2.5 Example: monkey and banana

The monkey and banana problem is often used as a simple example of problem solving. Our Prolog program for this problem will show how the mechanisms of matching and backtracking can be used in such exercises. We will develop the program in the non-procedural way, and then study its procedural behaviour in detail. The program will be compact and illustrative.

We will use the following variation of the problem. There is a monkey at the door into a room. In the middle of the room a banana is hanging from the ceiling. The monkey is hungry and wants to get the banana, but he cannot stretch high enough from the floor. At the window of the room there is a box the monkey may use. The monkey can perform the following actions: walk on the floor, climb the box, push the box around (if it is already at the box) and grasp the banana if standing on the box directly under the banana. Can the monkey get the banana?

One important task in programming is that of finding a representation of the problem in terms of concepts of the programming language used. In our case we can think of the 'monkey world' as always being in some *state* that can change in time. The current state is determined by the positions of the objects. For example, the initial state of the world is determined by:

(1) Monkey is at door.
(2) Monkey is on floor.
(3) Box is at window.
(4) Monkey does not have banana.

It is convenient to combine all of these four pieces of information into one structured object. Let us choose the word 'state' as the functor to hold the four components together. Figure 2.12 shows the initial state represented as a structured object.

Our problem can be viewed as a one-person game. Let us now formalize the rules of the game. First, the goal of the game is a situation in which the monkey has the banana; that is, any state in which the last component is 'has':

```
state( _, _, _, has)
```

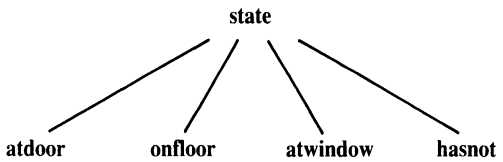

Figure 2.12 The initial state of the monkey world represented as a structured object. The four components are: horizontal position of monkey, vertical position of monkey, position of box, monkey has or has not the banana.

Second, what are the allowed moves that change the world from one state to another? There are four types of moves:

(1) grasp banana,
(2) climb box,
(3) push box,
(4) walk around.

Not all moves are possible in every possible state of the world. For example, the move 'grasp' is only possible if the monkey is standing on the box directly under the banana (which is in the middle of the room) and does not have the banana yet. Such rules can be formalized in Prolog as a three-place relation named **move**:

```
move( State1, M, State2)
```

The three arguments of the relation specify a move thus:

```
State1 --------> State2
          M
```

State1 is the state before the move, M is the move executed and **State2** is the state after the move.

The move 'grasp', with its necessary precondition on the state before the move, can be defined by the clause:

```
move( state( middle, onbox, middle, hasnot),     % Before move
      grasp,                                     % Move
      state( middle, onbox, middle, has) ).      % After move
```

This fact says that after the move the monkey has the banana, and he has remained on the box in the middle of the room.

In a similar way we can express the fact that the monkey on the floor can walk from any horizontal position P1 to any position P2. The monkey can do this regardless of the position of the box and whether it has the banana or not. All this can be defined by the following Prolog fact:

```
move( state( P1, onfloor, B, H),
      walk( P1, P2),                             % Walk from P1 to P2
      state( P2, onfloor, B, H) ).
```

Note that this clause says many things, including, for example:

- the move executed was 'walk from some position P1 to some position P2';
- the monkey is on the floor before and after the move;

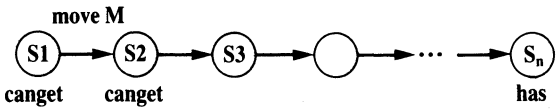

Figure 2.13 Recursive formulation of **canget**.

- the box is at some point B which remained the same after the move;
- the 'has banana' status remains the same after the move.

The clause actually specifies a whole set of possible moves because it is applicable to any situation that matches the specified state before the move. Such a specification is therefore sometimes also called a move *schema*. Due to the concept of Prolog variables such schemas can be easily programmed in Prolog.

The other two types of moves, 'push' and 'climb', can be similarly specified.

The main kind of question that our program will have to answer is: Can the monkey in some initial state S get the banana? This can be formulated as a predicate

canget(S)

where the argument S is a state of the monkey world. The program for **canget** can be based on two observations:

(1) For any state S in which the monkey already has the banana, the predicate **canget** must certainly be true; no move is needed in this case. This corresponds to the Prolog fact:

```
canget( state( _, _, _, has) ).
```

(2) In other cases one or more moves are necessary. The monkey can get the banana in any state S1 if there is some move M from state S1 to some state S2, such that the monkey can then get the banana in state S2 (in zero or more moves). This principle is illustrated in Figure 2.13. A Prolog clause that corresponds to this rule is:

```
canget( S1) :-
  move( S1, M, S2),
  canget( S2).
```

This completes our program which is shown in Figure 2.14.

