OF ARTIFICIAL INTELLIGENCE

Sophism: A Cretan saying all Cretans are liars!

25.1 INTRODUCTION

Learning AI should never be a theoretical exercise. It has to be augmented by programs that actually exhibit it. Naturally one requires a programming language for this. While numerous languages (such as BASIC, C and Java) have been used to write AI code, the better known languages are PROLOG and LISP. PROLOG was first developed by Alain Colmerauer of the University of Marseilles, France in the early 1970s and was used as a tool for Programming in Logic. PROLOG is known for its in-built depth-first search engine and for its ease in quick prototyping. A PROLOG program, unlike other computer language programs, comprises a description of the problem using a number of *facts* and *rules*. Execution is more of defining a goal that forces the PROLOG Inference Engine to search and move in a direction that yields one or more solutions that help achieve the goal.

Section 6.2 gave a brief description about PROLOG from the point of view of logic. In this chapter we try to cover the basic principles of programming in PROLOG to enable the reader to comprehend and program for AI. It may be noted that only those features that are prominent and essential to arm the reader to write AI programs are discussed. As will be seen the language is simple, more akin to English and thus quickly learnt. A variety of PROLOG compilers are currently available. The programs discussed herein have been written and compiled using LPA WIN-PROLOG. Most of the programs do not use LPA WIN-PROLOG specific predicates and thus will work fine on other PROLOG environments as well; for all others you will require the LPA WIN-PROLOG compiler.

25.2 CONVERTING ENGLISH TO PROLOG FACTS AND RULES

It is extremely simple to convert English sentences or facts into their PROLOG equivalents. Here are a few sentences converted into PROLOG.

English	PROLOG
The cakes are delicious.	delicious(cakes).
The pickles are delicious.	delicious(pickles).
Biryani is delicious.	delicious(biryani).
The pickles are spicy.	spicy(pickles).
Priya relishes coffee.	relishes(priya,coffee).
Priya likes food if they are delicious.	likes(priya, Food) *if* delicious(Food).
Prakash likes food if they are spicy and delicious.	likes(prakash, Food) *if* spicy(Food) *and* delicious(Food).

While the first four PROLOG equivalents are called *Facts* the last two conversions are termed as *Rules*. od is a variable whose value is to be found from the previous facts. Also note that while variables begin th an upper case letters, the constants (eg. *priya*, *pickles*, etc.) begin with lower case letters. If we just look the contents of the second column, you will be amazed to note that we have already written a *PROLOG* ogram to deduce what *Priya* and *Prakash* like! So here is how our first *PROLOG* program looks like-

```
/* This is a program that can be used to find
   what a person likes*/
delicious(cakes).                  % Cakes are delicious.
delicious(pickles).                % Pickles are delicious.
delicious(biryani).                % Biryani is delicious.
spicy(pickles).                    %Pickles are spicy.
relishes(priya,coffee).            %Priya relishes coffee.
likes(priya, Food):-               % Priya likes food if it is delicious.
        delicious(Food).
likes(prakash, Food) :-
        spicy(Food), delicious(Food). /*Prakash likes food if it is spicy and delicious.*/
```

Notice how the **if**s have been replaced by (**:-**) and the **and**s by (commas). Also every fact such as picy(pickles) or rule is terminated by a . (period). Comments could be of any of the following forms

```
% This comments only this line.
/* This comments everything in between the asterisks and the slashes.*/
```

25.3 GOALS

Now compile the program. The program is ready to accept *goals* which are more like queries. Suppose we wish to ask the program:

Which food items are delicious?

This, in PROLOG terminology, is called a *Goal* and is presented on the **?** – prompt as

```
?- delicious(Food).
Food = cakes
Press ; to get the remaining alternatives for Food viz.
Food = pickles;
Food = biryani
```

Note that the goal also ends with a . (period).
How do we find out the kind of food a person likes. Try the goal:

likes(priya,Food).

which means "*What food does Priya like?*".
The goal and the response of the system is shown below:

```
| ?- likes(priya,Food).
Food = cakes
Press ; to get the remaining alternatives for Food viz.
Food = cakes ;
Food = pickles ;
Food = biryani
| ?-
```

Now try the goal:
likes(prakash,Food).

The above program serves several goals. Apart from finding out what food *Priya* and *Prakash* can also find which food items are spicy and who relishes what. More complex goals could also be For instance you could ask other questions like:

Who relishes coffee and also likes pickles?

using the goal given below.

```
?- relishes(Who,coffee),likes(Who,pickles).
Who = priya
```

Alternatively, you could also query the likings of a person who relishes coffee as:

```
?- relishes(Who,coffee), likes(Who,Food).
```

for which you will get the following answers:

```
Who = priya ,
Food = cakes ;

Who = priya ,
Food = pickles ;

Who = priya ,
Food = biryani
```

Now do you like PROLOG or not? If the answer is negative then try writing code in any of the c conventional languages (C, BASIC, Java,…) to realize the same! Don't forget that the same PROLOG prog can still reason around and satisfy many other goals. Try these as goals

licious(cakes).

eaning - Are cakes delicious?

and
spicy(pickles).
Are pickles spicy?
likes(priya,biryani).
Does Priya like biryani?

ll these goals will prompt the PROLOG inference engine to yield a *Yes* as all of them are *True*.
y now you are already familiar with what *facts* and *rules* are and how *goals* serve to query information.
ou are also aware of *Variables* and *Constants* in PROLOG.

4 PROLOG TERMINOLOGY

a) Predicates
A *predicate* name is the symbol used to define a relation. For instance in relishes(priya, coffee). the symbol *relishes* is the predicate while the contents within viz. *priya* and *coffee* comprise its arguments. Predicates need not necessarily have arguments.

b) Clauses
Clauses are the actual rules and facts that constitute the PROLOG program.

c) Atoms
Atoms are basically symbols or names that are indivisible and are used in the program, files or as database items, among others. They are represented in single quotes as in
'Here is an atom'

(d) Character
Character lists are used in programs that manipulate text character by character. They are represented in double quotes and when printed appear as a list of ASCII codes of each letter in the character chain.

(e) Strings
Strings on the contrary, have a form that is in between an atom and a character list. They are represented between backward quotes (the key below *esc* on the keyboard) as –
`This is an string`.

(f) Arity
The number of arguments in a predicate forms its arity. It is represented by a */n* placed after the predicate name. *n* is the number of arguments. For instance the predicate *relishes/2* takes two arguments.

25.5 VARIABLES

In PROLOG, variables must begin with a capital letter and could be followed by other letters (upper or lower case), digits, underscores or hyphenations but no blanks. Examples of a few valid variables are given below:

ms

Some invalid variables are given below:

Menu Item-1
5Pro
foo

PROLOG also has a variable called the *anonymous* variable. This is represented by just an underscore (_). In a goal if information on a particular argument is not required then an anonymous variable could be used in its place. If you need to know only the color of a car and not its color from a program containing facts of the type car(Brand,Color) viz.

```
car(maruti,white).
car(fiat,black).
then the goal –
car(_,Color).
```

will provide only the colors of the cars available in the fact base and not the car names. In short the goal could be translated into the English query - Find only the color(s) of the car(s).

25.6 CONTROL STRUCTURES

In PROLOG it is possible to order a set of rules in two different ways. Examine the flow in the following program:

```
p:- a,b,c.
p:- d,e,f.
```

This means that *p* is true if either (*a and b and c*) is true or (*d and e and f*) is true. The comma as mentioned earlier implies an *and*. The same rules could be reordered as –

```
p:- a,b,c;d,e,f.
```

without changing the logical meaning.

PROLOG also allows an *if then* control structure. The -> and ; together form the *if_then* control structure. Thus in

```
a:- b -> c;d.
```

the right hand side of the rule is interpreted as

If b is true then verify the truth of c else find that of d.

25.7 ARITHMETIC OPERATORS

Some of the main arithmetic related predicates are shown in Table 25.1.

Predicate	Functionality
</2, >/2	Expression *less than, greater than*
= : =/2, =\=/2	Expression *equality, inequality*
is/2	Expression evaluator
seed/1	Seed the random number generator

An arithmetic expression can take any of the following forms:

(i) A number – could be an integer or a floating point. (E.g. 45)
(ii) A list wherein the elements are numbers. (E.g. [55, 45, 40])
(iii) A function represented by a compound term wherein the functor denotes a function predefined by the compiler and whose argument is itself an expression. (E.g. tan(1 + 4)/4)
(iv) A bracketed expression (An expression in a bracket) (E.g. (4*W))
(v) A variable that has been bound to an expression before the expression is evaluated. (E.g. *V* where *V* was initialized to $V = \sin(CX)$)

Some of the basic predefined arithmetic, trigonometric and logarithmic functions are shown in Table 25.2.

