13 LOGIC PROGRAMMING: PROLOG

13.1 HISTORY AND MOTIVATION

We now come to the third major programming paradigm represented by fifth-generation languages: *logic programming*, a step toward *nonprocedural programming.*

Nonprocedural Programming: Saying 'What' Instead of 'How'

Whenever things can be arranged in a series, it is natural to ask if there is a first or last element in the series. For example, once we know that one program can be *smaller* than another and perform the same function, it is natural to ask if there is a *smallest* program to do this function. Similarly, the notion of a faster program leads us to seek a fastest program, and the notion of a *better* program (by any criterion) leads us to ask if there is a *best* program.

One way we judge a programming language is whether it is a *higher-level* language than another language. A language is higher level than another if we can express the same program with less detail. That is, a higher-level language does more automatically. This is valuable because if less is done manually, there is less chance of human error. An alternative way of expressing this is that a higher-level language is *less procedural* than a lower-level language. In other words, in a higher-level language we can concentrate more on *what* is to be done, and less on *how* to do it.

The notion of higher level immediately suggests the notion of highest level. Can there be a highest-level language? Similarly, the notion of a less-procedural language suggests the notion of a least-procedural language or even a *nonprocedural* language. This would be a language in which the programmer stated only what was to be accomplished and left it to the computer to determine how it was to be accomplished. In this exercise we are taking technological extrapolation to the limit (Section 1.4).

Consider the following example. Suppose we wished to sort an array. How would we express this problem in a nonprocedural language? We would have to describe what we meant by sorting an array. For example, we might say that B is a sorting of A if and only if B is a permutation of A and B is ordered. We might also have to describe what we meant by

a permutation of an array and what we meant by an array being ordered. For the latter we might say that B is ordered if $B[i] \leq B[j]$ whenever $i < j$.

It would be the responsibility of the nonprocedural programming system to determine how to create an array B that is an ordered permutation of a given array A. Conceivably it might use any sorting algorithm, including a bubble sort, Shell-sort, or quick-sort. This selection is part of the procedural part of the programming process, which we are assuming is performed automatically.

Local Programming and Automatic Deduction

Nonprocedural programming turns out to be related to an active research area in artificial intelligence: automated theorem proving. The goal of automated theorem proving is the development of programs that can construct formal proofs of propositions stated in a symbolic language. How is this connected with nonprocedural programming? Suppose that for a given array A we wanted to prove the following proposition:

There is an array B that is a sorting of A.

One way to prove this proposition is to exhibit the array B, thus proving that such an array exists. This is exactly the strategy that would be chosen by many automated theorem-proving systems. Thus, a side effect of proving that the array A *can* be sorted is *construction* of the sorted array.

Logic programming makes explicit use of the observation, made in the early 1970s by Pay Hayes, Robert Kowalski, Cordell Green, and others, that applying standard deduction methods often has the same effects as executing a program. Programs are expressed in the form of propositions that assert the existence of the desired result. The theorem prover then must construct the desired result to prove its existence.

In this chapter we investigate logic programming as the third example of a fifth-generation programming technology. We illustrate logic programming by means of the logic-oriented language *Prolog*.

Prolog Uses Symbolic Logic as a Programming Language

In the early 1970s, Alain Colmerauer, Philippe Roussel, and their colleagues in the Groupe d'Intelligence Artificielle (GIA) of the University of Marseilles developed a programming language based on these ideas. It is called Prolog, which stands for "programming in logic." The GIA group developed a Prolog interpreter in Algol-W in 1972, in FORTRAN in 1973, and in Pascal in 1976. Since then Prolog interpreters and compilers have been developed for a number of computer systems, including personal computers.

Prolog has been growing in popularity as a language for artificial intelligence work since the mid-1970s; its proponents have suggested it as the successor to LISP for these applications. Although there are now many variants of logic programming, we will investigate Prolog since it is typical.[1]

---

[1] Prolog has not been standardized. In this chapter we follow the Edinburgh dialect described in Clocksin and Mellish (1984).

13.2 DESIGN: STRUCTURAL ORGANIZATION

A Program Is Structured Like the Statement of a Theorem

Consider the Prolog program in Figure 13.1. It is divided into three major parts, much like the statement of a mathematical theorem. First, we have a series of *clauses* that define the problem domain, which in this case is kinship relations. For example, the first line can be read "*X* is a parent of *Y* if *X* is the father of *Y*." Thus, the first two lines express the idea that *X* is a parent of *Y* if either *X* is the father of *Y* or *X* is the mother of *Y*. The third line can be read, "*X* is a grandparent of *Z* if, for some *Y*, *X* is a parent of *Y* and *Y* is a parent of *X*." The next two lines are a recursive definition of the ancestor relation: "*X* is an ancestor of *Z* if either *X* is a parent of *Z* or, for some *Y*, *X* is a parent of *Y* and *Y* is an ancestor of *Z*." You can see that it is fairly easy to translate common ideas about kinship into Prolog.

