Chapter 1
Introducing Prolog

1.1. THE IDEA OF PROLOG

Until recently, programming a computer meant giving it a list of things to do, step by step, in order to solve a problem. In Prolog, this is no longer the case. A Prolog program can consist of a set of facts together with a set of conditions that the solution must satisfy; the computer can figure out for itself how to deduce the solution from the facts given.

This is called LOGIC PROGRAMMING. Prolog is based on formal logic in the same way that FORTRAN, BASIC, and similar languages are based on arithmetic and simple algebra. Prolog solves problems by applying techniques originally developed to prove theorems in logic.

Prolog is a very versatile language. We want to emphasize throughout this book that Prolog can implement all kinds of algorithms, not just those for which it was specially designed. Using Prolog does not tie you to any specific algorithm, flow of control, or file format. That is, Prolog is no less powerful than Pascal, C, or C++; in many respects it is more powerful. Whether Prolog is the best language for your purposes will depend on the kind of job you want it to do, and we will do our best to equip you to judge for yourself.

Prolog was invented by Alain Colmerauer and his colleagues at the University of Aix-Marseille, in Marseilles, France, in 1972. The name stands for programming in logic. Today Prolog is used mainly for artificial intelligence applications, especially automated reasoning systems. Prolog was the language chosen for the Fifth Generation Project, the billion-dollar program initiated by the Japanese government.
in 1982 to create a new generation of knowledge-based computers. Commercially, Prolog is often used in expert systems, automated helpdesks, intelligent databases, and natural language processing programs.

Prolog has much in common with Lisp, the language traditionally used for artificial intelligence research. Both languages make it easy to perform complex computations on complex data, and both have the power to express algorithms elegantly. Both Lisp and Prolog allocate memory dynamically, so that the programmer does not have to declare the size of data structures before creating them. Both languages allow the program to examine and modify itself; thus, a program can "learn" from information obtained at run time.

The main difference is that Prolog has an automated reasoning procedure — an INFEERENCE ENGINE — built into it, while Lisp does not. As a result, programs that perform logical reasoning are much easier to write in Prolog than in Lisp. If the built-in inference engine is not suitable for a particular problem, the Prolog programmer can usually use part of the built-in mechanism while rewriting the rest. In Lisp, on the other hand, if an inference engine is needed, the programmer must supply it.

Is Prolog "object-oriented"? Not exactly. Prolog is a different, newer, and more versatile solution to the problem that object orientation was designed to solve. It is quite possible to organize a Prolog program in an object-oriented way, but in Prolog, that's not the only option available to you. Prolog lets you talk about properties and relations directly, rather than approaching them indirectly through an inheritance mechanism.

1.2. HOW PROLOG WORKS

Prolog derives its power from a PROCEDURAL INTERPRETATION OF LOGIC — that is, it represents knowledge in terms of procedure definitions, and reasoning becomes a simple process of calling the right procedures. To see how this works, consider the following two pieces of information:

[1] For any \( \mathbf{x} \), if \( \mathbf{x} \) is in Georgia, then \( \mathbf{x} \) is in the United States.
[2] Atlanta is in Georgia.

We will call a collection of information such as this a KNOWLEDGE BASE. We will call item [1] a RULE because it enables us to infer one piece of information from another, and we will call item [2] a FACT because it does not depend on any other information. Note that a rule contains an "if" and a fact does not. Facts and rules are the two types of CLAUSES.

A fact need not be a true statement about the real world; if you said Minneapolis was in Florida, Prolog would believe you. Facts are sometimes called GROUND CLAUSES because they are the basis from which other information is inferred.

Suppose we want to know whether Atlanta is in the United States. Clearly, [1] and [2] can be chained together to answer this question, but how should this chaining be implemented on a computer? The key is to express [1] and [2] as definitions of procedures:
[1'] To prove that \( x \) is in the United States, prove that \( x \) is in Georgia.
[2'] To prove that Atlanta is in Georgia, do nothing.

We ask our question by issuing the instruction:

Prove that Atlanta is in the United States.

This calls procedure [1'], which in turn calls procedure [2'], which returns the answer “yes.”

