4.900

Prolog Tutorial 1

by Clive Spenser

Prolog Tutorial 1

The contents of this manual describe the product, ***WIN-PROLOG***, version 4.500, and are believed correct at time of going to press. They do not embody a commitment on the part of Logic Programming Associates Ltd (LPA), who may from time to time make changes to the specification of the product, in line with their policy of continual improvement. No part of this manual may be reproduced or transmitted in any form, electronic or mechanical, for any purpose without the prior written agreement of LPA.

Logic Programming Associates Ltd
Studio 30
The Royal Victoria Patriotic Building
Trinity Road
London SW18 3SX
England

phone: +44 (0) 20 8871 2016
fax: +44 (0) 20 8874 0449
web: http://www.lpa.co.uk

4 February, 2005

Contents

Introduction to *LPA-*PROLOG for Windows

Author: Clive Spenser

A basic introduction to using the Console Window in LPA Prolog for Windows.

Show you how to use basic Prolog facilities and steps you through developing some simple programs.

Using the Console Window

You can type queries into the Console window at the ?- prompt.

For instance, there is a built-in predicate called 'write' which takes one argument. This is denoted in Prolog as *write/1* and it writes that argument out to the current output channel:

```
?- write( hello ). <enter>
helloyes
```

The yes indicates that the query succeeded.

If you mistype the query, you will generally get an error message.

```
?- write( hello .
! ---------------------------------------
! Error 42 : Syntax Error
! Goal     : eread(_37600,_37602)
```

You must not have a space before the first round bracket.

```
?- write (hello).
! ---------------------------------------
! Error 42 : Syntax Error
! Goal     : eread(_33392,_33394)
```

The system only tries to evaluate your query when a full-stop is entered, so don't forget to terminate your query with a full-stop. A variable in Prolog begins with an UpperCase letter or underscore.

```
?- write( X ).
_20190X = _
```

We can uses *nl/0* (takes no arguments), to separate the output due to the write statement to that of the system.

```
?- write( X ), nl.
_21022
X = _
```

The number beginning with the underscore is a system created variable.

We can use *write/1* to output more complex structures.

```
?- write( data(alpha,beta) ), nl.
data(alpha,beta)
yes
```

Matching Structures

Prolog uses pattern-matching, called 'unification' to match variables. This is a very powerful idea. We can use *=/2* to invoke unification.

The simplest queries only involve ground terms, i.e. terms without variables.

```
?- a=a.
yes

?- a=b.
no
```

The terms can be complex. The computation either succeeds or fails:

```
?- f(a(b)) = f(a(b)).
yes

?- f(a(b)) = f(b(a)).
no
```

We can introduce a variable on either left or right hand side of the equals sign.

```
?- X=a.
X = a

?- b=Y.
Y = b
```

Unification ensures that the variable will be returned instantiated such that the computation can succeed; i.e. it gets given a value which matches the other side.

More complex structures; now the variable returns a structure.

```
?- X=f(a).
X = f(a)

?- f(a)=X(Y).
X = f ,
Y = a
```

Now with variables on both sides, we can instantiate them simultaneously.

```
?- X(a)=b(Y).
X = b ,
Y = a

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

It matters where we put the variable; some structure just do not match

```
?- X(a) = Y(b(c)).
no
```

whereas, others do

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

We can repeat values as arguments:

```
?- a(X) = Y(b(b)).
X = b(b) ,
Y = a
```

We can repeat variables too (and their colour changes):

```
?- a(X(X)) = Y(b(b)).
X = b ,
Y = a
```

We can have the same variable on both sides of an equation

```
?- a(X(b)) = Y(b(X)).
X = b ,
Y = a
```

Sometimes this works; i.e. there is a solution

```
?- a(X) = X(a).
X = a
```

And sometimes it doesn't; i.e. there is no solution

```
?- a(X) = X(b).
no
```

We can join together as many queries as we want:

```
?- a=X, Y=b, Z=X(Y).
X = a ,
Y = b ,
Z = a(b)
```

Example 1 – who likes what?

We can create a new program in a program window by choosing File/New. We can then type in some initial data.