The first part of this Prolog program states a number of general principles; the second part states a number of particular facts. Thus, we see that Albert is the father of Jeffrey and that Alice is the mother of Jeffrey. This part of the Prolog program can be thought of as a *database* that defines interrelationships among the atomic individuals (people, in this example).

The parts of the Prolog program discussed so far all make *assertions.* To get the program to compute something, we must ask a *question.* This is the purpose of *goals*, which are

```
parent(X,Y)  :- father(X,Y).
parent(X,Y)  :- mother(X,Y).
grandparent(X,Z)  :- parent(X,Y), parent(Y,Z).
ancestor(X,Z)  :- parent(X,Z).
ancestor(X,Z)  :- parent(X,Y), ancestor(Y,Z).
sibling(X,Y)  :- mother(M,X), mother(M,Y),
                 father(F,X), father(F,Y), X \= Y.
cousin(X,Y)  :- parent(U,X), parent(V,Y), sibling(U,V).

father(albert, jeffrey).
mother(alice, jeffrey).
father(albert, george).
mother(alice, george).
father(john, mary).
mother(sue, mary).
father(george, cindy).
mother(mary, cindy).
father(george, victor).
mother(mary, victor).

:- ancestor(X, cindy), sibling(X, jeffrey).
```

Figure 13.1 Example Prolog Program

usually typed interactively to a Prolog system. We can see an example of a goal in the last line of the Prolog program in Figure 13.1:

```
:- ancestor(X, cindy), sibling(X, jeffrey).
```

This goal asks if there is an individual who is an ancestor of Cindy and a sibling of Jeffrey. In this case, there is such an individual satisfying the goal, so the Prolog system responds

```
X = george
```

We can find out if there are other individuals satisfying the goal by typing a semicolon after the previous answer.[2] In this case, there are no more, so the system responds `no`.

Some goals merely ask if a fact is provable on the basis of the assertions. For example, a Prolog system will respond as shown to these goals:

```
:- grandparent(albert, victor).
yes
:- cousin(alice, john).
no
```

Note that when the Prolog system responds `no` it does not necessarily mean that the goal is false; it means only that it could not be proved on the basis of the general rules and particular facts provided.

Some goals may require several individuals for their satisfaction and may be satisfiable in several ways. For example, if we type the goal

```
:- sibling(A,B).
```

we will receive the answer

```
A = jeffrey, B = george
```

There may be more solutions, so we type a semicolon after each, which yields the following answers:

```
A = jeffrey, B = george;
A = george, B = jeffrey;
A = cindy, B = victor;
A = victor, B = cindy;
no
```

Notice that the system responds `no` when there are no more solutions. Also notice that the system produces all possible ways of satisfying the goal, even if some of them are uninteresting.

■ ***Exercise 13-1:*** Show in detail how each of the above pairs satisfies the goal `:- sibling(A,B).`

[2] This is the convention on many Prolog systems, but it is far from standard.

■ ***Exercise 13-2:*** Show in detail that X = `george` satisfies the goal

```
:- ancestor(X,cindy), sibling(X,jeffrey).
```

■ ***Exercise 13-3:*** Show in detail that the goal `:- grandparent (albert,victor)` is satisfiable.

■ ***Exercise 13-4:*** Show in detail that the goal `:- cousin(alice,john)` is unsatisfiable.

■ ***Exercise 13-5:*** Recall the personnel file example of Chapter 9 (Section 9.1). How might the facts about Don Smith be represented in Prolog?

Proving the Theorem Generates the Answer

The Prolog system may arrive at the answer `X = george` by generating a proof such as the following: The goal is to find an `X` such that `ancestor (X, cindy)` and `sibling (X, jeffrey)`. Now, `sibling (X, jeffrey)` is true if there are individuals `M` and `F` such that `mother (M, jeffrey)`, `father (F, jeffrey)`, `mother (M, X)`, and `father (F, X)`. Now we see that if we set `M = alice` and `F = albert`, we will have `mother (M, jeffrey)` and `father (F, jeffrey)`. But it is now necessary to find an `X` such that `mother (alice, X)` and `father (albert, X)`; this `X` is `george`.

This is not our answer though since there may be other `X`'s that are siblings of Jeffrey; we must also determine if `ancestor (george, cindy)` is true. Now `ancestor (george, cindy)` is true if either `parent (george, cindy)` or there is a `Y` such that `parent (george, Y)` and `ancestor (Y, cindy)`. Further, `parent (george, cindy)` is true if either `father (george, cindy)` or `mother (george, cindy)` is true. But we are given `father (george, cindy)`, so the theorem is true with `X = george`.

