Introduction to Prolog  
Adapted by Vicki Allan from Paul Brna and Tamsin Treasure-Jones (Sept 2007)

Go to [http://www.amzi.com/](http://www.amzi.com/) to download a free version of prolog. There is a good tutorial associated with this product. You may choose to use it rather than this document. Use the IDE Quick Start link (on the webpage) to learn how to use the system.

Prolog is a logic language that is particularly suited to programs that involve symbolic or non-numeric computation. For this reason, it is a frequently used language in Artificial Intelligence where manipulation of symbols and inference about them is a common task.

In Prolog, there are no looping constructs. Instead, repetition is achieved by recursion.

Prolog consists of a series of rules and facts. A program is run by presenting some query and seeing if this can be proved against these known rules and facts. In this tutorial, we will attempt to give you a flavor of how all this is achieved and teach you how to write a simple program for yourself.

Here is a two-line program, meant to help you learn the mechanics of the editor and your listener.

```prolog
mortal(joe).
mortal(socrates).
```

From the Amzi! Interactive Development Environment (IDE), create a project and then select File/New from the main menu. Enter the program in the edit window, paying careful attention to upper and lowercase letters and punctuation. Next, start the Prolog listener by selecting Run/Run…. You want to run an interpreted project. You should see the typical listener prompt.  
`?-`  
Entering the source code in the Listener is called consulting. This happens automatically when you select Run. You should see comments in the Listener which show you it has consulted the source files.

Once you’ve loaded the program, try the following Prolog queries from the Listener prompt.

```prolog
?- mortal(socrates).
yes
?- mortal(X).
X = joe
```

This example shows a feature of Prolog called [unification](http://www.amzi.com/). Two things can be unified if they can be made to match. Thus, when you write `mortal(X)`, you are asking, “Is there a value for X which makes the statement true?”

### Simple Facts

In Prolog we can make some statements by using facts. Facts either consist of a particular item or a relation between items. For example, we can represent the fact that it is sunny by writing the program:

```
sunny.
```

We can now ask a query of Prolog by asking
?- sunny.

?- is the Prolog prompt. To this query, Prolog will answer yes. sunny is true because (from above) Prolog matches it in its database of facts.

Facts have some simple rules of syntax. Facts should always begin with a lowercase letter and end with a period. The facts themselves can consist of any letter or number combination, as well as the underscore _ character. However, names containing the characters -,+,*,/, or other mathematical operators should be avoided.

**Examples of Simple Facts**

Here are some simple facts about an imaginary world. /* and */ are comment delimiters

```
john_is_cold.  /* john is cold */
raining.       /* it is raining */
john_Forgot_His_Raincoat. /* john forgot his raincoat */
fred_lost_his_car_keys. /* fred lost is car keys */
peter_footballer. /* peter plays football */
```

These describe a particular set of circumstances for some character john. We can interrogate this database of facts, by again posing a query. For example: {note the responses of the Prolog interpreter are shown in italics}

```
?- john_Forgot_His_Raincoat.
yes
?- raining.
yes
?- foggy.
no
```

The first two queries succeed since they can be matched against facts in the database above. However, foggy fails (since it cannot be matched), and Prolog answers no since we have not told it this fact.

**Facts with Arguments**

More complicated facts consist of a relation and the items that this refers to. These items are called arguments. Facts can have arbitrary number of arguments from zero upwards. A general model is shown below:

```
relation(<argument1>,<argument2>,....,<argumentN>).
```

Relation names must begin with a lowercase letter

```
likes(john,mary).
```

likes is termed a *functor* or a *predicate*. It has *arity* two (as there are two parameters).

The above fact says that a relationship *likes* links john and mary. We can’t tell whether john likes mary or mary likes john. We only know that the relationship holds. Thus, you should always be clear and consistent how you intend to interpret the relation and when.
Names of the relations are defined by you. With the exception of a few relations that the system has built into it, the system only knows about relations that you tell it about.

