Intro to Prolog
Chapter 11

Prolog, which stands for PROgramming in LOGic, is the most widely available language in the logic programming paradigm using the mathematical notions of relations and logical inference. Prolog is a declarative language rather than procedural, meaning that rather than describing how to compute a solution, a program consists of a data base of facts and logical relationships (rules) that describes the relationships which hold for the given application. Rather then running a program to obtain a solution, the user asks a question. When asked a question, the run time system searches through the data base of facts and rules to determine (by logical deduction) the answer.

Often there will be more than one way to deduce the answer or there will be more than one solution, in such cases the run time system may be asked to backtrack and find other solutions. Prolog is a weakly typed language with dynamic type checking and static scope rules.

Prolog is typically used in artificial intelligence applications such as natural language interfaces, automated reasoning systems and expert systems. Expert systems usually consist of a data base of facts and rules and an inference engine, the run time system of Prolog provides much of the services of an inference engine.

Basic Propositional Logic

Propositional logic provides the foundation for programming in Prolog. A **proposition** is formed using the following rules:

- true and false are propositions
- variables p,q, r,… etc. that take on values true or false are propositions
- Boolean operators $\wedge$ $\vee$ $\neg$ $\Rightarrow$ and = used to form more complex propositions

Book uses $\supset$ in place of $\Rightarrow$ for implication, where $p \Rightarrow r$ is equivalent to $\neg p \vee r$.
For example, a proposition might be $p \vee q \wedge \neg r$ which ultimately evaluates to true or false.

Predicate logic expressions include all propositions and also include variables in the domain. A **predicate** is a proposition in which some of the Boolean variables are replaced by:

- Boolean-valued functions
- Quantified expressions

Here are some examples of Boolean-valued functions:

prime(n)	- true if n is prime
0 < x	- true if 0 < x
president-of-usa(x)	- true if x is the president of the USA

A predicate combines these kinds of functions using operators of the propositional calculus and the universal quantifier, $\forall$ (which reads "for all"), and the existential quantifier, $\exists$ (which reads "there exists").

For example:

$\forall x$ (speaks(x,Russian)) - true if everyone on the planet speaks Russian, false otherwise
Note that this is not true for ONLY Russian speakers, this applies to everyone

$\exists x$ (speaks(x,Russian)) - true if at least one person on the planet speaks Russian

$\forall x \exists y$ (speaks(x,y)) – true if for all people x, there exists a language y such that x speaks y; false otherwise

$\forall x$ ($\neg$literate(x) $\Rightarrow$ ($\neg$writes(x) $\wedge$ $\neg \exists y$(reads(x,y) $\wedge$ book(y)))) - true if every illiterate person x does not write and does not read a book y

A **tautology** is a proposition that is true for all possible values of their variables. A simple example is: $q \vee \neg q$. If q is true the entire expression is true. If q is false the expression is still true.

Predicates that are true for some assignment of values to their variables are called **satisfiable**. Those that are true for all possible assignments of values to their variables are called **valid**.

A prolog program is essentially an implementation of predicate logic.

Prolog Syntax

Prolog is based on facts, rules, queries, constants, and variables. Facts and rules make up the database while queries drive the search process. Facts, rules, and queries are made up of constants and variables. All prolog statements end in a period.

Facts

A fact is a proposition and begin with a lowercase alphabetic character and ends in a period. Here are some sample facts:

```
sunny.
```

This says that *sunny* is true.

```
superhero(spiderman).
```

This says that spiderman is a superhero. Note the lowercase. This distinction is important, because we'll use an initial uppercase letter to indicate a variable.

eats(spiderman, pizza).

This says that spiderman eats pizza.

Each fact that we enter describes the logical "world" that comprises the database of knowledge we can then reason over.

Rules

A rule is an implication like forward chaining in logic. In a rule, we can use boolean operators to connect different facts. The symbols used in prolog are as follows:

Predicate Calculus	Prolog	
$\wedge$	,	(comma)
$\vee$	;	(semicolon)
$\leftarrow$	:-	"if", note direction is left, not $\rightarrow$
$\neg$	not	

Here are some examples:

humid :- hot, wet.	Same as: $hot \wedge wet \rightarrow humid$
pleasant :- not(humid) ; cool.	Same as: $\neg humid \vee cool \rightarrow pleasant$

likes(bruno, spinach) :- not(likes(ted, spinach)).

Bruno likes spinach if Ted does not like spinach. Soon we'll extend this using variables instead of just spinach.