Prolog has its own notation for representing knowledge. Our sample knowledge base can be represented in Prolog as follows:

\[
\text{in\_united\_states}(X) :- \text{in\_georgia}(X).
\]

\[
\text{in\_georgia}(\text{atlanta}).
\]

Here \( \text{in\_georgia} \) and \( \text{in\_united\_states} \) are PREDICATES — that is, they say things about individuals. A predicate can take any fixed number of ARGUMENTS (parameters); for example,

\[
\text{female}(\text{sharon}).
\]

might mean “Sharon is female,” and

\[
\text{mother}(\text{melody}, \text{sharon}).
\]

might mean “Melody is the mother of Sharon.” A predicate that takes \( N \) arguments (for any number \( N \)) is called an \( N \)-PLACE PREDICATE; thus we say that \( \text{in\_georgia}, \text{in\_united\_states}, \) and \( \text{female} \) are ONE-PLACE PREDICATES, while \( \text{mother} \) is a TWO-PLACE PREDICATE. A one-place predicate describes a PROPERTY of one individual; a two-place predicate describes a RELATION between two individuals.

The number of arguments that a predicate takes is called its ARITY (from terms like unary, binary, ternary, and the like). Two distinct predicates can have the same name if they have different arities; thus you might have both \( \text{mother}(\text{melody}), \) meaning Melody is a mother, and \( \text{mother}(\text{melody}, \text{sharon}), \) meaning Melody is the mother of Sharon. We will avoid this practice because it can lead to confusion.

In some contexts a predicate is identified by giving its name, a slash, and its arity; thus we can refer to the two predicates just mentioned as \( \text{mother}/1 \) and \( \text{mother}/2 \).

Exercise 1.2.1

Give an example, in Prolog, of a fact, a rule, a clause, a one-place predicate, and a predicate of arity 2.

Exercise 1.2.2

In the previous example, we represented “in Georgia” as a property of Atlanta. Write a Prolog fact that represents “in” as a relation between Atlanta and Georgia.

Exercise 1.2.3

How would you represent, in Prolog, the fact “Atlanta is at latitude 34 north and longitude 84 west”? (Hint: More than one approach is possible. Second hint: It is OK to use numbers as constants in Prolog.)
1.3. VARIETIES OF PROLOG

An important goal of this book is to teach you how to write portable Prolog code. Accordingly, we will stick to features of the language that are the same in practically all implementations. The programs in this book were developed in Arity Prolog and ALS Prolog on IBM PCs and Quintus Prolog on Sun workstations. Most of them have also been tested in SWI Prolog, LPA Prolog, Cogent (Amzi) Prolog, and Expert Systems Limited’s Public Domain Prolog-2.¹

For many years, the de facto standard for Prolog was the language described by Clocksin and Mellish in their popular textbook, Programming in Prolog (1981, second edition 1984). This is essentially the language implemented on the DEC-10 by D. H. D. Warren and his colleagues in the late 1970s, and is often called “Edinburgh Prolog” or “DEC-10 Prolog.” Most commercial implementations of Prolog aim to be compatible with it.

In 1995 the International Organization for Standardization (ISO) published an international standard for the Prolog language (Scowen 1995). ISO Prolog is very similar to Edinburgh Prolog but extends it in some ways. Our aim in this book is to be as compatible with the ISO standard as possible, but without using features of ISO Prolog that are not yet widely implemented. See Appendix A for more information about ISO Prolog.

Finally, we must warn you that this book is not about Turbo Prolog (PDC Prolog), nor about Colmerauer’s Prolog II and Prolog III. Turbo Prolog is Prolog with data type declarations added. As a result, programs run faster but are largely unable to examine and modify themselves. Colmerauer’s Prolog II and III are CONSTRAINT LOGIC PROGRAMMING languages, which means they let you put limits on the value of a variable before actually giving it a value; this makes many new techniques available. The concepts in this book are certainly relevant to Turbo (PDC) Prolog and Prolog II and III, but the details of the languages are different.

Exercise 1.3.1

If you have not done so already, familiarize yourself with the manuals for the version of Prolog that you will be using.

Exercise 1.3.2

In the Prolog that you are using, does the query ‘?- help.’ do anything useful? Try it and see.

