Prolog Tutorial

Basic Logic Programming Tutorial

Motaz K. Saad

8/26/2008

Contents

Prolog Fundamentals

Sentences: Facts and Rules

A prolog programmer defines objects and relations, then defines rules about when these relations are true. For example, the sentence

Bill likes dogs.

shows a relation between the objects Bill and dogs ; the relation is likes.

Here is a rule that defines when the sentence Bill likes dogs is true:

Bill likes dogs if the dogs are nice.

A- Facts: What You Already Know

In Prolog, a relation between objects is called a fact. In natural language, a relation is symbolized by a sentence. In the predicate logic that prolog uses, a relation is summarized in a simple phrase-a fact-that consists of the relation name followed by the object or objects (enclosed in parentheses).

As with a sentence, the fact ends with a period (.).

Here are some more facts expressing "likes" relations in natural language:

Bill likes Cindy.

Cindy likes Bill.

Bill likes dogs.

Here are the same facts, written in prolog syntax:

likes (bill, cindy).

likes (cindy, bill).

likes (bill, Dogs).

Facts can also express properties as well as relations; in natural language "Kermit is green" and "Caitlin is a girl". Here are some Prolog facts that express these same properties:

green (Kermit).

girl (Caitlin).

B- Rules: What You Can Infer from Given Facts

a rule is a conclusion that is known to be true if one or more other conclusions or facts are found to be true. **In prolog, all rules have three parts: a head, an if symbol, and a body.**

- The head is the fact that would be true if some number of conditions were true. this is also known as the conclusion or the dependent relation.

- The if symbol separates the head of the rule from the body; it can be the characters :- or the word if.

- The body is the set of conditions (or list of facts) that must be true so that Prolog can prove that the head of the rule is true.

Here are some rules concerning a " likes" relation:

Cindy likes every thing that bill likes.

Caitlin likes every thing that is green.

Given these rules, you can infer from the previous facts some of the things that Cindy and Caitlin likes:

Cindy likes Cindy.

Caitlin likes Kermit.

To encode these same rules in to prolog, you only need to change the syntax a little, like this:

likes (cindy, Something) if likes (bill, Something).

likes (caitlin, Something) if green (Something).

In the first rule the head is likes (cindy, Something), and the body is likes (bill, Something). and in the second rule the head is likes (caitlin, Something), and the body is green (Something).

Queries

Once we give prolog a set of facts, we can proceed to ask questions concerning these facts; this is known as querying the prolog system. we can ask prolog the same type of questions that we would ask you about these relations. based upon the known facts and rules given earlier, you can answer questions about these relations, just as prolog can. in natural language, we ask you:

Does Bill like Cindy?

In Prolog syntax, we ask Prolog:

likes (bill, cindy).

Given this query, prolog would answer

Yes

Because prolog has a fact that says so. As a little more complicated and general question, we could ask you in natural language:

What does Bill like?

In Prolog syntax, we ask Prolog:

likes (bill, What).

Important notes: The second object-What-begins with a capital letter, while the first object-bill-does not. This is because bill is a constant a known value-but What is a variable. In general, constants begin with lower case letters and variable begin with upper case.

Prolog always looks for an answer to a query by starting at the top of the facts. It looks at each fact until it reaches the bottom, where there are no more. Given the query about what Bill likes, Prolog will return

What = dogs

What = cindy

This is because Prolog knows

likes (bill, cindy).

and

likes (bill, dogs).

If we were to ask you (and Prolog):

What does Cindy like?

likes (cindy, What).

Prolog would answer

What = dogs

What = cindy

What = bill

3 solutions

This is because Prolog knows that Cindy likes Bill, and the Cindy likes what Bill likes, and that Bill likes Cindy and dogs.

