In chapter 1 we have seen some examples of atoms and variables. In general, however, atoms can take more complicated forms—that is, strings of the following characters:

- upper-case letters A, B, ..., Z
- lower-case letters a, b, ..., z
- digits 0, 1, 2, ..., 9
- special characters such as *, <, >, :,

Atoms can be constructed in three ways:

1. Strings of letters, digits, and the underscore character, starting with a lower-case letter:
   - anna
   - nil
   - x25
   - x_25
   - x_2548
   - x_v
   - alpha_beta
   - procedure
   - missJones

2. Strings of special characters:

   When using atoms of this form, some care is necessary because some strings of special characters already have a predefined meaning. An example is :-.

Figure 2.1 Data objects in Prolog.
Atoms

- Start with lowercase letters
- Strings of special characters, e.g. `:-` is an atom
- Enclosed in single quotes, e.g. `'Tom'` is an atom
Anonymous variables

• Singleton variable – variable in a rule that is named but not used
• Anonymous variable – an unnamed variable in a rule
• Avoid singleton variables
Figure 2.2 Date is an example of a structured object: (a) as it is represented as a tree; (b) as it is written in Prolog.
Figure 2.2 Date is an example of a structured object: (a) as it is represented as a tree; (b) as it is written in Prolog.

- **Name is** `date`
- **Arity is** 3
- **Functor** `date`
- **Arguments** `(1, may, 2001)`
2.1 Data objects

simple objects

constants

variables

structures

data objects

Figure 2.1 Data objects in Prolog.

Atoms, numbers, variables and structures are all terms.
Chapter 2
Syntax and Meaning of Prolog Programs

We can, however, define the same name above, the arguments, and the arguments and the arguments are defined by two names, as already explained. The program in the program is an example of the data structure's simplicity and power. It is one of the reasons why Prolog is so naturally applied to problems that involve symbolic manipulation.

Syntactically, all data objects in Prolog are terms. For example, point may and date(point(1, may, 2001)) are terms. All structured objects can be pictured as trees (see Figure 2.2 for an example). The root of the tree is the functor, and the children of the root are the components. If a component is also a structure then it is a subtree of the tree that corresponds to the whole structured object.

Our next example will show how structures can be used to represent some simple geometric objects (see Figure 2.3). A point in two-dimensional space is defined by its three coordinates; a line segment is defined by two points; and a triangle can be defined by three points. Let us choose the following functors: point for points, seg for line segments, and triangle for triangles. Then the objects in Figure 2.3 can be represented as follows:

\[
\begin{align*}
S &: \text{seg(point(1,1), point(2,3))} \\
T &: \text{triangle(point(1,1), point(6,4), point(7,1))}
\end{align*}
\]

The corresponding tree representation of these objects is shown in Figure 2.4. The functor at the root of the tree is called the principal functor of the term. So seg is the principal functor in the term seg(point(1,1), point(2,3)).

If in the same program we also had points in three-dimensional space then we could use another functor, point3, say, for their representation:

\[
\text{point3(X,Y,Z)}
\]

Figure 2.3 Some simple geometric objects.
Figure 2.3 Some simple geometric objects.

P1 represented as point(1, 1).
Figure 2.3 Some simple geometric objects.

P1 represented as point(1, 1).
S represented as seg(point(1, 1), point(2, 3)).
PI represented as point(1, 1).
S represented as seg(point(1, 1), point(2, 3)).
T represented as
triangle(point(4, 2), point(6, 4), point(7, 1)).

Figure 2.3 Some simple geometric objects.
2.1 Data objects

Figure 2.4 Tree representation of the objects in Figure 2.3.
Figure 2.5 A tree structure that corresponds to the arithmetic expression \((a + b) \times (c - 5)\).
Chapter 2 Syntax and Meaning of Prolog Programs

According to the syntax of terms introduced so far this can be written, using the symbols ‘*’, ‘+’ and ‘-’ as functors, as follows:

\[ * ( + (a, b), - (c, 5) ) \]

This is, of course, a legal Prolog term, but it is not in the form that we would normally use. We would normally prefer the usual infix notation as used in mathematics. In fact, Prolog also allows us to use the infix notation so that the symbols ‘*’, ‘+’ and ‘-’ are written as infix operators. Details of how the programmer can define his or her own operators will be discussed in Chapter 3.