Variables

Variables are denoted in prolog by identifiers that start with an uppercase letter. For example:

likes(bruno, Food) :- not(likes(ted, Food)).

This says that Bruno likes any food that Ted does not like. Note that this is quite different from:

likes(bruno, food) :- not(likes(ted, food)).

The second statement is an atom named food, while the first is a variable that can represent any number of possible values.

Consider the following rules and facts:

```
likes(joe,Food) :-
      contains_cheese(Food),
      contains_meat(Food).
likes(joe,Food) :-
      greasy(Food).

likes(joe,chips).
contains_cheese(macaroni).
contains_cheese(lasagna).
contains_meat(lasagna).
greasy(french_fries).
```

In processing these rules, Prolog will **unify** the right hand side of the rule with any atoms that match the predicate. For the first rule, Food could be either macaroni or lasagna since both fit the criteria of contains_cheese. But then we AND this with contains_meat which leaves only lasagna. From these facts we can conclude that Joe likes chips, lasagna (cheese + meat), and french fries (greasy).

Queries

To query the database we can use prolog in an interpretive manner. Queries are generally made at the ?- prompt and consist of a predicate. For example, given the above data:

```
?- contains_meat(salad).
No

?- contains_meat(lasagna).
Yes

?- likes(joe,chips).
Yes

?- likes(joe, lasagna)
Yes

?- likes(joe, macaroni)
No
```

We can also make queries that include variables in them. Prolog will **instantiate** the variables with any valid values, searching its database in left to right depth-first order to find out if the query is a logical consequence of the specifications. Whenever Prolog finds a match, the user is prompted with the variables that satisfy the expression. If you would like to have Prolog continue searching for more matches, type ";" (meaning NO match yet). This may require prolog to backtrack to find some other matching

expressions. If you are satisfied with the match and would like Prolog to stop, type "y" (meaning Yes, accept).

Here is a query that finds all foods that contain cheese:

```
?- contains_cheese(X).
X = macaroni ;
X = lasagna ;
No
```

Since I hit ";" each time to not accept the matches, Prolog exhausts the possible foods with cheese and returns no. If I type "y" instead Prolog will return yes:

```
?- contains_cheese(X).
X = macaroni
Yes
```

We could query the database to find all the foods that Joe likes to eat:

```
?- likes(joe, X).
X = lasagna ;
X = french_fries ;
X = chips ;
No
```

We could also query the database to find all the people that like to eat lasagna:

```
?- likes(X, lasagna).
X = joe ;
No.
```

Right now nobody in the database likes macaroni so we get the following:

```
?- likes(X, macaroni).
No
```

Using SWI Prolog

We've now covered enough that we can write somewhat interesting programs. In this class we will be using a free implementation of prolog called SWI Prolog. It is already installed on bigmazzy and on the Windows machines. If you want to install it on your own Windows or Linux box, you can download it from http://www.swi-prolog.org/

Here we'll show how to get started with SWI Prolog in the Unix environment. It is very similar under Windows (in fact the Windows implementation just launches a Unix-like

shell). In Windows, files are loaded relative to the prolog directory selected as the 'Home' directory during installation.

To start, type **swipl** to invoke the SWI Prolog interpreter.

```
mazzy> swipl
Welcome to SWI-Prolog (Version 4.0.11)
Copyright (c) 1990-2000 University of Amsterdam.
Copy policy: GPL-2 (see www.gnu.org)

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

At this point, the interpreter is ready for you to type in queries. We have no predicates entered though, so we can enter them by typing either

consult(user). or the shortcut of **[user].**

We can then continue by typing in the facts and rules we would like:

```
?- [user].
|: likes(joe,Food) :-
|:        contains_cheese(Food),
|:        contains_meat(Food).
|: likes(joe,Food) :-
|:        greasy(Food).
|:
|: likes(joe,chips).
|: contains_cheese(macaroni).
|: contains_cheese(lasagna).
|: contains_meat(lasagna).
|: greasy(french_fries).
|: % user compiled 0.00 sec, 2,336 bytes

Yes
?-
```

To end the user input, hit **control-d**. Back from the ?- prompt we can now make our queries:

```
?- likes(joe, X).
X = lasagna ;
X = french_fries ;
X = chips ;
No
```

To exit the prolog interpreter, hit **control-d** again.