Table 25.2 *Some predefined Arithmetic, Trigonometric and Logarithmic functions*

Function	*Computes*
	Arithmetic
X + Y	Sum
X – Y	Difference
X*Y	Product
X/Y	Division
–X	Negative
X mod Y	Remainder after integer division
X^Y	X to the power Y
	Trigonometric
rand(X)	Pseudo random floating point number between 0 and X
sqrt(X)	Square root of X
sin(X), cos(X), tan(X), asin(X), acos(X), atan(X)	Sine, Cosine and Tangent of X as also arc values respectively.
	Logical
X«Y, X»Y	Logical shift arithmetic left or right respectively
\(X)	Logical negation of the integer X
a(X,Y), o(X,Y), x(X,Y)	Logical AND,OR,XOR of X and Y respectively

25.8 MATCHING IN PROLOG

How does PROLOG find a way to satisfy the goal or query posed to it? This depends on the goal. If the goal does not have any variables as in

relishes(priya,coffee).

then things are simple. A matching is performed predicate to predicate and argument to argument. If this goal matches any fact then it outputs a *Yes* (or *No*) as the case may be.

Things are a bit different for a case when the goal involves variables. Consider the simple case of "*Who relishes coffee?*" The corresponding goal is

relishes(Who,coffee).

Remember *Who* is a variable. This goal is matched with the first fact of *relishes* viz. *relishes(priya,coffee)* which causes the variable *Who* to be bound to the value *priya* which in turn is finally returned.

Now add the following facts to the above program:

fond(priya,driving).
fond(prakash,cartoons).

Compound goals like – Who relishes coffee and is also fond of driving? viz.

relishes(Who, coffee), fond(Who,driving).

will initially bind the first *Who* to *priya* and then try to find a match with this value for the second goal viz. *fond(priya,driving).*

Notice that *Who* has been bound to *priya*.

Since it could consistently find a value for *Who* that matches both the goals, the system returns *priya* as an answer.

The value bound in the first goal is thus propagated to the next goal and so on.

What about the goal

relishes(Who, coffee), fond(Who,cartoons). ?

Since the propagated value of *Who*, viz. *priya*, cannot bind to the variable *Who* in the second clause (obviously), the PROLOG inference engine *backtracks* to another fact for relishes to find a new value for *Who* that matches the first goal.

If you add the fact
relishes(prakash,coffee).

then this compound goal given above would in the second attempt, bind the variable *Who* to *prakash* and carry it forward to the second goal to finally satisfy it.

One should bear in mind that free variables having the same position in a predicate can bind to each other. For instance in the goal-

likes(priya,X).

e variable *X* is not bound (free) and hence will bind with the left hand side of the first rule viz.

likes(priya,Food).

which also has a variable as its second argument. We could thus say that *X* is mapped to the variable *Food* whose value is subsequently discovered if all the subgoals in the right hand side of the rule can are satisfied.

In brief one should bear in mind that PROLOG allows structures that comprise a name (or it could be an atom) with arguments enclosed in brackets. Two such terms *unify* only if their structure names, also called the functors, are same, the number of arguments within are same and every argument within it *unifies* with the corresponding one of the other.

25.9 BACKTRACKING

By now we have partially discussed what backtracking means. But let's finish it for good. Imagine you have entered a maze and are trying to search for something within. You would in the normal course always take the left turn (or right) consistently at every fork and continue. If you reach a dead end, you would return to the fork and try the right turn (or the left). Eventually you would search the whole maze. We have seen that in compound goals, PROLOG attempts to do the very same. The process is called *backtracking*. Whenever there is a fork or alternate paths to be discovered a backtracking point is put up and the same is visited in the event of a failure. Let's look at the following program.

```
likes(prakash, X) :-edible(X), tastes(X, sweet).
tastes(chocolates,sweet).
tastes(gourd,bitter).
tastes(toffees, sweet).
edible(chocolates).
edible(toffees).
edible(gourd).
```

What happens when you pose the goal:

likes(prakash,X).

The right hand side of the rule for *likes* is first triggered. So we have *edible(X)* interpreted as

Find an X such that X is edible

to be satisfied. There are three values that *X* could take (*chocolates*, *toffees* and *gourd*). The variable *X* is first bound to *chocolates*. The process of verifying whether chocolates are sweet continues using *tastes(chocolates, sweet)*. Since this too is true the system returns *chocolates* as an answer. If you press ; you can get further solutions for *X*.

the modified program below:

```
likes(prakash, X) :-edible(X), tastes(X, sweet),write(X),nl,fail.
tastes(chocolates,sweet).
tastes(gourd,bitter).
tastes(toffees, sweet).
edible(chocolates).
edible(toffees).
edible(gourd).
```

Note that we have added a new term called *fail* which effectively returns all possible answers one after the other. After the value of *X* has been bound to *chocolates* (as in the previous explanation) the next thing it encounters is a *write(X)* which writes the currently bound value of *X*. The *nl* forces a newline. Finally *fail* forces the PROLOG engine to imagine that the right hand side is not *true* though all previous clauses have succeeded in finding an apt value for *X*. *fail* thus makes the system feel that it has not succeeded in its attempt to satisfy a goal and therefore forces it to backtrack to the previous fork and find the next alternative solution for *X*. *fail* can be used when one needs to perform an exhaustive search of a tree and not just deliver a single instance of the correct solution. In other words the goal succeeds in finding all possible solutions by failing each time!

Thus, when we issue the goal:

```
| ?- likes(prakash, X).
```

here is what you get

```
chocolates
toffees
no
```

The final *no* crops up because after all solutions (two in this case) are found, the last search for another solution fails due to the *fail*!

This can be avoided by allowing an automatic success after the final failure of *likes*. Insert an additional terminating clause for *likes* as shown below.

```
likes(prakash, X) :-edible(X), tastes(X, sweet),write(X),nl,fail.
likes(_,_).          /*Terminating condition*/
tastes(chocolates,sweet).
tastes(gourd,bitter).
tastes(toffees, sweet).
edible(chocolates).
edible(toffees).
edible(gourd).
```

After all possible values of X have been found the second clause is triggered which in plain English states that *Anyone likes anyone*. Now try to interpret what would happen if this clause were to be placed prior to the original clause for *likes*.

Another way around is to use the following program.

```
do:-likes(prakash,X).
do.
likes(prakash, X) :-edible(X), tastes(X, sweet),write(X),nl,fail.
tastes(chocolates,sweet).
tastes(gourd,bitter).
tastes(toffees, sweet).
edible(chocolates).
edible(toffees).
edible(gourd).
```

The readers are urged to find the difference between these two programs.

25.10 CUTS

Written using an exclamatory mark (!), cuts form a way of preventing backtracking. Imagine what would happen if we burn the bridge on a river after crossing it. It would not be possible for us to return to find an alternate path, if required. The cut does just that. Inspect the sequence in which the subgoals on the right hand side are executed in following program-

```
do:- p(N),q(N),nl,write(N),!,r(N).
p(1).
p(2).
p(3).
p(4).
q(2).
q(3).
q(4).
r(3).
```

When the goal *do* is issued, the inference engine tries to satisfy *p(N)*; then *q(N)* after which it proceeds to satisfy *r(N)* which happens to be located after the cut (!). Naturally, after satisfying *p(N)* (meaning finding a value of *N* such that *p(N)* is true), if *q(N)* is not satisfied, the engine backtracks to another alternative for *p* and comes back to check whether *q* can be satisfied with the new state of *p*. But if *r* (or any alternatives of *r*) is not satisfied, given the currently satisfied states of *p* and *q*, *do* fails as the cut (!) prevents any further backtracking. In short once *p* and *q* are satisfied, it becomes mandatory that *r* be satisfied, else the rule *do* fails. The sequence below shows how *N* changes its values.

```
p(1) → q(1) False
       ← Backtrack to p
p(2) → q(2)
!      → r(2) False
       Cannot backtrack; thus fails.
```

What would happen if there is an alternative path for *do* as in -

```
do:- p(N),q(N),nl,write(N),!,r(N).
do.
```

The effect would remain the same, thanks to the cut that makes the inference engine forget about the fork at *do* in toto! Now try adding the fact *r(2)* and see the effect.

