2 Prolog: To Define a World

The purpose of this chapter is to help the learner
- perceive a Prolog program as a self-contained world description.
- state Prolog facts.
- recognize the roles of constants and variables.
- ask Prolog questions for
    confirmation of facts.
    reporting of variable values.

2.1 PROGRAMMING LANGUAGES

A computer is a tool that can solve many different kinds of problems. For a computer to solve a particular problem, it must be given information about that problem. Programming languages are a way for people to communicate with a computer about the particular problem that is to be solved. The programming language is used to provide information about the problem and about the requirements for finding the solution to the problem. There are many different programming languages, but their role is always to communicate specifics of a problem to a computer.

Problems generally fall into one of a number of categories. For example, a problem like space navigation involves a lot of numeric calculations. Other problems, like printing transcripts for students, require taking a quantity of known information and arranging it in a form that people can easily understand. Because of the way programming languages have been designed, different ones match with certain types of problems. That is, the languages have the vocabulary and means of expression to communicate requirements that are typical of certain kinds of problems.

Prolog

Prolog is a programming language designed for a particular kind of problem. It is a convenient way to describe a self-contained world of knowledge. Tasks that can be carried out by following precise specifications can be done in a Prolog world that knows those specifications. For example, translation from one computer language to another can be done by a Prolog world that knows the translation rules. Another role for Prolog worlds is as an information resource. The world will provide selected information at the request of a person. This is the Prolog role we will focus on during the first part of this book.

The Prolog description of the world will consist of facts and rules. Once we have the world description, we can ask questions about that world. The questions we ask and the answers Prolog provides make using Prolog like holding a conversation with the computer, a conversation about the world described in the program.

The questions we ask will not be limited to requests for simple facts; Prolog will use the rules to combine facts about the world to find answers to complex questions. The questions themselves are descriptive. They describe requirements of the desired answer rather than specifying how to find out the answer. The facts and rules describe the world in the Prolog program; the questions describe the kinds of answers Prolog should provide from the world. Because of its character, Prolog is called a descriptive language. Neither the world nor the questions will give Prolog instructions about what actions should be carried out find the answer. That process is embedded in the Prolog system.

Of course, Prolog's answers will be constrained by the limits of the data it has been given in the description of the world. A Prolog program knows only about the world of knowledge that has been specified. Similarly, the description of the Prolog world need not match the world of our normal experience. If we choose to define a world in which "up" is the same as "down", Prolog will not object. Prolog allows a programmer to define a knowledge base and then explore that knowledge with questions.

A Base of Facts

A fundamental part of a Prolog description of a world is the facts it contains. Each fact is like a sentence, declaring a piece of information about one or more objects in the world. A collection of facts for Prolog is called a *base* or *database*. For example,

```
capital(texas,austin).
```

This Prolog fact shows that there is a relation between two objects, texas and austin.

In providing Prolog with a set of facts, we will follow rules as to how we write them. These rules, which are called *syntax* rules, are quite simple but very important. The name that defines the relationship appears first, then the names of the objects appear within parentheses, separated by a comma. Like a sentence, the fact ends with a period (.).

Syntax rules: facts

- relationship name first
- objects in parentheses, comma between
- period (full stop) at end

Example: `feeds(charlie,dog).`

The relationship name and the object names start with lower-case letters. This is to indicate to Prolog that these are names of specific objects and that the names will not change; they are called constants. Notice that there are no spaces within the fact. If we want to use a name that includes a space, we use an underscore in the space,

```
capital(new_york,albany).
```

These two examples use the same relationship name, capital. We can interpret these facts as meaning "The capital of Texas is Austin" and "The capital of New York is Albany". The order of the object names and the interpretation of the facts are freely chosen by the programmer who designs the facts. It is necessary, however, that the programmer, having made the choices, be consistent when using them within a program.

We could imagine, then, a set of 50 facts like these examples that specifies the capitals of all 50 states in the United States. In each case, the state name would come first and the capital-city name second. In the same base of facts, we can include other facts about the same objects

```
weather(texas,summer,hot).
```

or other completely independent objects

```
color(sox,green).
```

Sometimes only one object name appears within the parentheses of the fact. Then the name in front of the parentheses usually represents an attribute or characteristic of the object. For example

```
female(mary).
```

Prolog requires that an attribute or relationship and its objects, known also as the *predicate* and its *arguments*, maintain the same pattern throughout a program. Most commonly, predicates we use will have two or three arguments, but occasionally we will need more. The number of arguments for a predicate is known as its *arity*. For example, the capital predicate we have been using has an arity of two. In creating a base of facts, the programmer has responsibility for choosing the pattern and putting the names of objects in the appropriate position in a predicate. The pattern will be based on the meaning of the facts to the programmer. The meaning of the program is called the *semantics* of the program. Semantics stands in contrast to syntax, which as we said earlier, is the way the program is written down. The computer's processing is based on syntax, not semantics.

EXERCISES 2.1

1. Write Prolog facts based on these sentences.
   The book is on the table.
   The book is on the third shelf.
   John owns a dog.
   Jim owns a black dog.
2. What might these facts mean? Give at least two interpretations.
   ```
   mother(sarah,jane).
   father(joe,sam).
   ```
3. An old joke: Which is better, a ham sandwich or eternal happiness? Well, nothing is better than eternal happiness, and a ham sandwich is better than nothing. Therefore, a ham sandwich is better than eternal happiness. What does this have to do with Prolog facts?