```
% halt
```

It is often inconvenient to have to enter our data every time we start prolog and go back and forth between user mode and query mode. We can direct prolog to load its facts from a file instead of from the keyboard. To do so, but all of the prolog code you want into a file in the working directory. SWI Prolog recognizes extensions of ".pl" as being prolog code. To load the file in prolog type either:

consult(filename). or **[filename].**

This will be the most common way you will input your data to prolog.

For example:

```
?- [myfile].
% myfile compiled 0.00 sec, 1563 bytes.
Yes
```

One command you may find useful is "trace" especially when we start working with recursive rules. To trace calls, use:

trace(predicate).

For example, to trace "likes" and "contains_cheese" we could issue the following:

```
?- trace(likes).
%        likes/2: [call, redo, exit, fail]

Yes
[debug]  ?- trace(contains_cheese).
%        contains_cheese/1: [call, redo, exit, fail]

Yes
[debug]  ?- likes(joe,X).
 T Call: (6) likes(joe, _G342)
 T Call: (7) contains_cheese(_G342)
 T Exit: (7) contains_cheese(macaroni)
 T Redo: (7) contains_cheese(_G342)
 T Exit: (7) contains_cheese(lasagna)
 T Exit: (6) likes(joe, lasagna)

X = lasagna
```

In this case Prolog displays every time we invoke a rule in the search for ways to satisfy the query. Prolog is invoking contains_cheese with a variable and getting back valid values for the predicate.

Upon using trace, we'll be in the debugger. To exit debug mode, enter **nodebug**.

```
[debug]  ?- nodebug.
```

Types and Expressions

Prolog is a weakly typed language. We have the following simple data types:

boolean , integer, real, atoms (character sequences)

Since a variable could be of many possible types, there are predicates to test what type a variable is:

var(V)	true if V is a variable
nonvar(NV)	true if NV is not a variable
atom(A)	true if A is an atom
integer(I)	true if I is an integer
real(R)	true if R is a floating point number
number(N)	true if N is an integer or real
atomic(A)	true if A is an atom or a number

Arithmetic expressions are evaluated using the built-in predicate **is** which operates in an infix fashion:

```
?- X is 4*4.
X = 16
```

It is important to note that we cannot use the = symbol here. The equals symbol unifies a variable with some value, but does not evaluate the value:

```
?- X = 4*4.
X = 4*4
```

We have not evaluated 4 times 4. Instead X is bound to 4*4, just like we could bind X to something like macaroni, cheese, or foo. In this case, we happened to choose a value that looks like an arithmetic expression.

We have available your standard arithmetic operators, with some syntax changes:

+ addition	- subtraction	* multiplication	
/ real division	// integer division	mod modulus	** power

Boolean Predicates allow us to compare values to one another. There are several interesting boolean predicates in Prolog that are not available in other languages, but we'll only cover the basics here.

```
A = B                    ; Unify A with B
```

Unification is not the same as assignment. It sets A equal to the matching pattern from B. It is the same as:

```
food(cheese).
?- food(X).
X = cheese;
```

In this case, when we query food(X) we unify or match A with cheese. The same thing is happening when we write A = B, e.g.:

```
A = foo ; A = bar.
A = foo;
A = bar;
No
```

Here Prolog is searching for matches that satisfy A. The user says "no" to discovered matches so none is found.

Here are some other booleans:

```
A == B          ; A identical to B
A \= B          ; A not identical to B
A < B           ; Numeric less than
A > B           ; Numeric greater than
A =< B          ; Numeric <=,  note that there is no <=
A >= B          ; Numeric >=, note that there is no =<
```

There is another form of <, > using terms, that we won't cover here.

User I/O

To output a string or variable use write:

write(X)	outputs value of X to screen
write('foo')	outputs foo

The value returned by write is always true.

To read into a variable use read:

```
read(X).
```

Read waits for the user to input a value, and what the user inputs is bound to X.

Normally we won't use read or write very often, but in some cases it may be useful, especially for debugging.

Comments

User comments are on lines preceded by %. These will be ignored by the interpreter or compiler.

Functions

Prolog does not provide explicit function types, but we can use rules or predicates to give us the same functionality. We can then use as arguments to our function the different variables that are passed in. With the aid of unification we will be able to manipulate these variables in clever ways to serve as both input and outputs depending on the context.

Let's define a predicate that returns the minimum of two other values. Since we don't have functions that can return values, we must return the value in a third parameter. What will really happen is Prolog will bind the third parameter to a value that satisfies the predicate. Prolog will try these definitions in order listed, from top to bottom.