Here is a simple program that helps select a car described by its brand name, color and cost.

```
car(palio,black, 450000).
car(hyundai, silver, 500000).
car(fiat, silver, 400000).
car(maruti, black, 20000).

get_car(Color,Cost_Less_Than):-
        car(Name, Color,Cost),!,Cost<Cost_Less_Than, nl, write(Name).
```

The goal

```
get_car(black,350000).
```

does not yield a solution in spite of the fact that one of the cars viz. the *maruti* satisfies the same. This is so because after the variables - *Name* and *Cost* are bound to *palio* and *450000*, the cut is crossed and the condition *350000 < 450000* is checked. Since the condition is not satisfied a backtracking is initiated but prevented due to the cut, yielding no solutions. Now re-order the facts for car with *car(maruti, black, 20000)* as the very first one and try the same goal.

25.11 RECURSION

A recursive procedure is one that calls itself. A simple recursive procedure often used for implementing loops, is given below –

```
repeat.
repeat:- repeat.
```

Such a program becomes handy when we have only one solution and we need to introduce backtracking. The repeat structure makes PROLOG's inference engine to believe that there are infinite number of solutions. Inspect the next program which will make things clearer.

```
getinput:-
redo,
grab(Ch),string_chars(C,[Ch]),
write(C),Ch= =13.
redo.
redo:- redo.
```

The *grab/1* predicate takes in a single character in ASCII directly from the keyboard while the *string_chars/2* predicate converts the input into a regular string which is displayed by *write*. The second argument of the latter predicate has to be a list which is why the variable *Ch* is presented as a list, a structure discussed in the next section. Both *grab* and *string_chars* are LPA PROLOG specific. You can always find the equivalent of these predicates from the documentations of the PROLOG compiler you are using.

The more important issue here is the role played by the predicate *redo*. As is obvious, the clauses for *redo* form a recursive loop. If you try to issue the goal as-

find that the program succeeds. This is so because the very first clause for *redo* itself says it is true.
things work to our benefit in the above program.
understand the working of the latter part of the above program. For the goal -*getinput*., the sequence
tion of subgoals is given below:

Execute *redo* which is essentially true.
Read a character from the keyboard and bind it to the variable *Ch*.
Convert it into a string form.
Display the character.
Check whether the input is the *Enter* key (ASCII code 13).
If yes, then it's a success; so quit.

backtrack to find alternatives of the previous subgoals. Since none of the predicates - *write*, *string_chars*
have alternatives, backtracking is done all the way back to *redo*. This has an alternative path
cond clause) which also says *redo*. This causes it to verify the truth of *redo* (again). Naturally this
ds thanks to the first clause for *redo* which is unconditionally true as in step (a).

Since the alternative path for *redo* has succeeded, carry on performing the sequence starting from step (b).

e now inspect some other typical PROLOG programs that use recursion.

n-counting:

following program counts down from N to 0.

```
count(0).
count(N):-
        write(`Old N=`),write(N),nl,
        N1 is N -1,
        write(`New N=`),write(N1),nl,
        count(N1).
```

Note that expression *N1 is N-1* is equivalent to the statement *N = N-1* in conventional languages. Interestingly,
ou replace it with *N = N-1*, PROLOG behaves differently. This is because it tries to verify whether the left
d side and right hand side are equal, which is never so. (What if we use *N = N-1*? Try and find out!)
When the goal *count(5)* is given the first clause for *count* fails making it backtrack to its second clause.
s decrements *N* and again calls *count* with *N=4*. The process continues till *N* equals zero and the first
use succeeds. Now try to explain what would happen if we rewrite the same program as

```
count(0).
count(N):-
        write(`Old N= `),write(N), nl,
        N1 is N -1,
        count(N1),
        nl,write(`New N= `),write(N1).
```

ctorial of a number

simple version of the program is shown below:

```
factorial(1,1):-!.
factorial(N,Fact_of_N):-
        Q is N-1,
        factorial(Q,Fact_of_Q),
        write(Fact_of_Q),nl,
        Fact_of_N is N*Fact_of_Q.
```

The first clause obviously states that factorial of 1 is 1. The cut ensures that this is the dead-end. The ond decrements *N* and recursively, calls *factorial*. The process continues till *N* becomes 1 and the first use terminates the recursion. The rest is mere unwinding by cumulatively multiplying 1 with 2, 2 with 3, 6 4 and so on. Run the program by giving the goal *factorial(4,X)* and inspect the output.
Another version of the program for computing the factorial of a number is given below:

```
factorial(N, Fact_of_N):-
factorial(N, Fact_of_N, 1, 1).

factorial(N, Fact_of_N, N, Fact_of_N):-!.

factorial(N,Fact_of_N, I, J):-
NextI is I + 1,
NextJ is J*NextI,
factorial(N,Fact_of_N,NextI,NextJ).
```

ote that the arity or the number of arguments the predicate (*factorial* in the present case) can vary. A trace ctorial(3,X) is shown below.

First clause *factorial(3,X)* calls *factorial(3,X,1,1)*

factorial(3,X,1,1) The second clause fails as the first and third arguments are not equal. The third clause is thus triggered.

NextI = 2, NextJ = 2

actorial(3,X,2,2) The second clause fails as the first and third arguments are still not equal.

NextI = 3, NextJ = 6

ctorial(3,X,3,6) The second clause succeeds this time as the first and third arguments are not equal and binds the ariable X to 6.

LISTS

PROLOG constitute elements separated by commas and enclosed within square brackets. The elements mprise of any data type. A list of integers looks like this

4, 8, 10, 12]

that of strings is given below:

[`jack`, `jill`,`jane1`].

A list essentially comprises two parts—a Head and a Tail. The former is the first element of the list while latter is the list comprising all other members. For instance the list of integers [2,4,8,10,12] has the integer as its head and the list [4,8,10,12] as its tail. List is a powerful structure which is why we also have a guage LISP which deals with just that.

If we wish to make a simple thesaurus of words we could start by adding facts as

```
thesaurus(`like`, `love`).
thesaurus(`like`, `fond`).
thesaurus(`like`, `similar`).
thesaurus(`like`, `akin`).
thesaurus(`give`, `offer`).
thesaurus(`give`, `bestow`).
thesaurus(`give`, `present`).
```

For each synonym a new fact has to be added. This makes the process tedious. Lists provide for a more mpact database as –

```
thesaurus(`like`, [`love`,`fond`,`similar`,`akin`]).
thesaurus(`give`, [`offer`,`bestow`,`present`]).
```

This could be further compacted by including the base word such as *like* as the head of the corresponding list.

```
thesaurus([`like', `love`,`fond`,`similar`,`akin`]).
thesaurus([`give', `offer`,`bestow`,`present`]).
```

One may now write a program to interpret the list as-
The head of the list is the base word while the tail contains its synonyms. We may also interpret it as

If a word belongs to the list, the rest of the words in the list are its synonyms.

We could also condense the thesaurus into just one list as-

```
thesaurus([[`like`, `love`,`fond`,`similar`,`akin`],[`give`, `offer`,`bestow`,`present`]]).
```

wherein all the words along with their respective synonyms form a list of lists.
A few sample programs that manipulate lists form interesting utilities and when coupled with other programs can reveal the true potential of the list structure. Some of them have been included below-

Appending an element to a list

```
add(Element, InList, [Element | InList]).
```

It is amusing to note that the above single statement can perform the following:

(a) Add an element to a list:
The clause for *add/3* takes in an element (*Element*) and the list to which it is to be added (*InList*) to give the output list with the element as its head and the *InList* as its tail..
For example,
add(1,[2,3,4],Out_List).
binds *Out_List to [1,2,3,4].*

(b) Verify whether an element is the head of the list as in –
add(1, [2,3,4],[1,2,3,4]).

(c) Get the head of a list whose tail is known.
add(Head, [2,3,4],[1,2,3,4]).

(d) Get the tail of a list whose head is known.
add(1, Tail, [1,2,3,4]).

Though (c) and (d) can be achieved using other techniques they have been stated here to emphasize the functionality of the same predicate.

Member of a list

```
member(X,[X|_]).
member(X,[_|Tail]):-member(X,Tail).
```

The clauses for *member* accept the member to be verified as the first argument and the list as the second argument.
The first clause states

X is member of the list if X is the Head of that list.

The second clause states

else check to see if X is the member of the tail of the list.

The *member* predicate in some PROLOG versions may be built into the compiler. Yet understanding the way it is written is important. The *member* predicate can also be used to extract the individual members of a given list as in the goal

```
member(X,[2,3,4,5]).
```

Append a list to another

The clauses for append are given below.

```
append([],L,L).
append([X|L1],L2,[X|L3]):-
append(L1,L2,L3).
```

The first argument, which is a list, is appended to the second, also a list, to give the third which is the appended version.

The first clause states that

if the list to be appended is an empty list then output the other list as the appended one.