2.2 QUERIES

When we have a base of facts, we can use Prolog to ask questions about the world we ~~have~~ described in the database. A query follows the same syntax rules that facts follow, but has a **?-** first. For example,

```
?- capital(oregon,salem).
```

This kind of question is a direct request for confirmation that a fact is in the base that has been given. If we have a base that consists of a set of facts, one giving the capital city for each of the 50 states in the United States, (see Figure 2.1) we could ask our question

```
?- capital(oregon,salem).
```

and Prolog would respond

```
yes
```

Figure 2.1

Facts — States and Capitals

```
capital(alabama,montgomery).         capital(montana,helena).
capital(alaska,juneau).              capital(nebraska,lincoln).
capital(arkansas,little_rock).       capital(nevada,carson_city).
capital(arizona,phoenix).            capital(new_hampshire,concord).
capital(california,sacramento).      capital(new_jersey,trenton).
capital(colorado,denver).            capital(new_mexico,senta_fe).
capital(connecticut,hartford).       capital(new_york,albany).
capital(delaware,dover).             capital(north_carolina,raleigh).
capital(florida,tallahassee).        capital(north_dakota,bismark).
capital(georgia,atlanta).            capital(ohio,columbus).
capital(hawaii,honolulu).            capital(oklahoma,oklahoma_city).
capital(idaho,boise).                capital(oregon,salem).
capital(illinois,springfield).       capital(pennsylvania,harrisburg).
capital(indiana,indianapolis).       capital(rhode_island,providence)
capital(iowa,des_moines).            capital(south_carolina,columbia).
capital(kansas,topeka).              capital(south_dakota,pierre).
capital(kentucky,frankfort).         capital(tennessee,nashville).
capital(louisiana,baton_rouge).      capital(texas,austin).
capital(maine,augusta).              capital(utah,salt_lake_city).
capital(maryland,annapolis).         capital(vermont,montpelier).
capital(massachusetts,boston).       capital(virginia,richmond).
capital(michigan,lansing).           capital(washington,olympia).
capital(minnesota,st_paul).          capital(west_virginia,charleston).
capital(mississippi,jackson).        capital(wisconsin,madison).
capital(missouri,jefferson_city).    capital(wyoming,cheyenne).
```

If we asked

```
?- capital(oregon,portland).
```

Prolog would respond

```
no
```

If we asked

```
?- capital(ontario,toronto).
```

Prolog would answer

```
no
```

not because Toronto is not the capital of Ontario, but because that fact is not in the information we provided in our description to Prolog. We describe a world in our Prolog base. Prolog can only respond from the data in the base, answering "no" if a fact is not in that world.

Prolog checks to see if a fact exists in a database by using pattern matching. When asked a question, Prolog compares the first fact in the database with the expression given in the question. If it does not match, Prolog moves on to the next fact to try the match, and continues the process until one of two things happens. Either it succeeds in making the match, whereupon it reports "yes" or it checks the base clear through to the end and fails to find the match, so it reports "no". Prolog responds "no" to both the questions

```
?- capital(oregon,portland).
?- capital(ontario,toronto).
```

because it does not find matching facts in the base. Similarly, small typographic errors create completely different words. Thus,

```
?- capital(texas,asutin).
```

would be answered

```
no
```

Because Prolog is matching the pattern of the fact with the pattern in the question, details such as the spelling and the order of the object names must be attended to carefully.

Variables

We will sometimes want Prolog to check a piece of information against a database and report a "yes" or "no" to us. More frequently, however, we will want Prolog to fill in the name of an object that belongs in a relationship. To do this, we will again ask a question that contains the pattern we are looking for, but in the place of the name we want, we put a variable name. It is called a *variable* because it can stand in place of various objects. This variability is in contrast to the names which always refer to the same objects, *constant* names. In Prolog, we specify that a name is a variable by beginning the name with an upper-case letter. For example, in

```
?- capital(nevada,City).
```

City is a variable because it starts with an upper-case letter, C. If we asked this question of our base in Figure 2.1, Prolog would respond with

```
City = carson_city
```

We could as well have used any set of letters to represent the variable as long as the first is upper-case. Thus the question

```
?- capital(washington,XYZ).
```

will get the response

```
XYZ = olympia
```

Since XYZ is a variable, we can use it again

```
?- capital(california,XYZ).
```

to be told

```
XYZ = sacramento
```

Different people choose different kinds of words for variable names. Some people prefer a letter like X which is short and fast to use. Others prefer words like City, which may convey more meaning. Meaningful names help programmers make fewer mistakes.

We can use variables for any of the objects within the parentheses.

```
?- capital(State,pierre).
```

will get the response

```
State = south_dakota
```

and for

```
?- capital(State,charleston).
State = south_carolina
```

but for

```
?- capital(ontario,City).
```

we will be told "no" since the expression in the question has no match in our database, no matter what object the variable stands for. We are not limited to one variable in each question.

```
?- capital(State,City).
State = alabama   City = montgomery
```

In this case, any one of the facts in the database would match. Prolog responded with the first pair of objects it found, those in the fact at the beginning of the database.

While variables can stand in place of any of the object names in a fact, they cannot be used in place of the relationship name. Arguments may be variables, but predicates may not.

```
?- Connection(oregon,salem).
**^syntax error **
```

EXERCISES 2.2

1. Write Prolog facts to match these statements, then give examples of queries with variables that could be used with the facts.
   The book is on the table.
   The book is on the third shelf.
   John owns a dog.
   Jim owns a black dog.

2. Given these facts
   ```
   owns(john,dog,midget).
   owns(john,cow,jersey).
   ```
   what will be the response to
   ```
   ?- owns(Thief,cow,jersey).   Thief = John
   ?- owns(john,A,midget).      A = dog
   ?- owns(john,cow,Breed).     [illegible]
   ```
3. Specify advantages and disadvantages for each of the variable names in Problem 2.

2.3 FURTHER RESPONSES ("OR")

When Prolog returns a constant name in response to a question with a variable, it pauses, waiting for the questioner to signal (by keying in a "return") that the answer is accepted. Instead of just a "return", the questioner may respond with a semicolon (;) followed by the "return". In Prolog, the ~~semicolon means 'OR'. Prolog interprets this~~ as a direction to look further through the database to find ~~another pattern~~ that matches the one in the question.

Consider a base that consists of facts about a product and the raw material from which it is made. Figure 2.2 shows such a base.