```
minimum(M,N,M) :- M =< N.
minimum(M,N,N) :- N =< M.
```

We could invoke this as:

```
?- minimum(4, 5, X).
X = 4;

?-minimum(10, 3, X).
X = 3;
```

The first invocation unifies M=4, N=5, and M with X = 4 and checks if 4<=5. Since it is this expression is true. Since this is true, we get back as an option X = 4. If we said no to this value, prolog would try the second definition. M = 4, N=5, and N is unified with X. We check to see if 5 <= 4 which it is not, so this returns false.

The second invocation works the other way around, where the first definition will fail but the second will succeed. In this way we have defined a function to determine the minimum by expressing what we want declaratively, but perform no procedural instructions. Prolog does the procedural search for us.

Example Programs

Now that we have mastered functions, let's try some sample programs.

Factorial

For the first one, lets try defining factorial. We must do this recursively, starting with a base case and moving on up, where n! equals n*(n-1)! given the base case of 0! equals 1.

```
factorial(0,1).              %% Defines that 0! equals 1
factorial(A,B) :-            %% Compute A!  return value via B
    A > 0,
    C is A-1,                %% Note "is" for arithmetic expression
    factorial(C,D),          %% Compute A-1 factorial, get value in D
    B is A*D.                %% Our return variable gets set to A*D
```

We can now use this to compute the factorial:

```
?- factorial(4, X).
X = 24
```

A trace may be helpful to understand what is happening:

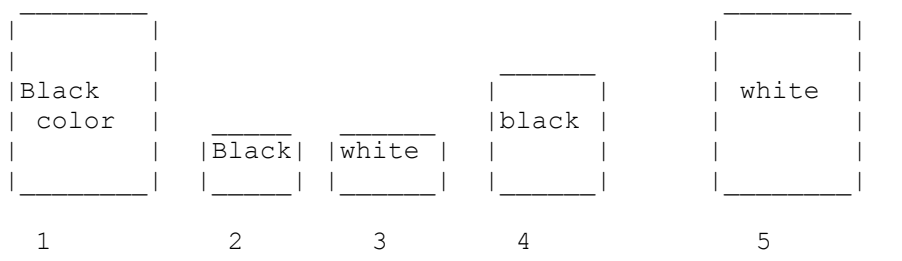
```
?- trace(factorial).
[debug]  ?- factorial(3,X).
 T Call: (6) factorial(3, _G354)
 T Call: (7) factorial(2, _G434)
 T Call: (8) factorial(1, _G437)
 T Call: (9) factorial(0, _G440)
 T Exit: (9) factorial(0, 1)
 T Exit: (8) factorial(1, 1)
 T Exit: (7) factorial(2, 2)
 T Exit: (6) factorial(3, 6)
```

Here we make the recursive calls until we get to the base case of factorial(0,1). Then the values propagate back up as the recursive calls exit. Note that we can't run this in reverse because we haven't specified enough information about A (e.g. we try to subtract 1 from it, but we need a starting value for A):

```
?- factorial(X,24).
ERROR: Arguments are not sufficiently instantiated
^  Exception: (7) _G233>0 ? Unknown option (h for help)
^  Exception: (7) _G233>0 ? nodebug
```

Constraints – Box Solver

For the second program, let's solve a small box problem. There are five boxes:




```
Ann, Bill, Charlie, Don, Eric own one box each but we don't know
which box.
Try to find each boxes' owner if you know that:
  Ann and Bill have boxes with the same color.
  Don and Eric have boxes with the same color.
  Charlie and Don have boxes with the same size.
  Eric's box is smaller than Bill's.
```

Here is Prolog code to solve the problem.

```
% First define numbers for each box.
getbox(1).   getbox(2).   getbox(3).   getbox(4).   getbox(5).

% Box number, color, size.
box(1,black,3).
box(2,black,1).
box(3,white,1).
box(4,black,2).
box(5,white,3).

owners(A,B,C,D,E):-
  getbox(A),  getbox(B),  getbox(C),  getbox(D),  getbox(E),
  A\=B,A\=C,A\=D,A\=E,
  B\=C,B\=D,B\=E,
  C\=D,C\=E,
  D\=E,
  box(A,ColorA,_), box(B,ColorA,_),  % Ann and Bill have same color
  box(D,ColorD,_), box(E,ColorD,_),  % Don and Eric have same color
  box(C,_,SizeC),  box(D,_,SizeC),   % Charlie and Don have same size
  box(E,_,SizeE),  box(B,_,SizeB),
  SizeE < SizeB.                     % Eric's smaller than Bill's
```

Here is the program in action;
?- owners(A,B,C,D,E).
A = 2
B = 4
C = 1
D = 5
E = 3 ;

SEND MORE MONEY

For the next program, let's solve the cryptarithmetic problem:

```
      S E N D
+     M O R E
-------------
=  M O N E Y
```

Given the constraints that each letter must be a digit from 0-9 and must be unique, we can determine a few constraints. M must equal 1. Then S must be greater than 8. But if M equals 1 then the largest we could get is 8+1+1=10 or 9+1+1=11. So O must be 0 or 1. But 0 cannot be 1 since M is 1, so O must be 0. We could work out more constraints, but let's stop there. We can now write a prolog program to solve this problem:

```
value(1).  value(2).  value(3).
value(4).  value(5).  value(6).
value(7).  value(8).  value(9).  value(0).          %% Define digit values
solve(S,M,E,O,N,R,D,Y) :-
        M is 1,                                     %% Our constraints first
        O is 0,
        value(S),
        S >= 8,
        value(Y),                                   %% Each var must be a value
        value(D),
        value(R),
        value(N),
        value(E),
        S \= M, S\=E, S\=O, S\=N, S\=R, S\=D, S\=Y,  %% Ensure uniqueness
        M\=E, M\=O, M\=N, M\=R, M\=D, M\=Y,
        E\=O, E\=N, E\=R, E\=D, E\=Y,
        O\=N, O\=R, O\=D, O\=Y,
        N\=R, N\=D, N\=Y,
        R\=D, R\=Y,
        D\=Y,
        Y is (D+E) mod 10,                          %% check if SEND+MORE
        C1 is truncate((D+E)/10),                   %%     = MONEY
        E is (C1+N+R) mod 10,                       %% could use // instead
        C2 is truncate((C1+N+R)/10),                %%     of truncate
        N is (C2+E+O) mod 10,
        C3 is truncate((C2+E+O)/10),
        O is (C3+S+M) mod 10.
```

Let's run the program:

?- solve(S,M,E,O,N,R,D,Y).

S = 9
M = 1
E = 5
O = 0
N = 6
R = 8
D = 7
Y = 2 ;

The program runs through the big list of AND's. When something is false, the program backtracks and tries another value. This process is continually repeated until finally we come across some values that match our constraints and make the right hand side true.

We could simplify this program a bit by using lists or permutations to check for uniqueness (more on lists later).

Lists

Given what we've covered, we can already write fairly sophisticated programs. However, to build more complex data types, we can make use of prolog's built in lists. To manipulate the lists, we must use recursion since prolog lacks iterative structures.

A list is denoted by square brackets, [].

> [] is the empty list.
> [1] is the list with the element 1 inside it.
> [3,1] is the list with elements 3 and 1 inside it.
> [foo, bar, zot] is the list with foo, bar, and zot in it.
> [[3,1], [2,2], [4,4]] is the list with three sublists

By embedding lists within lists, we can create fairly complex data structures.

The trickier part comes in matching lists. Prolog uses the vertical bar symbol, |, to separate the head element from the tail of the list. For example, if:

> [a, b, c] is matched to [X|Y] then X=a, Y=[b, c]
> [a, b, c] is matched to [X,Y | Z] then X=a, Y=b, Z = [c]
> [a, b, c] is matched to [W, X,Y|Z] then W=a, X=b, Y=c, Z=[]
> [a, b, c] does not match [V,W,X,Y | Z] since the number of elements differs.

Here is how we can construct lists and access elements of the list:

```
head([X|Y],X).
tail([X|Y],Y).
add(X,R,[R|X]).
```

Head takes the list as the first argument, and returns the first element of the list as the second argument.

Tail takes the list as the first argument, and returns the tail of the list as the second argument.

Add takes a list as the first argument and an element to add as the second argument and returns the list with the appended element as the third argument. Notice that we are using [R|X] constructively in this case, while previous examples used it to pick apart an existing list.

For example:

```
?- add([],a,X), add(X,b,Y), add(Y,c,Z), tail(Z,TAIL), head(Z,HEAD).