* Putting Fact, Rules, and Queries Together

1. Suppose you have the following facts and rules:

A fast car is funny. (Fact)

A big car is nice. (Fact)

A small car is practical. (Fact)

Bill likes a car if the car is funny. (Rule)

When you read these facts, you can deduce that Bill likes a fast car. In much the same way, Prolog will come to the same conclusion. If you where not given any facts about fast cars, then you would not be able to logically deduce what kind of a car Bill likes. You could take a guess at what kind of a car might be fun, but prolog only knows what you till it; Prolog does not guess.

2. An example demonstrating how Prolog uses rules to answer queries.

likes (ellen, tennis) .

likes (john, football).

likes (tom, baseball).

likes (eric, swimming).

likes (mark, tennis).

likes (bill, Activity) if likes (tom, Activity) .

It is clear that the first 5 lines are facts, but the last one is rule, this rule corresponds to the natural language statement.

Bill likes an activity if Tom likes that activity .

In this rule, the head is likes (bill, Activity) **,** and the body is

likes (tom, Activity)

If you give Prolog the following query likes (bill, baseball).

(In natural language;" Does Bill like tennis?") When attempting to find a solution to this query, Prolog will use the rule:

likes (bill, Activity) if likes (tom, Activity).

Because there is no fact in example about Bill liking baseball Prolog will replies yes as a solution of this query.

The way that Prolog will find the solution of the query It has combined the rule

likes (bill, Activity) if likes (tom, Activity).

With the fact

likes (tom, baseball)

To decide that

likes (bill, baseball). is true.

If we give Prolog the another following query

likes (bill, tennis).

Prolog replies **No** to the latest query (" Does Bill like tennis?") because

a. There is no fact that says Bill likes tennis.

b. Bill is relationship with tennis cannot be inferred using the given rule and the available facts.

Variable: General sentences

In Prolog, ***variables*** enable you to write general facts and rules and ask general questions.

To represent a variable in Prolog, you capitalize the first letter of that argument as we saw in the preceding discussion of rules, you saw this line:

likes (cindy, Something) if likes (bill, Something).

The object **Something** begins with a capital letter because it is a variable; it must be able to match any thing that Bill likes. It could equally

well have been called ***X or Zorro.***

The objects bill and cindy begin with lowercase letters because they are not variables-instead, they are symbols (constants).

Note: you can begin a symbol with a capital letter if you enclose it in quotes (like this: "Bill").

Causes and Predicates

From Natural Language to Prolog Programs

In the first section of this chapter we talked about facts and rules, relations general sentences, and queries. Those wards are all part of a discussion of logic and natural language. Now we are going to discuss the same ideas, but we are going to use more Prolog-ish words, like clauses, predicates, variables, and goals.

Clauses (Facts and Rules)

Basically, there are only two type of phrases that make up the Prolog language; a phrase can be either a ***fact*** or a ***rule.*** These phrases are known in Prolog as ***clauses.*** **The heart of a Prolog program is made up of clauses.**

Examples of Rules

1.This firs example shows a rule that can be used conclude whether a menu item is suitable for Diane.

Diane is vegetarian and eats only what her doctors tills her to eat.

Given a menu and the preceding rule, you can conclude if Diane can order a particular item on the menu. To do this, you must check to see if the item on the menu matches the constraints given.

a. Is Food_on_menu a vegetable?

b. Is Food_on_menu on the doctors list?

c. Conclusion: If both answers are yes, Diane can order Food_on_menu.

In Prolog, a relationship like this must be represented by a rule because the conclusion is based on facts. Here's one way of writing the rule:

```
diane_can_eat ( Food_on_menu) if
vegetable ( Food_on_menu) and
on_doctor_list ( Food_on_menu).
```

2. Suppose you want to make a Prolog fact that is true if ***person1*** is the parent of ***person2.*** This is easy enough; simply state the prolog fact

```
parent (paul, samantha).
```

This shows that Paul is the parent of Samantha. Put, suppose your prolog data base already has facts stating father relationships. For example, "Paul is the father of Samantha":

```
father (paul, samantha).
```

And you also have facts stating mother relationships; "Julie is the mother of Samantha":

```
mother (Julie, samantha).
```

If you already had a collection of facts stating these father/mother relationships, it would be a waste of time to write parent facts in to the database for each parent relationship.