Figure 2.2

Raw Materials and Products

```
made_into(plums,prunes).
made_into(grapes,raisins).
made_into(apples,sauce).
made_into(grapes,wine).
made_into(apples,wine).
made_into(plums,wine).
```

If asked

```
?- made_into(plums,What).
```

Prolog responds

```
What = prunes
```

and pauses. If the questioner responds with a ; Prolog continues

```
What = wine
```

Another ; from the questioner yields

```
no
```

from Prolog. There are only two matches in our database, so asking for a third match causes Prolog to report failure. In a similar manner, the question

```
?- made_into(What,wine).
```

gets the responses

```
What = grapes ;
What = apples ;
What = plums ;
no
```

We have already mentioned that questions can contain more than one variable. Like our example at the end of Section 2.2

```
?- capital(State,City).
```

this question with two variables will match with any fact in the base. Here, however, we will respond with semicolons to get all possible matches.

```
?- made_into(Source,Product).
Source = plums    Product = prunes ;
Source = grapes   Product = raisins ;
Source = apples   Product = sauce ;
Source = grapes   Product = wine ;
Source = apples   Product = wine ;
Source = plums    Product = wine ;
no
```

When Prolog responds to a query, a specific value is given for each of the variables in the query. The order in which values appear within the group (in the case above, a pair) depends on the particular words that programmer has chosen rather than the order in which they appear in the question. In this case, if the query had been

```
?- made_into(Fruit,Product).
```

the pairs would have been shown in this form:

```
Product = prunes   Fruit = plums
yes
```

This difference happens because of Prolog's way of keeping track of the names (called their internal representation). The order in which the pairs of constants appear, however, will be the same.

EXERCISES 2.3

Suppose you have the database

```
do(monday,call_agent).
do(friday,get_tickets).
do(tuesday,get_money).
do(monday,write_home).
do(friday,pack).
do(thursday,study).
do(tuesday,study).
```

1. What will be Prolog's responses to the following questions? (Assume repeated semicolons)

```
?- do(monday,What).
?- do(tuesday,What).
?- do(When,study).
```

2. Assuming repeated semicolons, how many responses will be given to

```
?- do(When,What).
```

2.4 CONJUNCTIONS ("AND")

More complex queries can be made in Prolog by using the conjunction "," which means "AND". For example, the question

```
?- made_into(What,raisins), made_into(What,wine).
```

asks "what thing is there that raisins are made of and wine is also made of?" A variable name within the question (between the ? and the .) can only represent one object name at a time. That is, when a variable appears more than once in a question, the only matches that can be made will have the same constant name in every place where that variable name appears. Thus Prolog will answer

```
What = grapes
```

and if the questioner responds with a semicolon and return Prolog will respond

```
no
```

as there are no more successful matches to the complete question. Even though part (the second part) has some matches in the database, no others match both parts. If, however, the question asked is

```
?- made_into(What,raisins), made_into(Else,wine).
```

Prolog responds

```
What = grapes   Else = grapes ;
What = grapes   Else = apples ;
What = grapes   Else = plums ;
no
```

as the questioner provides the semicolons. Prolog has made multiple matches and reported the constants for each variable each time. This is possible because What and Else are two separate variables in this question. They may be matched to the same constant, but do not have to be.

More interesting is a database with several different predicates. If we add some facts about colors to our world (Figure 2.3) we can ask "What red things can be made into wine"?

```
?- made_into(Thing,wine), color(red,Thing).
```

and be told

```
Thing = grapes ;
Thing = apples ;
no
```

or

```
?- made_into(Thing,Product), color(red,Product).
Thing = grapes   Product = wine ;
Thing = apples   Product = wine ;
no
```

Figure 2.3

Raw Materials, Products and Colors

```
made_into(plums,prunes).
made_into(grapes,raisins).
made_into(apples,sauce).
made_into(grapes,wine).
made_into(apples,wine).
made_into(plums,wine).
color(red,grapes).
color(white,grapes).
color(purple,plums).
color(red,apples).
color(red,wine).
color(white,wine).
```

Advanced Variable Uses

There are two uses of variables that do not follow the straightforward idea of the variable that we have discussed so far. These are uses that, while not important to the Prolog we have seen up to this point, will be very useful for more sophisticated programs.

Prolog has a special variable name, the underscore "_", which can be read "don't care". It is used mainly as a place holder and can match any constant, any time. Given the question

```
?- made_into(_,prunes).
```

Prolog will respond

```
yes
```

that prunes are made of something, according to the knowledge contained in the database. The variable _ is not associated with a specific name, so it is anonymous. A constant is the name of a specific object. A variable can represent a number of constants, but only represents one at any one time. In a question, the variable represents a specifc object and the constant name of that object is reported to the user. The anonymous variable indicates that there is some constant in the fact in the specified position, but that the questioner is not concerned about what object it is. If a question has two anonymous variables, the constants in those places need not be the same as they would be with a non-anonymous variable.

EXERCISES 2.4

Suppose you have the database

```
do(monday,call_agent).
do(friday,get_tickets).
do(tuesday,get_money).
do(monday,write_home).
do(friday,pack).
do(thursday,study).
do(tuesday,study).
```

1. I want to know what else I have to do on the same day I pack.
   ```
   ?- do(Day,pack), do(Day,Task).
   ```
   What will Prolog say?
2. How can I just ask if I set aside study time, never mind when?
3. How can I find out if I have other things to do on my study days?

SUMMARY

Prolog is a descriptive language, designed to define a knowledge base (usually called a database) and ask questions of that base. Prolog facts are part of the base. Prolog facts are made up of attribute/relationship names followed by object names (arguments). The programmer has responsibility for selecting the form of the facts and understanding the associated meaning. The names of objects in facts are typically constant names, although they can be variables. The attribute/relationship (predicate) names must be constants.