```
likes( clive, tea ).

likes( clive, beer ).

likes( brian, beer ).

likes( diane, soda ).
```

Some simple queries:

```
?- likes( brian, X ).
X = beer
```

Hitting the spacebar or semi-colon lets us see alternate solutions where they exist

```
?- likes( clive, X ).
X = tea ;

X = beer
```

What about trying to find two people who like the same thing.

```
?- likes(A,X), likes(B,X).
A = B = clive ,
X = tea ;

A = B = clive ,
X = beer ;

A = clive ,
X = beer ,
B = brian ;

A = brian ,
X = beer ,
B = clive ;

A = B = brian ,
X = beer ;

A = B = diane ,
X = soda
```

We can explicitly demand that the two likers are different.

```
?- likes(A,X), likes(B,X), A \= B.
A = clive ,
X = beer ,
B = brian ;
```

```
A = brian ,
X = beer ,
B = clive ;

no
```

Now lets add a rule:

```
likes( X, Y ) :-
  cold( X ),
  hot( Y ).
```

And some associated data:

```
cold( clive ).

hot( coffee ).

hot( cocoa  ).
```

We can then ask:

```
?- likes( clive, X ).
X = tea ;

X = beer ;

X = coffee ;

X = cocoa
```

Now we can better re-organize the rule and associated data:

```
likes( X, Y ) :-
  person(X, cold),
  drink( Y, hot ).

drink( coffee, hot   ).

drink( cocoa , hot   ).

drink( tea   , hot   ).

drink( soda  , cold  ).

person( clive, cold  ).

person( brian, hot   ).

person( diane, hot   ).
```

Notice now we get:

```
?- likes( clive, X ).
X = tea ;

X = beer ;

X = coffee ;
```

```
X = cocoa ;

X = tea ;
```

We can now extend the data:

```
drink( coffee, hot,  100  ).

drink( cocoa , hot,  120  ).

drink( tea   , hot,  150  ).

drink( soda  , cold, 220  ).

person( clive, cold, 100  ).

person( brian, hot,  150  ).

person( diane, hot,  200  ).
```

And we can ask if there are any hot drinks less than 125:

```
?- drink(X,hot,V), V<125.
X = coffee ,
V = 100 ;

X = cocoa ,
V = 120 ;
```

We can then introduce a new predicate. We use underscore to indicate that we don't care about the values of those variables. The test at the end of can_buy/2 checks that the number associated with the person is greater than that of the drink.

```
can_buy( X, Y ) :-
  person(X, _, V1),
  drink( Y, _, V2 ),
  V1 > V2.
```

We now get:

```
?- can_buy(X,Y).
X = brian ,
Y = coffee ;

X = brian ,
Y = cocoa ;

X = diane ,
Y = coffee ;

X = diane ,
Y = cocoa ;

X = diane ,
Y = tea ;

no
```

Simple Arithmetic

We can use is/2 to evaluate various mathematical expressions.

```
?- X is 7.
X = 7

?- X is 7+5.
X = 12

?- X is 7+5*2.
X = 17

?- X is (7+5)*2.
X = 24
```

Expressions can get quite complicated.

```
?- Y=5, Z=4, X is sqrt(Y*Y-Z*Z).
Y = 5 ,
Z = 4 ,
X = 3

?- Y is 5*5, Z is 4*4, X is sqrt(Y*Y-Z*Z).
Y = 25 ,
Z = 16 ,
X = 19.2093727122985
```

Simple Lists

Lists are widely used in Prolog. Lists are recursive data structures comprising of a first element of the list, often called the Head, and the rest of the list, often called the Tail.

At any point in time, only the Head is accessible. Lists can be manipulated using recursive procedures. There are several built-in predicates, bips, for list manipulation, e.g. **member/2**, **append/3**, **remove/3, sort/2**.

We can define other routines:

- to sum the elements in a list of numbers,
- to replace an element of a list,
- to convert an input list into an output list, e.g. to change a list of verbs into their past tenses.We can use member/2 to show membership of a list.

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

X = b ;

X = c ;

X = d ;

no

?- member(X, [a,b,c,d] ), member(Y, [g,f,e,d]).
X = a ,
Y = g ;

X = a ,
Y = f ;

X = a ,
```

And so on 4x4 = 16 solutions.

```
?- member(X, [a,b,c,d] ), member(Y, [g,f,e,d]), X=Y.
X = Y = d ;

no
```

There is one element which is a member of both lists, d.