The second clause may seem to confuse us initially! It takes the Head of the list *X* to be appended and adds it to the Head of the output (appended) list, *[X|L3]*, and calls *append* recursively with the Tail of the list to be appended, the list to which it is to be appended and *L3* as its arguments. Note that *L3*, the tail of the output list, gets values only while the unwinding of the recursive loop occurs. Inspect the trace below to comprehend the working of append.

append1([1,2],[4,5,6],O).

↓

X = 1
L1 = [2]
L2 = [4,5,6] L3 is free

↓

X = 2
L1 = []
L2 = [4,5,6] L3 is free

↓

L = [4,5,6] The first clause succeeds here binding L for the last recursive call thus triggering the unwinding.

↓

X = 2
L1 = []
L2 = [4,5,6]
L3 = [4,5,6] L3 is bound

↓

X = 1
L1 = [2]
L2 = [4,5,6]
L3 = [2,4,5,6]

L3 = [1,2,4,5,6]

↓

O = [1,2,3,4,6]

The member predicate can also be used in different ways. For instance it can be used to find the former elements of a list as in

```
| ?- append(A,[4,5,6],[1,2,3,4,5,6]).
A = [1,2,3]
```

ation as in

```
| ?- append([1,2,3],Z,[1,2,3,4,5,6]).
Z = [4,5,6]
```

It can be used to verify whether two known lists append to form another known list.

```
| ?- append([1,2,3], [4,5,6],[1,2,3,4,5,6]).
yes
```

Length of a list

The following program finds the length of a list.

```
length([],0).
length([_|Tail],Len):-
        length(Tail,Len1),
        Len is Len1+1.
```

The first clause which finally terminates the recursion, states

An empty list has zero length.

The second clause makes a recursive call to find the length of the tail of the list. The last subgoal computes the length of the list by incrementing *Len* while the recursion unwinds. The predicate can also be used to verify the length of a known list.

N^{th} member of a List

Given the index of an element, the following program can retrieve the associated element in a known list.

```
nthmember(1,[Head|_],Head):-!.
nthmember(N,[_|Tail],Item):-
N1 is N-1, nthmember(N1,Tail,Item).
```

The first clause states

If the index requested was 1 then the desired element is the Head of the list.

The second decrements the index *N* and recursively calls the predicate till the index is 1 so as to return the *Head* of the current *Tail* as the N^{th} member. As in the previous case this predicate too can be used to verify whether a known element is placed at the indexed location in the list.

The reader is urged to investigate list based programs for finding the index of an element in a list, replacing an element and inserting and deleting of elements within.

25.13 DYNAMIC DATABASES

PROLOG provides a novel way of inserting, modifying and deleting facts during runtime. It is thus possible to assert a statement, retract or modify it changing the behavior of the program in run time.

Let us take the example of creating a simple database of names to form a dynamic database. The program below also uses some other predicates which will be explained to clarify other issues of PROLOG.

```
ynamic name_db/1.   /*Declare name_db as a predicate that has just one argument, as
                        a dynamic one*/

ert_name:-
    write(`Enter name to be added correctly: `),
    add_name([]).
ert_name:-
    nl, write(`Name could not be entered; Possible Error!`).
iscard_name:-
    write(`Enter name to be deleted correctly: `),
    delete_name([]).
iscard_name:-
    nl, write(`Name could not be discarded; the entry does not exist!`).

dd_name(TmpList):-
    getb(Char), Char =\=13,string_chars(Str,[Char]),
    write(Str),add_name([Char\TmpList]).

dd_name(OutList):-
    nl,reverse_list(OutList,[],List),
    string_chars(String,List),
    assert(name_db(String)),
    write(`Added `), write(String), write(`to database`), nl.

elete_name(TmpList):-
    getb(Char),Char =\=13,string_chars(Str,[Char]),
    write(Str),delete_name([Char\TmpList]).
elete_name(OutList):-
    nl,reverse_list(OutList,[],List),
    string_chars(String,List),
    retract(name_db(String)),
    write(`Deleted`), write(String), write(`from database`), nl.

everse_list([],OutList,OutList).
everse_list([Head\Tail],TmpList,OutList):-
    reverse_list(Tail,[Head\TmpList],OutList).
```

e first statement informs the compiler that *name_db/1* is declared to be a dynamic predicate. Clauses for redicate can now be added or deleted during run time.

er_name/0 uses add*_name/1* to get the input name string to be added to the database. Since *getb/1* reads haracters entered via the keyboard, we need to collect and transform them into a string for proper storage. hat *add_name/1* is recursive and called with an empty list into which the characters are added one by The check for code 13 is done to see whether the Enter key is pressed and if so the output list containing haracters in reverse order is reversed using *reverse_list/3* to put them in the right form. The list is converted tring using *string_chars/2* and then stored into the dynamic database using *assert*.

sert and *retract* thus constitute a way to add or delete dynamic clauses for *name_db* during run time. The program also features predicates for *discard_name/0* which deletes the fact from the dynamic database *retract*.

ogether *assert* and *retract* form a powerful tool to manipulate the execution of code during run time. ompile the program and issue the goal

enter_name.

Enter the name to be added. Assume you typed *Prakash* for a name. Now you can go ahead to verify whether the same is entered in the database by using the goal

name_db(Name).

you would get the output as

| ?- Name = Prakash

You could add more names and imagine you accidentally entered the name *Devil* and wish to delete it. Use the goal

discard_name.

and enter the name *Devil*. Try issuing the goal *name_db(Devil)* again and lo and behold you will discover that the *Devil* has vanished!

Now that we have seen how a dynamic database can be generated and manipulated imagine a case wherein you could also add, delete or modify *code* on the fly. In the PROLOG environment this can be achieved by adding or deleting rules. Recollect the *count/1* program we used to comprehend recursion. Let's see how we can add the whole program dynamically. The actual program by itself will contain a line of code indicating that the clause for count/1 is dynamic i.e.

Issue the following goals –

```
|?:-dynamic count/1.
|?-assert((count(N):-(N = = 0 -> true; (N1 is N-1, write(N1),nl,count(N1))))).
```

Note that the clauses for *count/1* are within and have been written in a slightly different style. However, you will find that the code is semantically the same as the one discussed earlier for *count*.

Once the rule for *count* has been asserted you may proceed to issue goals that use *count* such as *count(4)*.

The above down counter could be easily retracted and an upcounter asserted in lieu of the same. This would obviously affect the behavior of the main program that uses *count* during run time.

Let's take one last look at the implications of such dynamic databases before we jump to the next section.

```
:-dynamic exec_something/1.
:-dynamic code_base/1.

do:-
        assert(code_base(assert((exec_something(X):- write(X),nl)))).
```

This does not just assert code (clauses) but also can form a dynamic database of clauses.

Try the following sequence of goals after compiling the above piece of code.

```
?- do.
yes.
? –code_base(Code),Code.
Code = assert((exec_something(_32868) :- write(_32868)))
?-exec_something('Hello there!').
Hello there!
yes
```

The numbers that follow the underscore are the unbound variables. With some minor modifications, one could use

```
assert(code_base(assert( (exec_something(another prolog program) ) ) ) ).
```

and accordingly maintain a database of programs in run time.

25.14 INPUT/OUTPUT AND STREAMS

PROLOG can read and write terms from one or several files. These thus form the streams of data that are either input (input stream) or output (output stream). The use of some typical built-in predicates (BIPS) is described below.

(i) *see/1 and seen/0*

see takes the input file name as its argument and opens the file for use. The terms within the file may be now read with other BIPS like *read/1* that reads a term or *get/1* that gets a character. The input file has to be closed after reading and this is done by the BIPS *seen/0*.

Assume that the input file *likings.pl* contains facts about *Who likes what food.*

```
likes(prakash,sandwiches).
likes(priya,eggs).
likes(shiva,cheese).
```

A typical program that reads and writes onto the console is given below

```
read_4rm_file:-
        see(`likings.pl`),
        redo,
        read(Likes),
        (Likes = end_of_file, nl, write(`Thats all folks!`),nl,!;
        likes(Name, Food), write(Name), write(`likes`), write(Food),nl,fail),
        seen.

redo.
redo:-
        redo.
```

The output on issue of the goal *read_4rm_file.* is shown below

```
| ?- read_4rm_file.
prakash likes sandwiches
priya likes eggs
shiva likes cheese

Thats all folks!
yes
```

(ii) *tell/1 and told/0*

These just perform the reverse operations as *see* and *seen* viz. writing to a file.

```
write_2_file:-
        write(`Enter term :`),
        tell(`likings.pl`),
        redo,
        read(Likes),
        (Likes = `Over`, told, nl, write(`Thanks for the inputs!`),nl;
        write(Likes),put(46), nl, fail).

redo.
redo:-
        redo.
```

The output looks as shown below

```
| ?- write_2_file.
Enter term :|: likes(jane,eggs).
|: likes(jack,burger)
.
|: Over.

Thanks for the inputs!
yes
```

25.14.1 Consulting and Reconsulting Knowledge Bases

The PROLOG system allows you to load and reload knowledge bases or programs. You can use *consult/1* to load a file to the system. Suppose you have written the clauses for *read_4rm_file* and *write_2_file* into a file named *my_file_io.pl*, then the program may be consulted (or loaded into the system) by

```
| ?- consult(`my_file_io.pl`).
```

Now all clauses within can be used.

