Chapter 1

Today's practitioner of Artificial Intelligence is handicapped unless thoroughly familiar with both Lisp and Prolog, for knowledge of the two principal languages of Artificial Intelligence is essential for a broad point of viewToday's practitioner of Artificial Intelligence is handicapped unless thoroughly familiar with both Lisp and Prolog, for knowledge of the two principal languages of Artificial Intelligence is essential for a broad point of view.

learning about goal-oriented programming through Lisp programs is like reading Shakespeare ina language other than English

an introduction to Prolog is important for all students of Computer Science, for there is no better way to see what the notion of whattype programming is all about.In particular, the chapters of this book clearly illustrate the difference between how-type and what-type thinking

Part A: The Prolog Language

Chapter 1: An overview of Prolog

Prolog is a programming language for symbolic, non-numeric computation. It is specially well suited for solving problems that involve objects and relations between objects. Figure 1.1 shows an example: a family relation. The fact that Tom is a parent of Bob can be written in Prolog as: `parent( tom, bob)` .

Here we choose parent as the name of a relation; tom and bob are its arguments

Family Tree Program

```
1 parent( pam, bob).
2 parent( tom, bob).
3 parent( tom, liz).
4 parent( bob, ann).
5 parent( bob, pat).
6 parent( pat, jim).
```

This program consists of six clauses. Each of these clauses declares one fact about the parent relation.

```
?- parent( bob, pat).
```

true *1*

```
?-
```

```
    parent( liz, pat).
```

false

```
?- parent( tom, ben).
```

false

```
?- parent( X, liz).
```

X = tom

Prolog's answer will not be just 'yes' or'no' this time. Prolog will tell us what is the (yet unknown) value of X such that the above statement is true

```
?- parent( bob, X)
```

X = ann
X = pat

We may now want to see other solutions. We can say that to Prolog (in most Prolog implementations by typing a semicolon), and Prolog will find other answer. If we request more solutions again, Prolog will answer 'no' because all the solutions have been exhausted

Our program can be asked an even broader question: Who is a parent of whom? Another formulation of this question is: Find X and Y such that X is a parent of Y. This is expressed in Prolog by:

```
?- parent(X, Y).
```

X = pam,
Y = bob
X = tom,
Y = bob
X = tom,
Y = liz
X = bob,
Y = ann
X = bob,
Y = pat
X = pat,
Y = jim

We can stop the stream of solutions by typing, for example, a period instead of a semicolon (this depends on the implementation of Prolog).

Our example program can be asked still more complicated questions like: Who is a grandparent of Jim? As our program does not directly know the grandparent relation this query has to be broken down into two step

(1) Who is a parent of Jim? Assume that this is some Y.

(2) Who is a parent of Y? Assume that this is some X.

```
?- parent( Y, jim), parent( X, Y).
```

X = bob,
Y = pat

If we change the order of the two requirements the logical meaning remains the same

```
?- parent( X, Y), parent( Y, jim).
```

X = bob,
Y = pat
false

In a similar way we can ask: Who are Tom's grandchildren

```
?- parent( tom, X), parent( X, Y).
```

X = bob,
Y = ann
X = bob,
Y = pat
false

Yet another question could be: Do Ann and Pat have a common parent? This can be expressed again in two stePs:

(1) Who is a parent, X, of Ann?

(2) Is (this same) X a parent of Pat?

```
?- parent( X, ann), parent( X, pat).
```

X = bob

Our example program has helped to illustrate some important points: o It is easy in Prolog to define a relation, such as the parent relation, by stating the n-tuples of objects that satisfy the relation.

(1)The user can easily query the Prolog system about relations defined in the program.

(2)A Prolog program consists of clauses. Each clause terminates with a full stop.

(3) The arguments of relations can (among other things) be: concrete objects, or constants (such as tom and ann), or general objects such as X and Y. Objects of the first kind in our program are called atoms. Objects of the second kind are called variables.

(I) Questions to the system consist of one or more goals. A sequence of goals, such as `parent( X, ann)` , `parent( X, pat)` means the conjunction of the goals:

X is a parent of Ann, and

X is a parent of Pat.

The word 'goals' is used because Prolog accepts questions as goals that are to be satisfied.

(4) An answer to a question can be either positive or negative, depending on PROLOG PROGRAMMING FOR ARTIFICIAL INTELLIGENCE whether the corresponding goal can be satisfied or not. In the case of a positive answer we say that the corresponding goal was satisfiable and that the goalsucceeded. Otherwise the gbalwas unsatisfiable anditfailed.

(5) If several answers satisfy the question then Prolog will find as many of them as desired by the user.