Example 2 - more lists

Lets start by considering a program to write out a list, one item at a time on a line.

We use the [Hd|Tl] construct to separate the Head of the list for processing, and then, having dealt with it, call the program again recursively on the Tail.

```
writer( [Hd|Tl] ) :-
  write( Hd ),
  nl,
  writer( Tl ).
```

We stop when there are no more elements in the list to process. This is the first clause.

```
writer( [] ).
```

Lets test it out:

```
?- writer([1,2,3,4]).
1
2
3
4
yes
```

Now, lets try and add up a list of numbers. We use a style of program that is very common and well established in Prolog circles.

i) we introduce an argument which acts an accumulator and represents the current intermediate value

ii) a 'base case' which is activated when there are no more elements to process, and which copies the accumulator value into another 'result' variable

In the general case, we process the Head element of the list by adding it to the interim Value we have so far to get a Next interim value. We then call the same routine again, recursively, on the Tail of the list. In this way, we work our way through the list, one element at a time.

```
adder( [Hd|Tail], Value, Answer ) :-
   Next is Hd+Value,
   adder( Tail, Next, Answer ).
```

When we reach the end of the list, we need to copy the value in the accumulator through to our final result.

```
adder( [], P, P ).
```

Now, lets try this:

```
?- adder( [1,2,3,4],0,P).
P = 10
```

We can introduce another predicate to seed the accumulator version with the correct

starting value; for addition, this is 0.

```
adder( List, Answer ) :-
    adder( List, 0, Answer ).

?- adder( [1,2,3,4], P).
P = 10
```

Example 3 – family tree

We can create a new program in a program window by choosing File/New. We can then type in some initial data about fathers.

```
father( ron,    clive ).

father( ron,    john ).

father( clive,    sean ).

father( clive, ruaidhri ).

father( john,    sarah ).
```

We can write a predicate which defines siblings as people having the same father. We need to include a simple check to make sure that the two siblings are indeed different people. Even though X and Y are not the same variable they are not necessarily different either. Unless we write something to the contrary, Prolog will indeed try to coerce to the same value at some stage.

```
sibling( X, Y ) :-
  father( Z, X ),
  father( Z, Y ),
  not X=Y.
```

We have two pairs of siblings in our data.

```
?- sibling(X,Y).
X = clive ,
Y = john ;

X = john ,
Y = clive ;

X = sean ,
Y = ruaidhri ;

X = ruaidhri ,
Y = sean ;
```

We can add some more basic data of people ages and sex.

```
person( clive,    male, 41 ).

person( john,     male, 45 ).

person( ruaidhri, male,  7 ).

person( sean,     male,  3 ).

person( sarah,  female, 18 ).

person( adele,  female, 81 ).
```

We can add the siblings' ages together using is/2.

```
sibling( X, Y, Total ) :-
  person( X, _, Age1 ),
  person( Y, _, Age2 ),
  father( Z, X ),
  father( Z, Y ),
  not X=Y,
  Total is Age1 + Age2.
```

And now we can get the combined ages for the sibling pairs.

```
?- sibling(X,Y,T).
X = clive ,
Y = john ,
T = 86 ;

X = john ,
Y = clive ,
T = 86 ;

X = ruaidhri ,
Y = sean ,
T = 10 ;

X = sean ,
Y = ruaidhri ,
T = 10 ;

no
```

Example 4 – simple network

We can then type in some initial data about network connections.

```
arc( a, b ).

arc( a, c ).

arc( b, c ).

arc( b, d ).
```

We can write a predicate which defines two nodes as being linked if they either have a (direct) arc, or you can there is an intermediate node you can go to which is linked.

```
link( X, Y ) :-
  arc( X, Y ).

link( X, Y ) :-
  arc( X, Z ),
  link( Z, Y ).
```

Now we can ask who is linked to who.

```
?- link(a,d).
yes

?- link(c,d).
no

?- link(a,X).
X = b ;

X = c ;

X = c ;

X = d ;

no
```

As we have two routes from a to c, one direct and one indirect, we get the solution twice.

