10,X)$.

Next we define the predicate $compoundNumber(N)$ that states that N is a compound number. The obvious way is to define a predicate $less(M,N)$ that find a number less than equal N and chekc for divisibility.

```
% less(X+, Y+) is true iff X is smaller than Y

less(0, 1).
less(0, Y):- Y > 0, Y1 is Y - 1, less(0, Y1).
less(X, Y):- X > 0, Y > 0, X1 is X-1, Y1 is Y-1, less(X1, Y1).
```

The next rule defines the $compNum(N)$ predicate using the above idea:

```
% compNum(X) is true iff X is a compound number

compNum(X):- less(Y, X), Y > 1, Y < X, divisible(X, Y).
```

This rule does not work properly because the variable Y have to be guessed and $less(X, Y)$ does not work properly when it has to guess. For this reason, we develop a new set of rules. The idea is to put the boundary to the possible values that we need to check for being a divisor of X. The rules are as follows.

```
% compoundNumber(X+) is true iff X is a compound number

compoundNumber(X):-
  X >= 0, X1 is X - 1,
  compoundInBound(X, X1).

compoundInBound(X, Y):-
  Y > 1, Y < X, divisible(X, Y).

compoundInBound(X, Y):-
  Y > 1, Y < X, \+ divisible(X, Y),
  Y1 is Y - 1,
  compoundInBound(X, Y1).
```

In the above, $compoundInBound(X, Y)$ defines a predicate that is true iff some integer between 2 and Y is divisible by X. The two rules check for the first divisor of X in the range $2 \ldots X - 1$.

Finally, the rules for checking the prime number is as follows.

```
% primeNumber(X) is true iff X is a prime number

primeNumber(X):-
  num(X), !,
  \+ compoundNumber(X).
```

Try to run this program and see what happends!