**Facts with Arguments Examples 1**

An example database, listing who eats what in some world model.

```prolog
eats(fred, oranges). /* Fred eats oranges */
eats(fred, T-bone steaks). /* Fred eats T-bone steaks */
eats(fred, mangoes).
eats(fred, meal(hamburger, frenchfries)).
eats(tony, meal(appetizer('marinated mushrooms'), main(turkey, potatoes, broccoli), dessert('Crème Brûlée'))).
eats(tony, apples). /* Tony eats apples */
eats(john, apples). /* John eats apples */
eats(john, grapefruit). /* John eats grapefruit */
eats(tony, meal(appetizer(mushrooms), main(turkey, potatoes, broccoli), dessert(pie))).
eats(jim, happyMeal).
eats(sarah, hamburger).
```

If we now ask some queries we would get the following interaction:

```prolog
?- eats(fred, oranges).
/* does this match anything in the database? */
yes /* yes, matches the first clause in the database */

?- eats(john, apples).
/* do we have a fact that says john eats apples? */
yes /* yes we do, clause 4 of our eats database */

?- eats(mike, apples).
/* how about this query, does mike eat apples */
no /* not according to the above database */

?- eats(fred, apples).
/* does fred eat apples */
no /* again no, we don’t know whether fred eats apples */
```

In Prolog, variables are typeless. They can bind to an atom or to a structure. How do we say something like "What does Fred eat"? We could type in something like

```prolog
?- eats(fred, what).
```

However, Prolog will say no. The reason for this is that *what* is a literal (not a variable) and does not match with mangoes or with meal(hamburger, frenchfries). In order to match arguments in this way, we must use a Variable. The process of matching items with variables is known as unification. Variables are distinguished by starting with a capital letter.

Thus returning to our first question we can find out what Fred eats by typing

```prolog
?- eats(fred, What).
What = oranges
yes
```

As a result of this query, the variable *What* has matched (or unified) with oranges. We say that the variable *What* now has the binding “What:oranges”. When we pose a query, if the query is successful, Prolog prints both the variable and the variable name, as we see above. If we wanted to know everything that Fred eats, we could type ‘;’ (the symbol for “or”) after the first response as shown below.

```prolog
?- eats(fred, What).
What = oranges ;
```
What = t_bone_steaks ;
What = mangoes ;
What = meal(hamburger, frenchfries) ;

We can also use unification to match structures.
?- eats(Who,meal(appetizer(A),main(turkey,I,J),R)).

Who = tony
A = mushrooms
I = potatoes
J = broccoli
R = dessert(pie)

Variable Example 1
Let's consider some examples using facts. First consider the following database.

loves(john,mary).
loves(fred,hobbies).

Now let's look at some simple queries using variables
?- loves(john,Who).  /* Who does john love? */
  Who=mary          /* yes , Who gets bound to mary */
  yes              /* and the query succeeds*/
?- loves(arnold,Who). /* does arnold love anybody? */
  no               /* no, arnold doesn't match anything in our database*/
?- loves(fred,Who).  /* Who does fred love? */
  Who = hobbies    /* Note the to Prolog Who is just the name of a variable, its */
  yes             /* semantic connotations are not picked up, hence Who unifies */
                  /* with hobbies */

Variable Example 2
Here are some more difficult object/variable comparisons. Consider the following database showing a library of cassette tapes. Notice the final argument to tape, which is the name of a favorite song from the album.

tape(1,van_morrison,astral_weeks,madam_george).
tape(2,beatles,sgt_pepper,a_day_in_the_life).
tape(3,beatles,abbey_road,something).
tape(4,rolling_stones,sticky_fingers,brown_sugar).
tape(5,eagles,hotel_california,new_kid_in_town).

Let's now look at some queries.
?- tape(5,Artist,Album,Fave_Song).  /* what are the contents of tape 5 */
  Artist=eagles
  Album=hotel_california
  Fave_Song=new_kid_in_town
  yes
?- tape(4,rolling_stones,sticky_fingers,Song).  /* find just song */
  Song=brown_sugar

Rules
So far we have looked at how to represent facts and to query them. Now we move on to rules. Rules allow us to make conditional statements about our world. Each rule can have several
variations, called clauses. These clauses give us different choices about how to perform inference about our world. Let's take an example to make things clearer.