■ ***Exercise 13-6:*** Determine if the following goals can be satisfied, and if so, exhibit the individuals that satisfy them:

```
:- sibling(X, cindy).
:- ancestor(albert, victor).
:- ancestor(john, X).
:- ancestor(jeffrey, mary).
:- grandparent(john, X), parent(Y, X), sibling(Y, jeffrey).
:- cousin(X,Y).
```

Clauses Are Constructed from Relationships

A Prolog program is constructed from a number of *clauses*. These clauses have three forms: *hypotheses* (or *facts*) such as

```
mother(mary, cindy).
```

goals, such as

```
:- ancestor(john,X).
```

and *conditions* (or *rules*) such as

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

In *pure* logic programming, the order of these clauses is irrelevant to the logic of the program, although it may be very important in Prolog (which is only *logic-oriented*). Issues relating to the ordering of clauses are discussed in Section 13.4.

The general form of a clause is

⟨head⟩ :– ⟨body⟩.

If ⟨head⟩ is omitted, the clause is considered a goal; if :– ⟨body⟩ is omitted, it is considered a hypothesis. Both the ⟨head⟩ and the ⟨body⟩ are composed of *relationships* (also called *predications* or *literals*), which are applications of a *predicate* (such as `parent`) to one or more *terms*, (such as `john` and `X`):

```
parent(john,X)
```

Relationships are intended to represent properties of and relations among the individuals of the problem domain. Terms may be (1) atoms, such as `john` or `25`; (2) variables, such as `X`; or (3) compound terms, which are described below. Note that many Prolog systems require variables to begin with an uppercase letter, and other terms to begin with a lowercase letter (or a nonletter). This is the convention adopted in this chapter, although there is little standardization among Prolog systems.[3]

Prolog, like many logic programming languages, allows at most one relationship in the ⟨head⟩ of a clause but any number in the ⟨body⟩. This restriction is called the *Horn clause form*; it permits an especially efficient kind of automatic deduction.

■ ***Exercise 13-7:*** To define `brother`, `sister`, `son`, and `daughter`, we need to know the sex of individuals. Add rules and facts to the program of Figure 13.1 to define these predicates. *Hint:* Define `male` and `female`.

Terms Can Be Compound

In all of the examples we have seen so far, the terms have been simple names—either constants such as `cindy` or variables such as `X`. Since terms represent individuals in the problem domain, constant terms represent specific individuals and variable names represent indefinite individuals. Although it is easy to show that any proposition can be expressed by using just constant and variable terms, it is often convenient to allow expressions as terms, which are called *compound terms*. These give us the ability to describe individuals without giving them individual names.

---

[3] Also, some Prolog systems use '←', 'if' or other symbols instead of ':–'.

The value of compound terms will be clearer if we look at an example. Suppose that we wanted to write a Prolog program to do symbolic differentiation. We must express the idea that the derivative of $U + V$ with respect to X is $DU + DV$, where DU is the derivative with respect to X of U and DV is the derivative with respect to X of V. These ideas are easy to write using compound terms:

```
d(X, plus(U,V), plus(DU,DV)) :- d(X,U,DU), d(X,V,DV).
```

The name `plus` is called a *functor* because it acts in many ways like a function. Notice, however, that in the compound term `plus(U,V)` no function is being called; rather this term is a data structure similar to the LISP list `(plus U V)`. A functor is just a *tag* that can be used on data structures (i.e., compound terms).

The above differentiation rule can be made even clearer if we allow (as some Prologs do) the use of infix functors:

```
d(X, U + V, DU + DV) :- d(X, U, DU), d(X, V, DV).
```

Here `DU + DV` does not mean arithmetic addition; it is simply an alternate notation for the compound term `plus(DU, DV)`.

To express this clause without the use of compound terms requires us to introduce a predicate `sum(X, Y, Z)`, which means X + Y = Z. The resulting clause for the derivative of a sum is

```
d(X, W, Z) :- sum(U, V, W), d(X, U, DU), d(X, V, DV),
              sum(DU, DV, Z).
```

This is considerably more difficult to read and also has some subtle difficulties that are discussed later (p. 458). We will see further applications of compound terms in Section 13.3, Data Structures. (See Figure 13.2 for a more complete symbolic differentiation program.)