Since you know that person1 is the parent of person2 if person1 is the father of person2 or if person1 is the mother of person2, then why not write a rule to convey these constraints? after stating these conditions in

natural language, it should be fairly simple to code this into a prolog rule by writing a rule that states the relationships.

```
parent (person1, person2) if father  (person1, person2).
parent (person1, person2) if mother (person1, person2).
```

These Prolog rules simply state that

```
person1 is the parent of person 2 if person1 is the father of person2.
person1 is the parent of person 2 if person1 is the mother of person2.
```

Variables (General Clauses)

In a simple query, you can use variables to ask Prolog to find who likes tennis. for examples,

```
likes (X, tennis)
```

This query uses the letter X as a variable to indicate an unknown person. Variable names in Prolog must begin with a capital letter, after which any number of letters (uppercase or lowercase), digits, or underline characters (_) can be used. For example, the following are valid variable names:

```
My_first_correct_variable_name
Sales_10_11_86
```

While the next three are invalid:

```
1stattempt
second_attempt
"disaster"
```

(Careful choice of variable names makes programs more readable. For example,

```
likes (Person, tennis).
```

is better than

```
likes (X, tennis).
```

Because ***Person*** makes more sense than X.) Now type the query

```
likes (Person, tennis).
```

How Variables Get Their Values

You may have noticed that Prolog has no assignment statement; this is a significant distinction between Prolog and other programming languages.

Variables in Prolong get their values by being matched to constants in facts or rules. Until it gets a value, a variable is said to be ***free***; when it gets a value, it becomes ***bound.*** But it only stays bound for the time needed to obtain one solution to the query; then Prolog unbinds it, backs up, and looks for alternative solutions.

This is a very important point: ***You can't store information by giving a value to a variable.*** Variables are used as part of the pattern-matching process, not as a kind of information storage.

Take a look of the following example, which uses program to demonstrate how and who variables get their values.

```
/* Example 1.2.2.1*/
  predicates
    likes (symbol, symbol).
  clauses
    likes (ellen, reading).
    likes (john, computers).
    likes (john, badminton).
    likes (leonard, badminton).
```

likes (eric, swimming).

likes (eric, reading).

Consider this query: Is there a person who likes both reading and swimming?

likes (Person, reading) and likes (person, swimming).

Prolog will solve the two parts of this query by searching the program's clauses from top to bottom. In the first part of the query

likes (Person, reading).

the variable ***Person*** is free; its value is unknown before prolog attempts to find a solution. On the other hand, the second argument, ***reading,*** is known. Prolog searches for a fact that matches the first part of the query.

The first fact in the program

likes (ellen, reading).

is a match (***reading*** in the fact matches ***reading*** in the query), so prolog binds the free variable ***Person*** to the value ***ellen***, the relevant value in the fact. At the same time, Prolog places a pointer in the list of facts indicating how far down the search procedure has reached. Next, in order for the query to be fully satisfied (find a person who likes both reading and swimming), the second part must also be fulfilled. Since Person is now bound to ellen, Prolog must search for the fact

likes (ellen, swimming).

Prolog searches for this fact from the beginning of the program, but no mach occurs (because there is no such fact in the program). The second part of query is not true when ***Person*** is ***ellen.*** Prolog now "unbinds" ***Person*** and attempts for another solution of the first part of the query with ***Person*** once

again a free variable. The search for another fact that fulfills the first part of the query starts from the pointer in the list of facts. Prolog looks for the next ***Person*** who likes reading and find the fact likes (eric,reading). Person is now bound to eric, and Prolog tries once again to satisfy the second part of the query, this time by looking in the program for the fact

likes (eric, swimming).

