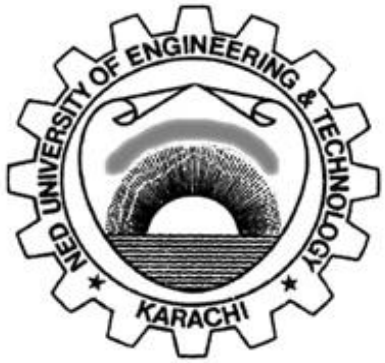
Workbook

Artificial Intelligence & Expert System (CT – 361)



Name _____

Roll No _____

Batch _____

Year _____

Department _____

Department of Computer Science & Information Technology
NED University of Engineering & Technology

Workbook

Artificial Intelligence & Expert System
(CT – 361)

Prepared by

Ms. Nazish Irfan
I.T Manager, CS&IT

Approved by

Chairman
Department of Computer Science & Information Technology

Table of Contents

S. No	Object	Page No	Signatures
1.	Introduction to PROLOG.	01	
2.	Understanding facts, its type and how to query.	07	
3.	Unifications.	10	
4.	Understanding Rules.	13	
5.	Input & Output Commands.	16	
6.	Operators – Arithmetic and Comparison.	19	
7.	Data types in PROLOG.	25	
8.	Lists and how to manipulate them.	32	

Note: The following software is used: WinProlog (Free version).

Lab # 1

Object:

Introduction to PROLOG.

.

Theory:

Introduction.

Prolog was invented in the early seventies at the University of Marseille. Prolog stands for PROgramming in LOGic. It is a logic language that is particularly used by programs that use non-numeric objects. For this reason it is a frequently used language in Artificial Intelligence where manipulation of symbols is a common task. Prolog differs from the most common programming languages because it is declarative language. Traditional programming languages are said to be procedural. This means that the programmer specify how to solve a problem. In declarative languages the programmers only give the problem and the language find himself how to solve the problem.

What is a Prolog program?

Programming in Prolog is very different from programming in a traditional procedural language like Pascal. In Prolog you don't say how the program will work. Prolog can be separated in two parts:

The Program.

The program, sometimes called Database is a text file (*.pl) that contain the facts and rules that will be used by the user of the program. It contains all the relations that make this program.

The Query.

When you launch a program you are in query mode. This mode is represented by the sign ? - at the beginning of the line. In query mode you ask questions about relations described in the program.

Loading a program.

First you have to launch your Prolog compiler. When Prolog is launched the ?- should appear meaning you are in query mode. The manner to launch a program depends of your compiler. We can load a program by typing the command consult[filename]. When you have done this you can use all the facts and rules that are contained in the program.

Facts and Rules.

A prolog program consists of facts and rules. The following are some examples of facts.

car(rusty).
car(dino).
colour(rusty, blue).
colour(dino, red).

These facts state that rusty and dino are both cars. They also state the colour of each car. Notice that a fact has two parts, a property or attribute part such as car or colour and an object part, such as rusty or dino. In prolog, the property is called the functor. The prolog syntax requires that every fact be terminated by a full stop.

A rule is more complex than a fact, roughly speaking, it relates various facts. The following is an example of a rule relating the facts given above.

expensive(Thing) :- car(Thing), colour(Thing, red).

This rule states that a Thing is expensive (has the property expensive) if the Thing is a car (has the car property) and the Thing is red (has the red property).

Notice that :- means if and , means and. A rule has a head, the part before the if, and a body, the part after the if.

Collectively, facts and rules are known as **clauses**. A collection of clauses constitute a **program**, i.e.

car(rusty).
car(dino).
colour(rusty, blue).
colour(dino, red).
expensive(Thing) :- car(Thing), colour(Thing, red).

The above clauses would be written in a file and the file loaded into the prolog interpreter. When a prolog program executes, it calculates the properties of object, that is all it does. In the above program, some of the properties of some objects can be calculated very easily. For example, the fact that dino has the car property is stated directly in the program. Prolog can be asked to calculate this property of dino as follows. Within the prolog interpreter, type car(dino). at the ?- prompt. The ?- prompt is also known as the **query prompt.**

?- car(dino).
After typing return, prolog replies.
Yes

It is important to understand in detail how prolog is able to produce the answer Yes in response to the query car(dino).. The interpreter reads the query and extracts the functor (property) car and the object dino. It then goes through each line of the program, always starting from the first line to find a fact that has the same functor as the query, i.e. car. Prolog finds the fact car(rusty).. The functor of this fact is the same as that in the query but the object does not match. As a result, this fact does not match the query and prolog moves on to the next line of the program. On the next line, it finds car(dino). which does match the query and hence prolog replies Yes.

If the query had been
?- car(rover).

prolog would not have been able to find a match in any of the rows of the program and would therefore have replied.
No

Suppose that the user does not know which objects are cars, i.e. the user cannot enter the query car(dino). because although he is interested in the car property but does not know the names of any of the cars in the program. The best query that the user can manage is.

?- car(X).

where X is a variable and stands for any object that has the car property. In prolog, variables are always written with an upper case initial letter. Objects or constants are written with a lower case initial letter. In the prolog rule above, Thing is a variable because of the initial upper case T.

In response to the above query, prolog replies.
X = rusty
Yes

Again, prolog does this by reading the query, extracting the functor and variable. Starting from the first line of the program, the functor in the query is matched against that of the fact or rule in the program. If the functor matches, then the variable of the query is matched against the constant rusty. Since the variable does not yet have a value, it can be given the value rusty and as a result a match is produced. If the variable already had a value, different to rusty then it would not have matched.