As the last example we consider some simple electric circuits shown in Figure 2.6.

The right-hand side of the figure shows the tree representation of these circuits. The atoms r1, r2, r3 and r4 are the names of the resistors. The functors par and seq respectively.

**Figure 2.6** Some simple electric circuits and their tree representations: (a) sequential composition of resistors r1 and r2; (b) parallel composition of two resistors; (c) parallel composition of three resistors; (d) parallel composition of r1 and another circuit.
According to the syntax of terms introduced so far this can be written, using the symbols '*' and 'J as functors, as follows:

\[
* (+ (a, b), - (c, 5))
\]

This is, of course, a legal Prolog term, but it is not in the form that we would normally use. We would normally prefer the usual infix notation as used in mathematics. In fact, Prolog also allows us to use the infix notation so that the symbols '*' and 'J are written as infix operators.

Details of how the programmer can define his or her own operators will be discussed in Chapter 3.

As the last example we consider some simple electric circuits shown in Figure 2.6. The right-hand side of the figure shows the tree representation of these circuits. The atoms rl, 12, 13 and 14 are the names of the resistors. The functors par and seq respectively.

\[
\begin{align*}
&\text{seq} \\
&\text{par} \\
&\text{par} \\
&\text{par}
\end{align*}
\]

Figure 2.6 Some simple electric circuits and their tree representations: (a) sequential composition of resistors r1 and r2; (b) parallel composition of two resistors; (c) parallel composition of three resistors; (d) parallel composition of r1 and another circuit.
According to the syntax of terms introduced so far this can be written, using the symbols ‘*’, ‘*’ and ‘J as functors, as follows:

\[ *\left( +\left(r_1, r_2\right), -\left(c, 5\right)\right) \]

This is, of course, a legal Prolog term, but it is not in the form that we would normally use. We would normally prefer the usual infix notation as used in mathematics. In fact, Prolog also allows us to use the infix notation so that the symbols ‘*’, ‘*’ and ‘J are written as infix operators. Details of how the programmer can define his or her own operators will be discussed in Chapter 3.

As the last example we consider some simple electric circuits shown in Figure 2.6. The right-hand side of the figure shows the tree representation of these circuits. The atoms rl, 12, 13 and 14 are the names of the resistors. The functors par and seq respectively. The <seq(11,12) pat(rl,12) par(11,par(r2,par(11,seq(par(11,seq(12,seq(12,r1)))))))>

Figure 2.6 Some simple electric circuits and their tree representations: (a) sequential composition of resistors r1 and r2; (b) parallel composition of two resistors; (c) parallel composition of three resistors; (d) parallel composition of r1 and another circuit.
Figure 2.7 Matching triangle( point(1,1), A, point(2,3) ) = triangle( X, point(4,Y), point(2,Z) )
The following example will illustrate how matching alone can be used for interesting computation. \( \text{Let us return to the simple geometric objects of Figure 2.4, and define a piece of program for recognizing horizontal and vertical line segments.} \)

'Vertical' is a property of segments, so it can be formalized in Prolog as a unary relation. Figure 2.8 helps to formulate this relation. A segment is vertical if the x-coordinates of its end-points are equal, otherwise there is no other restriction on the segment. The property 'horizontal' is similarly formulated, with

\[
\text{point}(X, Y_1) \quad \text{point}(X, Y) \quad \text{point}(X_1, Y) \quad \text{point}(X, Y) \quad \text{point}(X_1, Y)
\]

\[
\text{Figure 2.8 Illustration of vertical and horizontal line segments.}
\]
Matching
= is the matching operator

(1) If S and T are constants then S and T match only if they are the same object.

(2) If S is a variable and T is anything, then they match, and S is instantiated to T. Conversely, if T is a variable then T is instantiated to S.

(3) If S and T are structures then they match only if
   (a) S and T have the same principal functor, and
   (b) all their corresponding components match.

The resulting instantiation is determined by the matching of the components.
Definitions

Clause – a rule or a fact

Instance of clause $c$ – $c$ with each variable substituted by some term.

, is conjunction “and”

; is disjunction “or”

$P :\neg Q ; R$. is the same as

$P :\neg Q$.

$P :\neg R$. 
Declarative meaning – what?