1.4. A PRACTICAL KNOWLEDGE BASE

Figure 1.1 shows a Prolog knowledge base that describes the locations of certain North American cities. It defines a single relation, called located_in, which relates a city to a larger geographical unit. The knowledge base consists of facts such as

¹Users of ESL Public Domain Prolog-2 must select Edinburgh-compatible syntax by adding the line ‘:- state(token_class,_,dec10).’ at the beginning of every program. Note that ESL Prolog-2 has nothing to do with Colmerauer’s Prolog II.
Sec. 1.4. A Practical Knowledge Base

\% File GEO.PL
\% Sample geographical knowledge base

/* Clause 1 */ located_in(atlanta, georgia).
/* Clause 2 */ located_in(houston, texas).
/* Clause 3 */ located_in(austin, texas).
/* Clause 4 */ located_in(toronto, ontario).
/* Clause 5 */ located_in(X, usa) :- located_in(X, georgia).
/* Clause 6 */ located_in(X, usa) :- located_in(X, texas).
/* Clause 7 */ located_in(X, canada) :- located_in(X, ontario).
/* Clause 8 */ located_in(X, north_america) :- located_in(X, usa).
/* Clause 9 */ located_in(X, north_america) :- located_in(X, canada).

Figure 1.1 A simple Prolog knowledge base.

"Atlanta is located in Georgia," "Houston is located in Texas," and the like, plus rules such as "X is located in the United States if X is located in Georgia."

Notice that names of individuals, as well as the predicate located_in, always begin with lowercase letters. Names that begin with capital letters are variables and can be given any value needed to carry out the computation. This knowledge base contains only one variable, called X. Any name can contain the underscore character (_). Notice also that there are two ways to delimit comments. Anything bracketed by /* and */ is a comment; so is anything between \% and the end of the line, like this:

/* This is a comment */
\% So is this

Comments are ignored by the computer; we use them to add explanatory information and (in this case) to number the clauses so we can talk about them conveniently.

It is not clear whether to call this knowledge base a program; it contains nothing that will actually cause computation to start. Instead, the user loads the knowledge base into the computer and then starts computation by typing a QUERY, which is a question that you want the computer to answer. A query is also called a GOAL. It looks like a Prolog clause except that it is preceded by '?' — although in most cases the Prolog implementation supplies the '?' and you need only type the goal itself.

Unfortunately, we cannot tell you how to use Prolog on your computer because there is considerable variation from one implementation to another. In general, though, the procedure is as follows. First use a text editor to create a file of clauses such as GEO.PL in Figure 1. Then get into the Prolog interpreter and type the special query:

?- consult('geo.pl').

(Remember the period at the end — if you don’t type it, Prolog will assume your query continues onto the next line.) Prolog replies
to indicate that it succeeded in loading the knowledge base.

Two important notes: First, if you want to load the same program again after escaping to an editor, use `reconsult` instead of `consult`. That way you won’t get two copies of it in memory at the same time. Second, if you’re using a PC, note that backslashes (\) in the file name may have to be written twice (e.g., `consult('c:\myprog.pl')` to load `C:\MYPROG.PL`). This is required in the ISO standard but not in most of the MS-DOS Prologs that we have worked with.

As soon as `consult` has done its work, you can type your queries. Eventually, you’ll be through using Prolog, and you can exit from the Prolog system by typing the special query

```
?- halt.
```

Most queries, however, retrieve information from the knowledge base. You can type

```
?- located_in(atlanta,georgia).
```

to ask whether Atlanta is in Georgia. Of course it is; this query matches Clause 1 exactly, so Prolog again replies “yes.” Similarly, the query

```
?- located_in(atlanta,usa).
```

can be answered (or, in Prolog jargon, SOLVED or SATISFIED) by calling Clause 5 and then Clause 1, so it, too, gets a “yes.” On the other hand, the query

```
?- located_in(atlanta,texas).
```

gets a “no” because the knowledge base contains no information from which the existence of an Atlanta in Texas can be deduced.

We say that a query SUCCEEDS if it gets a “yes” answer, or FAILS if it gets a “no” answer.

Besides answering yes or no to specific queries, Prolog can fill in the blanks in a query that contains variables. For example, the query

```
?- located_in(X,texas).
```

means “Give me a value of X such that \( \text{in}(X,\text{texas}) \) succeeds.”