It is important to note that a variable without a value may be given a value when there is an attempt to match it. A variable without a value is called unbound. A variable with a value is called bound. So in the above calculation, two steps take place, firstly the unbound variable X is bound to rusty and then the variable, with the value rusty is successfully matched to rusty.

Looking at the program, rusty is not the only car. How can the user find out that dino is also a car? The prolog interpreter allows the user to remove the value of a variable, i.e. to make in unbound, and to force prolog to find another value for the variable that will match. To do this, prolog always remembers at which point in the program a variable was bound to a value. It remembers, for example, that X was bound to rusty at the first line of the program.

If the user enters ; prolog goes to the point at which the variable X was given the value rusty, removes the value rusty so that X is unbound and moves to the next line of the program so that the same match cannot be made again. The interpreter now finds the fact car(dino). Again, X is unbound and so it can be bound to dino, at which point the query matches the fact and prolog outputs.
X = dino
Yes

It the user once more enters ; in order to ask for a different match, the binding of X to dino is removed and the search for a match starts from the line below where the dino binding was made. No more matches can be found and so prolog outputs No.

Whenever, there is a choice about the value that can be bound to a variable, there is another branch in the tree. Prolog execution is best understood by constructing by hand the search tree of matches. The user can query the program about the colour of various cars. The query colour(rusty, blue). will return Yes because the functor and both constants match. The query colour(rusty, red). will return No because the functor and only one constant matches. The query colour(rusty, X). will return X = blue Yes and the query colour(Y, X). will return Y = rusty, X = blue and if requested, also Y = dino, X = red.

Another query the user may enter is expensive(rusty).. This query matches expensive(Thing) because the variable Thing is unbound and is consequently bound to rusty. Prolog cannot print Yes however, because it does not know that rusty is expensive, it knows only that a Thing is expensive if it is a car and that car is red. Whenever the head of a rule is matched, prolog tries to match the body of the rule.

In the example, expensive(Thing) in the program is matched to the query expensive(rusty). And so prolog tries to match car(Thing) where Thing is bound to rusty. We have worked through this query before. Queries that prolog initiates are called goals. In fact all queries are often called goals. We would say that expensive(Thing) is a goal and that car(Thing) is a subgoal. To satisfy the subgoal, prolog starts at the first line (as always) and finds the fact car(rusty)., i.e. a match because the variable Thing has the value rusty. The rule body has another part that must also be matched. Recall that , means and. The subgoal that prolog now tries to match is colour(Thing, red). where Thing is bound to rusty. This subgoal will fail and prolog will print No. Again, it is not necessary for the user to try to guess the names of expensive cars and try each one. The user may enter expensive(X).. This query matches expensive(Thing). This raise an interesting situation. There are two variables, X and Thing, neither of which are bound. The rule in prolog is that two unbound variables share. This means that they can be given a binding but only one binding that both of them must share. In effect, when variables share they are made to have equal values, if and when they are given values.

To satisfy the head of a rule, i.e. expensive(Thing), the body of the rule must be satisfied. Prolog now tries to match car(Thing) where Thing is unbound but shared with X. We have worked through this query before. Prolog starts at the first line (as always for a new goal) and finds the fact car(rusty)., i.e. a match because the variable Thing can be given the value rusty. Recall that Thing shares with X and so X also has the value rusty.

The rule body has another part that must also be matched because , means and. The subgoal that prolog now tries to match is colour(Thing, red). where Thing is bound to rusty. This subgoal will fail. In the previous example, Thing was bound to rusty because the user had put rusty in the query. Obviously, prolog will not try to replace the user's constant values in a query. In the current example, however, the user query contains a variable X, which prolog currently has bound to rusty. This binding can be undone and a new search made continuing on from where X was given the value

rusty. This is what happens now. Prolog will unbind the last variable binding, in this case, Thing is bound to rusty, and try to find another match for car(Thing). The search for a new match starts from just after the point where the old match was made. A match is found with Thing bound to dino. The next subgoal of the rule body is now rematched, from the first line of the program. This is the subgoal colour(Thing, red). where Thing is bound to dino. This subgoal is matched, we also say it succeeds, and when the body of a rule succeeds, so does the head. This is to say that expensive(Thing) succeeds and Thing is bound to dino.

Exercise:

1. Modify the example given above and add another fact and rule stating that if the car is white it is cheap and its name is dumbo.

Lab # 2

Object:
Understanding facts, its type and how to query.

Theory:
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 full stop. 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.

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>).

The arguments can be any legal Prolog term. The basic Prolog terms are an integer, an atom, a variable or a structure. Various Prolog implementation enhance this basic list with other data types, such as floating point numbers, or strings.
Example:
likes(john,mary).

In the above fact john and mary are two atomes. Atoms are usally made from letters and digits with lowercase characters. The underscore (_) can also be used to separe 2 words but is not allowed as the first character. Atoms can also be legally made from symbols. The followings are legal atoms:
hello
zz42
two_words

The followings are not legal atoms:
Hello
4hello
_Hello
two words
two-words

You can use single quotes to make any character combination a legal atom.
'two words'
'UpperCase'
'12444'

The fact likes(john,mary). say that there is a relation between john and mary. It can be read as either john likes mary or mary likes john. This reversibility can be very useful to the programmer, however it can also be a source of mistakes. You have to be clear on how you intend to interpret the relation.