X = [a]
Y = [b, a]
Z = [c, b, a]
TAIL = [b, a]
HEAD = c
```

This created the list Z equal to [c, b, a] (we put new elements on the front). Then we extracted the tail and the head from this list.

Prolog includes a predicate to test for membership of a list. It is member:

```
?- member(x,[a,b,x,y,z]).
Yes
?- member(x,[a,b,c]).
No
```

Although this is built-in, it is a useful exercise to write our own membership function. We can do so recursively fairly simply:

```
mymember(X, [X|T]).                    %% If X is the head, then true.
mymember(X, [Y|T]) :- mymember(X, T).  %% Recurse on tail of list
```

The recursive part of this function traces mymember with smaller and smaller lists each time. If X is a member of the list, eventually it will be the head of the list and the first predicate will be true. Otherwise, this function will recurse through the entire list without

finding the element and give back false. We had to list mymember(X, [X|T]) first to terminate recursion since this will be checked before the second definition.

Also notice that in the recursive definition, we declared a variable Y to refer to anything that is in the head position of the list. However, we didn't use the variable anywhere on the right side of the rule. We just wanted some variable there that is not equal to X. In cases like this, we can instead use something called the anonymous variable. It is denoted by an underscore and serves as a placeholder for a variable whose value we really don't care about. We can rewrite this as:

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

The functionality will be identical to before. This is a good programming practice where applicable, since it tells the programmer and interpreter that certain variables are used solely for pattern-matching purposes, but not for variable binding.

Prolog contains a number of other useful built-in predicates that deal with lists. Here are some of them:

```
length(L,X)          -  X is set to the length of L
e.g. length([a,b,c],X).
        X=3;

reverse(L,R)         - R is set to the reverse of L
e.g. reverse([a,b,c],R).
        R = [c,b,a]

union(L1,L2,U)       - U is the union of L1 and L2
e.g. union([1,2,3],[2,4],U).
        U = [1,2,3,4]

intersection(L1,L2,I)  - I is the intersection of L1 and L2
e.g. intersection([1,2,3],[2,4],I).
        I = [2]
```

List Examples

Here are a few more short examples of working with lists.

The following will increment each element in the list by one, and return a new list of incremented values:

```
inc(X,Y):- Y is X+1.
incAll([],[]).
incAll([H|T],[R|RT]):- inc(H,R),incAll(T,RT).
```

The “inc” predicate returns X+1 through Y.
The “incAll” predicate takes an input list and the second parameter is the output list of values incremented by one. To terminate the recursion, we define the empty list incremented is just the empty list.

In the more general case, “incAll” is split up in terms of a head and tail for the input and output. We define the lead element of the output list to be the head of the input, incremented by one. We follow this up by defining that the tail must also conform to incAll.

Example usage:

```
?- incAll([1,4,6],X).
X = [2, 5, 7] ;
```

The following will remove duplicate elements from a list:

```
removedups([],[]).
removedups([X|Rest], Result) :-
      member(X,Rest),
      removedups(Rest,Result).
removedups([X|Rest], [X|Rest1]) :-
      not(member(X,Rest)),
      removedups(Rest,Rest1).
```

RemoveDups takes as the first argument the list to remove duplicates. The list without duplicates is returned in the second argument.

If the list is empty, the list without duplicates is also empty.

If the head of the list is a member of the rest of the list, it is a duplicate. We don’t include X in the result by invoking removedups with Rest.

If the head of the list is not a member of the rest of the list, it is not a duplicate. Make sure this element is included in the list by concatenating it on front of the outgoing parameter, [X|Rest1].

Example usage:

```
?- removedups([1,1,2,2,5,6,7,5,9],X).
X = [1, 2, 6, 7, 5, 9] ;
```

The following will remove all occurrences of an element from a list:

```
removeall([],_,[]).
removeall([H|T],H,T2):-
   removeall(T,H,T2).
removeall([H|T],X,[H|T2]):-
   X\=H,
   removeall(T,X,T2).
```

The first parameter is the list we would like to remove an element from. The second parameter is the element we want to remove. The third parameter is the list with the element removed.

In this example, we return back the empty list if the input list is empty.
If the head of the input list is the same as our element, then return back removeall of the tail.
If the head of the input list is not the same as the element, concatenate the head onto the list we are returning and invoke removeall.

Example usage:

```
?- removeall([a,b,c,b,b,b,a,a,d,b,b],b,X).
X = [a, c, a, a, d] ;
```

Example: Mergesort

Here is code to sort a list using merge sort. How about we just give the code and talk about it in class. It's really short compared to writing a mergesort routine in Java!

```
mergesort([],[]).    /* covers special case */
mergesort([A],[A]).
mergesort([A,B|R],S) :-
   split([A,B|R],L1,L2),
   mergesort(L1,S1),
   mergesort(L2,S2),
   merge(S1,S2,S).

split([],[],[]).
split([A],[A],[]).
split([A,B|R],[A|Ra],[B|Rb]) :-  split(R,Ra,Rb).

merge(A,[],A).
merge([],B,B).
merge([A|Ra],[B|Rb],[A|M]) :-  A =< B, merge(Ra,[B|Rb],M).
merge([A|Ra],[B|Rb],[B|M]) :-  A > B,  merge([A|Ra],Rb,M).