Here we run into another unique feature of Prolog — a single query can have multiple solutions. Both Houston and Austin are in Texas. What happens in this case is that Prolog finds one solution and then asks you whether to look for another. This continues until all alternatives are found or you stop asking for them. In some Prologs, the process looks like this:

```
?- located_in(X,texas).
X = houston
More (y/n)? y
X = austin
More (y/n)? y
no
```
The "no" at the end means there are no more solutions.

In Arity Prolog, the notation is more concise. After each solution, the computer displays an arrow (\(\rightarrow\)). You respond by typing a semicolon (meaning look for more alternatives) or by hitting Return (meaning quit), like this:

```prolog
?- located_in(X, texas).
X = houston \(\rightarrow\) ;
X = austin \(\rightarrow\) ;
no
```

In Quintus Prolog and many others, there isn't even an arrow; the computer just pauses and waits for you to type a semicolon and then hit Return, or else hit Return by itself:

```prolog
?- located_in(X, texas).
X = houston ;
X = austin ;
no
```

Also, you'll find it hard to predict whether the computer pauses after the last solution; it depends partly on the way the user interface is written, and partly or exactly what you have queried. From here on, we will present interactions like these by printing only the solutions themselves and leaving out whatever the user had to type to get the alternatives.

Sometimes your Prolog system may not let you ask for alternatives (by typing semicolons, or whatever) even though alternative solutions do exist. There are two possible reasons. First, if your query has performed any output of its own, the Prolog system will assume that you've already printed out whatever you wanted to see, and thus that you're not going to want to search for alternatives interactively. So, for example, the query

```prolog
?- located_in(X, texas), write(X).
```

displays only one answer even though, logically, there are alternatives. Second, if your query contains no variables, Prolog will only print "yes" once no matter how many ways of satisfying the query there actually are.

Regardless of how your Prolog system acts, here's a sure-fire way to get a list of all the cities in Texas that the knowledge base knows about:

```prolog
?- located_in(X, texas), write(X), nl, fail.
```

The special predicate `write` causes each value of \(X\) to be written out; `nl` starts a new line after each value is written; and `fail` forces the computer to backtrack to find all solutions. We will explain how this works in Chapter 2. For now, take it on faith.

We say that the predicate `located_in` is NONDETERMINISTIC because the same question can yield more than one answer. The term "nondeterministic" does not mean that computers are unpredictable or that they have free will, but only that they can produce more than one solution to a single problem.

Another important characteristic of Prolog is that any of the arguments of a predicate can be queried. Prolog can either compute the state from the city or compute the city from the state. Thus, the query
Introducing Prolog  Chap. 1

?- located_in(austin,X).

retrieves the names of regions that contain Austin, and

?- located_in(X,texas).

retrieves the names of cities that are in Texas. We will call this feature REVERSIBILITY
or INTERCHANGEABILITY OF UNKNOWNS. In many — but not all — situations, Prolog
can fill in any argument of a predicate by searching the knowledge base. In Chapter
3 we will encounter some cases where this is not so.

   We can even query all the arguments of a predicate at once. The query

?- located_in(X,Y).

means “What is in what?” and each answer contains values for both X and Y (Atlanta
is in Georgia, Houston is in Texas, Austin is in Texas, Toronto is in Ontario, Atlanta is
in the U.S.A., Houston is in the U.S.A., Austin is in the U.S.A., Toronto is in Canada,
and so forth). On the other hand,

?- located_in(X,X).

means “What is in itself?” and fails — both occurrences of X have to have the same
value, and there is no value of X that can successfully occur in both positions at the
same time. If we were to add New York to the knowledge base, this query could
succeed because the city has the same name as the state containing it.

Exercise 1.4.1

   Load GEO.PL into your Prolog system and try it out. How does your Prolog system
respond to each of the following queries? Give all responses if there is more than one.

   ?- located_in(austin,texas).
   ?- located_in(austin,georgia).
   ?- located_in(what,texas).
   ?- located_in(atlanta,what).

Exercise 1.4.2

   Add your hometown and state (or region) and country to GEO.PL and demonstrate
that the modified version works correctly.

Exercise 1.4.3

   How does GEO.PL respond to the query ’?- located_in(texas,usa).’? Why?