You may also wish to reload a modified version of the same file. For this you can use *reconsult(`my_file_io.pl`)*. Naturally the older version is overwritten. Multiple knowledge bases may also be loaded or reconsulted. For instance –

```
consult([`my_first_program`, `my_data.dat`]).
```

If the file extension is missing it is assumed to be a *.pl* (PROLOG program) file.

25.15 SOME ASPECTS SPECIFIC TO LPA PROLOG

Logic Programming Associates (LPA) PROLOG offers quite a few toolboxes that allow writing enhanced PROLOG programs. This book illustrates a couple of them in brief. The readers are advised to go through the documentation available online at *http://www.lpa.co.uk/*. The site currently hosts free trial versions of its goodies.

25.15.1 Chimera Agent Engine

The new buzz word in AI is the *agent* – of course the *Intelligent Agent*. We discussed the concept of an agent in brief in section 16.3 but this entity may still seem alien. In the following sections we will describe an agent

...hen go ahead to writing a sample program to realize agents using LPA's Chimera Agent Engine. ...e word *Chimera* possibly stands for a grotesque mythical female being with a lion's head, a goat's ...a tail that resembles that of a serpent, all contrary characteristics, Chimera in the current context ...be an agent that is part server and part client and features properties of a component in a distributed ...environment. Well, you can always impart more features to it! As mentioned earlier, before we ...k at it, we need to comprehend the concept of an agent.

...– Their Definition and Characteristics

...16.3 we had opened up the term *agent* whose definition was described as vague. Several researchers ...ided definitions based on the work carried out by them. Discussing each of them would be like ...the Pandora's box (much like the definitions of intelligence and AI!). Russel and Norvig have described ...to be something, that is capable of perceiving the environment which it is in, using sensors and ...on this environment through effectors. A robot could rate high as an example of an agent. It can ...environment using sensors and then act accordingly. Naturally it changes the state of the environment ...act (since its position, for instance, will have changed, its battery power drained by a fraction due to ...after the elapse of time). It may look as if this definition allows a printer driver or for that matter a mere ...r program to be classified as an agent. These programs can be visualized to be sensing (perceiving) ...from a user within its environment, acting (processing the inputs) and effecting the environment ...by displaying or printing the results). At a very low level such programs too could be categorized as ...but we do not wish to open a debate on this. The definitions put forth by many should be always ...ed from the standpoint of a few already propounded characteristics agents should possess. Not all ...possess all these characteristics. However more the number of these characteristics it possesses, more ...agent. Table 25.3 lists some of the major characteristics of an agent.

...ble 25.3 *Typical Characteristics of an Agent*

definitions available may be all the more that the definition of an agent is not all that complete and remains should therefore be cautious and take a second and serious look at anything that we feel is being termed an agent.

A Chimera Agent

A Chimera agent has basically two components—a socket and a handler. The former works on the well known Windows sockets (Winsock) and is used by the agents to communicate amongst each other. It supports TCP/IP network communications. The latter handles the events and governs what should happen when the agent perceives something. The body of the handler is generally a PROLOG program which makes the agent do what it is designed to, when the event is reported. In order to use the Chimera Agent system it is necessary to load the same into the PROLOG system. This is done by –

```
:-ensure_loaded(system(chimera)).
```

as the very first clause of every agent program.

Agent Handler, Creation and Link Establishment

Table 25.4 shows the development of two simple Chimera agents named *me* and *you*. Note that the Chimera system has been loaded in two separate instantiations of WIN-PROLOG.

Every agent has to have a *handler/3* which defines what is to be done when an event takes place. Naturally for every event there ought to be a handler clause. The agent *me* has a handler predicate *me_handler* while the other *you* has *you_handler* to take care of these tasks. All handlers have an arity of 3. The explanation of each of them is as given below.

```
handler(Name, Link,Event)
```

The first argument is the name of the agent (*you*, for instance) that is seeking this agent (*me*), the second is integer that designates the link between the two agents and the third is the actual event that has occurred d which needs to be serviced. The *me_handler/3* in the Table 25.4 writes the *Name*, *Link* and *Event*, finds at is to be done next with the input event and then sends (posts) back the processed data to the querying nt. The *you_handler* just writes the data received.

Now that the handlers are in place we need to effect the formal creation of the agents. An agent can be ted with the predicate *agent_create/3* comprising the *Name* of the agent, the associated *handler/3* predicate e and the *Port* number through which it can be accessed. After both the agents have been created the next is to establish a connection between them using *agent_create/4* which has a different interpretation. This icate has four arguments. The first is the *name* of the local agent which is being connected. The second is nk number which is used at later stages to define this link. The third and fourth arguments comprise the dress of machine at which the other agent is running and the associated port number.

e scene is now set for the agents to communicate. Passing information can be done by using *agent_post/3* uses the posting agent's name, the link number that describes the connection and the event (data) to be d on. When the data is received at the other end, the corresponding handler of that agent is invoked.

copying the programs given in the columns Agent#1 and Agent#2 of Table 25.4 on two different files pened in two instantiations of WIN-PROLOG. Compile both in their respective instantiations and issue ls in the order shown in the Table 25.4. Try and interpret the data being received by tracing the clauses respective handlers.

	These lines are mandatory and load the CIPA Chimera Agent System.	
	Handlers define what is to be done when a request comes from an agent with an arity of three comprising of the Name of the agent, the Link number and an Event, it services a request from either the local or remote agent. For instance, the first clause for me_handler handles an agent on the specified link that sends an event of the form *to_do(Something)*. You could now write PROLOG code (as has been done) within the body of the clause to process *Something* and maybe return the processed data to the requesting agent.	
	This creates the agent using the Chimera predicate agent_create/3. The first argument is the name of the agent, the second is its associated handler predicate name and the last is the TCP/IP port address through which communication can be effected.	create_You_agent:- agent_create(you, you_handler,4000).
	This clause establishes a connection between the created agent and another running on another/same machine in a different port and assigns a link number for future reference. agent_create/3, the predicate that performs this task has the following arguments in this order: Name of the agent, Link number, IP-address and Port.	establish_connection_with_Me_agent:- agent_create(me,2,'202.143.90.94',3000).

Agent#1 WIN-PROLOG Instantiation#1	Comments	Agent#2 WIN-PROLOG Instantiation#2
Name: **me**	Agent: Name	Name: **you**
...loaded(system(chimera)).	These lines are mandatory and load the LPA Chimera Agent System.	:-ensure_loaded(system(chimera)).
...dler(Name, ...(DoThis)):- ...Name - Link - ...DoThis)), ...to_do_with(DoThis,NowDoThis), ...ssage(Name, ...NowDoThis)), nl.	Handlers define what is to be done when a request comes from an agent. With an arity of three comprising of the Name of the agent, the Link number and an Event, it services a request from either the local or remote agent. For instance, the first clause for *me_handler* handles an agent on the specified link that sends an event of the form *to_do(Something)*. You could now write PROLOG code (as has been done) within the body of the clause to process *Something* and maybe return the processed data to the requesting agent.	you_handler(Name, Link,get(Stuff)):- /*You could process Stuff here and send it back to the agent me.*/ nl,write(`Recd:—`), write(Stuff). /*The following handler clause below is a catch all clause*/ you_handler(Name, Link,_):- nl,write(`Ooooops`).
...dler(Name, ...(NowDoThis)):- ...NowDoThis). ...following handler clause ...is a catch all clause*/ ...ndler(Name, Link,_):- ...Pardon Me!`),nl. ...to_do_with(`Hello`,`Welcome`). ...to_do_with(`How are ...Fine Thank u`). ...to_do_with(`Pardon ...Thats ok`).		
...e_Me_agent:- ...t_create(me, ...ndler,2000).	This creates the agent using the Chimera predicate agent_create/3. The first argument is the name of the agent, the second is its associated handler predicate name and the last is the TCP/IP port address through which communication can be effected.	create_You_agent:- agent_create(you, you_handler,4000).
...blish_connection_with_You_agent:- ...t_create(me,1,`202.141.80.66`,4000).	This clause establishes a connection between the created agent and another running on another/same machine in a different port and assigns a link number for future reference. agent_create/4, the predicate that performs this task, has the following arguments in that order: Name of the agent, Link number, IP address and Port.	establish_connection_with_Me_agent:- agent_create(you,0,`202.141.80.66`,2000).