1.2 Extending the example program by rules

Our example program can be easily extended in many interesting ways. Let us first add the information on the sex of the people that occur in the parent relation. This can be done by simply adding the following facts to our program:

```
1 female( pam).
2 male( tom).
3 male( bob).
4 female( liz).
5 female( pat).
6 female( ann).
7 male( jim).
```

```
?- female(pat).
```

```
Clauses of female/1 are not together in the source-file
Earlier definition at line 1
Current predicate: male/1
Use :- discontiguous female/1. to suppress this message
```

```
Clauses of male/1 are not together in the source-file
Earlier definition at line 2
Current predicate: female/1
Use :- discontiguous male/1. to suppress this message
```

true *1*

The relations introduced here are male and female. These relations are unary (or one-place) relations. A binary relation like parent defines a relation between pairs of objects; on the other hand, unary relations can be used to declare simple yes/no properties of objects. The first unary clause above can be read: Pam is a female. We could convey the same information declared in the two unary relations with one binary relation, sex, instead.

An alternative piece AN OVERVIEW OF PROLOG of program would then be:

```
1 sex( pam, feminine).
2 sex( tom, masculine).
3 sex( bob, masculine).
```

```
?-
```

```
sex( pam, feminine).
```

```
Failed to set breakpoint at line 1
```

true *1*

As our next extension to the program let us introduce the offspring relation as the inverse of the parent relation. We could define offspring in a similar way as the parent relation; that is, by simply providing a list of simple facts about the offspring relation, each fact mentioning one pair of people such that one is an offspring of the other. For example

```
1 offspring( liz, tom)
```

However, the offspring relation can be defined much more elegantly by making use of the fact that it is the inverse of parent, and that parent has already been defined. This alternative way can be based on the following logical statement:

```
1 For all X and Y,
2     Y is an offspring of X if
3     X is a parent of Y.
```

This formulation is already close to the formalism of Prolog. The corresponding Prolog clause which has the same meaning is:

```
1 offspring( Y, X) :- parent( X, Y).
```

This clause can also be read as:

```
1 For all X and Y,
2     if X is a parent of Y then
3     Y is an offspring of X.
```

Prolog clauses such as

```
1 Your Prolog rules and facts go here ...
```

are called rules. There is an important difference between facts and rules. A fact like

is something that is always, unconditionally, true. On the other hand, rules specify things that may be true if some condition is satisfied. Therefore we say that rules have:

1/ a condition part (the right-hand side of the rule) and

2/ a conclusion part (the left-hand side of the rule).

The conclusion part is also called the headof a clause and the condition part the body of a clause. For example:

```
offspring( y, X) :- parent( X, y) .
```

head \\\\\\\ body

If the condition `parent( X, Y)` is true then a logical consequence of this is `offspring( Y, X)` .

```
1 parent( pam, bob).
2 parent( tom, bob).
3 parent( tom, liz).
4 parent( bob, ann).
5 parent( bob, pat).
6 parent( pat, jim).
7
8 offspring( Y, X) :- parent( X, Y).
```

```
?- offspring( liz, tom).
```

true *1*

specification of the mother relation can be based on the following logical statement:

For all X and Y.

X is the mother of Y if

X is a parent of Y and

X is a female.

```
1 parent( pam, bob).
2 parent( tom, bob).
3 parent( tom, liz).
4 parent( bob, ann).
5 parent( bob, pat).
6 parent( pat, jim).
7 female( pam).
8 female( liz).
9 female( pat).
10 female( ann).
11 male( tom).
12 male( bob).
13 male( jim).
14 offspring( Y, X) :-
15     parent( X, Y).
16 mother( X, Y) :-
17     parent( X, Y),
18     female( X).
19 grandparent(X, Z) :-
20     parent( X, Y),
21     parent( Y, Z).
22 sister( X, Y):-
23     parent( Z,X),
24     parent( Z,Y),
```

```
25      female( X).
```

A comma between two conditions indicates the conjunction of the conditions, meaning that both conditions have to be true.

```
?- mother( X, jim)
```

X = pat

```
?- sister( ann, pat).
```

true *1*

```
?- sister( X, pat).
```

X = ann
X = pat
false

So, Pat is a sister to herself?! This is probably not what we had in mind when defining the sister relation. However, according to our rule about sisters Prolog's answer is perfectly logical. Our rule about sisters does not mention that X and Y must not be the same if X is to be a sister of Y. As this is not required Prolog (rightfully) assumes that X and Y can be the same, and will as a consequence find that any female who has a parent is a sister of herself