Consider the following

'All people are mortal':

We can express this as the following Prolog rule

\[
\text{mortal}(X) : - \text{person}(X).
\]

The clause can be read in two ways (called either a declarative or a procedural interpretation). The declarative interpretation is "For a given X, X is mortal if X is a person." The procedural interpretation is "To prove the main goal that X is mortal, prove the subgoal that X is a person."

To continue our previous example (which defined Rule 1 as person(socrates)), let's define the fact 'Socrates is a person' so that our program now looks as follows:

\[
\text{mortal}(X) : - \text{person}(X).
\]

\[
\text{person}(\text{socrates}).
\]

If we now pose the question to Prolog

\[
?- \text{mortal}(\text{socrates}).
\]

The Prolog interpreter would respond as follows:

yes

Why is this? Well in order to solve the query ?- mortal(socrates), we used the rule we saw previously. This said that in order to prove someone mortal, we had to prove them to be people. Thus, from the goal, Prolog generates the subgoal of showing person(socrates).

In the above example, we were able to match person(socrates) against the database. In Prolog, we say that the subgoal succeeded, and as a result, the overall goal succeeded. We know when this happens because Prolog prints yes. We might wish to see if there is somebody who is mortal. This is done by the following line.

\[
?- \text{mortal}(P).
\]

\[
P = \text{socrates}
\]

yes

This means that Prolog was able to prove the goal (mortal(P)) by binding the variable P to socrates. This was done by proving someone was mortal by proving the subgoal that they were a person. Prolog thus asked if there was any P that was a person. This matches against the clause person(socrates) thereby binding P to socrates. This binding is then passed back to the parent goal, and the results in the printout we saw above.

Sometimes we may wish to specify alternative ways of proving a particular thing. This we can do by using different rules and facts with the same name. For example, we can represent the sentence Something is fun if it is a red car or a blue bike or it is ice cream as follows:

\[
\text{fun}(X) : - /* an item is fun if */
\]

\[
\text{red}(X), /* the item is red */
\]

\[
\text{car}(X). /* and it is a car */
\]

\[
\text{fun}(X) : - /* or an item is fun if */
\]

\[
\text{blue}(X), /* the item is blue */
\]

\[
\text{bike}(X). /* and it is a bike */
\]

\[
\text{fun}(\text{ice\_cream}). /* and ice cream is also fun */
\]
This program says that we have three ways of finding out if something is fun. These three options are represented in Prolog by three clauses of the predicate fun. Just like we saw for pure facts, Prolog will start from the first clause (be it a rule or fact) of fun and try that. If that does not succeed, Prolog tries the next clause. Prolog only fails when it runs out of rules or facts to try.

All identically-named variables within a particular rule (e.g. all occurrences of, say, X in the first fun rule above) are constrained to have one and the same instantiation for each solution to a particular query. Identical variable names in separate rules are totally independent, just as if different variable names had been used. Because the consequences of the preceding two sentences are so fundamental to Prolog (and so often misunderstood), we recommend that you read them again.

Thus, variable name scoping is per-individual rule (often called a clause). The same variable name may appear in different clauses of a rule, or rules with different names. Each time it is treated as something specific to its context. A variable X may occur in the program many times with many different bindings.

**Examples of Rules**

Consider the following program:

```
fun(X) :- red(X),
         car(X). /*All three instances of X are identical */
fun(X) :- blue(X),
         bike(X).
car(vw_beatle).
car(ford_escort).
bike(harley_davidson).
red(vw_beatle).
red(ford_escort).
blue(harley_davidson).
```

Let's now use the above program and see if a harley_davidson is fun. To do this we can ask Prolog the following question.

```
?- fun(harley_davidson).
/* to which Prolog will reply */
yes /* to show the program succeeded */
```

To execute this query, Prolog will first see if harley_davidson is red, however only vw_beatles and ford_escorts are defined as being red. Hence, the query `red(harley_davidson)` will fail. This means that the first clause of fun will fail. As a result, we now try the second clause of fun. This will mean that we will attempt the subgoals `blue(harley_davidson)` and `bike(harley_davidson)`. Both these goals match facts in the database. As a result, fun succeeds.