This time it finds a match (the last clause in the program), and the query is fully satisfied. Prolog returns

Person =eric

1 solution

Anonymous Variables:-

Anonymous variables enable you to unclutter your programs. If you only need certain information from a query, you can use anonymous variables to ignore the values you don't need. In Prolog, the anonymous variable is represented by a lone underscore ("_").

The following parents example demonstrate how the anonymous variable is used.

/* Example 1.2.2.2*/

predicates

male (symbol).

female (symbol).

parent (symbol, symbol).

clauses

male (bill).

male (joe).

female (sue).

female (tammy).

parent (bill, joe).

parent (sue, joe).

parent (joe, tammy).

The anonymous variable can be used in place of any other variable. the difference is that the anonymous variable will never get set to a value. For example, in the following query, you need to know which people are parents, but you don't need to know who their children are. Prolog realizes

that each time you use the underscore symbol in the query, you don't need information about what value is represented in that variable's place.

```
Goal : parent (parent, _)
```

Given this query, Prolog replies

```
parent =bill
parent =sue
parent =joe
3 solutions
Goal:-
```

In this case, because of the anonymous variable, Prolog finds and reports three parents, but it does not report the values associated with the second argument in the ***parent*** clause. Anonymous variables can be also used in facts. The following Prolog facts

```
owns (_, shoes).
eats (_).
```

could be used to express the natural language statements

```
Everyone own shoes.
Everyone eats.
```

The anonymous variable match anything. A named variable would work equally well in most cases, but its name would serve no useful purpose.

Goals

Up to now, we've been using the word query when talking about the questions you ask Prolong. From now on we're going to refer to them by their more common name: ***goals.*** This should make sense:

when you query Prolong, you are actually giving it a goal to accomplish ("Find an answer to this question, if one exists...").

Goals can be simple, such as these two:

```
likes (ellen, swimming).
likes (bill, what).
```

or they can be more complex. in the "Variable" section of this chapter, you saw a goal made up of two parts:

```
likes (Person, reading) and like (Person, swimming).
```

A goal made up of two or more parts is known as a ***compound goal***, and each part of the compound goal is called a ***subgoal.*** (Note: In Prolog rule, each condition is also called a subgoal.)

Compound Goals: Conjunctions and Disjunctions

You can use a compound goal to find a solution where both subgoal A ***and*** subgoal B are true (a ***conjunction***), or a solution where subgoal A ***or*** subgoal B is true (a ***disjunction***). Here's an example program illustrating this idea:

/* Example 1.2.3 */

```
predicates
  car (symbol, real, integer, symbol, integer)
  truck (symbol, real, integer, symbol, integer)
clauses
```

```
car (chrysler, 130000, red, 12000).
car (ford, 90000, 4, gray, 25000).
car (datsun, 8000, 1, red, 30000).
truck (ford, 8000, 6, blue, 8000).
truck (datsun, 50000, 5, orange, 20000).
truck (toyota, 25000, 2, black, 25000).
```

Load and run this program, then try the goal

```
car (Make, Odometer, Years_on_road, Body, 25000).
```

This goal attempts to find a car described in the clauses that costs exactly $25,000. Prolog replies

```
Make=ford, Odometer=9000, Years_on_road=4, Body=gray
1 Solution
Goal:_
```

But this goal is slightly unnatural, since you'd probably rather ask a question like:

Is there a car listed that costs less than $25,000?

You can get Prolog to search for a solution by setting this compound goal:

```
car (Make, Odometer, Years_on_road, Body, cost) and      /* subgoal A and*/
Cost < 25000.                                            /* subgoal B*/
```

This is known as a conjunction. To fulfill this compound goal, Prolog will try to solve the subgoals in order. First, it will try to solve

```
car (Make, Odometer, Years_on_road, Body, Cost).
```

and then

```
Cost < 25000.
```

with the variable ***Cost*** referring to the same value in both subgoals. Try it out now.