Two types of questions can be asked of a Prolog database. Prolog will answer "yes" or "no" to questions for confirmation of a fact. If Prolog is asked questions with variables, it will return constant names that will create a match between patterns in the question and patterns in the base. A Prolog query can be used both to check for a true instance in the base or to find a constant that specifies a true instance.

Prompting Prolog with a semicolon will cause it to continue a search for more possible matches. Questions can contain a conjunction, with a comma used to join the parts. A variable name stands for one constant at any one time, even if it appears in more than one place in a conjunction.

SYNTAX SUMMARY

Facts

```
  predicate(<object1>,<object2>,...).
```

Questions

```
  ?- predicate(<object1>,<object2>,...).
  ?- predicate1(<object1.1>,<object1.2>,...),
               predicate2(<object2.2>,...),...
```

constants begin with lower-case letters

variables begin with upper-case letters
  _ anonymous variable

, and
; additional answers

EXPERIMENTING WITH CHAPTER 2

Purpose:
  To see: interactive, conversational nature
          entering facts
          asking questions
          consulting files and user

Computers, being general purpose tools, are able to do many things. For a computer to do Prolog, it must have a large set of intructions to tell it how. These instructions (called software) are stored somewhere, either on a disk that you put in the computer or in a permanent storage area in the computer system.

When you are ready to use Prolog on a computer, you (the user) must indicate that intention to the computer, so the system will make the Prolog software available. How you do this will depend on your computer system. For at least one system, you type in

```
  >Prolog
```

and the computer responds by printing.

```
  Prolog-10  version 3.3
  Copyright (C) 1981 by D. Warren, F. Pereira and L. Byrd
  | ?-
```

The ?- is the Prolog prompt that tells the user Prolog is "ready and waiting". When you begin, your Prolog database is empty. There are two ways to enter facts into the database. The direct way is to "consult the user" (you). You type in [user]. and the Prolog prompt changes to a bar without the ?-.

```
  | ?- [user].
  |
```

Prolog is now ready to accept the facts you want to enter. Facts that have typing errors in them may or may not be accepted. Ones with punctuation errors usually will be rejected. When you type an error, you wlll probably get a message telling you there is a syntax error. If you cannot tell from the messages, you may have to check your system manual to find out what to do to recover from the error.

When you are done entering facts, you must indicate this to Prolog. For many systems, you will key in ^Z (hold down the control key while you key in either upper or lower case z). Others require a different signal, which should be indicated in your system manual. The ?- prompt will now appear. If you type listing., you will see a list of the facts in your database. If you type a predicate name in parentheses between listing and its period, you will see a list of just that set of predicates.

Another way of building a database for Prolog is to use your computer system's editor to create a file. The editor is another piece of software, not part of Prolog. The Prolog facts can be entered one per line. An advantage to using the editor method is that you can correct typing errors more easily. When you have the facts all entered and leave the editor, you save the file with a name. Now call on Prolog, and in response to the ?- prompt, type the file name in square brackets with a period at the end. Sometimes the file name must be enclosed in single quote marks. This consults the file and puts the facts from the file in the database. For example, if my file is named STAR, the Prolog interaction would look like this.

```
Prolog-10  version 3.3
Copyright (C) 1981 by D. Warren, F. Pereira and L. Byrd
| ?- [star].
star consulted  88 words  0.08 sec.
yes
| ?-
```

You can consult several files, one after another. This will put the files' contents together in the database. Again, you can look at the base by typing "listing." If you want to add more facts you can also consult [user] as above. Prolog systems usually have a mechanism for removing some parts from the database, but this mechanism varies from system to system. Check your system manual for specific instructions.

Now that you have some entries in the database, you are ready to make queries of the base. To ask Prolog a question, you simply type the question after the ?- prompt. Prolog gives you its answer and you respond with a semicolon to find alternative answers or with a return to terminate this question. You can then ask another question to which Prolog again responds. Working in this manner is called *interactive* computer use. The example below shows a complete interaction with Prolog, including building the base and asking questions.

```
Prolog-10  version 3.3
Copyright (C) 1981 by D. Warren, F. Pereira and L. Byrd
| ?- [star].
star consulted  88 words  0.08 sec.
yes
| ?- listing.
```

```
do(monday,call_agent).
do(friday,get_tickets).
do(tuesday,get_money).
do(monday,write_home).
do(friday,pack).
do(thursday,study).
do(tuesday,study).

yes

| ?- do(When,pack).

When = friday ;

no

| ?- [user].

| do(monday,shop).
| ^Z

| ?- listing.

do(monday,call_agent).
do(friday,get_tickets).
do(tuesday,get_money).
do(monday,write_home).
do(friday,pack).
do(thursday,study).
do(tuesday,study).
do(monday,shop).

yes
| ?- do(monday,What.
** syntax error **
do(monday,What
**here**
| do(monday,What).

What = call_agent ;
What = write_home ;
What = shop ;

no

| ?-^Z

core     641012 (18944 lo-seg + 4101068 hi-seg)
heap     2048 = 1301 in use + 747 free
global   1451 = 16 in use + 1435 free
local    1024 = 16 in use + 1008 free
trail     511 = 0 in use + 511 free
0.83 sec. runtime
```

These last six lines report some statistics on the amount of computer time and space your Prolog interaction used.

If you want to be able to reuse a database, you want to store it in a file. If you used an editor to build the base, it will still be stored in the original file. If you build or modify the base through [user] you may want to save the active base. Your system may allow it; some Prologs do not. Check your system manual.

When you are done with Prolog, key in ^Z to indicate you want to stop. Most Prolog systems also create a file, called a log that records the interaction between the user and Prolog. Check your system for a file, perhaps called Prolog.log.