Exercise 1.4.4  (for PC users only)

   Does your Prolog require backslashes in file names to be written double? That is, to load
C:\MYDIR\MYPROG.PL, do you have to type consul('c:\mydir\myprog.pl')?
Try it and see.
1.5. UNIFICATION AND VARIABLE INSTANTIATION

The first step in solving any query is to match — or UNIFY — the query with a fact or with the left-hand side (the HEAD) of a rule. Unification can assign a value to a variable in order to achieve a match; we refer to this as INSTANTIATING the variable. For example, the query

?- located_in(austin,north_america).

unifies with the head of Clause 8 by instantiating X as austin. The right-hand side of Clause 8 then becomes the new goal. Thus:

Goal: ?- located_in(austin,north_america).
Clause 8: located_in(X,north_america) :- located_in(X,usa).
Instantiation: X = austin
New goal: ?- located_in(austin,usa).

We can then unify the new query with Clause 6:

Goal: ?- located_in(austin,usa).
Clause 6: located_in(X,usa) :- located_in(X,texas).
Instantiation: X = austin
New query: ?- located_in(austin,texas).

This query matches Clause 3. Since Clause 3 does not contain an "if," no new query is generated and the process terminates successfully. If, at some point, we had had a query that would not unify with any clause, the process would terminate with failure.

Notice that we have to instantiate X two times, once when we call Clause 8 and once again when we call Clause 6. Although called by the same name, the X in Clause 8 is not the same as the X in Clause 6. There is a general principle at work here:

Like-named variables are not the same variable unless they occur in the same clause or the same query.

In fact, if we were to use Clause 8 twice, the value given to X the first time would not affect the value of X the second time. Each instantiation applies only to one clause and only to one invocation of that clause. However, it does apply to all of the occurrences of that variable in that clause; when we instantiate X, all the X's in the clause take on the same value at once.

If you've never used a language other than Prolog, you're probably thinking that this is obvious, and wondering why we made such a point of it; Prolog couldn't possibly work any other way. If you're accustomed to a conventional language, we want to make sure that you don't think of instantiation as storing a value in a variable. Instantiation is more like passing a parameter. Suppose you have a Pascal procedure such as this:

procedure p(x:integer); { This is Pascal, not Prolog! }
begin
  writeln('The answer is ',x)
end;

If you call this with the statement

\[ p(3) \]

you are passing 3 to procedure \( p \) as a parameter. The variable \( x \) in the procedure is instantiated as 3 but only for the duration of this invocation of \( p \). It would not be correct to think of the value 3 as being "stored" in a location called \( x \); as soon as the procedure terminates, it is gone.

One uninstantiated variable can even be unified with another. When this happens, the two variables are said to share, which means that they become alternative names for a single variable, and if one of them is subsequently given a value, the other one will have the same value at the same time. This situation is relatively uncommon, but there are programs in which it plays a crucial role. We will discuss unification and instantiation at greater length in Chapter 3.

**Exercise 1.5.1**

What would happen to GEO.PL if Clauses 5 and 6 were changed to the following?

\[
\text{located_in}(Y, \text{usa}) :- \text{located_in}(Y, \text{georgia}).
\]

\[
\text{located_in}(Z, \text{usa}) :- \text{located_in}(Z, \text{texas}).
\]

**Exercise 1.5.2**

Disregarding the wisdom of this section, a beginning Prolog student loads GEO.PL and has the following dialogue with the computer:

\[ ?- \text{located}_\text{in}(\text{austin},X). \]

\[ X = \text{texas} \]

\[ ?- \text{write}(X). \]

\[ X \text{ is uninstantiated} \]

Why didn't the computer print 'texas' the second time? Try this on your computer. What does your computer print when you try to write out an uninstantiated variable?

### 1.6. BACKTRACKING

If several rules can unify with a query, how does Prolog know which one to use? After all, if we unify

\[ ?- \text{located}_\text{in}(\text{austin}, \text{usa}). \]

with Clause 5, we generate

\[ ?- \text{located}_\text{in}(\text{austin}, \text{georgia}). \]

which fails. However, if we use Clause 6, we generate

\[ ?- \text{located}_\text{in}(\text{austin}, \text{texas}). \]

which succeeds. From the viewpoint of our query, Clause 5 is a blind alley that does not lead to a solution.

The answer is that Prolog does not know in advance which clause will succeed, but it does know how to back out of blind alleys. This process is called backtracking.