PROLOG I

1. Facts

A prolog program is a set of facts and rules describing relations. Lets consider a relation r between cities where $f(a, b)$ belongs to the relation if there is a direct flight from city a to city b. The following facts are written in a prolog program called flight.

```
flight(a,b).
flight(a,c).
flight(b,c).
flight(c,d).
flight(d,e).
flight(e,a).
```

Some things to note here: Each statement ends with ., similar to the semicolon in $C++$. I use lower case letters here and this is important. Names which start with upper-case letters are reserved for variables. So for instance

```
fLight(a,b).
```

would be a valid fact (different from flight(a,b)), but

```
Flight(a,b).
```

is invalid and would result in an error-message.

I load my program from the intepreter by

```
[flight].
```

and can now ask questions. For instance.

```
flight(a,b).
```

results in true, and

```
flight(e,d).
```

results in false. It is more useful to think of the result false as meaning "it is not possible to deduce from the statements in the program that flight(e,d) is true." and not "flight(e,d) is false".

Now lets try with variables

```
?- flight(a,X).
X = b ;
X = c.
```

Two things to note here. First X denotes a variable since it starts with an upper-case letter. It can be replaced with any name with upper-case letters. For instance you may try

```
flight(A,ThisIsMyVariable).
```

Also, I press ; to tell the intepreter that I would like to get more answers than the first $X = b$.

So what is happening here: Prolog is working through your program looking for possibilities for replacing X with something else to get a true fact. It finds that the replacement $X = b$ gives the fact $flight(a, b)$ which is true and returns it's finding. You can think of $flight(a, X)$. as the question "for which X is flight(a,X) true?".

We can use logical operators to ask more complicated questions. For instance

```
flight(a,b), flight(b,a).
```

is prologs notation for "flight(a,b) and flight(b,a)". We get false since flight(b,a) cannot be deduced from your program. On the other hand

```
flight(a,b); flight(b,a).
```

will result in true as ; is prologs way of denoting "or".

```
\+ flight(b,a).
```

results in true. This is similar to the operation "not", but should be thought of as "it is not possible to show that flight(b,a). is true from the statements in the program". This is an important distinction.

We look at some more examples with variables.

```
?- flight(a,X), flight(X,a).
```

asks for an airport X which has flights to and from a. Note the difference with

```
?- flight(a,X), flight(Y,a).
```

Lets try to find out if there are any pair of airports with flights in both directions. We ask

```
?- flight(X,Y), flight(Y,X).
```

and get false.

2. Rules

A rule is written as

```
body :- head
```

where the intepretation is that "if head is true, then body is true". It is Prolog's notation for the symbol $\leftarrow$ in logic.

We want to add a rule which says that whenever there is a flight from an airport X to an airport Y, there is always a flight in the opposite direction.

The following

```
flight(a,b).
flight(a,c).
flight(b,c).
flight(c,d).
flight(d,e).
flight(e,a).
flight(X,Y) :- flight(Y,X).
```

is a good first try, but leads to an infinite loop.

If we query

```
?- flight(b,a).
```

we now get true. Prolog cannot find flight(b,a). among the facts, but replacing X=b we do have flight(a,b). and then by the rule it deduces that flight(b,a). must also be true.

If we query

```
?- flight(a,a).
```

prolog will never stop running out of rules to try. The following program does what we want, and breaks the loop

```
f(a,b).
f(a,c).
f(b,c).
f(c,d).
f(d,e).
f(e,a).

flight(X,Y) :- f(X,Y); f(Y,X).
```

3. Recursion

Consider the following program

```
f(a,b).
f(a,c).
f(b,c).
f(c,d).
f(d,e).
flight(X,Y) :- f(X,Y).
flight(X,Y) :- f(X,Z), flight(Z,Y).
```

It contains facts describing direct flights between cities. The relation flight describes flying between cities with stops. The first rule says that a direct flight is a flight. The second says that a flight can be composed of a direct flight and a flight. This is an example of recursion.

We query

```
flight(a,b).
```

and get true. Although there are no facts for flight, prolog uses the rule flight(X,Y) :- f(X,Y). and the fact f(a,b). to deduce flight(a,b). The query

```
flight(a,c).
```

also returns true. This time rule 2 is used with Z=b together with rule one for flight(b,c).

The query

```
flight(a,X).
```

will give all aiports we can fly to from a. Change the program to

```
f(a,b).
f(a,c).
f(b,c).
f(c,d).
f(d,e).
flight(X,Y) :- f(X,Y), write("rule 1 "),
                  write(X), write(" "), write(Y), write("\n").
flight(X,Y) :- f(X,Z), flight(Z,Y),  write("rule 2 "),
                 write(X), write(" "), write(Y), write("\n").
```

to see how prolog uses the rules.

4. Lists

Lists in prolog are written as e.g.

```
[1,2,3,4]
```

The empty list is []. A list consists of a head and a tail. The head of [1,2,3,4] is 1 and the tail is the list [2,3,4]. The empty list is special and does not have a head and a tail. The tail of [1] is the empty list. The head of [1,2] is 1 and the tail is the list [2].