We can also ask our program to find fun items for us. To do this we can pose the following question.

```
?- fun(What).
To which Prolog will reply
What=vw_beatle
yes
```
Let's see how Prolog deals with this query. First, we will try the first clause of fun. This results in us trying the goal red(What). This succeeds matching the first clause of red with the binding What=vw_beatle. Now we attempt the goal car(vw_beatle). This matches the first clause of car, and, as a result, the fun goal succeeds.

**Output**

You can also write strings to the output.

```prolog
% This is the syntax for comments.
% MORTAL - The first illustrative Prolog program
mortal(X) :- person(X).
/*The :- is called the neck. It is sometimes read as if. X is mortal if X is a person*/
person(socrates). /* Fact. Socrates is a person.*/
person(plato).
person(aristotle).

mortal_report:- /*Evaluate mortal report, by evaluating each of the following.*/
  write('Known mortals are:'),nl,
mortal(X),
  write(X),nl, /* not undone on backtracking. Won't retry as only one way to succeed */
fail. /* Because this always fails, it forces other things to be tried.*/
mortal_report. /* after first clause exhausts all possibilities, this clause makes the
  functor succeed, which is important if it is called from somewhere else */
```

Note that functors of different arities (parameter length) are different. Thus mortal(X,Y) will not be confused with mortal(X).

**Search**

**Introducing Backtracking**

When we typed a semi-colon after the system responded, we asked the system to backtrack and find another solution. We can also have backtracking in rules. For example consider the following program.

```prolog
hold_party(X):-birthday(X),happy(X).
birthday(tom).
birthday(fred).
birthday(helen).
happy(mary).
happy(jane).
happy(helen).
```

Notice that we normally keep predicates of the same name together. If we now pose the query

```prolog
?- hold_party(Who).
```

In order to solve the above, Prolog first attempts to find a clause of birthday, as it is the first subgoal of birthday. This binds X to tom. We then attempt the goal happy(tom). This will fail, since it doesn't match the above database. As a result, Prolog backtracks. This means that Prolog goes back to its last choice point and sees if there is an alternative solution. In this case, this means going back to a previous state and attempting to find another clause of birthday. This time
we can use clause two, binding X to fred. This then causes us to try the goal happy(fred). Again this will fail to match our database. As a result, we backtrack again. This time we find clause three of birthday, and bind X to helen, and attempt the goal happy(helen). This goal matches against clause 3 of our happy database. As a result, hold_party will succeed with X=helen.

**Cut**

Consider the prolog code below:

```prolog
grade(Score,a) :- Score >= 90.
grade(Score,b) :- Score >= 80.
grade(Score,c) :- Score >= 65.
grade(Score,d) :- Score >= 50.
grade(Score,f).
```

If I enter, grade(55,Grade), the system responds Grade = d. However, if I enter a semi-colon, the system will respond, Grade = f. This undesirable behavior can be fixed by adding additional tests.

```prolog
grade(Score,a) :- Score >= 90.
grade(Score,b) :- Score > 90, Score >= 80.
grade(Score,c) :- Score < 80, Score >= 65.
grade(Score,d) :- Score < 65, Score >= 50.
grade(Score,f) :- Score < 50.
```

This solves the incorrect results on backtracking, but makes the tests less efficient. Also, when an “a” grade is received, the system still backtracks through all the other rules (which all fail).

Sometimes the automatic backtracking is inefficient, and we want to limit it. The cut predicate has the effect of telling Prolog not to pass back through this point when it is looking for alternative solutions. You are saying, “If you get to this point, this is the only solution I want to consider.” Thus, the `!` acts as a marker, back beyond which Prolog will not go. When it passes this point all choices that is has made so far are `set`; i.e. they are treated as though they were the only possible choices.