EXERCISES CHAPTER 2

1. Identify the Prolog concept or concepts described by each phrase below.
    a) The part of a fact that shows the relationship.
    b) The mechanism for joining more that one fact in a query.
    c) The mechanism for requesting multiple responses to a query.
    d) The component that can be matched to constants or variables.
    e) How a world is described in Prolog.
    f) The variable that can be read as "don't care".
    g) What Prolog means when it reports a constant for a variable.
    h) How Prolog responds to a query about a fact with no variables.
2. For each statement, find one answer.
    a) In the following Prolog sentence, find the predicate.
    ```
    carpet(Room,beige).
    ```
    b) In the following Prolog sentence, find an argument.
    ```
    carpet(Room,beige).
    ```
    c) In the following Prolog sentence, find a constant.
    ```
    carpet(Room,beige).
    ```
    d) In the following Prolog sentence, find a variable.
    ```
    carpet(Room,beige).
    ```
3. Write Prolog sentences for the following.
    a) A set of facts with the name and birthplace of everyone in your family.
    b) a query to see who was born where.
    c) A query to see if two people were born in the same place.
    d) Additional facts for the database about the month people were born.
    e) Pick a month and write a query to see if anyone was born then.
    f) Pick a town and a month and write a query to see who was born there, then.

Extending the World Definition: Prolog Rules

3

The purpose of this chapter is to help the learner:

- define and use Prolog rules, including
  - simple rules.
  - compound rules.
  - multiple rules.
  - rules based on rules.
- understand and use recursive rules.
- recognize the sources of some problems in using recursion.

3.1 RULES

We use Prolog by asking questions. Facts form the basis for the answers that Prolog gives in response to the questions. Facts are statements that are true in our database, and thus are the fundamental building blocks of the database that describes a world. We can expand the description of the world we have defined by adding *rules* to the database. These rules, which are built on facts, or on other rules and facts, add another dimension to the informational power of Prolog's responses.

A Prolog rule has two parts:

1. a conclusion and
2. the requirements for the conclusion.

The rule specifies that the conclusion will be considered true if the requirements component is found to be true. For Prolog to answer a question based on a rule, it uses the data available in the base that tells it about the requirements component. If all the requirements can be found to be true, then the conclusion is true.

Syntax of Rules

The general form for Prolog rules is

```
<conclusion> :- <requirements>.
```

The conclusion, which is called the *head* of the rule, is followed by **:-** which is followed by one or more requirements. The part after the **:-** is called the *body* of the rule. The **:-** can usually be read as "if." A fact is actually a rule with no body.

Syntax Rules: Rules

- conclusion in head (one predicate)
- requirements in body (zero or more predicates)
- :- between head and body, if any
- period (full stop) at end

As an example of a rule, we might say we can conclude that "the sky is blue today if today's weather outlook is described as fair." In Prolog, the rule in this example could be expressed as

```
sky(blue) :- outlook(fair).
```

Assume we have a database with some facts from today's weather report,

```
outlook(fair).
high(seventies).
low(fifties).
rain(none).
```

to this base we add the rule

```
sky(blue) :- outlook(fair).
```

Now if we ask Prolog

```
?- sky(blue).
```

it will respond

```
yes
```

Since we asked about sky(blue), Prolog looked for a matching pattern in the data base. It found the pattern as the head of a rule, the rule that the sky is blue if the outlook is fair. Prolog then had to establish the truth of the items in the body. In this case, it found the fact that the outlook is fair. Prolog concluded from this fact, through the rule we have given it, that the sky is blue. Note that Prolog, having found the rule and discovered that there is a body containing requirements, goes back to the beginning of the database to begin searching for the facts to satisfy the requirements.

The structure of a Prolog rule allows us to specify a conclusion in the head that depends on the requirements indicated in the body. A fact is a rule with no requirement component, so it is a conclusion that is always true.

EXERCISES 3.1

1. Using the above database about weather plus the rule
   `need(umbrella) :- rain(heavy).`
   What will Prolog respond to
   `?- need(umbrella).` no
2. Write a rule that indicates one may wear shorts if the high is in the seventies. wear(shorts):- high(seventies)
3. Write a rule that says to plan a picnic if there is no rain.
   plan(picnic):- rain(none)

3.2 RULES WITH VARIABLES

?- Plan(Picnic).

While the previous rule is interesting, it might be more useful to define a rule that made reference to a specific day. With this kind of rule, we could generalize over a number of days' weather. For example,

```
color(sky,blue,Day) :- weather(Day,fair).
```

Within this rule, there are constants and variables. The variable, Day, allows the rule to be applied to more than one instance of fair weather. The constants show the specifics of the rule, that fair weather means a blue sky. The variable stands for one single day throughout the rule. That is, fair weather Monday tells us about Monday's sky color only, not any other day's. In Prolog, this is indicated by the use of the same variable name in both the conclusion and the requirements of the rule. The variable in the rule can refer to different days at different times, but it will refer to only one day at any one time.

A base about this week's weather might include

```
weather(sunday,fair).
weather(monday,overcast).
weather(tuesday,fair).
weather(wednesday,fair).
weather(thursday,overcast).
weather(friday,rainy).
weather(saturday,overcast).
color(sky,blue,Day) :- weather(Day,fair).
```

The rule with variables is part of the database along with the facts.

Now if we ask

```
?- color(sky,blue,Day).
```

Prolog will respond (with our entering the semicolons and returns)

```
Day = sunday ;
Day = tuesday ;
Day = wednesday ;
no
```

We have written our query using Day for the variable. Prolog will not confuse our use of Day as the variable name in the question with our use of the same word in the rule. The word Day in our query is not the same as the word Day in the rule from Prolog's point of view. A variable is considered the same for all appearances in one Prolog sentence, up to the period at the end. A sentence is also known as a *clause*; facts and rules are both clauses, and so are queries. A variable must be associated with the same constant throughout a clause but only within that clause. This is called the *scope* of the variable.