We now extend the arcs to have shapes, and add some more arcs.

```
arc( a, b, circle ).

arc( a, c, square ).

arc( b, c, circle ).

arc( b, d, circle ).

arc( b, e, square ).
```

```
arc( c, e, circle ).

arc( d, e, square ).

arc( e, f, circle ).

arc( f, c, square ).
```

Now we can modify our link program to ensure that all arcs used are of the same type.

```
link( X, Y, Shape ) :-
  arc( X, Y, Shape ).

link( X, Y, Shape ) :-
  arc( X, Z, Shape ),
  link( Z, Y, Shape ).
```

We now get:

```
?- link(a,c,circle).
yes
```

Or with a variable for the shape:

```
?- link(a,c,S).
S = square ;

S = circle ;
```

We can ask 'who can a link to and using what shape':

```
?- link(a,P,L).
P = b ,
L = circle ;

P = c ,
L = square ;

P = c ,
L = circle ;

P = d ,
L = circle ;

P = e ,
L = circle ;

P = f ,
L = circle ;
```

Example 5 – more arithmetic

Let's look to generate a list of N numbers.

We use a counter and generate a list one element at a time by adding that number to the list we have accumulated so far and passing that expended list on into the recursive call.

```
gen( N, ListSoFar, Answer ) :-
    N > 0,
    N1 is N-1,
    gen( N1, [N|ListSoFar], Answer ).
```

When we the counter reaches zero, there are no more elements to add to the list, we have finished and we return the current value of the accumulator by unifying it with an extra 'result' variable.

```
gen( 0, List, List ) .
```

We can introduce another predicate to seed the recursive version with the correct starting value; for list generation, this is the empty list, [].

```
gen( N, List ) :-
    gen( N, [], List ).

?- gen( 4, P).
P = [1,2,3,4]
```

So by combining this with our adder program we get:

```
?- gen(5,L), adder(L,P).
L = [1,2,3,4,5] ,
P = 15

?- gen(15,L), adder(L,P).
L = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] ,
P = 120
```

Example 6 – more networks

Similarly, we can add in another dimension which indicates the cost or length of the connection:

```
arc( a, b, circle,  4 ).

arc( a, c, square, 11 ).

arc( b, c, circle,  5 ).

arc( b, d, circle,  4 ).

arc( b, e, square,  2 ).

arc( c, e, circle,  3 ).

arc( d, e, square,  3 ).

arc( e, f, circle,  4 ).

arc( f, c, square,  1 ).
```

We need to keep adding the new cost to our cost so far:

```
link( X, Y, Shape, SoFar, Total ) :-
  arc( X, Y, Shape, Val ),
  Total is SoFar+Val .

link( X, Y, Shape, SoFar, Total ) :-
  arc( X, Z, Shape, Val ),
  NextSoFar is SoFar+Val,
  link( Z, Y, Shape, NextSoFar, Total ) .
```

Now we get:

```
?- link(a,c,circle,0,Cost).
Cost = 9 ;

no

?- link(a,c,Shape,0,Cost).
Shape = square ,
Cost = 11 ;

Shape = circle ,
Cost = 9 ;

no
```

Example 7 – simple expert system

Let's construct a simple expert system. First some:

```
question(q0,   'are you rich?').

question(q1a,  'do you prefer a hot climate?').

question(q1b,  'do you prefer a hot climate?').

question(q2aa, 'do you prefer a destination near to home?').
```

We will start at level0. We ask the first question, and then branch according to the answer.

```
process0 :-
  ask( q0, Answer ),
  ( Answer=yes ->process1a;
    Answer= no ->process1b ).
```

The next level, level1, looks similar, except we have two cases to consider:

```
process1a :-
  ask( q1a, Answer ),
  ( Answer=yes ->process2aa;
    Answer= no ->process2ab ).

process1b :-
  ask( q1b, Answer ),
  ( Answer=yes ->process2ba;
    Answer= no ->process2bb ).
```

For the next level, level2, we can do some branching and some terminating solutions:

```
process2aa :-
  ask( q2aa, Answer ),
  ( Answer=yes ->process3aaa;
    Answer= no ->process3aab ).

process2ab :-
 write( alaska    ),
 nl.

process2ba :-
 write( penang    ),
 nl.

process2bb :-
 write( russia    ),
 nl.
```