Note that the cut always appears where a predicate can appear (never, for example, as arguments to a predicate). It is treated just like any other predicate, and it always succeeds. For our example, using the cut would give us:

```prolog
grade(Score,a) :- Score >= 90, !.
grade(Score,b) :- Score >= 80, !.
grade(Score,c) :- Score >= 65, !.
grade(Score,d) :- Score >= 50, !.
grade(Score,f) :- !.
```

In summary, the effect of the cut is as follows:

1. Any variables which are bound to values at this point cannot take on other values
2. No other versions of predicates called before the cut will be considered
3. No other subsequent versions of the predicate at the head of the current rule will be considered
4. The cut always succeeds.
Basically, any more answers to the current query must come from backtracking between the point of the cut and the end of the current rule.

However, the cut should be used cautiously. Consider the following code.

```prolog
succeed(Student) :-
    smart(Student), workhard(Student).
succeed(Student) :- lucky(Student).

smart(josh).
smart(ty).
smart(liz).
smart(sally).
workhard(liz).
workhard(ty).
lucky(joe).
lucky(josh).

The query succeed(Who) yields (on backtracking) ty, liz, joe, josh.

If I insert a cut as shown below, the query succeed(Who) yields “no”.

```prolog
succeed(Student) :-
    smart(Student), !, workhard(Student).
succeed(Student) :- lucky(Student).
```

Let us see why this is so. Who initially matches “joe” and then the cut is reached. Since workhard(joe) does not succeed and backtracking is prohibited by the cut, the rule fails.

Try putting the cut at the end.

```prolog
succeed(Student) :-
    smart(Student), workhard(Student),!.
succeed(Student) :- lucky(Student),!.
```

The query succeed(Who) yields “ty” only. However, succeed(liz), succeed(josh), and succeed(joe) all give “yes”.

In summary, the cut can be useful, but generally, you should NOT use it.

**Recursion**

As is commonly the case in many programming tasks, we often wish to repeatedly perform some operation either over a whole data-structure, or until a certain point is reached. The way we typically do this in Prolog is by recursion. This simply means a program calls itself typically until some final point is reached. Frequently in Prolog what this means is that we have a first fact that acts as some stopping condition followed up by some rule(s) that performs some operation before reinvoking itself.

Consider an ancestral example.