The number of arguments is the arity of the predicate. A predicate with 2 arguments will be called by predicate_name/2. You can have different predicates with the same name if they have a different arity.

How to query.
Once you have entered the facts in a program you can ask prolog about it. An example program can be:

```
eats(fred,oranges).        /* 'Fred eats oranges' */
eats(tony,apple).          /* 'Tony eats apple'   */
eats(john,apple).          /* 'John eats apple'   */
```

If we now ask some queries we would get the followings things :

```
?- eats(fred,oranges).     /* does this match anything in the database? */
yes                        /* yes, that matchs the first clause */
?- eats(john,apple).
yes
?- eats(mike,apple).
no                          /* there is no relation between mike and apple */
```

Exercise:

1. Enter the above program into Prolog and execute the queries shown below:

   Facts & Rules

   ring(Person, Number) :- location(Person, Place), phone_number(Place, Number).
   location(Person, Place) :- at(Person, Place).
   location(Person,Place) :- visiting(Person, Someone), location(Someone, Place).

   phone_number(rm303g, 5767).
   phone_number(rm303a, 5949).
   at(dr_jones, rm303g).
   at(dr_mike, rm303a).
   visiting(dr_mike, dr_jones).

   Queries.

   ?- location(dr_bottaci, Pl).
   ?- ring(dr_mike, Number).
   ?- ring(Person, 5767).
   ?- ring(Person, Number).
   ?- ring(dr_jones, 999).

2. Consider the question provided in exercise of LAB#2. Assume that dr_mike leaves the office of dr_jones and instead visits the office of dr_frankenstein. Modify the above program to represent this new situation. You will need to make up some room and phone numbers. Test the program.

Lab # 3

Object:
Unifications.

Theory:
Simples unifications.
How can we ask something like "what does Fred eat?" If we have the following program:
eats(fred,oranges).

How do we ask what fred eats ? We could write something like this :
?- eats(fred,what).

But Prolog will say no. The reason is that Prolog can't find the relation eats(fred,what) in its database. In this case we have to use a variable which will be unified to match a relation given in the program. This process is known as unification. Variables are distinguished from atoms by starting with a capital letter. Here are some examples of variables:

```
X                   /* a single capital letter*/
VaRiAbLe            /* a word beginning with an upper case letter */
Two_words           /* two words separated with an underscore     */
```

Now that we know how to use a variable, we can ask the same question as before using the variable what instead of an atom.

```
?- eats(fred,What)
What=oranges
Yes
```

In this case Prolog try to unified the variable with an atom. "What=oranges" means that the query is successful when what is unified with oranges. With the same program if we ask:
?- eats(Who,oranges).

In this example we ask who eats oranges. To this query Prolog should answer :

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

Now if we ask:

```
?- eats(Who,apple).
No
```

Prolog answer no because he can't find in his database any atom than can match with this relation. Now if we only want to know if something is eated by anyone and we don't care about that person we can use the underscore. The '_' can be used like any variable. For example if we ask eats(fred,_) the result will be:
?- eats(fred,_).

Yes

The result will be yes because Prolog can find a relation of eat between fred and something. But Prolog will not tell use the value of '_'.
Now if we have the following program:
eats(fred,apple).
eats(fred,oranges).

Now if we ask:
?- eats(fred,What).

The first answer will be "What=apple" because that is the unification that match the first relation of eats with fred in the database. Then prolog will be waiting for you to press a key. If you press enter Prolog will be ready for a new query. In most implementation if you press the key ";" then Prolog will try to find if there is any other successful unification. Prolog will give the second result "What=orange" because this the second one in the program. If you press again the key ";" then Prolog will try to find a third unification. The result will be "no" because he is not able to find any other successful unification.
?- eats(fred,What).
What=apple;
What=oranges;
no

Variables unification example.
Consider this example of program that could be used by a library:
book(1,title1,author1).
book(2,title2,author1).
book(3,title3,author2).
book(4,title4,author3).

Now if we want to know if we have a book from the author2 we can ask:
?- book(_,_,author2).
yes

If we want to know which book from the author1 we have:
?- book(_,X,author1).
X=title1;
X=title2;

Exercise:

1. Write a program to implement the following:
   likes(fred,cola).
   likes(fred,cheap_cigars).
   likes(fred,monday_night_football).

   likes(sue,jogging).
   likes(sue,yogurt).
   likes(sue,bicycling).
   likes(sue,noam_chomsky).

   likes(mary,jogging).
   likes(mary,yogurt).
   likes(mary,bicycling).
   likes(mary,george_bush).

   Queries:
   ?- likes(fred,cola).
   ?- likes(fred,X).
   ?- likes(fred,X).
   ?- likes(Y,jogging).

Lab # 4

Object:
Understand Rules.

Theory:
Rules.
Consider the following sentence:
'All men are mortal'.

We can express this thing in Prolog by:
mortal(X) :- human(X).

The clause can be read as 'X is mortal if X is human'. To continue with this example, let us define the fact that Socrates is a human. Our program will be:
mortal(X) :- human(X).
human(socrates).