A goal \( G \) is true (that is, satisfiable, or logically follows from the program) if and only if:

1. there is a clause \( C \) in the program such that
2. there is a clause instance \( I \) of \( C \) such that
   a. the head of \( I \) is identical to \( G \), and
   b. all the goals in the body of \( I \) are true.

\[
\text{Query, } ?- \ G.
\]

\[
\begin{align*}
\text{head} & : G \\
\text{body} & : Q, R, S.
\end{align*}
\]
Procedural meaning – how?

Figure 2.9 Input/output view of the procedure that executes a list of goals.
Program (data base)

big( bear). % Clause 1
big( elephant). % Clause 2
small( cat). % Clause 3

brown( bear). % Clause 4
black( cat). % Clause 5
gray( elephant). % Clause 6
dark( Z) :-
    black( Z). % Clause 7: Anything black is dark
dark( Z) :-
    brown( Z). % Clause 8: Anything brown is dark

Goal list
?- dark( X), big( X).

Execution trace
dark( X), big( X).
Program (data base)
big( bear). % Clause 1
big( elephant). % Clause 2
small( cat). % Clause 3

brown( bear). % Clause 4
black( cat). % Clause 5
gray( elephant). % Clause 6

dark( Z) :-
    black( Z). % Clause 7: Anything black is dark
dark( Z) :-
    brown( Z). % Clause 8: Anything brown is dark

Goal list
?- dark( X), big( X).

Execution trace
dark( X), big( X).
Program (data base)

big( bear).       % Clause 1
big( elephant).   % Clause 2
small( cat).      % Clause 3

brown( bear).     % Clause 4
black( cat).      % Clause 5
gray( elephant).  % Clause 6

dark( Z) :-
    black( Z).     % Clause 7: Anything black is dark

dark( Z) :-
    brown( Z).     % Clause 8: Anything brown is dark

Goal list

?- dark( X), big( X).

Execution trace

black( X), big( X).
Program (data base)

- big( bear).  % Clause 1
- big( elephant).  % Clause 2
- small( cat).  % Clause 3

- brown( bear).  % Clause 4
- black( cat).  % Clause 5
- gray( elephant).  % Clause 6

dark( Z) :-
  black( Z).  % Clause 7: Anything black is dark

dark( Z) :-
  brown( Z).  % Clause 8: Anything brown is dark

Goal list

?- dark( X), big( X).

Execution trace

black( X), big( X).
Program (data base)

big( bear).      % Clause 1
big( elephant).  % Clause 2
small( cat).     % Clause 3

brown( bear).    % Clause 4
black( cat).     % Clause 5
gray( elephant). % Clause 6

dark( Z) :-
    black( Z).     % Clause 7: Anything black is dark
dark( Z) :-
    brown( Z).     % Clause 8: Anything brown is dark

Goal list

?- dark( X), big( X).

Execution trace

black( X), big( X).
Program (data base)
big( bear).      % Clause 1
big( elephant).  % Clause 2
small( cat).     % Clause 3

brown( bear).    % Clause 4
black( cat).     % Clause 5
gray( elephant). % Clause 6
dark( Z) :-
   black( Z).     % Clause 7: Anything black is dark
dark( Z) :-
   brown( Z).     % Clause 8: Anything brown is dark

Goal list
?- dark( X), big( X).

Execution trace
big( cat).
Program (data base)
big( bear). % Clause 1
big( elephant). % Clause 2
small( cat). % Clause 3

brown( bear). % Clause 4
black( cat). % Clause 5
gray( elephant). % Clause 6
dark( Z) :-
  black( Z). % Clause 7: Anything black is dark
dark( Z) :-
  brown( Z). % Clause 8: Anything brown is dark

Goal list
?- dark( X), big( X).

Execution trace
big( cat).
Program (data base)

big( bear).  % Clause 1
big( elephant). % Clause 2
small( cat).   % Clause 3

brown( bear). % Clause 4
black( cat).  % Clause 5
gray( elephant). % Clause 6
dark( Z) :-
    black( Z).  % Clause 7: Anything black is dark
dark( Z) :-
    brown( Z).  % Clause 8: Anything brown is dark

Goal list

?- dark( X), big( X).