```prolog
parent(john,paul).  /* paul is john's parent */
parent(paul,tom).   /* tom is paul's parent */
parent(tom,mary).   /* mary is tom's parent */
```
ancestor(X,Y):- parent(X,Y). /* someone is your ancestor if they are your parent */
ancestor(X,Y):- parent(X,Z), /* or somebody is your ancestor if they are the parent */
        ancestor(Z,Y). /* of someone who is your ancestor */

The above program finds ancestors, by trying to link up people according to the database of parents at the top to the card. So let's try it out by asking

?- ancestor(john,tom).

The first clause of ancestor looks to see if there exists a clause that could match the goal parent(john,tom). This fails to match, as a result we try the second clause of ancestor. We now pose the query parent(john,Z). This results in us choosing clause one of parent and binding Z=paul. As a result, we now pose the recursive query ancestor(paul,tom). Applying the ancestor rule again, we first try the first clause. This means we check for parent(paul,tom). which successfully matches the second clause of parent. As a result the goal ancestor(paul,tom) succeeds. This in turn leads to the goal ancestor(john,tom) succeeding and Prolog responding yes.

**Infinite Recursion**

Consider our example above, but interchange the order of the goals for the second ancestor clause.

parent(john,paul).
pARENT(paul,tom).
pARENT(tom,mary).
ancestor(X,Y):- parent(X,Y).
ancestor(X,Y):-
        ancestor(Z,Y),parent(X,Z).

Notice, when you query ancestor(john,Who), you get the same series of answers as before (paul, tom, mary), but then the recursion goes into an infinite series of calls.

If the two ancestor clauses are reversed (as shown below), the recursion produces no results at all before going into an infinite series of calls.

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

The lesson to be learned is this: With recursion, take a small step towards the answer before plunging into a recursive call.

**Recursion Exercise**

Given facts such as
Bob is taller than Mike.
Mike is taller than Jim
Jim is taller than George
Write a recursive program that will determine that Bob's height is greater than George's.
Recursion Exercise

logans→riverheights→providence→millville→nibley→hyrum

A one way road links 6 towns (as shown above). Write a program that can work out if you can travel on that road between two specific cities. For example. Here are two sample program behaviors.

?- can_get(logan, millville).
yes

?- can_get(providence, logan).
no

Lists
So far we have only considered simple items as arguments to our programs. However, in Prolog a very common data-structure is the list. The internal storage of a list is important for a variety of reasons. Not only does the internal storage dictate which operations are intrinsic, but it also provides for efficient garbage collection (as all cells are the same size). Lists are recursive and are stored as “first element” and “rest of list” (which is a list). Because of this representation, finding the first element of a list is a basic operation, but finding the last element requires searching the entire list. Since basic cells are the same size, garbage collection is easier.

Lists themselves have the following syntax. They always start and end with square brackets, and each of the items they contain is separated by a comma. Here is a simple list
[a, freddie, A_Variable, apple]
We could add to our eats information by using a list as follows:
eats(liz, [pizza, garlicbread, sprite, salad]).

Prolog also has a special facility to split the first part of the list (called the head) away from the rest of the list (known as the tail). We can place a special symbol | (pronounced 'bar') in the list to distinguish between the first item in the list and the remaining list. For example, consider the following.
[first, second, third] = [A|B]
where A = first and B=[second, third]
The unification here succeeds. A is bound to the first item in the list, and B to the remaining list.

List Example 1
Here are some example simple lists

[a, b, c, d, e, f, g]
[apple, pear, bananas, kiwi]
[ ] /* this is a special list, it is called the empty list because it contains nothing */

Now let's consider some comparisons of lists:
[a, b, c] unifies with [Head|Tail] resulting in Head=a and Tail=[b, c]
[a] unifies with [H|T] resulting in H=a and T=[]
[a, b, c] unifies with [a|T] resulting in T=[b, c]
[a, b, c] doesn't unify with [b|T]
[] doesn't unify with [H|T]
[] unifies with []. Two empty lists always match

**List Example 2**
Consider the following fact.

\[
p([H|T], H, T).
\]

Let's see what happens when we ask some simple queries.

\[
?- p([a,b,c], X, Y).
X=a\nY=[b,c]\nyes
\]

\[
?- p([a], X, Y).
X=a\nY=[]\nyes
\]

\[
?- p([], X, Y).
no
\]

**List Searching**

We can use lists within facts and rules. One common way of using lists is to store information within a list and then subsequently search for this information when we run our programs. In order to search a list, Prolog inspects the first item in a list and then goes on to repeat the same process on the rest of the list. This is done by using recursion. The search can either stop when we find a particular item at the start of the list or when we have searched the whole list, in which case the list to be searched will be the empty list. In order to do this, we have to be able to selectivly pull the list apart. We have already seen how we can go about doing this. In the previous section, we showed how to take the head and a tail of a list:

\[
[\text{Head}|\text{Tail}]
\]

This method constitutes the basis of the searching method. We shall use it to pull apart a list, looking at the first item each time, recursively looking at the tail, until we reach the empty list [], when we will stop.

**List Searching: Example**

Consider the following problem. How can I see if a particular item is on a particular list? For example, I want to test to see if item apples is on the list [pears, tomatoes, apples, grapes]. One possible method of doing this is by going through the list, an item at a time, to see if we can find the item we are looking for. The way we do this in Prolog is to say that we could definitely prove an item was on a list if we knew that the target item was the first one on the list. i.e.

\[
on(\text{Item},[\text{Item}|\text{Rest}]). /* is the target item the head of the list */
\]
Otherwise we could prove something was on a list if we could prove that although it didn't match the existing head of the list, it nonetheless would be on the remaining list if we disregarded the first item and just considered the rest of the list i.e.