Because of their scope, Day in the rule and Day in the query are different to Prolog. They happen to appear in the same position in a matching predicate, but it is the position instead of the word that is important. We could use another variable instead. Prolog has its own internal representation for the variables in rules so that we are free to use any variable we choose in questions. For example, we might choose When.

```
?- color(sky,blue,When).
When = sunday ;
When = tuesday ;
When = wednesday ;
no
```

If we add another rule to our base which says

```
color(sky,grey,Day) :- weather(Day,overcast).
```

and ask a more general question

```
?- color(sky,Which,When).
```

Prolog will tell us.

```
When = sunday       Which = blue ;
When = tuesday      Which = blue ;
When = wednesday    Which = blue ;
When = monday       Which = grey ;
When = thursday     Which = grey ;
When = saturday     Which = grey ;
no
```

Notice that Prolog came to the blue-sky rule first and reported all the days with blue sky before moving on to the grey-sky days. Even though we used Day as the variable in both rules, the two rules are separate and there is no association between them.

One thing Prolog cannot do is work from conclusions back to requirements. The rule is not symmetric. For example, if we add the fact

```
color(sky,blue,christmas)
```

then ask

```
?- weather(christmas,fair).
```

Prolog will answer

```
no
```

To help understand this directionality, think of the :- as an arrow pointing to the left and the arrow pointing from the requirements to the conclusion. The reason underlying this asymmetry is that Prolog rules mean

conclusion "if" requirements

not

conclusion "if and only if" requirements

Thus

```
happy(jean) :- day(christmas).
```

means "Jean will be happy if it is Christmas" not "Jean will be happy only if it is Christmas" or "It must be Christmas if Jean is happy".

EXERCISES 3.2

1. If we added a rule to our weather database
   `bask(sun,When) :- weather(When,fair)`
   and ask
   `?- bask(sun,Day).`
   how will Prolog respond?
2. Write a rule that says to take an umbrella on a rainy day.
3. Write a rule that says to take an umbrella on an overcast day.

3.3 RULES WITH CONJUNCTIONS

Usually, our rules are designed for conclusions contingent on more than one requirement. To specify more than one requirement, we list them in the right hand part of the rule, with commas (,) between them. The comma can be read "and" just as it was with questions in Chapter 2. Thus, for example,

```
happy(birders,Day) :- weather(Day,fair), active(birds,Day).
```

could be read "Birders are happy on a day if the weather that day is fair and birds are active that day."

If we use our database from above and add this rule about birders being happy, we will have the database:

```
weather(sunday,fair).
weather(monday,overcast).
weather(tuesday,fair).
weather(wednesday,fair).
weather(thursday,overcast).
weather(friday,rainy).
weather(saturday,overcast).
color(sky,blue,Day) :- weather(Day,fair).
color(sky,grey,Day) :- weather(Day,overcast).
happy(birders,Day) :- weather(Day,fair), active(birds,Day).
```

If we now ask

```
?- happy(birders,Day).
```

Prolog will respond

```
no
```

This happens because Prolog has only enough data in its database to find out about one part of the requirements specification. There are facts about weather but none about birds being active. Prolog could not make the conclusion because it could not confirm both parts of the specified requirements. If we add

```
active(birds,sunday).
active(birds,tuesday).
active(birds,thursday).
```

then ask

```
?- happy(birders,When).
```

Prolog tells us

```
When = sunday ;
When = tuesday ;
no
```

Notice that Prolog only reports days when both parts of the requirements component can be confirmed with the variable representing the same specific day (see Figure 3.1). According to our rules, even though the birds were active Thursday, Thursday's overcast weather prevented birders from being happy.

Figure 3.1

Weather and Birding

```
weather(sunday,fair).
weather(monday,overcast).
weather(tuesday,fair).
weather(wednesday,fair).
weather(thursday,overcast).
weather(friday,rainy).
weather(saturday,overcast).
color(sky,blue,Day) :- weather(Day,fair).
color(sky,grey,Day) :- weather(Day,overcast).
happy(birders,Day) :- weather(Day,fair), active(birds,Day).
active(birds,sunday).
active(birds,tuesday).
active(birds,thursday).
```

When you as a programmer are writing the Prolog form of a rule you are defining, you may have trouble deciding how many variables to use and where to put them. For many people, it is easier to develop the correct clause if they focus on how they would *confirm* that a constant meets the rule than if they focus their thinking on how to *generate* the constants they want. Instead of thinking about getting information out of the database, think about an individual or an object and list the specifications that must be true in the database about that individual or object. After the rule is written, be sure that the form of the requirements in the body of a rule matches the form of the facts in the database.

EXERCISES 3.3

1. Write a rule that says birders will have mixed feelings about a day when it is rainy and birds are active.
2. According to our base, when will this happen?

3.4 RULES BASED ON OTHER RULES

The requirements in a rule need not only be facts; they can also be conclusions from other rules. For example

```
skyeyes(Person,Day) :- color(sky,Hue,Day),
                       color(eyes,Hue,Person).
```

would allow us to ask whose eyes matched the sky some day this week. It would only work if our earlier database (Figure 3.1) were augmented with some facts about the color of some people's eyes.

For example

```
color(eyes,grey,sue).
```

Three pairs of days and "sue" would be reported in response to

```
?- skyeyes(Who,When).
Who = sue   When = monday ;
Who = sue   When = thursday ;
Who = sue   When = saturday ;
no
```

In this particular example, the variable, Hue, appears only in the requirements component and not in the conclusion component. Nonetheless, Hue is involved in the matching needed to follow this rule. We will return to this example in Chapter 4 when we discuss the way Prolog follows its pattern-matching path through these rules.

In this rule notice that the second part of the body is shown on the next line. Prolog clauses can be spread out like this because it is the period that signals the end of the clause, not the end of a line. The clause may be broken up wherever a space is appropriate. Different layouts are chosen partly due to the size of the rules and partly due to programmer preference. Easy readability should be the main goal in choosing layout.

