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.

According to the declarative semantics of Prolog we can, without affecting the declarative meaning, change

(1) the order of clauses in the program, and
(2) the order of goals in the bodies of clauses.

The **predecessor** procedure consists of two clauses, and one of them has two goals in the body. There are, therefore, four variations of this program, all with the same declarative meaning. The four variations are obtained by

(1) swapping both clauses, and
(2) swapping the goals for each order of clauses.

The corresponding four procedures, called **pred1**, **pred2**, **pred3** and **pred4**, are shown in Figure 2.16.

There are important differences in the behaviour of these four declaratively equivalent procedures. To demonstrate these, consider the **parent** relation as shown in Figure 1.1 of Chapter 1. Now, what happens if we ask whether Tom is a predecessor of Pat using the four variations of the **predecessor** relation:

```
?- pred1( tom, pat).

yes

?- pred2( tom, pat).

yes

?- pred3( tom, pat).

yes

?- pred4( tom, pat).
```

In the last case Prolog cannot find the answer. This is manifested on the terminal by a Prolog message such as 'More core needed'.

Figure 1.11 in Chapter 1 showed the trace of **pred1** (in Chapter 1 called **predecessor**) produced for the above question. Figure 2.17 shows the corresponding traces for **pred2**, **pred3** and **pred4**. Figure 2.17(c) clearly shows that **pred4** is hopeless, and Figure 2.17(a) indicates that **pred2** is rather inefficient compared to **pred1**: **pred2** does much more searching and backtracking in the family tree.

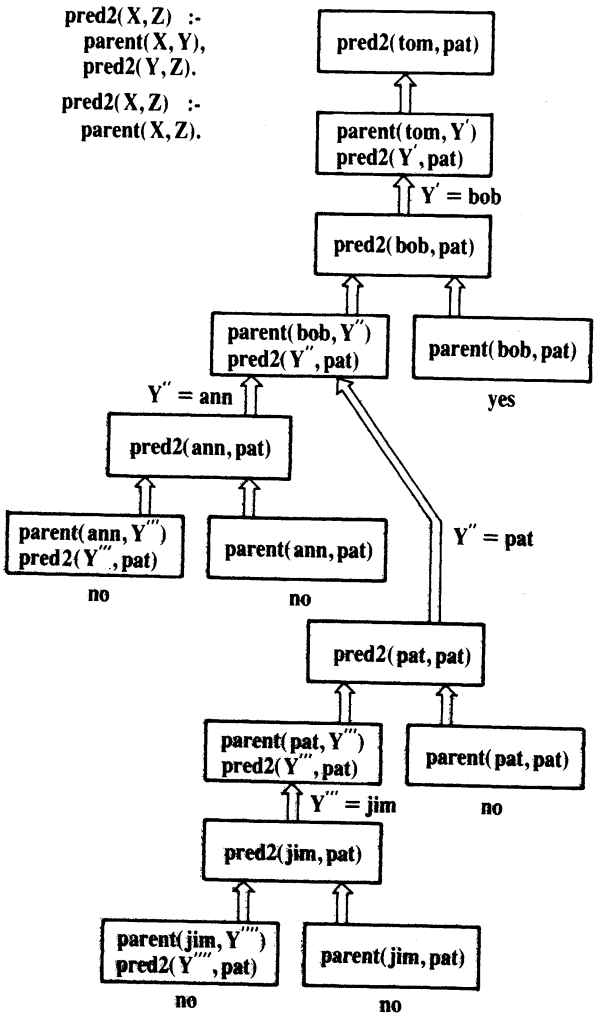
This comparison should remind us of a general practical heuristic in problem solving: it is often useful to try the simplest idea first. In our case, all the versions of the **predecessor** relation are based on two ideas:

- the simpler idea is to check whether the two arguments of the **predecessor** relation satisfy the **parent** relation;

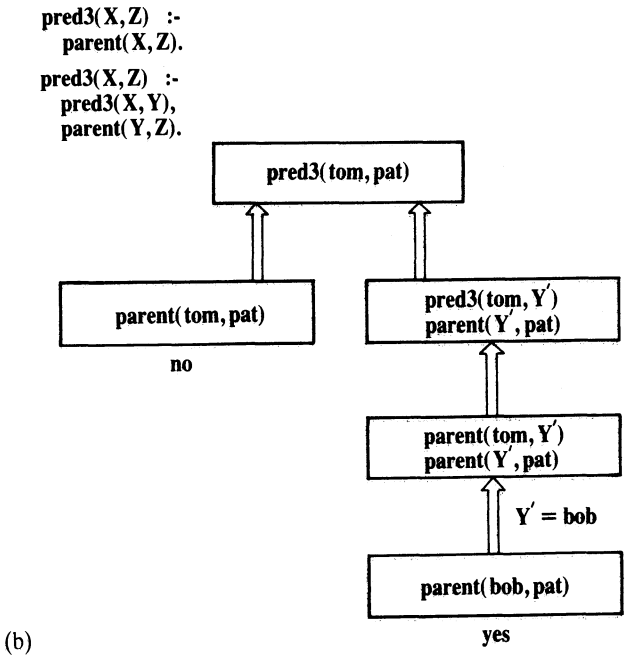
- the more complicated idea is to find somebody 'between' both people (somebody who is related to them by the **parent** and **predecessor** relations).