	Notice that the link between agents *me* and *you*, formed on the left side column using agent_create/4, is numbered 1 and that on the right side that establishes a connection between agents *you* and *me* is numbered 0.	
post_to_You_agent(String):- *agent_post(Name,1,to_do(`foo`)).*	In order to post a message to the other agent the Chimera system provides the agent_post/3 predicate which has the name of the posting agent, link between the posting and the receiving agent and the event that triggers the relevant handler clause at the receiver.	*post_to_Me_agent(String):-* *agent_post(Name,0,to_do(String)).*
| ?- create_Me_agent. *| ?-* *establish_connection_with_You_agent.* *| ?- post_to_You_agent(`My My`).*	Goals to be given on two separate instantiations of WIN-PROLOG.	| ?- create_You_agent. | ?- establish_connection_with_Me_agent. | ?- post_to_Me_agent(`Hello`).

A Three-Agent Scenario

Let's now look into a scenario where three agents communicate. Table 25.5 provides the PROLOG source code for each of these. Ensure that you change the IP addresses to those of the machines where the agents are to run. Naturally the three agents viz. *antonym*, *synonym* and *sentence* are to be loaded and compiled on three separate instantiations of WIN-PROLOG. For brevity these have been christened as *ant*, *syn* and *sen* respectively in our discussions. The *ant* provides the antonym of a given word and also passes this information to the agent *syn* that provides for its synonym. The latter also proactively contacts *sen* and requests it to provide an example sentence for the input word. Together the three agents provide the user with antonyms, synonyms and an example sentence that uses the input word. The prime objective of discussing this is to emphasize the fact that one can build a Multi-Agent System (MAS) using Chimera. Agents could talk things over resolve mutually and bring back useful information. The distributed nature also allows the flexibility of computation being carried out elsewhere and also the reuse of code already built.

If you inspect the three pieces of code you will observe that after the Chimera system is loaded, the handlers do the basic jobs of finding the antonyms, synonyms and example sentences. They of course post messages to the other agents to derive the information. It can be seen that first the user query is passed on to the *ant* which immediately provides the antonym of the word (provided of course the same exists within its database). It also sends the word to *syn* that in turn provides the synonym and passes on the word to *sen*. This searches for a sentence within its database which has the input word and sends it straight to *ant* thus completing a loop. Figure 25.1 depicts this flow of information. The striped arrows indicate the communication links established between the agents while the chequered ones depict the flow of information. Ensure that you issue the goals as per the schedule indicated in the last row of the Table 25.5 lest a connection is attempted before the other agent comes into being.

Other Chimera predicates

Some other useful predicates made available within the Chimera system include

(a) agent_dict/2

This predicate takes in the link number to give the name of the agent on the link.

| ?- agent(0,Agent).

Table 25.5 Three Chimera Agents—ant, syn and sen

ANTONYM AGENT (ANT) WIN-PROLOG instantiation#1	SYNONYM AGENT (SYN) WIN-PROLOG instantiation#2	SENTENCE AGENT (SEN) WIN-PROLOG instantiation#3
:-ensure_loaded(system(chimera)). antonym_handler(Name,Link,get_info String)):- get_antonym(String,Antonym), write(`The Antonym of the word `), write(String), write(`is: `), write(Antonym), nl, write(`Hold on! Let me try & get u more information on this`),nl, agent_post(antonym,1,get_synonym (String)),nl. antonym_handler(Name,[],get_info1 (String)):- write(`A2`), write(String),nl, get_antonym(String,Antonym), write(`The Antonym of the word `), write(String), write(`is `), write(Antonym),nl, write(`Hold on! Let me try & get u more information on this`),nl, agent_post(antonym,1,get_synonym (String)), write(`A2`), write(` Posted String to synonym`),nl. antonym_handler(Name,Link,take_ syn(String)):- write(`The Synonym of the word is: `), write(String),nl. antonym_handler(Name,Link,take_ sentence(String)):- write(`A4`), write(String),nl, write(`Here is an example sentence: `), write(String),nl. antonym_handler(Name,Link,Event):- nl,write(`ANT:`). get_antonym(String,Antonym):- antonym(String,Antonym).	:-ensure_loaded(system(chimera)). :-dynamic agent_link/3. synonym_handler(Name,Link,get_ synonym(String)):- get_syn(String,Synonym), agent_link(synonym,antonym,Lnk), agent_post(Name,Lnk,take_ syn(Synonym)), agent_link(Name,sentence,Lnk1), agent_post(Name,Lnk1,send_ example(String)),nl. synonym_handler(Name,Link,Event):- nl, write(`SYN:`). get_syn(String,Synonym):- synonym(String,Synonym). get_syn(String,Synonym):- synonym(Synonym, String). synonym(`like`, `love`). synonym(`up`,`above`). synonym(`beautiful`, `pretty`). synonym(`perfect`,`class`). create_synonym_agent:- agent_create(synonym,synonym_ handler,Port). establish_connection_with_antonym_ agent:- agent_create(synonym,Link,`202.141. 80.66 `,2000), assert(agent_link(synonym,antonym, Link)). establish_connection_with_sentence_ agent:- agent_create(synonym,Link,`202.141. 80.66 `,6000), assert(agent_link(synonym,sentence, Link)).	:-ensure_loaded(system(chimera)). :-dynamic agent_link/3. sentence_handler(Name,Link,send_ example(String)):- get_sentence(String,Sentence), agent_link(sentence,antonym,Lnk), agent_post(Name,Lnk,take_sentence (Sentence)),nl. sentence_handler(Name,Link,Event):- nl,write(`SEN:`). get_sentence(String,Sentence):- sentence(WordList,Sentence), member(String,WordList),!. get_sentence(_,`Sorry I cannot help you!`). sentence([`love`,`hate`,`superior`],`love is superior to hate`). sentence([`love`,`beautiful`,`true`, `nothing`], `nothing is as beautiful as true love`). sentence([`perfect`], `God is perfect`). create_sentence_agent:- agent_create(sentence,sentence_handler, 6000). establish_connection_with_antonym_ agent:- agent_create(sentence, Link, `202. 141.80.6 6`, 2000), assert(agent_link(sentence, antonym, link))

```
get_antonym(String,Antonym):-
antonym(Antonym,String).

get_antonym(_,`Sorry: Cannot
comprehend this word`).

antonym(`like`, `hate`).
antonym(`up`,`down`).
antonym(`beautiful`, `ugly`).
antonym(`perfect`,`imperfect`).

create_antonym_agent
:-
agent_create(antonym,antonym_
handler,2000).

establish_connection_with_antonym_
agent:-
agent_create(antonym,0,`202.141.80.66
`,2000).
establish_connection_with_synonym_
agent
:-
agent_create(antonym,1,`202.141.80.66
`,49152).

get_word_info(String):-
agent_post(antonym,[],get_info1
(String)).
```

Goals to be issued in the order shown (there could be other correct orders too.)

1. create_antonym_agent. 4. establish_connection_with_ antonym_agent. 8. get_word_info(`love`).	2. create_synonym_agent. 5. establish_connection_with_ antonym_agent. 6. establish_connection_with_ sentence _agent.	3. create_sentence_agent. 7. establish_connection_with_ antonym_agent.