Lets write a short program which checks if an element is a member of a list

```
mem(X,[X|T]).
mem(X,[H|T]) :- mem(X,T).
```

The program says that X is a member of a list if it is the head of the list (rule 1), or if it is a member of the tail of the list (rule 2).

One remark: The variable names H in rule 2 and T in rule 1 are not used for anything. Prolog allows us to skip naming variables we are not going to use. We do that as follows.

```
mem(X,[X|_]).
mem(X,[_|T]) :- mem(X,T).
```

The underscore says that something is there, but that we do not care what.

Member is already defined in prolog with the name member.

Lets write a program which computes the length of a list.

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

or with underscores

```
len([],0).
len([_|T],N) :- len(T,M), N is M+1.
```

The program says that the length of an empty list is zero. You query the program by writing for instance

```
len([1,2,3,4],4).
len([1,2,3,4],3).
```

where the first is true, since the list has length 4, and the second is false. If you want to find out the length of a list you can query

```
len([1,2,3,4],L).
```

where L is any variable name. Prolog searches for the value of L which results in true and returns 4. The natural way to view the two arguments is that the first is the list we want to find the length of, and the second is the results.

Note the use of the word "is" here. It says that if we compute the right side M+1 then the result should be equal to the left side N. Always put what you want to compute on the right side. You might be tempted to write "N==M+1" instead of "N is M+1", but that would not work, since you are not forcing Prolog to do any kind of evaluation of "M+1". If you try "2+3 is 1+4", you will also get false, since "2+3" is not evaluated. If you want to evaluate on both sides of the equation you can write "2+3 =:= 1+4" which would be true.

Lets write a program which compares two lists, returns true if they are equal, and false otherwise.

```
com([],[]).
com([H1|T1],[H2|T2]) :- H1 == H2, com(T1,T2).
```

We are not doing computations here, we really want H1 and H2 to be identical. Writing "H1 is H2" would also work on a list of integers, but would result in an error message on other lists.

Lets write a program which removes one occurence of an element from a list.

```
rem(X, [X|T], T).
rem(X, [H|T1], [H|T2]) :- rem(X,T1,T2).
```

and a program which removes all occurences.

```
rem(X, [],[]).
rem(X, [X|T], Result) :- rem(X,T,Result).
rem(X, [H|T], [H|Result]) :- dif(X,H), rem(X,T,Result).
```

Note that dif(X,H) ensures that X and H are different when we apply this rule.

Exercises.

Exericse 1.

a) Write a Prolog program which counts the number of occurences of the number 1 in a list.

b) Write a Prolog program which increments (i.e. adds 1) to each element in a list of numbers.

c) Write a Prolog program which counts the number of even numbers in a list of integers.

d) Write a Prolog program which removes all even integers from a list of integers.

Exercise 2.

a) Write a Prolog program which squares a number.

b) Write a Prolog program which calculates the factorial function $n!$ of a non-negative number n. Recall that $0! = 1$ and $n! = n \cdot (n-1)!$.

Exercise 3.

Avoid using built-in list-predicates when solving this problem.

a) Write a program which changes each element to a list containing that element. For example [a,b,c] should become [[a],[b],[c]].

b) Write a Prolog program which checks if an element belongs to a list of list. For instance, both the elements x and a belongs to the list of lists [x,x,[a,b],[b,c],d].

Exericse 1.

a)

```
numone([],0).
numone([H|T],N) :- H == 1, numone(T,M), N is M+1.
numone([H|T],N) :- H \= 1, numone(T,N).
```

b)

```
addone([],[]).
addone([H1|T1],[H2|T2]) :- addone(T1,T2), H2 is H1+1.
```

c) Write a Prolog program which counts the number of even numbers in a list of integers.

```
numeven([],0).
numeven([H|T], N) :- H mod 2 =:= 0, numeven(T,M), N is M-1.
numeven([H|T], N) :- H mod 2 =:= 1, numeven(T,N).
```

d) Write a Prolog program which removes all even integers from a list of integers.

```
remeven([],[]).
remeven([H|T], [H|Res] ) :- H mod 2 =:= 1, remeven(T,Res).
remeven([H|T], Res) :- H mod 2 =:= 0, remeven(T,Res).
```

Answers.

Exercise 2.

a)

```
square(X,Y) :- Y is X*X.
```

b)

```
fac(0,1).
fac(N1,Res1) :- N1>0, N2 is N1-1, fac(N2, Res2), Res1 is Res2 * N1.
```

Exercise 3.

a)

```
env([],[]).
env([H],[[H]]).
env([H|T1],[[H]|T2]) :- env(T1,T2).
```

b)

```
extmem(X,[X|T]).
extmem(X,[H|T]) :- extmem(X,H).
extmem(X,[H|T]) :- extmem(X,T).
```