To correct our rule about sisters we have to add that X and Y must be different. We will see in later chapters how this can be done in several ways, but **for the moment we will assume that a relation different is already known to Prolog**

```
 1 parent( pam, bob).
 2 parent( tom, bob).
 3 parent( tom, liz).
 4 parent( bob, ann).
 5 parent( bob, pat).
 6 parent( pat, jim).
 7 female( pam).
 8 female( liz).
 9 female( pat).
10 female( ann).
11 male( tom).
12 male( bob).
13 male( jim).
14 offspring( Y, X) :-
15     parent( X, Y).
16 mother( X, Y) :-
17     parent( X, Y),
18     female( X).
19 grandparent(X, Z) :-
20     parent( X, Y),
```

```
    parent( Y, Z).
different(X, Y) :-
    X = Y, !, fail ; true.
sister( X, Y):-
    parent( Z,X),
    parent( Z,Y),
    female( X),
    different( X, Y).
```

```
?- sister( X, pat).
```

X = ann
false

Some important points of this section are:

1/ Prolog programs can be extended by simply adding new clauses.

2/ Prolog clauses are of three types: facts, rules and questions.

3/ Facts declare things that are always, unconditionally true.

4/ Rules declare things that are true depending on a given condition.

5/ By means of questions the user can ask the program what things are true.

6/ Prolog clauses consist of the headand the body.The body is a list of" goals separated by commas. Commas are understood as conjunctions.

7/ Facts are clauses that have the empty body. Questions only have the body. Rules have the head and the (non-empty) body.

8/ In the course of computation, a variable can be substituted by another object. we say that a variable becomes instantiated.

9/ Variables are assumed to be universally quantified and are read as .for all'. Alternative readings are, however, possible for variables that appear only in the body. For example

```
hasachild( X) :- parent( X, y) .
```

can be read in two ways:

(a) For all X and Y,

if X is a parent of Y then

X has a child.

(b) For all X,

X has a child if

there is some Y such that X is a parent of y.

1.3 A recursive rule definition

Let us add one more relation to our family program, the predecessorelation. This relation will be defined in terms of the parent relation. The whole definition can be expressed with two rules. The first rule will define the direct (immediate) predecessors and the second rule the indirect predecessors. We say that some X is an indirect predecessor of some Z if. there is a parentship chain of people between X and Z

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

/* direct predecessor (Base Condition)*/
predecessor(X, Z) :-
    parent( X, Z).
/* indirect predecessor*/
predecessor( X, Z) :-
    parent( X, Y),
    predecessor(Y, Z).
```

```
?- predecessor( pam, X).
```

X = bob
X = ann
X = pat
X = jim
false

```
parent( pam, bob). % Pam is a parent of Bob
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
female( pam). % Pam is female
female( liz).
female( pat).
female( ann).
male( tom). % Tom is male
male( bob).
male( jim).
offspring( Y, X) :-
    parent( X, Y).
mother( X, Y) :-
    parent( X, Y),
    female( X).
grandparent(X, Z) :-
    parent( X, Y),
```

```
    parent( Y, Z).
different(X, Y) :-
    X = Y, !, fail ; true.
sister( X, Y):-
    parent( Z,X),
    parent( Z,Y),
    female( X),
```

1.4 How Prolog answers question

A question to Prolog is always a sequence of one or more goals. To answer a question, Prolog tries to satisfy all the goals. What does it mean to satisfy a goal? To satisfy a goal means to demonstrate that the goal is true, assuming that the relations in the program are true. In other words, to satisfy a goal means to demonstrate that the goal togically follows from the facts and rules in the program

. If the question containi variables, Prolog also has to find what are the particular objects (in place of variables) for wtriitr the goals are satisfied. The particular instantiation of variables to these objects is displayed to the user. If Prolog cannot demonstrate for some instantiation of vaiiables that the goals logically follow from the program, then Prolog's answer to the question will be 'no'.

An appropriate view of the interpretation of a Prolog program in mathematical terms is then as follows: Prolog accepts facts and rules as a set of axioms, and the user's question as a conjectured theorem;then it tries to prove this theorem - that is, to demonstrate that it can be logically derived from the axioms.

We will illustrate this view by a classical example. Let the axioms be:

All men are fallible.

Socrates is a man.

A theorem that logically follows from these two axioms is:

Socrates is fallible.

The first axiom above can be rewritten as:

For all X, if X is a man then X is fallible.

Accordingly, the example can be translated into Prolog as follows:

```
fallible( X) :- man( X). % All men are fallible
man( socrates). % Socrates is a man
```

```
?- fallible(socrates) % Socrates is fallible?
```

true *1*

. Instead of starting with simple facts given in the program, Prolog starts with the goals and, using rules, substitutes the current goals with new goals, until new goals happen to be simple facts

Prolog will try to satisfy this goal. In order to do so it will try to find a clause in the program from which the above goal could immediately'follow