Now if we ask to prolog:
?- mortal(socrates).
Prolog will respond:
yes

In order to solve the query -? mortal(socrates). Prolog will use the rule we have given. It says that in order to prove that someone is mortal we can prove that he is human. So from the goal mortal(socrates) Prolog generate the sub goal human(socrates). We can still use variables. For example we might want to know who is mortal:
?- mortal(X).
Then Prolog should respond:
P=socrates

This means that Prolog was able to succeeds the goal by unifying the variable X to socrates. Again this was done by using the sub goal human(X). Sometimes we may wish to specify alternatives ways to prove something. We can do this by using different rules and facts with the same name. For example, we can represent the sentence 'Something is fun if it is a PC running UNIX or am old amiga or an ice cream' with the following program:

```
fun(X) :-                       /* something is fun if      */
        pc(X),                  /* it is a pc and           */
        unix(X).                /* it is running unix       */
fun(X) :-                       /* or it is fun if          */
        old(X),                 /* it is old and            */
        amiga(X).               /* it is an amiga           */
fun(ice_cream).                 /* the ice_crean is also fun */
```

This program says that there are three ways to know if an object is fun or not. Like for pure facts, Prolog will start from the first clause (a clause can be a rule or a fact) of fun and try it. If that does not succeeds Prolog will try the next clause. If there is no

more clauses then it fails and Prolog responds ' no '. We can also see in this example that the 'and' is represented by a ',' and the 'or' by different clause. If needed the 'or' can also be represented by ';'. In the previous examples when we was asking eats(fred,What) and pressing the key ';' to see the following results we was in fact asking 'or'. All identically-named variables in a rule (for example X in the last rule we've seem) are of course considered as one unique variable and must have the same instantiation for each solution in a particular query. Identical variables names in different rules are considerate as different variables and are totally independent, this is the same as if we had used different names. The following program:

```
fun(X) :-
        pc(X),
        unix(X).
fun(X) :-
        old(X),
        amiga(X).
```

Will be seen by Prolog as:

```
fun(X_1) :-
            pc(X_1),
            unix(X_1).

fun(X_2) :-
            old(X_2),
            amiga(X_2).
```

How to add a rule with a program.

It is possible to add new rules or facts with the instruction Assert(fact1) which will add the fact called fact1. The predicates added with this command are considerate like any other in the source of the program.

The instructions.

The useful instructions are:

assert(c).	Add the rule c in the database.
retract(c).	Remove the c from the database.
asserta(c).	Add c at the beginning of the database.
assertz(c).	Add c at the end of the database.

Example.

```
?- sunny.
no.
?- assert(sunny).
yes.
?- sunny.
yes

?- retract(sunny).
yes.
?- sunny.
no.
```

Exercise:

1. Write a prolog program to create a family tree by creating facts and rules based on the information given below:

   Parveen is the parent of Babar.
   Talib is the parent of Babar.
   Talib is the parent of Lubna.
   Parveen is the parent of Lubna.
   Talib is male.
   Babar is male.
   Parveen is female.
   Lubna is female.

Lab # 5

Object:
Input & Output Commands.

Theory:
At this time we have seen how we can communicate with prolog using the keyboard and the screen. We will now see how we can communicate with prolog using files.

Prolog can read and write in a file. The file where Prolog read is called input and the file where Prolog write is called output. When you run Prolog the default output is your screen (the shell) and the input is your keyboard. If you want to use files for that you have to tell it to Prolog using commands.

Read and Write.
Sometimes a program will need to read a term from a file or the keyboard and write a term on the screen or in a file. In this case the goals write and read can be used.
read(X).
Read the term from the active input and unified X with it.
write(Y).
Write the term Y on the active output.

Examples.
Calculating the cube of an integer.
If we have the following program:
*cube(C,N) :- C is N * N * N*

If you ask something like cube(X,3) then Prolog would respond X=9. Suppose that we want to ask for other values than 3, we could write this program like this:

```
cube :-
        read(X), calc(X).                    /* read X then query calc(X). */
        calc(stop) :- !.                     /* if X = stop then it ends   */
        calc(X) :- C is X * X * X, write(C),cube.   /* calculate C and write it then
                                                       ask again cube. */
```

Now if we ask Prolog cube, Prolog will ask use a value and give us the result until we write stop.

Treating the terms of a file.
If we wanted to treat each term of a file, we could use this query:
?- see(filename), treatfile.\index{see(filename)}

using the following program:
read(Term)
treatfile :- read(Term), treat(Term).

```
        treat( end_of_file ) :- !.
        treat(Term) :- treatterm(Term),treatfile.
treatterm :- [the treatment you want to do to each term.]
```

ASCII characters.
It is of course possible to use ASCII code in Prolog. For example if you type:
?- put(65), put(66), put(67).

Prolog will write ABC.

We can also write:
?- put('A'), put('B'), put('C').

The output will be similar.

Tab.
The built-in predicate tab(N) causes N spaces to be output:

```
?- write(hi), tab(1), write(there),nl.
hi there
true.

?- write(hi), tab(15), write(there),nl.
hi              there
true.
```

Using another program.
It is possible to load another Prolog program using a program. Two predicates have been made for this:

consult(program1) to add in the database all the predicates contained in program1 and reconsult(program2)to reload the predicates contained in program2.

Exercise:

1. Write code in prolog to generate the following output:
   5 plus 8 is 13.
   X=13

   3 multiply by 3 is 9.
   X = 9.

2. Write code in prolog to generate table of any number.

3. Write code in prolog to print your name and roll no on screen.

Lab # 6

Object:
Operators – Arithmetic and Comparison.