Note: The subgoal Cost < 25000 involves the relation less than, which is built into the Prolog system. The less than relation is no different form any other relation involving two numeric objects, but it is more natural to place the symbol for it between the two objects.

You can also use a compound goal to find if one subgoal or another is true. In natural language, you might ask

Is there a car listed that costs less than $25,000?, or is there a truck listed that costs less than $20,000?

Prolog will search for a solution if you set this compound goal:

```
car (Make, Odometer, Years_on_road, Body, Cost) and Cost < 25000 or
                                                         /* subgoal A or */
truck (Make, Odometer, Years_on_road, Body, Cost) and Cost < 20000.
                                                         /* subgoal B*/
```

This kind of compound goal is known as a disjunction. This one sets up the two subgoals as alternatives, much as though they where two clauses for the same rule. Prolog will then find any solution that satisfies either of the subgoals. To fulfill this compound goal, Prolog will try to solve the first subgoal ("find a car..."), which is composed of these subgoals:

```
car (Make, Odometer, Years_on_road, Body, Cost.)
```

and

```
Cost < 25000
```

If a car is found, the goal will succeed; if not, Prolog will try to fulfill the second compound goal ("find a truck"), made up of the subgoals

```
truck (Make, Odometer, Years_on_road, Body, Cost ),
```

and

```
Cost < 20000.
```

Comments

It's good programming style to include comments in your program to explain things that might not be obvious to someone else (or to you in six months). This makes the program easy for you and others to understand. If you choose appropriate names for variables, predicates, and domains, you'll need fewer comments, since the program will be more self-explanatory.

Multiple-line comments must begin with the characters /*(slash, asterisk) and end with the characters /*(asterisk, slash). To set off single-line comments, you can use these same characters, or you can begin the comments with a percent sign (%).

```
/*This is an example of a comments*/
% This is also a comment
/* * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* and so are these three lines          */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* you can also nest a Prolog comment/* within a comment */ like this*/
```

Match

In the Previous sections of this chapter, we've talked about Prolog "matching answers to questions"," finding a match", "matching conditions with facts"," matching variables with constants", In this section we explain what we mean when when we use the term "match".

There are several ways Prolog can match one thing to another.

Obviously, **identical structures match each other;**

parent (joe, tammy) matches parent (joe, tammy).

However, **a match usually involves one or more free variables.** For example, with X free,

parent (joe, X) matches parent (joe, tammy).

and X takes on (is bound to) the value tammy.

If X is already bound, it acts exactly like a constant. Thus , if X is bound to the value tammy, then

parent (joe, X) matches parent (joe, tammy) but

parent (joe, X) would not matches parent (joe, millie).

The second instance doesn't match because, once a variable becomes bound, its value can't change.

How could a variable already be bound when Prolog tries to match it with something?

Remember that variables don't store values-they only stay bound for the length of time needed to find (or try to find) one solution to one goal. So the only way a variable could be bound before trying a match is that the goal involves more than one step, and the variable became bound in a previous step. For example,

parent (joe, X) and parent (X, jenny).

is a legitimate goal ; it means, "Find someone who is a child of Joe and a parent of Jenny". Here X will already be bound when the subgoal parent (X, jenny) is reached.

If there is no solution to parent (X, jenny), Prolog will unbind X and go back and try to find another solution to parent (joe, X), then see if parent (X, jenny) will work with the new value of X.

Tow free variables can even match each other. For example, parent (joe, X) matches parent (joe, Y).

binding the variables X and Y to each other. Also long as the binding lasts, X and Y are treated as a single variable, and of one of them gets a value the other one will immediately have, the same value. When free variables are bound to each other like this, they're called **free sharing variables**.

Note: In Prolog, Variable binding (values) are passed in two ways: in and out. The direction in which a value is passed is referred to as its flow pattern. When a variable is passed into a clause, it is an input parameter, signified by (i); when passed out of a clause, a variable is an output parameter, signified by (o).