```
on(Item,[DisregardHead|Tail]):-  
on(Item,Tail).
```

We now have a program consisting of a fact and a rule for testing if something is on a rule. To recap, it sees if something is the first item in the list. If it is we succeed. If it is not, then we throw away the first item in the list and look at the rest.

**Exercises:** CountAll: counts the number of things in a list. Our solution is [here](#).
Factor: creates a list of factors of a number. Our solution is [here](#).
MyReverse: reverses the list. Our solution is [here](#).
HalfList: create a list containing every other element of a list. Our solution is [here](#).
InsertNew: insert an element only if not in original list. Our solution is [here](#).
Flatten: removes all nesting levels of the list. Our solution is [here](#).

**List Construction.**
The last example showed how we could strip off the front items on a list and recursively search the rest. We can also use the same method to build lists. For example take one existing list, say List1. We can make a new list List2 which contains List1 but with the new head mynew, as follows:

```
List2 = [mynew|List1]
```

This again is pure unification. List2 now consists of the head prolog plus whatever List1 was bound to. If List1 were to be bound e.g. List1 = [list,c,pascal,basic], List2 would now be the list [mynew,lisp,c,pascal,basic].
We can use this construction technique to build lists during recursion. We'll present three examples of this.

**List Construction Example**
An example use of list construction is when we wish to create a new list out of two existing lists. We'll illustrate this by defining a predicate called append that takes three arguments. The first two are lists. The predicate uses these two lists to produce a third list which combines the original two. For example,