Theory:
For the arithmetic operators, prolog has already a list of predefined predicates. These are:
*=, is, <, >, =<, >=, ==, =:=, /, *, +, -, mod, div*

In Prolog, the calculations are not written like we have learned. They are written as a binary tree. That is to say that:
*y*5+10*x* is written in Prolog as *+(*(y,5),*(10,x)).*

But Prolog accept our way of writing calculations. Nevertheless, we have to define the rules of priority for the operators. For instance, we have to tell Prolog that * has higher priority than +..Prolog allows the programmer to define his own operators with the relations next:
Op(P, xfy, name).

where P is the priority of the operators(between 1 and 1200), xfy indicates if the operator is infix(xfx,xfy,yfx) or postfix(fx,fy).The name is, of course, the name of the operator. Note that the prior operator has the lowest priority. Prolog has already these predefined operators:
Op(1200,xfx,':-').
Op(1200,fx,[:-,?-]).
Op(1100,xfy,';').
Op(1000,xfy,',').
Op(700,xfx,[=,is,<,>,=<,>=,==,=:=]).
Op(500,yfx,[+.-]).
Op(500,fx,[+,-,not]).
Op(400,yfx,[,/,div]).*
Op(300,xfx,mod).

Arithmetic Operators
Prolog use the infix operator 'is' to give the result of an arithmetic operation to a variable.
X is 3 + 4.
Prolog responds.
X = 7
yes

When the goal operator 'is' is used the variables on the right must have been unified to numbers before. Prolog is not oriented calculations, so, after a certain point, it approximates the calculations and doesn't answer the exact number:

?- X is 1000 + .0001
X = 1000
yes

Comparison Operators.
Comparison operators can be classified into Arithmetic and term comparison operators.

Less than or equal to(=<).
Compares *Arg1* to *Arg2*, succeeding if *Arg1* is less than or equal to *Arg2*. Both the arguments must be a number (integer or float) or arithmetic expression.
Arg1=< Arg2
Example:
?- 1=<2.
yes
?- 1=<1.
yes
?- 2=<1.
no
?- 1.5=<2.
yes
?- 2=<1+2.0.
Yes

Less than (<).
Compares Arg1 to Arg2, succeeding if Arg1 is less than Arg2. Both the arguments must be a number (integer or float) or arithmetic expression.
Arg1< Arg2
Example:
?- 1<2.
yes
?- 1<1.
no
?- 1.0<2.
yes
?- 2+3<6.
Yes
?- 5<3+1.
no

Greater than or equal to(>=).
Compares *Arg1* to *Arg2*, succeeding if *Arg1* is greater than or equal to *Arg2*. Both the arguments must be a number (integer or float) or arithmetic expression.
Arg1>= Arg2

Example:
?- 2>=1.
yes
?- 2>=3.
No
?- 3+4>=7.9.
no
?- 7.5>=7+0.5.
yes

Greater than (>).
Compares Arg1 to Arg2, succeeding if Arg1 is greater than Arg2. Both the arguments must be a number (integer or float) or arithmetic expression.
Arg1> Arg2
Example:
?- 2>1.
Yes
*?- 3*3>10.*
no
?- 10>8+1.
yes
?- 11.5>10+1.
yes

Equals to (=:=).
Compares *Arg1* to *Arg2*, succeeding if *Arg1* and *Arg2* are equal. Both the arguments must be a number (integer or float) or arithmetic expression.
Arg1=:= Arg2
Example:
?- 1=:=1.
yes
?- 1=:=2.
no
?- 2+3=:=5.
yes
*?- 12=:=6*2.*
yes

Not equals to (=\=).
Compares Arg1 to Arg2, succeeding if Arg1 and Arg2 are not equal. Both the arguments must be a number (integer or float) or arithmetic expression.
Arg1=\= Arg2
Example:
?- 1=\=1.
no

?- 2=\=1.
yes
*?- 3*3=\=4+5.*
no
*?- 6*3=\=10+3.*
yes

Term Operators:
Order of Prolog terms:

Variables	→	*oldest variables precedes younger variables.*
Numbers	→	*float precedes integers.*
Atoms	→	*ordered alphabetically.*

Less than or equal to(@=<)
Succeeds if *Arg1* is equal to or before *Arg2* in the order of terms. Arguments may be any prolog term.
Arg1@=< Arg2
Example:
?- ab@=<aa.
no
?- a@=<1.
no
?-1@=<1.
Yes

Less than (@<).
Succeeds if *Arg1* is before *Arg2* in the order of terms. Arguments may be any prolog term.
Arg1@< Arg2

Greater than or equal to(@>=).
Succeeds if *Arg1* is equal to or after *Arg2* in the order of terms. Arguments may be any prolog term.
Arg1@>= Arg2

Greater than (@>).
Succeeds if *Arg1* is after *Arg2* in the order of terms. Arguments may be any prolog term.
Arg1@> Arg2
Example:
?- 1@>1.0.
no
?- 2.0@>ab.
no

Equals to(==).
Succeeds if *Arg1* and *Arg2* are literally identical. Arguments may be any prolog term.
Arg1== Arg2

Not equals to(\==).
Succeeds if *Arg1* and *Arg2* are not literally identical. Arguments may be any prolog term.
Arg1\== Arg2

Equals to (=:=).
Compares *Arg1* to *Arg2*, succeeding if *Arg1* and *Arg2* are equal. Both the arguments must be a number (integer or float) or arithmetic expression.
Arg1=:= Arg2
Example:
?- 1=:=1.
yes
?- 1=:=2.
no
?- 2+3=:=5.
yes
*?- 12=:=6*2.*
yes

Not equals to (=\=).
Compares *Arg1* to *Arg2*, succeeding if *Arg1* and *Arg2* are *not* equal. Both the arguments must be a number (integer or float) or arithmetic expression.
Arg1=\= Arg2
Example:
?- 1=\=1.
no
?- 2=\=1.
yes
*?- 3*3=\=4+5.*
no
*?- 6*3=\=10+3.*
yes

Exercise:

1. Write a program to compare ages by defining facts and rules from the information provided below. Use appropriate operators where necessary.