Multiple Rules

In addition to complex rules that use "and" in the requirements component, rules can use "or" (;) to specify more than one possible set of requirements for a single conclusion. For this circumstance, however, there is a better solution. In our database, we can have more than one rule that yields the same conclusion. These multiple rules will have the same predicate in their heads but different bodies. Say we use our database about weather and birders, add facts

```
observed(rarebird,wednesday).
observed(rarebird,friday).
```

and add the rule

```
happy(birders,Day) :- observed(rarebird,Day).
```

Our complete database is now

```
weather(sunday,fair).
weather(monday,overcast).
weather(tuesday,fair).
weather(wednesday,fair).
weather(thursday,overcast).
weather(friday,rainy).
weather(saturday,overcast).
color(sky,blue,Day) :- weather(Day,fair).
color(sky,grey,Day) :- weather(Day,overcast).
happy(birders,Day) :- weather(Day,fair), active(birds,Day).
happy(birders,Day) :- observed(rarebird,Day).
active(birds,sunday).
active(birds,tuesday).
active(birds,thursday).
observed(rarebird,wednesday).
observed(rarebird,friday).
```

Now we have two ways to conclude that birders will be happy. If we ask

```
?- happy(birders,When).
When = sunday ;
When = tuesday ;
When = wednesday ;
When = friday ;
no
```

The first two days were reported by using the first rule; the second two by using the second rule. If we add another fact,

```
  observed(rarebird,tuesday).
```

and ask our question again

```
  ?- happy(birders,When).
  When = sunday ;
  When = tuesday ;
  When = wednesday ;
  When = friday ;
  When = tuesday ;
  no
```

We have Tuesday reported to us twice, once by each of the two rules. There are two sets of requirements that both lead to the same conclusion. Prolog, following its top to bottom order, first found the rule about active birds and reported each day that met that requirement. Next, Prolog found the rarebird rule and reported days contingent on that. Tuesday met the requirements both times.

If two predicates have the same name but different arity, Prolog treats them as entirely different rules.

```
  a_name(Part1,Part2) :-
  a_name(Thing) :- ......
```

These two rules are not seen as similar by Prolog. Anytime that a rule is used the shape of the components must match. Thus, the variation in the arity makes a match to both rules impossible.

EXERCISES 3.4

Suppose you had a database:

```
     married(ann,abe).
     mother(ann,bet).
     mother(ann,cat).
     father(Man,Child) :- married(Woman,Man),
                          mother(Woman,Child).
     parent(Person,Child) :- mother(Person,Child).
     parent(Person,Child) :- father(Person,Child).
```

1. What will be Prolog's response to:
```
     ?- mother(ann,Whom).   whom = bet, cat
     ?- father(abe,Whom).   whom=bet, cat. no
     ?- parent(Who,bet).    who=ann, abe
```
2. Write queries to find out
   a) who is married to whom
   b) who has two parents known to the database

? - married (whom, what)
[illegible]

ADVANCED MATERIAL, for top students

3.5 RECURSION

In addition to having rules that use other rules as part of their requirements, we can have rules that use themselves as part of their requirements. For example, suppose we have an automobile named Bessy that several people have owned.

```
  owned(bessy,Person) :- bought(bessy,Person,Seller),
                         owned(bessy,Seller).
```

This rule may be read as "A person owns the car, Bessy, if that person bought Bessy from someone and that someone was Bessy's owner." This kind of rule is called *recursive* because the relationship in the conclusion appears again (recurs) in the body of the rule, where the requirements are specified. Recursive rules are useful when a relationship carries from one object to the next and from that object on to another. In this case, ownership of Bessy moves from person to person. Our understanding of ownership passing along is expressed syntactically in the recurrence in the rule.

While the relationship appears both in the head and body of the rule, the objects, at least some of which are represented by variables, are different in the two places. The pattern of the relationship, moving from object to object, is defined by the way the variables in the rule are related. Notice the way the variables Person and Seller appear in different places in the rule above. The people represented by the variables take different roles in different parts of the rule.

A recursive rule is a way of generating a chain of relationships. For a recursive rule to be effective, however, there must be some place in this chain of relationships where the recursion stops. In our example, someone bought Bessy from the manufacturer.

```
  owner(bessy,Person) :- bought(bessy,Person,manufacturer).
```

This stopping condition must be answerable in the database like any other rule. It can use other rules or facts, but should not use the recursive rule. Every time a recursive step along the chain is taken, some progress is made toward the stopping condition. In our example, each time we chain from a purchaser to the seller, we are getting closer to the original owner.

To specify a recursive rule for Prolog, we need a multiple rule definition. First we give the rule for the stopping condition and then the recursive rule. Occasionally, there will be more than one stopping condition. In those instances all the stopping rules should be written into the database before the recursive rule. Below is an example database and examples of questions and responses. Later (in Chapter 4) we will consider the process Prolog uses to keep track of the recursion. It is more important now to learn what Prolog does than to investigate the underlying process.

In this base, the first two lines make up the rule for ownership and the last four lines are facts.

```
owned(bessy,Person) :- bought(bessy,Person,manufacturer).
owned(bessy,Person) :- bought(bessy,Person,Seller),
        owned(bessy,Seller).
bought(bessy,abe,ben).
bought(bessy,ben,carl).
bought(bessy,carl,fred).
bought(bessy,fred,manufacturer).
```

We can inquire about any one of these people owning Bessy.

```
?- owned(bessy,abe).
yes
?- owned(bessy,sam).
no
```

Or we can present the more general question.

```
?- owned(bessy,Who).
Who = fred ;
Who = abe ;
Who = ben ;
Who = carl ;
no
```

Note the order in which these names appeared. Putting the stopping case first applies to the components of the multiple rule that defines the recursion; it is not a restriction on the ordering of the facts that the rule will use to reach its conclusions. The facts above could be in any order. For example,

```
bought(bessy,fred,manufacturer).
bought(bessy,ben,carl).
bought(bessy,abe,ben).
bought(bessy,carl,fred).
```