Here is a sample goal:

?- mergesort([4,3,6,5,9,1,7],S).
S=[1,3,4,5,6,7,9]
```

Example Program: U2 Crossing a Bridge

Allegedly, this is one of the questions for potential Microsoft employees.

"U2" has a concert that starts in 17 minutes and they must all cross a bridge to get there. All four men begin on the same side of the bridge. You must help them across to the other side. It is night. There is one flashlight. A maximum of two people cross at one time. Any party who crosses, either 1 or 2 people, must have the flashlight with them. The flashlight must be walked back and forth, it cannot be thrown, etc. Each band member walks at a different speed. A pair must walk together at the rate of the slower man's pace.

Bono - 1 minute to cross
Edge - 2 minutes to cross
Adam - 5 minutes to cross
Larry - 10 minutes to cross

For example - If Bono and Larry walk across first, 10 minutes have elapsed when they get to the other side of the bridge. If Larry then returns with the flashlight, a total of 20 minutes has passed and you have failed the mission.

We can model this problem by making "moves" from one state to the next. We must know what the goal state is and what states can lead to what other states. We will model states through the 5-tuple composed of l/r. The first element indicates what side of the bridge each Bono is on (left or right). The second element indicates what side the Edge is on. The third indicates what side Adam is on. The fourth indicates what side Larry is on. The fifth indicates what side the flashlight is on.

For example:	l,l,l,l,l	is the start state, everyone/flashlight is on the left
	r,r,l,l,r	Bono and Edge on right with flash, Adam and Larry on the left side
	r,r,r,r,r	goal state, everyone on the right

Our strategy will be to make moves and then remember each state that we visit by storing the state into a list. Before we visit a new state, we'll check to see if it already exists in the list of visited states. If so, we won't visit that state. This will eliminate the possibility of forming infinite loops in our search. Along the way we will also compute the total amount of time we have used. If the time is greater than 17 then we stop.

Here is a predicate that will tell us the opposite of a side:

```
opp(l,r).
opp(r,l).
```

For example, opp(l,X). gives us X=r, which is the opposite of l.
Next, we encode how long each person takes to cross:

```
time(bono,1).
```

```
time(edge,2).
time(adam,5).
time(larry,10).
```

We will need the maximum of two times, since we must proceed at the pace of the slower man. Here is a predicate to get the max of two numbers:

```
max(T1,T2,T2):- T1 =< T2.
max(T1,T2,T1):- T1 > T2.
```

Next, if we ever reach the goal state, print out a message:

```
% Goal state
move(state(r,r,r,r,r,T),_) :- T < 18, write('Goal achieved!'),nl.
```

The possible moves we can make are:

Bono goes alone Edge goes alone
Adam goes alone Larry goes alone
Bono and Edge go together
Bono and Adam go together
Bono and Larry go together
Edge and Adam go together
Edge and Larry go together
Adam and Larry go together

We will write a predicate for each of these possible moves. In a less brute-force solution we could combine many of these predicates into one by using lists, but in this case we will define a separate predicate for each move.

Here is the move for Bono going alone:

```
move(state(B,E,A,L,B,T),LST) :-
  opp(B,New),
  not(member(state(New,E,A,L,New),LST)),
  time(bono,T2),
  NewT is T + T2,
  NewT < 18,
  move(state(New,E,A,L,New,NewT),[state(New,E,A,L,New)|LST]),
  write('Bono crosses.'), nl.
```

In the state, B is used for both Bono and the Flashlight position, so this is only valid if Bono is on the same side as the Flashlight. First, we get the opposite side. Then make sure the new state, if Bono went to the other side, is not already in our list LST of visited states. Next we get how long it takes Bono to cross, add it to our current time, make sure it is still under the time limit, then move to the new state, adding our new state to the list of visited states. If successful, we write out that Bono crossed.

The other moves for each person moving alone are similar:

```
% Take edge alone
move(state(B,E,A,L,E,T),LST) :-
```

```
  opp(E,New),
  not(member(state(B,New,A,L,New),LST)),
  time(edge,T2),
  NewT is T + T2,
  NewT < 18,
  move(state(B,New,A,L,New,NewT),[state(B,New,A,L,New)|LST]),
  write('Edge crosses.'), nl.

