Prolog – Four

August 27, 2003

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 out input 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!