Prolog Basic Program Sections

The Clauses Section

The clauses section is the heart of the prolog program, this is where you put the facts and rules. Clauses for a given predicates must be placed together in the clauses section. A sequence of clauses defining a predicate is called ***a procedure*** .

The technique which used when we want to satisfy a goal is: prolog will start at top of the clauses section ,looking at each fact and rule as it searches for a match. When prolog goes down through the clauses section it place internal pointer next to each clauses that matches the current subgoal .If that clauses is not part of the solution, then prolog returns to set pointer and looks for another match, ***This is backtracking , which we mentioned in chapter1***.

The Predicates Section

The predicate section is where you declare your predicates and the domain (types) of the arguments to your predicates . If you define your own predicate in the clauses section of a Prolog program , you must declare it in ***a predicates*** section.

There is a built in predicates such as ***write , makewindow,nl*** and so on. So you don't need to declare any of prolog built in predicates that you use in your program.

How to declare user_defined predicates:

A predicate declaration begins with the predicate name followed by an open left parenthesis. After the predicate name and the open parenthesis come zero or more arguments to the predicate. ***predicate Name (argument_type1, argument_type2,--,argument_typeN)*** Each argument type is followed by a comma , and the last argument type is followed by the right parenthesis , a predicate declaration is not followed by a period. The argument types are either standard domains or domains that you've declared in the domains section.

Predicate Names

The name of the predicate must begin with a letter, followed by a sequence of a letters , digits, and underscores letter as the first letter in the predicate name predicate names can be up to 250 characters long. You can't use spaces, the minus sign, asterisks, or slashes in predicate names . Valid naming characters in prolog consist of the following:

Uppercase letters : A,B,...,Z
Lowercase letters : a,b,...,z
Digits : 0,1,...,9
Underscore charecter :_

Some legal Predicate names	some illegal predicate names
fact	[fact]
is_a	*is_a*
has_a	has/a
patternCheckList	pattern-check-list
choose_Menu	choose Menu
first_10	>first_10

Predicate Arguments:-

The ***argument to the predicates*** must belong to known Prolog domain . A domain can be ***a standard domain ,*** or it can be one you declare in the domains section.

Example:-

1. if you declare a predicate my_predicate(symbol, integer) in the predicates section, like this:

```
predicates
    my_predicate(symbol, integer)
```

you don't need to declare its arguments' domain in a domains section, because symbol and integer are standard domains. but if you declare a predicate my_predicate in the predicates section , like this:

```
predicate
    my_predicate(name, number)
```

you will also need to declare name (a symbol) and number (an integer) as belonging to the standard domains symbol and integer, like this:

```
domains
    name =symbol
    number=integer
predicates
    my_predicate(name, number)
```

2.This program show some more predicate and domain declarations:

```
domains
    person, activity = symbol
    car, make, color = symbol
    mileage, year_on_read,
    cost=integer
predicates
    likes(person, activity)
    parent(person,person)
    can_buy(person, car)
    car(make, mileage, year_on_read, color, cost)
    green(symbol)
    ranking(symbol, integer)
```

Multiple Arity:-

The arity of a predicate is the number of arguments that it takes. You can have two predicates with the same name but different arity.

You must group different arity versions of a given predicate name together in both the predicates and the clauses sections of your program , other than this restriction , the different arities are treated as completely different predicates.

/*Example of different arity*/

```
domains
    person = symbol
predicates
    father(person)              /*This person is a father*/
    father(person, person)      /*One person is a father of the
```

other person*/

```
clauses
    father (Man) :-
    father (Man, _).
    father(adem, seth).
    father(ahmed, ali).
```

The Domains Section

Domain in prolog are like type in Pascal. They enable you to give distinctive names to different kinds of data .In Prolog program , the arguments to a predicate belong to domain , theses can be standard domains, or special domains that you specify. The **domains** section serves two very useful purposes.