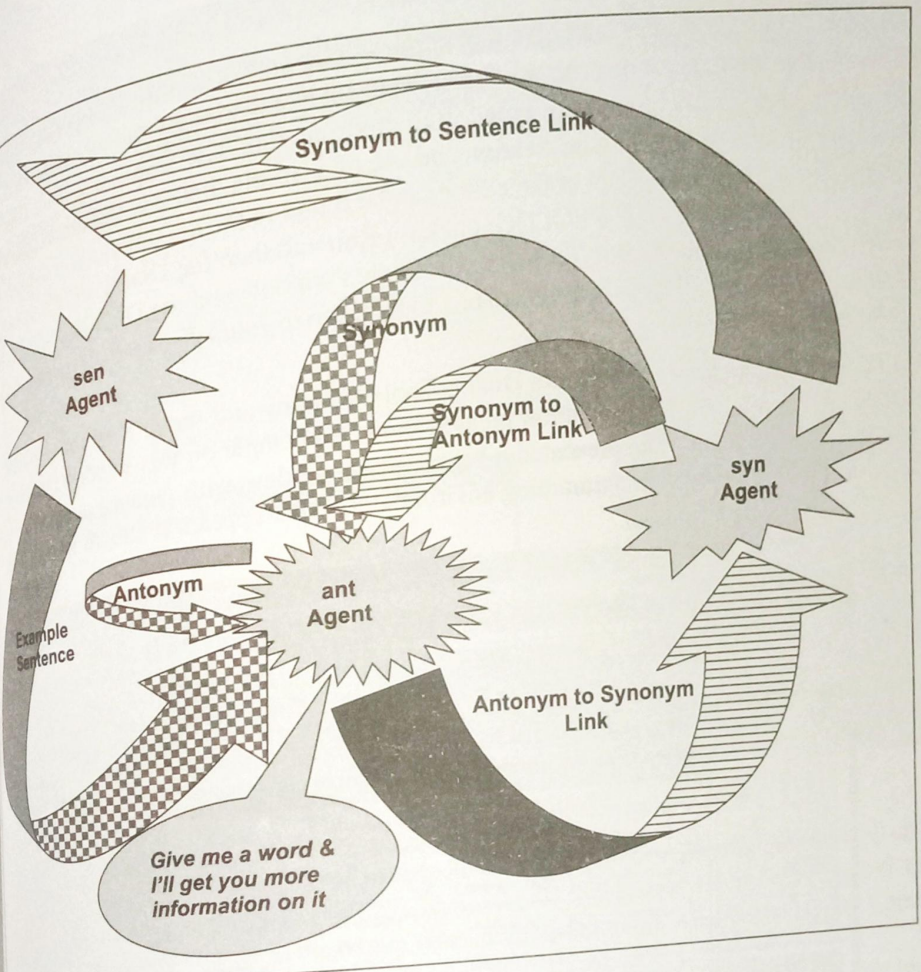

Fig. 25.1 *Flow of Information Amongst the Three Agents*

The second *agent_data* predicate has an arity of 5 and given the agent name and link number returns a random atom which is the Winsock Socket handler, a string representing the host IP address and the associated port.

```
| ?- agent_data(me, 0, Sock, HostIP, Port).
```

would possibly result in –

```
Sock = hdlsnfgj,
IP = 202.141.80.66,
Port = 4000
```

agent_close/1

The predicate takes in the name of the agent and shuts itself down. It is always best to close an agent when it has performed its functions. Try to incorporate the same as event handlers in the above programs.

Bear in mind that the above agents are not yet intelligent. You could make them so by providing private handler programs that will facilitate intelligent responses by the respective agents. Having talked about their functionality and implementation using Chimera we go ahead to providing a better method of working with such systems.

25.15.2 Creating Dialogues

Dialogues provide the visual effect and give a better aesthetic appearance to inputs and outputs. Though the same does not have anything to do with AI, it has been included here to encourage readers to write lively programs in the otherwise monotonously typed drab environment. The following discussion is solely LPA WIN-PROLOG specific. We discuss the creation and use of a simple dialog using the WIN-PROLOG Dialog Editor plug-in. A complete discussion of the editor can be found in the documentation available online in the LPA website. After invoking WIN-PROLOG; click on Run and then the Dialog Editor option.

Three windows pop up; the Dialog Tool Box, the Dialog window and the Dialog Code window. Move the cursor over each of the control icons on the Tool box to familiarize yourself with their names. Try the following sequence to build a simple dialog.

(a) Click on the *static/label* tool on the Dialog Tools Window and move the mouse pointer to the New Dialog Window. With the left button pressed draw a rectangle on this window. Double click on this static label and you will be presented a bigger Style window with numerous options, most of which conform to the Windows programming environment. What appears is shown in Fig. 25.2.

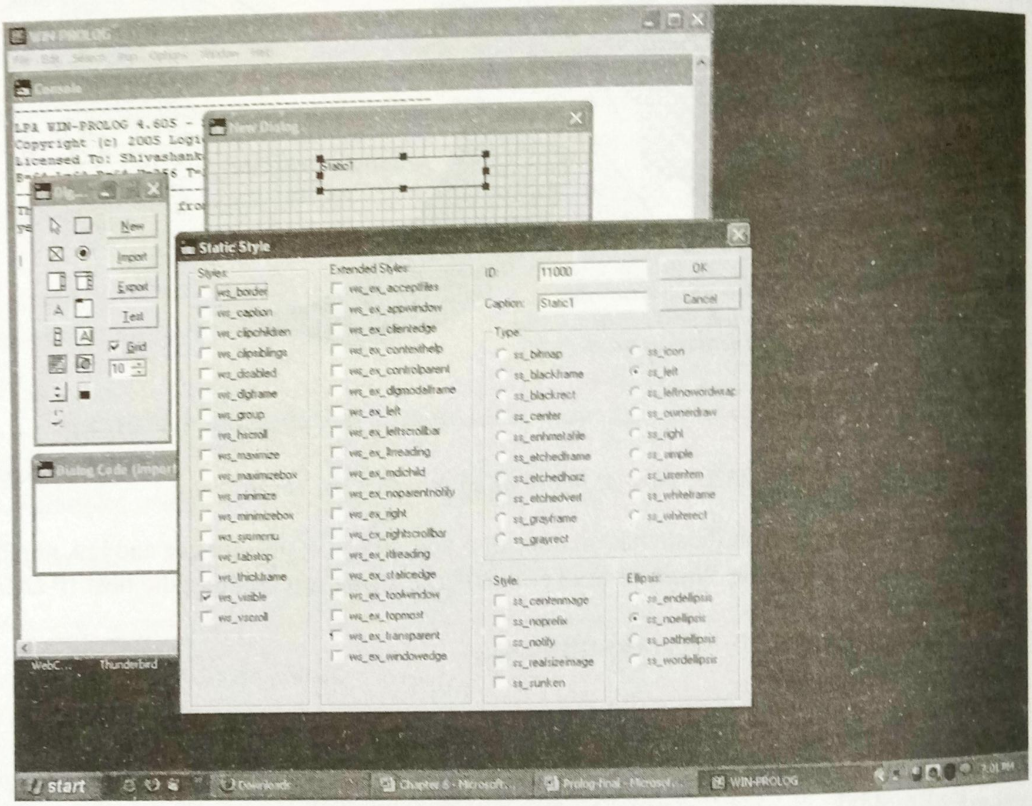

Fig. 25.2 *The Dialog Editor*

Change the caption to "*Enter your name below*", and click the *ss_center* button. The former will change the text (*Static1*) that appears in the static control while the latter will place the text in the centre. Note a number that appears against ID. This acts as the control's handle. Change this to 1 and click OK. What you will see is shown as Fig. 25.3.

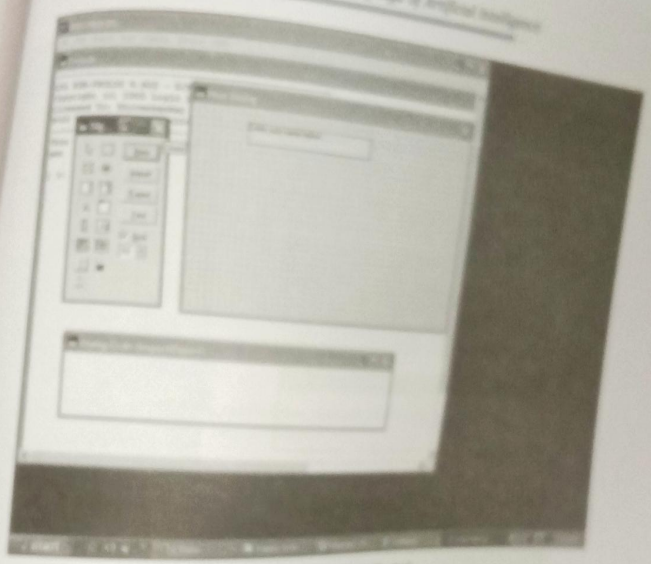

Fig. 25.3 *The Static Control*

Now add an *Edit* control in a similar manner below the static control. Double click and the same kind of window that appeared before will pop up. Change the ID to 2 and the remove the Caption and click OK. Finally add a button control to the dialog beneath the edit control, change the ID and Caption to 3 OK. Finally add a button control to the dialog beneath the edit control, change the ID and Caption to 3 and "OK" respectively in a similar manner. The output is shown in Fig. 25.4.