The formulation of **canget** is recursive and is similar to that of the **predecessor** relation of Chapter 1 (compare Figures 2.13 and 1.7). This principle is used in Prolog again and again.

```
% Legal moves

move( state( middle, onbox, middle, hasnot),
      grasp,                                    % Grasp banana
      state( middle, onbox, middle, has) ).

move( state( P, onfloor, P, H),
      climb,                                    % Climb box
      state( P, onbox, P, H) ).

move( state( P1, onfloor, P1, H),
      push( P1, P2),                            % Push box from P1 to P2
      state( P2, onfloor, P2, H) ).

move( state( P1, onfloor, B, H),
      walk( P1, P2),                            % Walk from P1 to P2
      state( P2, onfloor, B, H) ).

% canget( State): monkey can get banana in State

canget( state( _, _, _, has) ).                 % can 1: Monkey already has it

canget( State1) :-                              % can 2: Do some work to get it
  move( State1, Move, State2),                  % Do something
  canget( State2).                              % Get it now
```

Figure 2.14 A program for the monkey and banana problem.

We have developed our monkey and banana program in the non-procedural way. Let us now study its *procedural* behaviour by considering the following question to the program:

```
?- canget( state( atdoor, onfloor, atwindow, hasnot) ).
```

Prolog's answer is 'yes'. The process carried out by Prolog to reach this answer proceeds, according to the procedural semantics of Prolog, through a sequence of goal lists. It involves some search for right moves among the possible alternative moves. At some point this search will take a wrong move leading to a dead branch. At this stage, backtracking will help it to recover. Figure 2.15 illustrates this search process.

To answer the question Prolog had to backtrack once only. A right sequence of moves was found almost straight away. The reason for this efficiency of the program was the order in which the clauses about the **move** relation occurred in the program. The order in our case (luckily) turned out to be quite suitable. However, less lucky orderings are possible. According to the rules of the game, the monkey could just as easily try to walk here or there

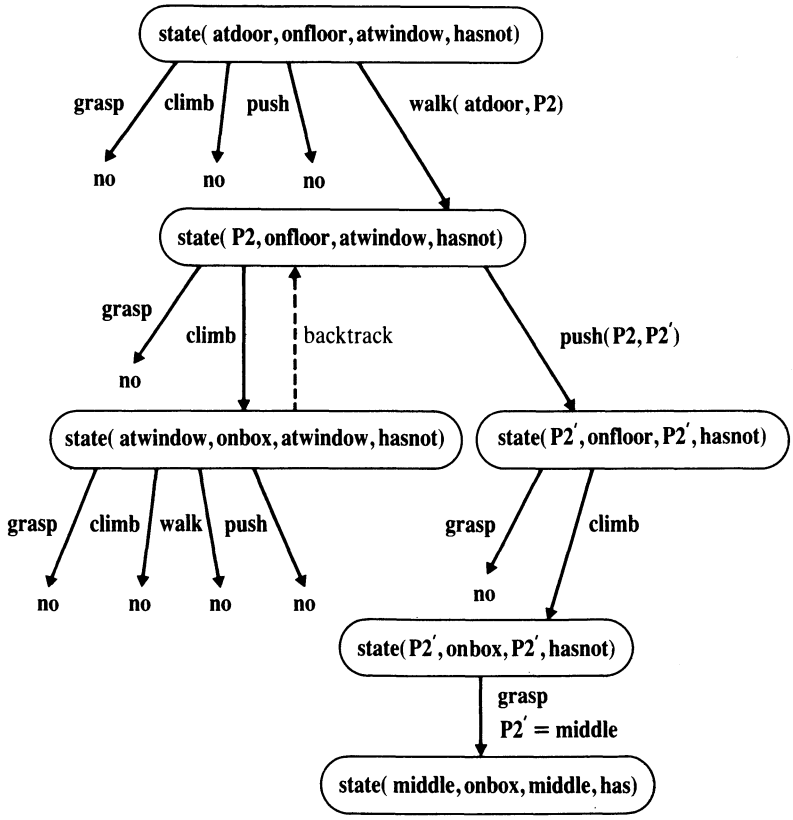

Figure 2.15 The monkey's search for the banana. The search starts at the top node and proceeds downwards, as indicated. Alternative moves are tried in the left-to-right order. Backtracking occurred once only.

without ever touching the box, or aimlessly push the box around. A more thorough investigation will reveal, as shown in the following section, that the ordering of clauses is, in the case of our program, in fact critical.

2.6 Order of clauses and goals

2.6.1 Danger of indefinite looping

Consider the following clause:

```
p :- p.
```

This says that 'p is true if p is true'. This is declaratively perfectly correct, but

procedurally is quite useless. In fact, such a clause can cause problems to Prolog. Consider the question:

```
?- p.
```

Using the clause above, the goal p is replaced by the same goal p; this will be in turn replaced by p, etc. In such a case Prolog will enter an infinite loop not noticing that no progress is being made.