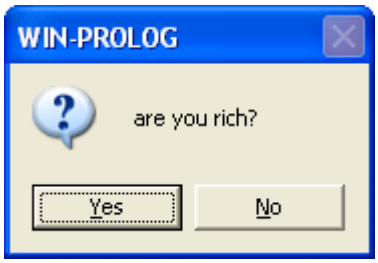
And then add the missing level3 case:

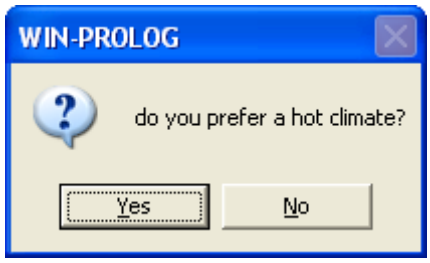
```
process3aaa :-
 write( tioman    ),
 nl.

process3aab :-
 write( jamaica   ),
 nl.
```

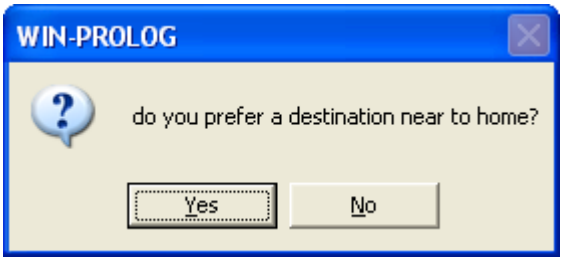
Now all we need is something to ask the questions; LPA offers message_box/3:

```
ask( Q, A ) :-
  question( Q, Text ),
  message_box( yesno, Text, A ).
```

This is a very simple solution which uses a naming convention for each new level. Better, would be to use parameters, i.e. process(1a), process(1b), etc.








```
?- process0.
tioman
yes
```

Example 8 – simple snacks and combinations

Let's start with some simple data (facts) about food:

```
appetiser( soup ).

appetiser( salad ).


entree( beef ).

entree( duck ).

entree( lamb ).

entree( chicken ).


pudding( ice_cream ).

pudding( fruit ).

pudding( cake ).
```

Now, we can have a rule which is that a snack comprises a starter, followed by a main_course and then a dessert:

```
snack( S, M, D ) :-
  appetiser(    S ),
  entree( M ),
  pudding(    D ).
```

This allows us to generate all the possible snack combinations and backtrack thru them:

```
?- snack(A,B,C).
A = soup ,
B = beef ,
C = ice_cream ;

A = soup ,
B = beef ,
C = fruit ;

A = soup ,
B = beef ,
C = cake ;

A = soup ,
B = duck ,
C = ice_cream
```

By extending the data 'sideways', we can make it slightly more interesting. We can add two arguments, one indicating the value or cost of the item and the second the calorific value.

```
appetiser(     soup,  50,  60 ).

appetiser(    salad,  80,  80 ).


entree(    beef, 100, 120 ).

entree(    duck, 120,  80 ).

entree(    lamb, 140,  90 ).

entree( chicken, 140,  90 ).


pudding( ice_cream, 20,  80 ).

pudding(     fruit, 40,  40 ).

pudding(      cake, 80,  20 ).
```

We can then extend our snack program to return the totals for the three courses

```
snack( S, M, D, TotalCost, TotalCals ) :-
  appetiser(     S, SV, SC ),
  entree( M, MV, MC ),
  pudding(     D, DV, DC ),
  TotalCost is SV+MV+DV,
  TotalCals is SC+MC+DC .

?- snack(A,B,C,D,E).
A = soup ,
B = beef ,
C = ice_cream ,
D = 170 ,
E = 260 ;

A = soup ,
B = beef ,
C = fruit ,
D = 190 ,
E = 220 ;

A = soup ,
B = beef ,
C = cake ,
D = 230 ,
E = 200 ;
```

We can add an extra argument to our snack program which indicates the maximum cost for any generated snack; now we will get less solutions.

```
snack( S, M, D, TotalCost, TotalCals, Upper ) :-
  appetiser(     S, SV, SC ),
  entree( M, MV, MC ),
  pudding(     D, DV, DC ),
  TotalCost is SV+MV+DV,
  TotalCals is SC+MC+DC,
  TotalCost =< Upper .

?- snack(A,B,C,D,E,200).
A = soup ,
B = beef ,
C = ice_cream ,
D = 170 ,
E = 260 ;

A = soup ,
B = beef ,
C = fruit ,
D = 190 ,
E = 220 ;

A = soup ,
B = duck ,
C = ice_cream ,
D = 190 ,
E = 220
```