Whenever we ask for all the owners of Bessy, however, Fred will be reported first. His ownership is based on the first part of the rule, so it is verified first. The order in which the others will be reported will depend on their order in the base, not, say, on the order in which Bessy was acquired. Thus, using the order of facts in the second case above, the query and the response are

```
?- owned(bessy,Who).
Who = fred ;
Who = ben ;
Who = abe ;
Who = carl ;
no
```

Here is another example with a recursive rule. The example is built on parent/child relationships. Family tree descriptions are some of the most straightforward examples of Prolog recursion. They are common textbook examples because families are full of relationships that move from object (person) to object (person). Again, the non-recursive part of the rule for ancestor is specified before the recursive part of the rule.

```
mother(ann,bet).
mother(ann,cat).
mother(bet,may).
mother(may,nan).
parent(Person,Child) :-mother(Person,Child).
ancestor(Person,Other) :- parent(Person,Other).
ancestor(Person,Other) :- parent(Person,Middle),
        ancestor(Middle,Other).

?- ancestor(Older,Younger).

Younger = bet   Older = ann ;
Younger = cat   Older = ann ;
Younger = may   Older = bet ;
Younger = nan   Older = may ;
Younger = may   Older = ann ;
Younger = nan   Older = ann ;
Younger = nan   Older = bet ;
no
```

EXERCISES 3.5

1. Add these three predicates to the database immediately above.
```
married(abe,ann).
father(Man,Child) :- mother(Woman,Child),
married(Man,Woman).
parent(Person,Child) :- father(Person,Child).
```
Now how will Prolog respond to
```
?- ancestor(Older,Younger).
```
2. Sally invented a new game called Glump. She taught it to Joe, who taught it to Sam and May. May taught it to Leo and he, in turn, taught it to his sister. Write a database using a recursive rule that will tell us who knows how to play Glump.

3.[redacted] PROBLEMS WITH RECURSION

Sometimes in Prolog [redacted] rules do not result in the behavior we had planned. This misbehavior is n[redacted]t of providing the wrong rule; it has to do with the mechanism Prolog uses to [redacted] at conclusions from rules. You can avoid a number of problems by keeping [redacted] few pointers for writing recursive rules.

EXERCISES 3.6

1. Here is an earlier [redacted]e with some rules added. Prolog will not answer the query correctly. Why[redacted]

```
married(ann,abe)[redacted]
mother(ann,bet).
mother(ann,cat).
mother(Woman,Child) :-[redacted]ther(Man,Child),
          married(Woman[redacted]n).
father(Man,Child) :- mothe[redacted]Woman,Child),
          married(Woman,Ma[redacted]
parent(Person,Child) :- mothe[redacted]erson,Child).
parent(Person,Child) :- father[redacted]son,Child).
?- mother(ann,Whom).
```

2. What is a potential problem with this rule a[redacted]how should it be corrected?

```
do(First_half,Second_half) :-
          do(First_quarter,Second_half[redacted]
          check(First_half,First_quarter[redacted]
```

SUMMARY

Rules can be added to facts in a database. The conclusion of a rule is in the rule's head and the requirements for the conclusion are in the rule's body. Rules can have constants and variables. The body of a rule may include a conjunction and may include the conclusion of another rule as part of its requirements. There may be more than one rule that specifies the same conclusion. In the case of recursion, there will usually be one that specifies the stopping case and one that gives the recursive rule[redacted]ful choice and ordering of components in the body of the recursive rule, along[redacted]ting the stopping case first, will help guarantee that recursive rules behave as [redacted]

SYNTAX SUMMARY

Rules

```
<conclusion> :- <requirement1>,
        <requirement2>,...,<requirementN>.
```

EXPERIMENTING WITH CHAPTER 3

Purpose:

To see: adding rules to a base
making queries based on rules
interrupting processing

Rules are added to the database using the same mechanism as is used with facts, either through consulting [user] or consulting a file that has been created with the system's editor.

When you do listing, you may find that Prolog has replaced the variable names in your rules with another kind of name. Prolog also may arrange the rules with different spacing from the way you wrote them, but it will not rearrange the order of the predicates in the body of the rule. Prolog groups like predicates together, but within the group, it maintains the order in which the predicates were entered.

By the time you are using rules, you may find you want to tell Prolog to stop what it is doing. This is particularly important if Prolog seems to be doing nothing for longer than the time it normally takes to respond. The process may have gone into an uncontrolled recursion. Telling Prolog to stop is called an *interrupt*. Check your system manual to find out how this is done. On some systems, a ^C (CONTROL C) interrupts Prolog and presents the user with some choices about what to do next.

EXERCISES CHAPTER 3

1. Identify the Prolog concept or concepts described by each phrase below.
   a) The contents of a Prolog database.
   b) The two parts of a Prolog rule.
   c) The part of a rule containing the requirements component.
   d) When Prolog uses a rule, the first part that will be matched.
   e) The two ways objects can be represented in rules.
   f) The number of predicates in the head of a rule and the number in the body.
   g) Rules in which the predicate from the conclusion also appears in the requirements.
   h) The two parts that are almost always included in a recursive rule.
2. Using the Prolog database given below, select the number in front of the line or lines that show an example of the terms below.

```
[1] knows(sally) :- knows(joe).
[2] knows(Hearer) :- tells(Hearer,Teller), knows(Teller).
[3] tells(sue,sally).
[4] knows(joe).
```

   a) fact
   b) compound rule
   c) multiple rule
   d) recursive rule
   e) stopping case
   f) recursive case
3. Write Prolog databases to describe these situations.
   a) Shop at the grocery store that has the best vegetables.
   b) Shop at the market that has the lowest price on turkey.
   c) Shop at a store if it has good vegetables and beef on special.
   d) Shop at Ben's market.
   e) Four students saw the note Tom wrote to Joe. They sit in a row in class: Tom, Mary, Sally, Joe. Each one looked at the note before passing it on.