Of the four variations of the **predecessor** relation, **pred1** does simplest things first. On the contrary, **pred4** always tries complicated things first. **pred2** and **pred3** are in between the two extremes. Even without a detailed study of the execution traces, **pred1** should be clearly preferred merely on the grounds of the rule 'try simple things first'. This rule will be in general a useful guide in programming.

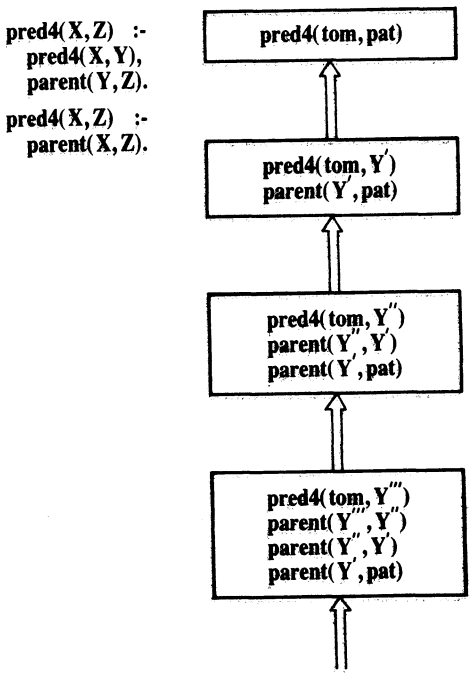
Our four variations of the **predecessor** procedure can be further compared by considering the question: What types of questions can particular variations answer, and what types can they not answer? It turns out that **pred1**



(a)



(b)



(c)

Figure 2.17 The behaviour of three formulations of the **predecessor** relation on the question: Is Tom a predecessor of Pat?

and **pred2** are both able to reach an answer for any type of question about predecessors; **pred4** can never reach an answer; and **pred3** sometimes can and sometimes cannot. One example in which **pred3** fails is:

```
?- pred3( liz, jim).
```

This question again brings the system into an infinite sequence of recursive calls. Thus **pred3** also cannot be considered procedurally correct.

2.6.3 Combining declarative and procedural views

The foregoing section has shown that the order of goals and clauses does matter. Furthermore, there are programs that are declaratively correct, but do not work in practice. Such discrepancies between the declarative and procedural meaning may appear annoying. One may argue: Why not simply forget about the declarative meaning. This argument can be brought to an extreme with a clause such as

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

which is declaratively correct, but is completely useless as a working program.

The reason why we should not forget about the declarative meaning is that progress in programming technology is achieved by moving away from procedural details toward declarative aspects, which are normally easier to formulate and understand. The system itself, not the programmer, should carry the burden of filling in the procedural details. Prolog does help toward this end, although, as we have seen in this section, it only helps partially: it sometimes does work out the procedural details itself properly, and sometimes it does not. The philosophy adopted by many is that it is better to have at least *some* declarative meaning rather than *none* ('none' is the case in most other programming languages). The practical aspect of this view is that it is often rather easy to get a working program once we have a program that is declaratively correct. Consequently, a useful practical approach that often works is to concentrate on the declarative aspects of the problem, then test the resulting program on the computer, and if it fails procedurally try to rearrange the clauses and goals into a right order.

2.7 Remarks on the relation between Prolog and logic

Prolog is related to mathematical logic, so its syntax and meaning can be specified most concisely with references to logic. Prolog is indeed often defined that way. However, such an introduction to Prolog assumes that the reader is familiar with certain concepts of mathematical logic. These concepts are, on the other hand, certainly not necessary for understanding and using Prolog as a

programming tool, which is the aim of this book. For the reader who is especially interested in the relation between Prolog and logic, the following are some basic links to mathematical logic, together with some appropriate references.

Prolog's syntax is that of the *first-order predicate logic* formulas written in the so-called *clause form* (a form in which quantifiers are not explicitly written), and further restricted to *Horn clauses* only (clauses that have at most one positive literal). Clocksin and Mellish (1981) give a Prolog program that transforms a first-order predicate calculus formula into the clause form. The procedural meaning of Prolog is based on the *resolution principle* for mechanical theorem proving introduced by Robinson in his classical paper (1965). Prolog uses a special strategy for resolution theorem proving called SLD. An introduction to the first-order predicate calculus and resolution-based theorem proving can be found in Nilsson 1981. Mathematical questions regarding the properties of Prolog's procedural meaning with respect to logic are analyzed by Lloyd (1984).

Matching in Prolog corresponds to what is called *unification* in logic. However, we avoid the word unification because matching, for efficiency reasons in most Prolog systems, is implemented in a way that does not exactly correspond to unification (see Exercise 2.10). But from the practical point of view this approximation to unification is quite adequate.

Exercise

2.10 What happens if we ask Prolog:

```
?- X = f( X).
```

Should this request for matching succeed or fail? According to the definition of unification in logic this should fail, but what happens according to our definition of matching in Section 2.2? Try to explain why many Prolog implementations answer the question above with:

```
X = f(f(f(f(f(f(f(f(f(f(f(f( ...
```

Summary

So far we have covered a kind of basic Prolog, also called 'pure Prolog'. It is 'pure' because it corresponds closely to formal logic. Extensions whose aim is to tailor the language toward some practical needs will be covered later in the book (Chapters 3, 5, 6, 7). Important points of this chapter are:

- Simple objects in Prolog are *atoms, variables* and *numbers*. Structured objects, or *structures,* are used to represent objects that have several components.