   Age of aslam is 11 years.
   Age of asif is 13 years.
   Age of afsheen is 17 years.
   Age of manal is 16 years.
   Age of amir is 30 years.
   Age of falak is 33 years.
   Age of sobia is 40 years.
   Age of sheheryar is 44 years.
   Age of rehan is 52 years.
   Age of erum is 64 years

Lab # 7

Object:
Data types in PROLOG.

Theory:
Terms
The central data structure in Prolog is that of a term. There are terms of four kinds: atoms, numbers, variables, and compound terms. Atoms and numbers are sometimes grouped together and called atomic terms.

Atoms: Atoms are usually strings made up of lower-and uppercase letters, digits, and the underscore, starting with a lowercase letter. The following are all valid prolog atoms:
elephant, b, abcXYZ, x_123, another_pint_for_me_please

On top of that also any series of arbitrary characters enclosed in single quotes denotes an atom.
'This is also a Prolog atom.'

Finally, strings made up solely of special characters like + - * = < > : & (check the manual of your Prolog system for the exact set of these characters) are also atoms.

Examples.
+, ::, <------>, ***

Examples of legal and illegal atom names are:

```
cats                    % ok
dogs                    % ok
cats and dogs           % wrong, contains white space
cats-and-dogs           % wrong, contains graphic symbol
cats_and_dogs           % ok, '_' is a character in Prolog
cats$dogs               % ok, '$' is a character in Prolog
Cats                    % wrong, begins with upper case ASCII (a variable)
_dogs                   % wrong, begins with underscore (a variable)
==>                     % ok, all graphic characters
;;$=                    % wrong, '$' is not a graphic.
;;+                     % ok
'Anything ;; - At all'  % ok, in single quotes
'New York'              % ok, in single quotes
'can''t'                % ok, with '' indicating ' inside atom
```

Numbers.
All Prolog implementations have an integer type: a sequence of digits, optionally preceded by a - (minus). Some also support floats. The ISO standard recognizes two types of numbers: integers and, for decimal numbers, floats.

Integers can optionally be entered using hexadecimal, octal, binary or character code notation. To do this, precede the number with 0x, 0o, 0b or 0'.

For example:
?- X = 16.
X = 16

?- X = 0o7777.
X = 4095

?- X = 0b111111.
X = 63

Integers expressed in decimal notation, if larger than an internal integer, will be promoted to the appropriate decimal type. For example:
?- X = 2000000000000000000000000000.
X = 2.000000e+029
Yes

Negative Numbers.
Negative numbers are entered with a preceding minus (-) sign. Care must be taken to ensure that there is no space between the minus and the number for single numbers. This is because the minus (-) is an operator, and (-) something can ambiguously be interpreted as a structure. This ambiguity is not a problem when minus (-) is used as an operator with two arguments. For example:
?- display(-3).
-3

?- display(- 3).
-(3)

?- display(2 - 3).
-(2,3)

Decimal Numbers.
For decimal numbers, such as 3.3, Amzi! supports three options, including ISO standard floats:

- single precision floating point numbers.
- double precision floating point numbers.
- infinite precision real numbers.

Real numbers allow for any number of digits on either side of the decimal places.

Variables.
Variables are strings of letters, digits, and the underscore, starting with a capital letter or an underscore. Examples: Variables in Prolog are called 'logical' variables. They are not the same as conventional program variables, which typically refer to an element in memory of a specific type.
X, Elephant, _4711, X_1_2, MyVariable, _

The following are valid variable names:
Var Var_2
_var_3 X
Leona Ivan

Two Prolog variables with the same name represent the same variable if they are in the same clause. Otherwise they are different variables (which just happen to have the same name). That is, the scope of For example, a query looking for a customer's phone number, in a clause that has lots of other information, might look like this:
?- customer('Leona', _, _, _, PHONE, _, _).
PHONE = 333-3333

a variable name is the clause in which it appears.

The last one of the above examples (the single underscore) constitutes a special case. It is called the anonymous variable and is used when the value of a variable is of no particular interest.

Compound terms.
Compound terms are made up of a functor (a Prolog atom) and a number of arguments (Prolog terms, i.e. atoms, numbers, variables, or other compound terms) enclosed in parentheses and separated by commas. The following are some examples for compound terms:
is_bigger(horse, X), f(g(X, _), 7), 'My Functor'(dog)

It's important not to put any blank characters between the functor and the opening parentheses, or Prolog won't understand what you're trying to say. In other places, however, spaces can be very helpful for making programs more readable.

The sets of compound terms and atoms together form the set of Prolog predicates. A term that doesn't contain any variables is called a ground term.