```
?- append([a,b,c],[one,three],Result).
Result = [a,b,c,one,three]
```

*Note, append expects two lists as arguments. If you use it with simple arguments, such as append(one,two,Result) you get something quite different Result=[one|two]*

The way we append in Prolog uses both list construction and list deconstruction. Recall that previously we saw how we could glue an item onto the front of a list to produce a new list. Thus, we can produce the list [c,one,three] from the list [one,three] by saying NewList = [c|[one,three]]. This results in NewList being bound to the list [c,one,three]. This is the clue to how we write append in Prolog. We can recursively split the first list into single items by
the deconstruction techniques discussed earlier. We can then use the construction method above to glue together a new list, which we shall illustrate next.

Note, we can’t do \([a|b c d]\) as after the bar you must have a list.

In order to write append, we shall first search through the first list, and taking each item in turn, add it to the second list. This we'll do recursively, by searching for the end of the first list, and then adding the items in the first list to the items in the second list in reverse order. Thus, the last item in the first list is the first to be added to the second list. We illustrate this more clearly with an example. Let’s see what the code to do this look like. First, we must deal with the case when the first list is an empty list. In this case, the result of appending the two lists will just be the second list. Thus this gives

\[
\text{append}([], \text{List}, \text{List}).
\]

\[
\text{append}([\text{Head}|\text{Tail}], \text{List2}, [\text{Head}|\text{Result}]):-
\]

\[
\text{append}(\text{Tail}, \text{List2}, \text{Result}).
\]

Notice, the functor we want it to match first (as a base case) must be listed first.

**List Construction Example 2**

Let's now consider a second example of constructing lists. Taking the list \([1,12,3,14,5,8]\), let’s construct a new list out of the old list that contains only numbers greater than 6. To do this we can search the list as we have seen before. If the item passes our test, then we can add it to the output list. If it does not, then we can discard the item. To make a start, if the list to search is empty, we will return the empty list. Hence, we get the base case of the recursion.

\[
\text{sift}([], []).
\]

Next we need to say that if the item passes our test, put it in the output list

\[
\text{sift}(\text{[X|T]}, \text{[X|Result]}):- \ X > 6, \ /*\ is\ X\ greater\ than\ 6 */
\]

\[
\text{sift}(\text{T}, \text{Result}). \ /*\ if\ so\ then\ go\ find\ the\ rest */
\]

Otherwise we are will discard the head and look for other hits in the tail

\[
\text{sift}(\text{[ThrowAway|Tail]}, \text{Result}):- \ /*\ discard\ the\ head */
\]

\[
\text{sift}(\text{Tail}, \text{Result}). \ /*\ and\ look\ in\ the\ tail */
\]

Let us now see how our program responds to the query

\[
?- \text{sift}([1,12,3,14,5,8], \text{Result}).
\]

To this goal, we first unify with clause 2. The result is to evaluate the subgoal \(1 > 6\), this clearly fails, so we move on to to the third clause and try the goal \(\text{sift}([12,3,14,5,8], \text{Result})\). This again matches on the second clause of sift. We now try the subgoal \(12 > 6\), this succeeds, and we now attempt the goal \(\text{sift}([3,14,5,8], \text{Result})\). However, notice what also happens. By using this clause, we also place 12 on our output list. Stepping further through our example, we see that greater than subgoal in clause 2 succeeds for 14 and 8, till finally we get the goal \(\text{sift}([], \text{Result})\). This matches the first goal, with Result bound to \([],\). As we come out of the recursion, we see that clause 2 builds up the result for to the list \([8]\), then \([14,8]\), then finally \([12,14,8]\), before the query finally succeeds.
Creating lists from structures

We can go between a structure and a list by the operator =.. which is used as “structure =..List”. Note, the predicate \textit{atom(Item)} returns true if the Item is not a structure or a list.

For example:
\begin{verbatim}
eats(tony, meal(appetizer(mushrooms), main(turkey, potatoes, broccoli), dessert(pie))).
eats(liz, [pizza, garlicbread, sprite, salad]).
eat(Who) :-
    eats(Who, What),
    write(Who), nl, writeList(What).
writeList([]).
writeList(What) :- atom(What), write(What), nl.
writeList([H|T]) :- writeList(H), writeList(T).
writeList(What) :- What =.. [H|T], writeList([H|T]).
?- eat(liz).
liz
pizza
garlicbread
sprite
salad
?- eat(tony).
tony
meal
appetizer
mushrooms
main
turkey
potatoes
broccoli
dessert
pie
\end{verbatim}

List Construction Exercise 1

Write a program to delete all references of a particular item from a list. It should have three arguments. The list you wish to use, the item to delete, and the resulting list. Here are some example of it behavior

\begin{verbatim}
?- delete_all([a,b,a,c,a,d],a,Result).
Result = [b,c,d]
?- delete_all([a,b,a,c,a,d],b,Result).
Result = [a,a,c,a,d]
?- delete_all([a,b,a,c,a,d],prolog,Result).
Result = [a,b,a,c,a,d]
\end{verbatim}

Once you have attempted a solution, compare it to our solution.

List Construction Exercise 2

Write a program to replace all occurrences of one item in a list with another. It should have four arguments: (1) the input list, (2) the item to replace, (3) the item to replace it with, and (4) the resulting list. Here are some example of its behavior

\begin{verbatim}
?- delete_all([a,b,a,c,a,d],a,Result).
Result = [b,c,d]
?- delete_all([a,b,a,c,a,d],b,Result).
Result = [a,a,c,a,d]
?- delete_all([a,b,a,c,a,d],prolog,Result).
Result = [a,b,a,c,a,d]
\end{verbatim}
Once you have attempted a solution, compare it to our solution.

**Updating the Data Base**

You can update the data base of facts as follows:

asserta(X). Adds the clause X as the first clause for its predicate. (Will not be undone on backtracking)
assertz(X). same as asserta only it adds the clause X as the last clause for its predicate.
retract(X). Removes X from the data base.
abolish(Name,Arity). removes all predicates with Name and Arity from dynamic data base.
Read about these options in the tutorial that comes with AMZI Prolog.

To save the contents of your dynamic data base, simply type

tell("myout.pro"). // sets output to go to file myout.pro.
listing. // does a listing of all current predicates, sending output to myout.pro.

**Equality Operators**

The operator is used to assign arithmetic values:
toFarenheight(C,F) :- F is C*9/5+32

The operator = does unification.
The operator == tests if two operands unify with no variable bindings taking place.
The operator =:= checks for arithmetic equality.

**Underscore**

The variable _ is used when we don’t care to reference the variable specifically. It is a no-name variable.
So, for example, if we want to replace X with Y in a list, we could code it as:

replace(_,_,[],[]).
replace(X,Y,[X|Rest],Y[XRest]):-
    replace(X,Y,Rest,XRest).
replace(X,Y,[Z|Rest],Z[XRest]):- replace(X,Y,Rest,Xrest).