Example 9 – more snacks and combinations

Now in our final version we change three more things:

- a couple of extra courses
- a test_for_chewiness which check we do not mix our chewiness types:
- wrap up the answers into a list

```
appetiser(     soup,  50,  60 ).

appetiser(    salad,  80,  80 ).


entree(    beef, 100, 120, chewy ).

entree(    duck, 120,  80, chewy ).

entree(    lamb, 140,  90, chewy ).

entree( chicken, 140,  90, easy ).


pudding( ice_cream, 20,  80, easy ).

pudding(     fruit, 40,  40, easy ).

pudding(      cake, 80,  20, chewy ).


drink(         tea, 10,  20 ).

drink(      coffee, 20,  10 ).


postre(     brandy, 30,  30 ).

postre(     whisky, 40,  40 ).

snack( [ S(SV), M(MV), D(DV), W(WV), P(PV) ] ) :-
  appetiser(     S, SV, SC ),
  entree( M, MV, MC, MT ),
  pudding(     D, DV, DC, DT ),
  test_for_chewiness( MT, DT ),
  drink(       W, WV, WC ),
  postre(      P, PV, PC ) .

test_for_chewiness( M, M ).
```

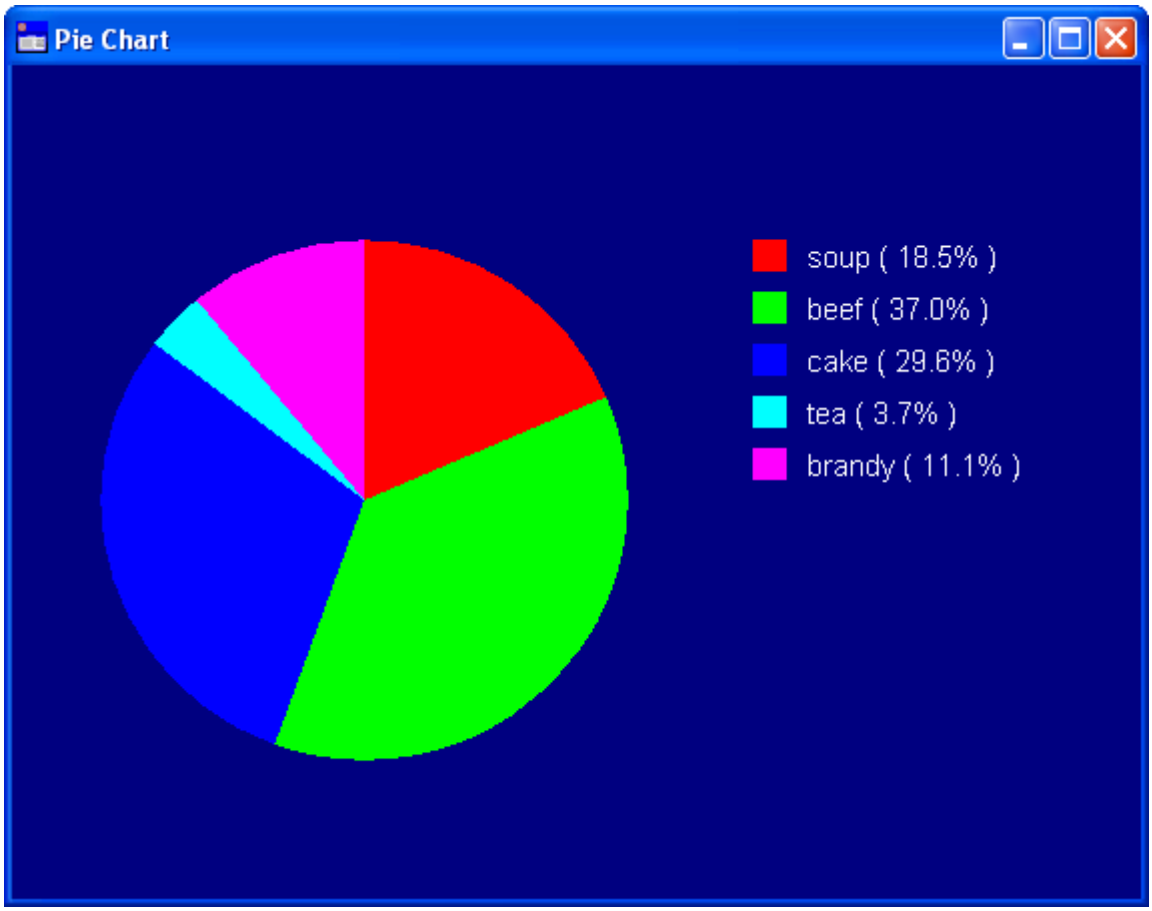
Now we can run this:

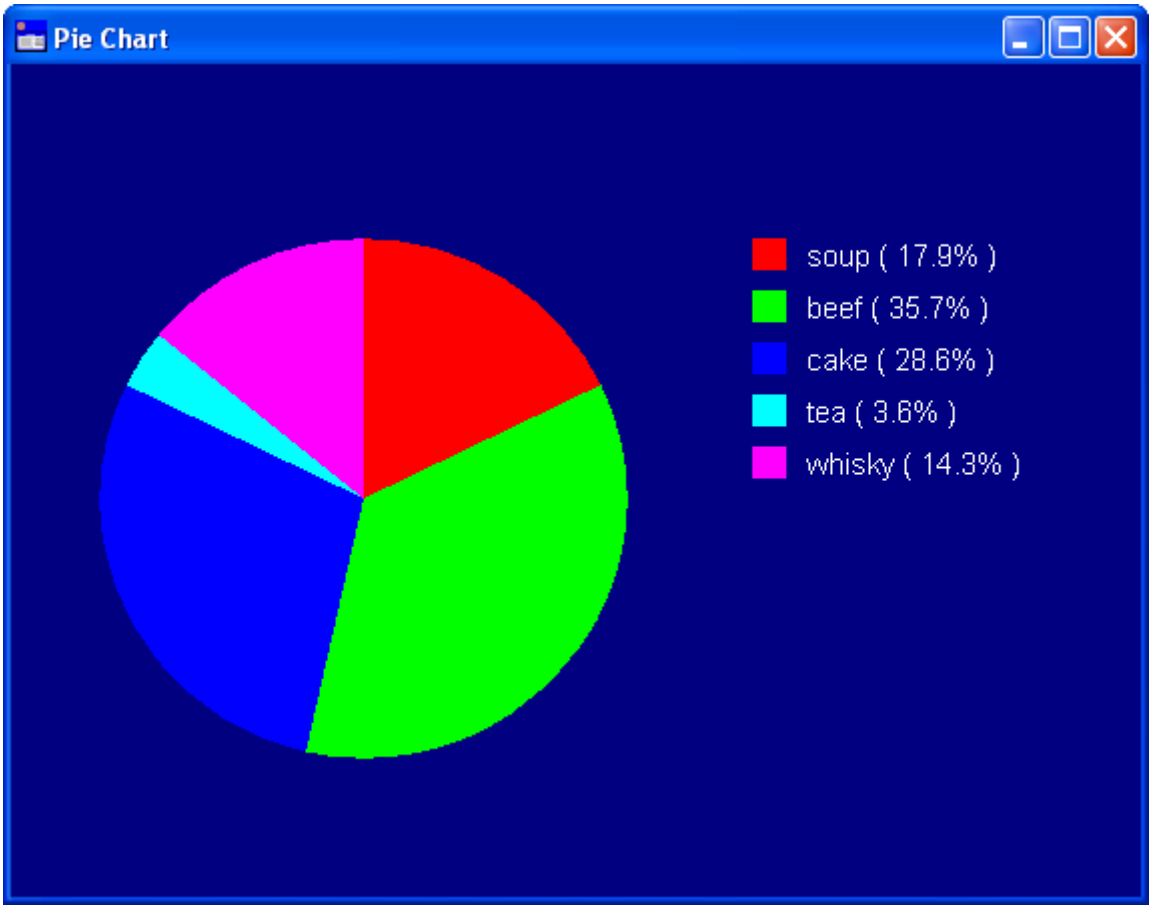
```
?- snack(X).
X = [soup(50),beef(100),cake(80),tea(10),brandy(30)] ;

X = [soup(50),beef(100),cake(80),tea(10),whisky(40)] ;

X = [soup(50),beef(100),cake(80),coffee(20),brandy(30)] ;

X = [soup(50),beef(100),cake(80),coffee(20),whisky(40)]
```