Execution trace

big( cat).
Program (data base)

big( bear). % Clause 1
big( elephant). % Clause 2
small( cat). % Clause 3

brown( bear). % Clause 4
black( cat). % Clause 5
gray( elephant). % Clause 6
dark( Z) :-
    black( Z). % Clause 7: Anything black is dark
dark( Z) :-
    brown( Z). % Clause 8: Anything brown is dark

Goal list

?- dark( X), big( X).

Execution trace

big( cat).

Fail
Program (data base)

big( bear).       % Clause 1
big( elephant).   % Clause 2
small( cat).      % Clause 3

brown( bear).     % Clause 4
black( cat).      % Clause 5
gray( elephant).  % Clause 6
dark( Z) :-
    black( Z).      % Clause 7: Anything black is dark
dark( Z) :-
    brown( Z).      % Clause 8: Anything brown is dark

Goal list

?- dark( X), big( X).

Execution trace

black( X), big( X).

Backtrack to black( cat).
Program (data base)

big( bear). % Clause 1
big( elephant). % Clause 2
small( cat). % Clause 3

brown( bear). % Clause 4
black( cat). % Clause 5
gray( elephant). % Clause 6
dark( Z) :-
  black( Z). % Clause 7: Anything black is dark
dark( Z) :-
  brown( Z). % Clause 8: Anything brown is dark

Goal list

?- dark( X), big( X).

Execution trace

black( X), big( X).
Program (data base)
big( bear). % Clause 1
big( elephant). % Clause 2
small( cat). % Clause 3

brown( bear). % Clause 4
black( cat). % Clause 5
gray( elephant). % Clause 6
dark( Z) :-
    black( Z). % Clause 7: Anything black is dark
dark( Z) :-
    brown( Z). % Clause 8: Anything brown is dark

Goal list
?- dark( X), big( X).

Execution trace
black( X), big( X).
Program (data base)

big(bear). % Clause 1
big(elephant). % Clause 2
small(cat). % Clause 3

brown(bear). % Clause 4
black(cat). % Clause 5
gray(elephant). % Clause 6

dark(Z) :-
    black(Z). % Clause 7: Anything black is dark

dark(Z) :-
    brown(Z). % Clause 8: Anything brown is dark

Goal list
?- dark(X), big(X).

Execution trace
black(X), big(X).
Fail
Program (data base)

big( bear). % Clause 1
big( elephant). % Clause 2
small( cat). % Clause 3

brown( bear). % Clause 4
black( cat). % Clause 5
gray( elephant). % Clause 6
dark( Z) :-
  black( Z). % Clause 7: Anything black is dark
dark( Z) :-
  brown( Z). % Clause 8: Anything brown is dark

Goal list

?- dark( X), big( X).

Execution trace

black( X), big( X).

Backtrack to dark( Z).
Program (data base)

big( bear). % Clause 1
big( elephant). % Clause 2
small( cat). % Clause 3

brown( bear). % Clause 4
black( cat). % Clause 5
gray( elephant). % Clause 6
dark( Z) :-
    black( Z). % Clause 7: Anything black is dark
dark( Z) :-
    brown( Z). % Clause 8: Anything brown is dark

Goal list
?- dark( X), big( X).

Execution trace
dark( X), big( X).
Program (data base)
big( bear). % Clause 1
big( elephant). % Clause 2
small( cat). % Clause 3

brown( bear). % Clause 4
black( cat). % Clause 5
gray( elephant). % Clause 6
dark( Z) :-
    black( Z). % Clause 7: Anything black is dark
dark( Z) :-
    brown( Z). % Clause 8: Anything brown is dark

Goal list
?- dark( X), big( X).

Execution trace
dark( X), big( X).
Program (data base)

big( bear). % Clause 1
big( elephant). % Clause 2
small( cat). % Clause 3

brown( bear). % Clause 4
black( cat). % Clause 5
gray( elephant). % Clause 6

dark( Z) :-
    black( Z). % Clause 7: Anything black is dark

dark( Z) :-
    brown( Z). % Clause 8: Anything brown is dark

Goal list
?- dark( X), big( X).

Execution trace
brown( X), big( X).
Program (data base)

big( bear). % Clause 1
big( elephant). % Clause 2
small( cat). % Clause 3

brown( bear). % Clause 4
black( cat). % Clause 5
gray( elephant). % Clause 6
dark( Z) :-
    black( Z). % Clause 7: Anything black is dark
dark( Z) :-
    brown( Z). % Clause 8: Anything brown is dark

Goal list
?- dark( X), big( X).