- First , you can give meaningful names to domains even if they are the same as domains that already exist.

- Second , special domain declarations are used to declare data structures that are not defined by the standard domains.

Examples :-

1.Here's an example to illustrate how declaring domains help to document your predicates:

Frank is a male who is 45 years old.

with standard domains, you come up with the following predicate declaration :

person (symbol, symbol, integer)

The following declaration will help you to understand what the arguments in the predicate declaration stand for:

```
domains
  name , sex = symbol
  age = symbol
predicates
  person(name , sex, age)
```

One of the main advantage of this is that Prolog can catch type errors, like the following obvious mistake:

```
same_sex(X,Y) :-
person (X,Sex,_),
person(Sex, Y, _).
```

Even though ***name*** and ***sex*** are both defined as ***symbol ,*** they are not equivalent to each other. This enables prolog to detect an error if you accidentally swap them. This feature is very useful when your programs get very large and complex. You might be wondering why we don't use special domains communicate the argument so much better. The answer is that once an argument is typed to a specific domain, that domain can't be mixed with another domain , even if the domain are the same so, even though ***name*** and ***sex*** are of the same (***symbol***)**,** they can't be mixed.

2.This next example program will yield a ***type error*** when run:

```
domains
  product, sum = integer
predicates
  add_em_up(sum, sum, sum)
```

```
multiply_em(product, product, product)
clauses
  add_em_up(X,Y, Sum) :-
  Sum = X+Y.
  multiply_em(X,Y, Product) :-
  Product=Y*X .
```

This program does two things: It adds and it multiplies . Given the goal

```
add_em_up(32, 54, Sum).
```

Prolog will come up with

```
Sum = 86
1 solution
```

which is the sum of the two integers you supplied the program. On the other hand , this program will also multiply two arguments with the ***multiply_em*** predicate . Now experiment with this program. If the need to figure out what the product of 31 and 13 is , you could enter the goal:

```
multiply_em(31, 13, Product).
```

Prolog would then respond with the correct answer.

```
Product = 403
1 solution
```

but suppose you need the sum of 42 and 17; the goal for this would be

```
add_em_up(42, 17, Sum).
```

Now you need to double the product of 31 and 17 , so you write the following goal :

```
multiply_em(31, 17, Sum), add_em_up(Sum, Sum, Answer).
```

you would expect Prolog to return

```
Sum = 527, Answer = 1054
1 solution
```

But, instead, you get a type error. What happened is that you tried to pass the resulting value of ***multiply_em*** ***(***that is , of domain ***product***), into the first and second arguments in ***add_em _up*** , which have domains of ***sum*** . This yields a type error because ***product*** is a different domain than ***sum*** . Even though both domains are really of type ***integer*** , they are different domains. So, if a variable is used in more than one predicate within a clause, it must be declared the same in each predicate.

The Goal Section

Up until now, you have been entering goals at the goal : prompt in the Dialog window . these are known as ***external goals ,*** because you must provide them at run time . Using external goals is fine if you always want to run your programs from within Prolog's development environment.

However , if you plan to compile your program to standalone executable program, you can hardly expect the end user of the program to enter queries at the goal: prompt.

Prolog incorporates a goal section that must be included if you intend to compile your program to an executable program. goals given in the goal section are known as ***internal goals,*** because they are part of your program's source text and they are compiled right in the program.

Essentially , the goal section is the same as the body of a rule :it's simply a list of subgoals. There are two differences between the goal section and a rule :

1. The goal keyword is ***not*** followed by :-(the if symbol).
2. Prolog automatically executes the goal when the program runs.

It's as if Prolog makes a call to goal , and the program runs , trying to satisfy the body of the goal rule . If the subgoals in the goal section all succeed, then the program terminates successfully . If while the program is running, a subgoal in the goal section fails, then the program is said to have failed .