You may now see how the dialog would look like during run time by clicking on the Test button on the Tool box. Resizing and moving of these controls can be easily done using the mouse. Generating the code for this is simple. Just click on Export and you will see the code in Dialog Code (sometimes referred to as IE) window. Cut and paste this code into a new file within the WIN-PROLOG environment. You will note that a clause for the predicate new_dialog has been generated. This dialog can be used only after it has been invoked by a call to this clause viz.

```
| ?- new_dialog.
```

The dialog is now ready for use by a program. You will of course have to display the window using the predicate wshow/2 with the window name and number as its arguments. The dialog will also require a handle for which the predicate window_handler/2 with arguments viz. the dialog name and the handle name, is to be invoked.

Inspect the code below to see how things work. More importantly look at what the event handlers do by correctly the text of the dialog typing, moving the cursor, etc.) with the handler code. Most of this part is based on Windows programming. Compile and issue the goal start_dialog. Use the dialog and interpret the handlers. One typical handler clause is described below. Based on this, the others can be easily comprehended.

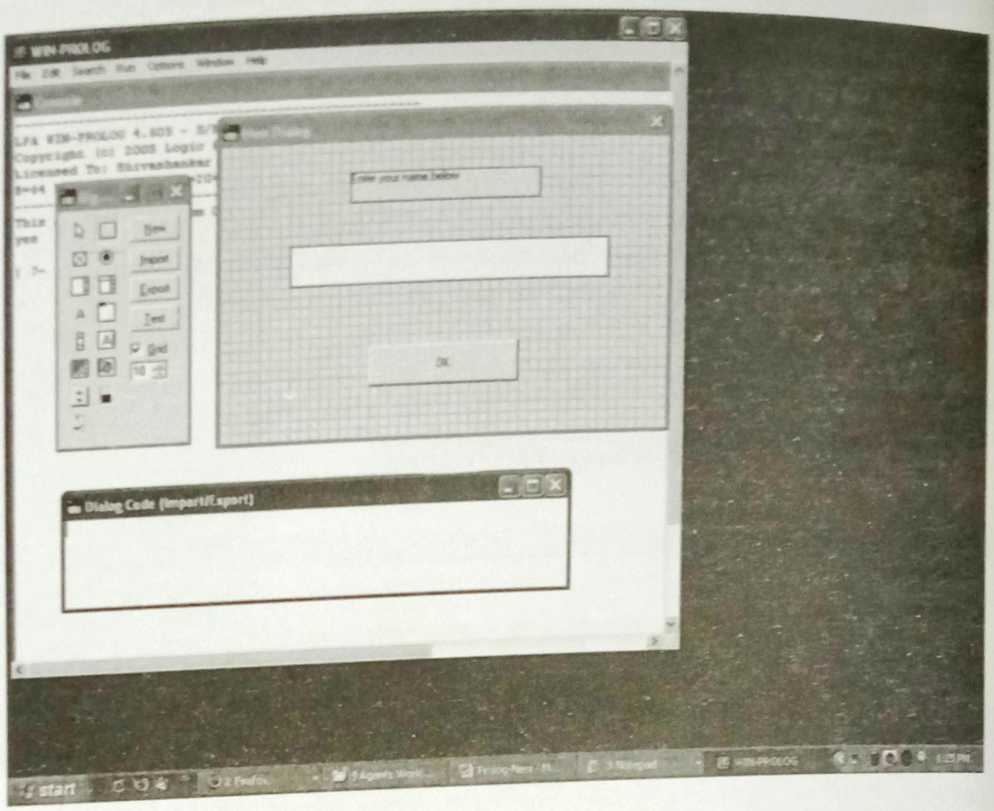

Fig. 25.4 *The Final Dialog*

```
start_dialog:-
new_dialog,
wshow(new_dialog,1),
window_handler(new_dialog, new_dialog_handler).
new_dialog_handler((new_dialog,2),WinMsg, Data,Result):-
WinMsg = msg_mouseenter,
!,
wtext((new_dialog,1), `You are doing well - continue!` ),
window_handler(new_dialog,Message,Data,Result).

new_dialog_handler((new_dialog,2),WinMsg,Data,Result):-
WinMsg = msg_mousemove,
!,
wtext((new_dialog,1), `You are doing well - continue!` ),
window_handler(new_dialog,Message,Data,Result).

new_dialog_handler((new_dialog,2),WinMsg,Data,Result):-
WinMsg = msg_mousehover,
!,
wtext((new_dialog,1), `Enter your name!` ),
window_handler(new_dialog,Message,Data,Result).
%write(WinMsg),wclose(new_dialog).
```

```
new_dialog_handler((new_dialog,2),WinMsg,Data,Result):-
WinMsg = msg_mouseleave,
!,
wtext((new_dialog,1), `Enter your name!` ),
window_handler(new_dialog,Message,Data,Result).
%write(WinMsg),wclose(new_dialog).


new_dialog_handler((new_dialog,3),WinMsg, Data,Result):-
WinMsg = msg_button,
!,
wtext((new_dialog,1),`Hello - You have a nice name!` ),
wtext((new_dialog,2), " ),
window_handler(new_dialog,Message,Data,Result).
%write(WinMsg),wclose(new_dialog).


new_dialog_handler((new_dialog,3),WinMsg, Data,Result):-
WinMsg = msg_mousehover,
!,
wtext((new_dialog,3), `Rt button 2 continue Lt 2 Quit` ),
window_handler(new_dialog,Message,Data,Result).

new_dialog_handler((new_dialog,3),WinMsg, Data,Result):-
WinMsg = msg_mouseleave,
!,
wtext((new_dialog,3), `OK` ),
window_handler(new_dialog,Message,Data,Result).


new_dialog_handler((new_dialog,3),WinMsg,_,_):-
WinMsg = msg_rightdown,
!,
write(WinMsg),wclose(new_dialog).


new_dialog_handler(Window,Message,Data,Result):-
write(Window - Message - Data - Result),nl,ttyflush,
window_handler(Window,Message,Data,Result).

new_dialog_handler((new_dialog,2),WinMsg, Data,Result):-
WinMsg = msg_mousemove,
!,
wtext((new_dialog,1), `You are doing well - continue!` ),
window_handler(new_dialog,Message,Data,Result).
```

The handler takes in the first argument (*control_handler,ID*), and returns the message from the window (WinMsg) and the data and result pertaining to the dialog/window. Based on the message or event returned, which in the present case is a movement of the mouse, the next part of the code is executed. *wtext/2* facilitates reading or writing of text onto or from a control. It has two arguments—the control handle and its ID and the

text to be written or read from the control. If the text is a variable, it reads the text on the control; else it places the text on it. The *window_handler/4* is called again to effect a cyclic process and to maintain the event driven actions. Its first argument is the dialog handle and the second the message. The third and fourth pertain to the data and the result respectively. Both these provide information specific to the window. Note the outputs reflected on the console due to the last clause of the dialog handler. Observe the mouse movement and events and the actual messages (*WinMsg*) received by the program. You could use these for enhancing the event handling process.

SUMMARY

This chapter introduced the concept of programming in PROLOG. Most of the key issues in PROLOG programming have been covered to facilitate the reader to kick-start the process of writing code for AI programs. In this chapter we have revisited the concept of an agent and discussed ways to implement real agents using LPA's Chimera Agent system. The creation and use of dialogs in the WIN-PROLOG system have also been discussed.

EXERCISES

1. A family tree is always something many of us would want to maintain. Try using PROLOG to code a family tree wherein you could query for a person's father, mother, son, daughter, siblings, ancestors, descendents and the like. Also depict the outputs in the form of a tree.
2. Make a repository of utilities that manipulate lists. These could include sorting, finding the sum of the numbers of the list, flattening a list (converting for instance [a,[b,[c],p],q] to [a,b,c,p,q]), union and intersection of two lists, etc.
3. Attempt to write a program that could change its behavior (code) based on interactions with the user.
4. PROLOG has been widely used in natural language processing applications. Given a set of context free grammar rules, write a program that will accept a sentence and verify whether it conforms to the grammar.
5. Write an agent program that will play the game of tic-tac-toe with you. Add a dialog window that will facilitate the input and display the results.
6. Most of the time we learn a game by observing. With reference to exercise 6, can you implement another agent that just observes the two of you (the agent in exercise 6 and you) playing tic-tac-toe and *learns* to play the game? After a certain period this observer and learner should be in a position to play the game with you! (Hint: You could notify the observing and learning agent as to who has won the game along with the final board positions.)
7. In section 6.2 we discussed bidirectional search. Construct two agents which traverse a graph from the two ends of a given path connected by nodes, to discover the shortest path between the source and the destination. One agent delves from the source while the other from the destination. Ensure proper communication between them so that each is aware of the nodes visited by the other.