% Take adam alone
move(state(B,E,A,L,A,T),LST) :-
  opp(A,New),
  not(member(state(B,E,New,L,New),LST)),
  time(adam,T2),
  NewT is T + T2,
  NewT < 18,
  move(state(B,E,New,L,New,NewT),[state(B,E,New,L,New)|LST]),
  write('Adam crosses.'), nl.

% Take larry alone
move(state(B,E,A,L,L,T),LST) :-
  opp(L,New),
  not(member(state(B,E,A,New,New),LST)),
  time(larry,T2),
  NewT is T + T2,
  NewT < 18,
  move(state(B,E,A,New,New,NewT),[state(B,E,A,New,New)|LST]),
  write('Larry crosses.'), nl.
```

Next, encode the case where both Bono and the Edge cross together:

```
move(state(B,B,A,L,B,T),LST) :-
  opp(B,NewB),
  not(member(state(NewB,NewB,A,L,NewB),LST)),
  time(bono,TB),
  time(edge,TE),
  max(TB,TE,TMax),
  NewT is T + TMax,
  NewT < 18,
  move(state(NewB,NewB,A,L,NewB,NewT),[state(NewB,NewB,A,L,NewB)|LST]),
  write('Bono and Edge cross.'), nl.
```

This starts by defining B for both Bono, Edge, and the Flashlight, so this predicate is only valid of all three are on the same side. We proceed similar to the case of where Bono went alone, except now we flip the side for both Bono and the Edge and get the larger of the crossing time between Bono and the Edge.

The remaining moves are similar to the above, with variables switched for the different band members that move:

```
% Take bono and adam
move(state(B,E,B,L,B,T),LST) :-
  opp(B,NewB),
  not(member(state(NewB,E,NewB,L,NewB),LST)),
  time(bono,TB),
```

```
  time(adam,TA),
  max(TB,TA,TMax),
  NewT is T + TMax,
  NewT < 18,
  move(state(NewB,E,NewB,L,NewB,NewT),[state(NewB,E,NewB,L,NewB)|LST]),
  write('Bono and Adam cross.'), nl.

% Take bono and larry
move(state(B,E,A,B,B,T),LST) :-
  opp(B,NewB),
  not(member(state(NewB,E,A,NewB,NewB),LST)),
  time(bono,TB),
  time(larry,TL),
  max(TB,TL,TMax),
  NewT is T + TMax,
  NewT < 18,
  move(state(NewB,E,A,NewB,NewB,NewT),[state(NewB,E,A,NewB,NewB)|LST]),
  write('Bono and Larry cross.'), nl.

% Take edge and adam
move(state(B,E,E,L,E,T),LST) :-
  opp(E,New),
  not(member(state(B,New,New,L,New),LST)),
  time(edge,T1),
  time(adam,T2),
  max(T1,T2,TMax),
  NewT is T + TMax,
  NewT < 18,
  move(state(B,New,New,L,New,NewT),[state(B,New,New,L,New)|LST]),
  write('Edge and Adam cross.'), nl.

% Take edge and larry
move(state(B,E,A,E,E,T),LST) :-
  opp(E,New),
  not(member(state(B,New,A,New,New),LST)),
  time(edge,T1),
  time(larry,T2),
  max(T1,T2,TMax),
  NewT is T + TMax,
  NewT < 18,
  move(state(B,New,A,New,New,NewT),[state(B,New,A,New,New)|LST]),
  write('Edge and Larry cross.'), nl.


% Take adam and larry
move(state(B,E,A,A,A,T),LST) :-
  opp(A,New),
  not(member(state(B,E,New,New,New),LST)),
  time(adam,T1),
  time(larry,T2),
  max(T1,T2,TMax),
  NewT is T + TMax,
  NewT < 18,
  move(state(B,E,New,New,New,NewT),[state(B,E,New,New,New)|LST]),
  write('Adam and Larry cross.'), nl.
```

Here is the program in action:

```
?- move(state(l,l,l,l,l,0),[]).
Goal achieved!
Bono and Edge cross.
Edge crosses.
Adam and Larry cross.
Bono crosses.
Bono and Edge cross.
```

Ultimately, what we have done is define what we want the outcome to be. Prolog has done the searching for us to tell us how to instantiate variables to satisfy our desired outcome.

Prolog programs generally do not run extremely efficiently; an equivalent program in C or C++ or Java will likely run much faster. However, some programs are much easier to code in Prolog but requires a different way to think about solving the problem than in an imperative language.