This example is a simple way of getting Prolog to loop indefinitely. However, similar looping could have occurred in some of our previous example programs if we changed the order of clauses, or the order of goals in the clauses. It will be instructive to consider some examples.

In the monkey and banana program, the clauses about the **move** relation were ordered thus: grasp, climb, push, walk (perhaps 'unclimb' should be added for completeness). These clauses say that grasping is possible, climbing is possible, etc. According to the procedural semantics of Prolog, the order of clauses indicates that the monkey prefers grasping to climbing, climbing to pushing, etc. This order of preferences in fact helps the monkey to solve the problem. But what could happen if the order was different? Let us assume that the 'walk' clause appears first. The execution of our original goal of the previous section

```
?- canget( state( atdoor, onfloor, atwindow, hasnot) ).
```

would this time produce the following trace. The first four goal lists (with variables appropriately renamed) are the same as before:

(1) **canget(state(atdoor, onfloor, atwindow, hasnot))**

The second clause of **canget** ('can2') is applied, producing:

(2) **move(state(atdoor, onfloor, atwindow, hasnot), M′, S2′),**
canget(S2′)

By the move **walk(atdoor, P2′)** we get:

(3) **canget(state(P2′, onfloor, atwindow, hasnot))**

Using the clause 'can2' again the goal list becomes:

(4) **move(state(P2′, onfloor, atwindow, hasnot), M″, S2″),**
canget(S2″)

Now the difference occurs. The first clause whose head matches the first goal above is now 'walk' (and not 'climb' as before). The instantiation is

S2″ = state(P2″, onfloor, atwindow, hasnot). Therefore the goal list becomes:

(5) **canget(state(P2″, onfloor, atwindow, hasnot))**

Applying the clause 'can2' we obtain:

(6) **move(state(P2″, onfloor, atwindow, hasnot), M‴, S2‴),**
canget(S2‴)

Again, 'walk' is now tried first, producing:

(7) **canget(state(P2‴, onfloor, atwindow, hasnot))**

Let us now compare the goals (3), (5) and (7). They are the same apart from one variable; this variable is, in turn, P′, P″ and P‴. As we know, the success of a goal does not depend on particular names of variables in the goal. This means that from goal list (3) the execution trace shows no progress. We can see, in fact, that the same two clauses, 'can2' and 'walk', are used repetitively. The monkey walks around without ever trying to use the box. As there is no progress made this will (theoretically) go on for ever: Prolog will not realize that there is no point in continuing along this line.

This example shows Prolog trying to solve a problem in such a way that a solution is never reached, although a solution exists. Such situations are not unusual in Prolog programming. Infinite loops are, also, not unusual in other programming languages. What *is* unusual in comparison with other languages is that the declarative meaning of a Prolog program may be correct, but the program is at the same time procedurally incorrect in that it is not able to produce an answer to a question. In such cases Prolog may not be able to satisfy a goal because it tries to reach an answer by choosing a wrong path.

A natural question to ask at this point is: Can we not make some more substantial change to our program so as to drastically prevent any danger of looping? Or shall we always have to rely just on a suitable ordering of clauses and goals? As it turns out programs, especially large ones, would be too fragile if they just had to rely on some suitable ordering. There are several other methods that preclude infinite loops, and these are much more general and robust than the ordering method itself. These techniques will be used regularly later in the book, especially in those chapters that deal with path finding, problem solving and search.

2.6.2 Program variations through reordering of clauses and goals

Already in the example programs of Chapter 1 there was a latent danger of producing a cycling behaviour. Our program to specify the **predecessor** relation

in Chapter 1 was:

```
predecessor( X, Z) :-
  parent( X, Z).

predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

Let us analyze some variations of this program. All the variations will clearly have the same declarative meaning, but not the same procedural meaning.

```
% Four versions of the predecessor program

% The original version

pred1( X, Z) :-
  parent( X, Z).

pred1( X, Z) :-
  parent( X, Y),
  pred1( Y, Z).

% Variation a: swap clauses of the original version

pred2( X, Z) :-
  parent( X, Y),
  pred2( Y, Z).

pred2( X, Z) :-
  parent( X, Z).

% Variation b: swap goals in second clause of the original version

pred3( X, Z) :-
  parent( X, Z).

pred3( X, Z) :-
  pred3( X, Y),
  parent( Y, Z).

% Variation c: swap goals and clauses of the original version

pred4( X, Z) :-
  pred4( X, Y),
  parent( Y, Z).

pred4( X, Z) :-
  parent( X, Z).
```

Figure 2.16 Four versions of the **predecessor** program.