Strings.
A string is an alternate way of representing text. Unlike atoms, which are stored in a table and are represented in terms as integers, strings are represented as the string itself. They are useful for textual information that is for display purposes only. Strings unify on a character-by-character basis with each other, and with atoms. A string is denoted by text enclosed in matching backquotes (`). Strings may also have embedded formatting characters exactly like atoms (as described in the section on Escape Characters). For example:
`This is a long string used for\ndisplay purposes`.

To represent the backquote (`) within a string use two backquotes (``).

Horn Clause.
A sentence like this is called a Horn Clause. In prolog the above sentence would look like this:
pred4(A) :- pred1(A, B), pred2(B, C), pred3(C, D).

Note that the implication sign is reversed, commas are used for conjunction, a period is used to end the sentence.

Structures.
Structures are the fundamental data types of Prolog. A structure is determined by its name (sometimes called the *principal functor*) and its arguments. The functor is an atom and the arguments may be any Prolog terms, including other structures. A structure is written as follows:
name(arg1, arg2, ... , argn)

There must be no space between the name and the opening parenthesis "(". The number of arguments in a structure is called the *arity*. The maximum arity of a structure is 4095. Structures are used to represent data. Following are some examples of a structure whose functor is 'likes' and whose arity is 2.
likes(ella, biscuits).
likes(zeke, biscuits)
likes(Everyone, pizza)

Here are some more complex nested structures.
file(foo, date(1993, 6, 15), size(43898))
tree(pam, left(tree(doyle, left(L2), right(R2))), right(R1))
sentence(nounphrase(det(the), noun(dog)), verbphrase(verb(sleeps)))

Structures are also the heads of Prolog clauses, and the goals of the bodies of those clauses. For example:
friends(X, Y) :- likes(X, Something), likes(Y, Something).

Lists.
Lists are used to represent ordered collections of Prolog terms. Lists are indicated by squared brackets '[' and ']'. A list with a known number of elements can be written down, separated by commas within the brackets. The elements can be arbitrary Prolog terms, including other lists, e.g.:
[1, 2, 3]
[alpha, 4]
[f(1), [2, a], X]

Here the first list has three elements, the numbers **1**, **2** and **3**. The second list has two elements. The third list also has three elements. The first is a structure of arity 1, the second a sub-list with two elements, and the third element is a variable called **X**.
Logically, a list can be considered to have two elements:
HEAD - the first element of the list
TAIL - a list of the remaining elements in the list

This way of viewing a list is very useful for recursive predicates that analyze lists. At each level of recursion, the HEAD can be used, and the TAIL passed down to the next level of recursion. It is important to remember that the TAIL is always another list.
This pattern is represented with a vertical bar: [HEAD | TAIL].
We can unify this pattern with a list to see how it works:
?- [HEAD|TAIL] = [a,b,c,d].
HEAD = a
TAIL = [b, c, d]

More than one element in the HEAD can be specified:

?- [FIRST, SECOND|TAIL] = [a,b,c,d].
FIRST = a
SECOND = b
TAIL = [c, d]

An empty list is represented by empty square brackets, []. It is important to note that [] is NOT a list, but an atom. However, it is what is used for the TAIL when there is no tail.
?- [W, X, Y, Z|TAIL] = [a,b,c,d].
W = a
X = b
Y = c
Z = d
TAIL = []

[] is then a useful element for recognizing that a recursive list predicate has reached the end of the list. For example, consider the following predicate that writes each element of a list on a new line:
write_list([]). % empty list, end.
write_list([A|Z]) :-
write(A), % write the head
nl,
write_list(Z). % recurse with the tail
Using it:
?- write_list([apple, pear, plum, cherry]).
apple
pear
plum
cherry
yes

While lists are stored more efficiently than structures, you can think of a list as a nested structure of arity two. The first argument is the head, the second argument is the same structure representing the rest of the list. The special atom, [], indicates the end of the nesting. This nature of lists can be seen using the display/1 predicate, where a dot (.) is used as the functor of the structure.
?- display([a, b, c]).
.(a, .(b, .(c, [])))

While the normal list notation is easier to read and write, sometimes it is useful to think of the structure notation of lists when trying to understand predicates that manipulate lists.

Character Lists.
Lists whose elements are character codes are often used in Prolog, especially in parsing applications. Prolog recognizes a special syntax to make this use more convenient.
A string of characters enclosed in double quotes (") is converted into a list of character codes. Use two double quotes("") to indicate an embedded double quote character. The following examples (using member/2 from the list library) illustrate its

use, and also show predicates for converting between character lists and atoms and strings:
For example:
?- X = "abc".
X = [0w0061, 0w0062, 0w0063]
yes

?- member(0'b, "abc").
yes

?- atom_codes(abc, X).
X = [0w0061, 0w0062, 0w0063]
yes

?- atom_codes(A, "abc").
A = abc
yes

?- string_list(S, "abc").
S = `abc`
yes

?- string_list(`abc`, L).
L = [0w0061, 0w0062, 0w0063]
yes

?- [0'a, 0'b, 0'c] = "abc".
yes

?- member(0'b, "abc").
yes

?- member(0x62, "abc").
yes
?- member(98, "abc").
Yes

Character Constants.
Because Unicode characters are unsigned ints and are often referred to by their hexadecimal value, character constants are represented by the a two byte hexadecimal code, using 0w to indicate a wide character. atom_codes/2, string_list/2 and the single character code notation (0'c) all use the character constants. For example:
?- atom_codes(duck, X).
X = [0w0064,0w0075,0w0063,0w006b]
yes

?- atom_codes(X, [0w64, 0w75, 0w63, 0w6b]).
X = duck
yes

Exercise:

1. Write program in prolog to implement all data types you have studied above.

Lab # 8

Object:
Lists and how to manipulate them.