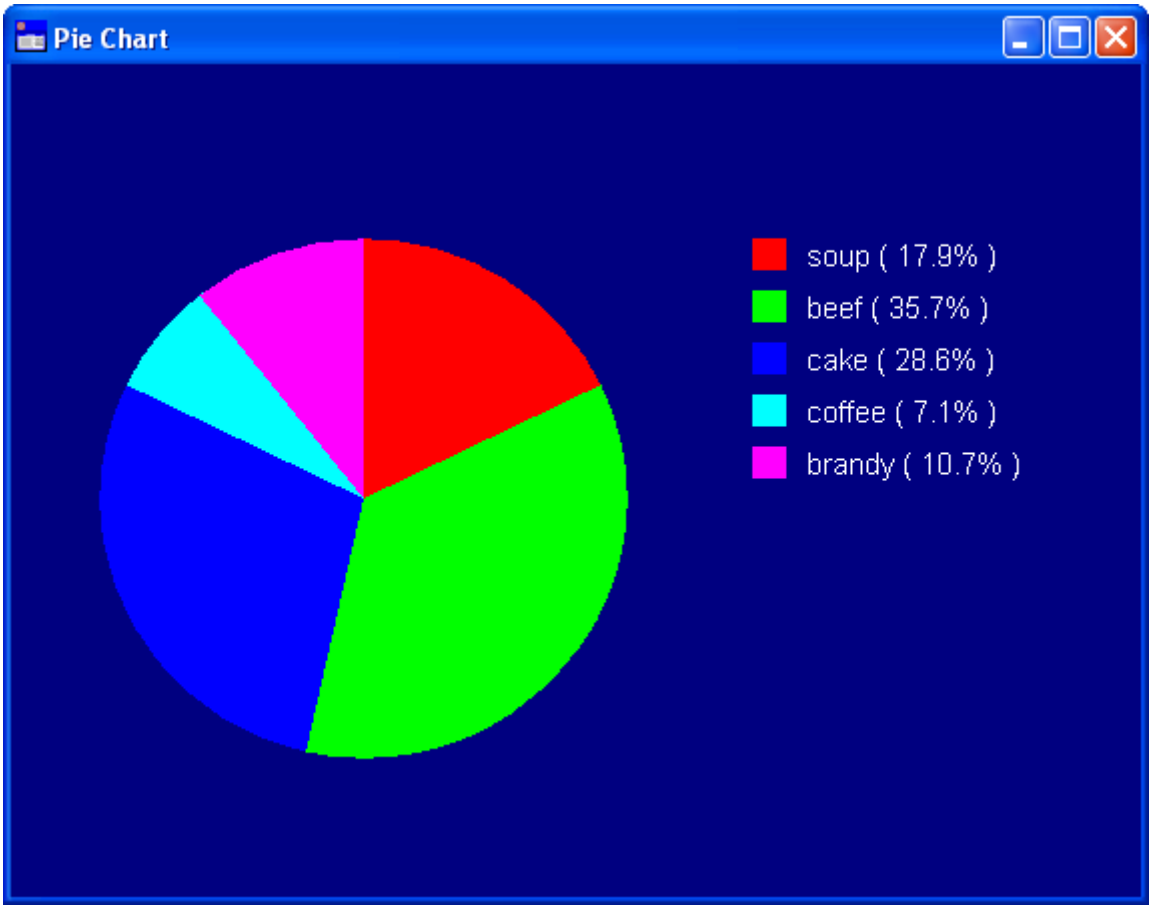
This is now the right format for the piechart program. We can load this using a call to *ensure_loaded/1* or by using either the open or load menu options from the File menu.

Now when we run the flowing query we can backtrack thru the solutions and see them displayed graphically.

```
?- snack(X), show_pie(``,X).
```




Pie Chart
soup (17.9%)
beef (35.7%)
cake (28.6%)
tea (3.6%)
whisky (14.3%)


Pie Chart
soup (17.9%)
beef (35.7%)
cake (28.6%)
coffee (7.1%)
brandy (10.7%)