Example:-

here's is an example design to show you the difference between internal and external goals:-

```
predicates
 run(char)
goal                    /*This is an internal goal.*/
 run(X)
clauses
 run(X) :-
 makewindow (1,7,7,"Hello world program",0,0,25,80),
 write("hello world (first)"),
 readchar(X),
 removewindow.
 run (X):-
 write("hello world (second)"),
 readchar(X).
```

In this example program, Prolog creates a window and write Hello world (first) into the window . Notice , however, that only Hello world (first) is written ; the second clause for ***run*** is never used . ***With an internal goal, Prolog only searches for the first solution .***Also, the value to which

X is bound is not displayed.

Other program sections

Now you are familiar with clauses , predicates, domains, and goal sections of Prolog program, we'll tell you a little bit about some other commonly used program sections: the database section, the **constants** section, and the various global sections.

The Database Section

A Prolog program is a collection of fact and rules. sometimes, while the program is running, you might want to update (change, remove, or add) some of the facts the program operates on. In such a case, the facts constitute a ***dynamic*** or ***internal*** database; it can change while the program is running. Prolog includes a special section for declaring the facts in the program that are to be a part of the dynamic (or changing) database; this is the **database** section .

The keyword **database** declares the **database** section. It is here that you declare the facts to be included in the dynamic database. Prolog includes a number of built in predicates that allow easy use of the dynamic

database.

The Constant Section

You can declare and use symbolic constants in your Prolog programs. A constant declaration section is indicated by the keyword **constants**, followed by the declaration themselves, using the following syntax:

```
<Id> = <Macro definition>
```

<Id> is the name of your symbolic constant, and***<Macro definition>*** is what you're assigning to that constant. Each***<Macro definition>*** is terminated by a new line character, so there can only be one constant declaration per line. Constant declared in this way can then be referred to later in the program.

Constant declared in this way can then be referred to later in the program. Consider the following program fragment:

```
constants
```

```
zero = 0
one = 1
two = 2
hundred = (10*(10-1)+10)
p1 = 3.141592
age = 3
slash_fill = 4
red  = 4
```

before compiling your program, Prolog will replace each constant with the actual string to which it corresponds. For instance.....,

```
A = hundred*34, delay(A),
setfillstyle(slash_fill, red),
Circumf = pi*Diam,
.....
```

will be handled by the compiler in exactly the same way as

```
.....,
A = (10*)10-1)*34, delay(A),
setfillstyle(4,4),
circumf = 3.141592*Diam,
.....
```

There are a few restrictions on the use of symbolic constants.

1. The difinition of a constant can't refer to it self. For example:

   ```
   my_number = 2*my_number/2          /*It is not allowed*/
   ```

   will generate the error massage recursion in constant definition
2. The system does not distinguish between uppercase and lowercase in a **constants** declaration. Consequently, when a **constants** identifier is used in the **clauses** section of a program, the first letter must be lowercase to avoid confusing constants with variables. So, for example, the following is valid construction:

   ```
   constants
    Two = 2
   goal
    A = two, write(A).
   ```
3. There can be several **constants** declaration sections in a program, but constants must be declared before they are used.
4. constant identifiers are global and can only be declared once. Multiple declaration s of the same identifier will result in the error massage

/*constant identifier can only be declared once.*/

The Global Sections

Prolog allows you to declare some domains, predicates, and clauses in your program to be global (rather than local); you do this by setting aside separate global domains, global predicates, and global database sections at the top of your program.

Compiler directives

Prolog provides several ***compiler directives*** you can add to your program to tell the compiler to treat your code in specified ways when compiling. You can also set most of the compiler directives from the Options/Compiler Directives menu item in the Prolog system. The two basic Compiler directives are there:-

1. the include Directive

You can use the include directive to save yourself from having to type certain procedures in again and again

2. The trace and shorttrace Directives It lets you walk through your program at run time one step at a time, looking at each CALLED clause and RETURNed value.