Theory:
Lists are powerful data structures for holding and manipulating groups of things. In Prolog, a list is simply a collection of terms. The terms can be any Prolog data types, including structures and other lists. Syntactically, a list is denoted by square brackets with the terms separated by commas. For example, a list of alcohol is represented as *[Tequila,Whisky,Vodka,Martini,Muscat,Malibu,Soho,Epita]*.

This gives us an alternative way of representing the locations of things. Rather than having separate location predicates for each thing, we can have one location predicate per container, with a list of things in the container.
list_where([Tequila,Whisky,Vodka], bathroom).
list_where([Martini,Muscat], kitchen).
list_where([Malibu,Soho], under_the_bed).
list_where([Epita], everywhere).

The empty list is represented by a set of empty brackets []. This is equivalent to the nil in other programming language. For our example in this section, it can describe the lack of things in a place:
list_where([], cave).

The Unification works on lists just as it works on other data structure of Prolog. With that, we now can ask questions about lists to prolog:
?- [_,_,X] = [lesson, work, sleeping].
X = sleeping
?- list_where(X, under_the_bed).
X = [Malibu,Soho]

Notice that there is impractical method of getting a list of elements in the first example, because of Prolog won't unify unless both list have the same number of elements. At last, the special notation for list structures.
[X | Y]

This structure is unified with a list, X is bound to the first element of the list, called the head. Y is bound to the list of remaining elements, called the tail. Note that the tail is considered as a list for Prolog and the empty list does not unify with the standard list syntax because it has no head. Here is an example:
?- [X|Y] = [a, b, c, d, e].
X = a
Y = [b, c, d, e]

?- [X|Y] = [].
No

The empty list does not unify with the standard list syntax because it has no head:
?- [X|Y] = [].
no

This failure is important, because it is often used to test for the boundary condition in a recursive routine. That is, as long as there are elements in the list, a unification with the [X|Y] pattern will succeed. When there are no elements in the list, that unification fails, indicating that the boundary condition applies. We can specify more than just the first element before the bar (|). In fact, the only rule is that what follows it should be a list.
?- [First, Second | Q] = [water,gin,tequila,whisky].
First = water
Second = gin
Q = [tequila,whisky]

We have said a list is a special kind of structure. In a sense it is, but in another sense it is just like any other Prolog term. The last example gives us some insight into the true nature of the list. It is really an ordinary two-argument predicate. The first argument is the head and the second is the tail. This predicate is represented by a period(.). To see this notation, we use the built-in predicate display, which writes lits in using this syntax.
?- X = [T|Q], display(X).
.(_01,_02)

From this examples it should be clear why there is a different syntax for lists. The easier syntax makes for easier reading, but sometimes obscures the behavior of the predicate. It helps to keep this "real" structure of lists in mind when working with predicates that manipulate lists.

How to manipulate list.
For lists to be useful, there must be easy way to access, add, and delete list elements. Moreover, we should not have to concern ourselves about the number of list items, or their order.

Two Prolog features enable us to accomplish this easy access. One is a special notation that allows reference to the first element of a list and the list of remaining elements, and the other is recursion.

These two features are very useful for coding list utility predicates, such as member, which finds members of a list, and append, which joins two lists together. List predicates all follow a similar strategy--try something with the first element of a list, then recursively repeat the process on the rest of the list.

The first one we will look at is member. As with most recursive predicates, we will start with the boundary condition, or the simple case. An element is a member of a list if it is the head of the list.
member(T,[T|Q]).

This clause also illustrates how a fact with variable arguments acts as a rule. The second clause of member is the recursive rule. It says an element is a member of a list if it is a member of the tail of the list.
member(X,[T|Q]) :- member(X,Q).

As with many Prolog predicates, member can be used in multiple ways. If the first argument is a variable, member will, on backtracking, generate all of the terms in a given list.
?- membre(X, [muscat, soho, martini]).
X = muscat;
X = soho;
X = martini;
No

Another very useful list predicate builds lists from other lists or alternatively splits lists into separate pieces. This predicate is usually called append. In this predicate the second argument is appended to the first argument to yield the third argument. For example.
?- append([a,b,c],[d,e,f],X).
X = [a,b,c,d,e,f]

It is a little more difficult to follow, since the basic strategy of working from the head of the list does not fit nicely with the problem of adding something to the end of a list. append solves this problem by reducing the first list recursively. The boundary condition states that if a list X is appended to the empty list, the resulting list is also X.
append([],X,X).

The recursive condition states that if list X is appended to list [T|Q1], then the head of the new list is also H, and the tail of the new list is the result of appending X to the tail of the first list.
append([T|Q1],X,[T|Q2]) :- append(Q1,X,Q2).

Real Prolog magic is at work here, which the trace alone does not reveal. At each level, new variable bindings are built, that are unified with the variables of the previous level. Specifically, the third argument in the recursive call to append is the tail of the third argument in the head of the clause. These variable relationships are included at each step.

Exercise:

1. Write a program to implement list in prolog.