Execution trace
brown( X), big( X).
Program (data base)

big( bear).      % Clause 1
big( elephant).  % Clause 2
small( cat).     % Clause 3

brown( bear).    % Clause 4
black( cat).     % Clause 5
gray( elephant). % Clause 6

dark( Z) :-
    black( Z). % Clause 7: Anything black is dark

dark( Z) :-
    brown( Z). % Clause 8: Anything brown is dark

Goal list

?- dark( X), big( X).

Execution trace

brown( X), big( X).
Program (data base)

big( bear). % Clause 1
big( elephant). % Clause 2
small( cat). % Clause 3

brown( bear). % Clause 4
black( cat). % Clause 5
gray( elephant). % Clause 6
dark( Z) :-
    black( Z). % Clause 7: Anything black is dark
dark( Z) :-
    brown( Z). % Clause 8: Anything brown is dark

Goal list

?- dark( X), big( X).

Execution trace

big( bear).
Program (data base)

big( bear). % Clause 1
big( elephant). % Clause 2
small( cat). % Clause 3

brown( bear). % Clause 4
black( cat). % Clause 5
gray( elephant). % Clause 6

dark( Z) :-
    black( Z). % Clause 7: Anything black is dark
dark( Z) :-
    brown( Z). % Clause 8: Anything brown is dark

Goal list

?- dark( X), big( X).

Execution trace

big( bear).
Program (data base)

big( bear).  % Clause 1
big( elephant). % Clause 2
small( cat).  % Clause 3

brown( bear).  % Clause 4
black( cat).  % Clause 5
gray( elephant). % Clause 6
dark( Z) :-
    black( Z). % Clause 7: Anything black is dark
dark( Z) :-
    brown( Z). % Clause 8: Anything brown is dark

Goal list
?- dark( X), big( X).

Execution trace
X = bear
procedure execute (Program, GoalList, Success);

Input arguments:
  Program: list of clauses
  GoalList: list of goals

Output argument:
  Success: truth value; Success will become true if GoalList is true with respect to Program

Local variables:
  Goal: goal
  OtherGoals: list of goals
  Satisfied: truth value
  MatchOK: truth value
  Instant: instantiation of variables
  H, H', B1, B1', . . ., Bn, Bn': goals

Auxiliary functions:
  empty(L): returns true if L is the empty list
  head(L): returns the first element of list L
  tail(L): returns the rest of L
  append(L1, L2): appends list L2 at the end of list L1
  match(T1, T2, MatchOK, Instant): tries to match terms T1 and T2; if succeeds
    then MatchOK is true and Instant is the corresponding instantiation of variables
  substitute(Instant, Goals): substitutes variables in Goals according to instantiation Instant
begin
  if empty(GoalList) then Success := true
  else
    begin
      Goal := head(GoalList);
      OtherGoals := tail(GoalList);
      Satisfied := false;
      while not Satisfied and "more clauses in program" do
        begin
          Let next clause in Program be
          \( H : - B_1, \ldots, B_n \).
          Construct a variant of this clause
          \( H' : - B_1', \ldots, B_n' \).
          match(Goal, H', MatchOK, Instant);
          if MatchOK then
            begin
              NewGoals := append([B1', \ldots, Bn'], OtherGoals);
              NewGoals := substitute(Instant, NewGoals);
              execute(Program, NewGoals, Satisfied)
            end
          end;
        end;
      Success := Satisfied
    end
  end;
Reordering clauses and goals

• Reordering can have a big effect on efficiency.

• In extreme cases, reordering can cause an infinite recursive loop.
Figure 2.13 The complete execution trace to satisfy the goal \texttt{anc2( tom, pat)}. All the alternative paths in the large left subtree fail, before the right-most path succeeds.
anc3(\(X, Z\)):-
    parent(\(X, Z\)).

anc3(\(X, Z\)):-
    anc3(\(X, Y\)),
    parent(\(Y, Z\)).

**Figure 2.14** The execution trace to satisfy the goal \(\text{anc3}(\text{tom}, \text{pat})\).
Figure 2.15 Infinite execution trace to satisfy the goal \texttt{anc4( tom, pat)}. 