13.3 DESIGN: DATA STRUCTURES

There Are Few Primitives and No Constructors

We will see in the following subsection that there are essentially no data structure constructors in Prolog. Rather, data structures are defined implicitly by their properties. The same approach could be used to define all data types, including traditionally primitive types such as the integers. For example, the natural numbers and natural number arithmetic could be defined by clauses such as these:

```
sum(succ(X), Y, succ(Z)) :- sum(X, Y, Z).
sum(0, X, X).
dif(X, Y, Z) :- sum(Z, Y, X).
```

The first clause says that the sum of the successor of X plus Y is the successor of Z, *if* the sum of X and Y is Z. The second clause says that the sum of zero and X is X. The third clause

says that the difference of X and Y is Z, *if* the sum of Z and Y is X. In effect, this is a recursive definition of arithmetic. If we type the goal

```
:- sum(succ(succ(0)), succ(succ(succ(0))), A).
```

(meaning $2 + 3 = A$), we will get the answer

```
A = succ(succ(succ(succ(succ(0)))))
```

(meaning 5).

Although this definition of addition and subtraction is correct, it would be very inefficient to use since all computers implement integer arithmetic directly. Hence, all Prolog systems build in certain predicates and functions for basic arithmetic. Unfortunately, as usually implemented, the logical properties are compromised. For example, addition cannot be done "backward" as is done in the above definition of `dif`. This is discussed further in Section 13.4, p. 471.

The small number of built-in data types and operations in Prolog is an example of the Simplicity Principle. The uniform treatment of all data types as predicates and terms is an example of the Regularity Principle.

■ ***Exercise 13-8:*** Explain in detail the satisfaction of the goal corresponding to $2 + 3 = A$.

■ ***Exercise 13-9:*** Write a goal corresponding to $4 - 2 = D$ and show its satisfaction.

Compound Terms Can Represent Data Structures

Prolog does not have a fixed set of data structure constructors as Pascal and Ada do. Rather, Prolog can operate directly with a logical description of the properties of a data structure. LISP-style lists form a good example of this.

Recall that in Chapter 9, Section 9.3, we saw these equations, which form an abstract definition of LISP lists:

`(car (cons X L))` = X
`(cdr (cons X L))` = L
`(cons (car L) (cdr L))` = L, for nonnull L
where L is a list and X is an atom or list.

It turns out that of these only the first two are necessary, since we can prove the third if we assume that every list is either `nil` or the result of a `cons` call (you will prove this in an exercise).

We take the first two equations as a basis for a Prolog definition of lists:

```
car( cons(X, L), X).
cdr( cons(X, L), L).
```

This means that if we take the `car` of `cons(`X, L`)`, we'll get X, and if we take the `cdr` of `cons(`X, L`)`, we'll get L. Notice that `car` and `cdr` are predicates; for example, `car(`L, A`)` means that the `car` of L is A. On the other hand, `cons` is a functor; thus, `cons(a,nil)` is a compound term with components `a` and `nil`.

To complete this implementation of lists, we need to define the predicates `null` and

`list`. When is something a list? There are two ways to get a list in LISP: (1) take the *primitive* list, `nil`, or (2) *construct* a list by applying `cons` to an arbitrary value and a list. In other words, *X* is a list if either it is `nil` or it is a result of `cons`:

```
list(nil).
list(cons(X,L)) :- list(L).
```

Since the only way to get a null list is as a result of `nil`, we write

```
null(nil).
```

Notice that we have not written a rule for `null(`*L*`)` for the case when *L* is nonnull: Prolog will take a list to be nonnull when it cannot prove that it is null (via the fact stated above).

There is something rather unusual going on here: There is no separate representation for lists; lists are represented by the expressions that construct them. Thus, we might type into a LISP system

```
(car (cons '(a b) '(c d)))
  (a b)
```

To accomplish the same thing in Prolog, we would enter the goal:

```
:- car( cons( cons(a, cons(b,nil)),
              cons(c, cons(d,nil)) ), X).
X = cons(a,cons(b,nil))
```

We can see that the answers are equivalent although the LISP notation for constant lists is much clearer. To solve this readability problem, some Prolog systems allow infix operators in compound terms. For example, if '.' were used for `cons`, the above goal and its solution would be[4]

```
:- car( (a.b.nil).(c.d.nil), X )
X = a.b.nil
```

Indeed, most Prolog systems make a special case of the dot operator and allow the abbreviation

`[`X_1`,` X_2 `,...` X_n`]` = X_1`.`X_2`.`$\cdots$`.`X_n`.nil`

In addition, `nil` is usually written `[]`. Thus, we could write

```
:- car( [[a,b],c,d], X).
X = [a,b]
```

as expected. Do not let the syntactic sugar fool you, though: Lists are just compound terms.

■ ***Exercise 13-10:*** Given that every list *L* is either `nil` or the result of a `cons`, prove

`(cons (car` *L*`) (cdr` *L*`))` = *L*, `for nonnull` *L*

from the other two list equations.

---

[4] Note that '.' is right-associative: `a.b.nil` means `a.(b.nil)`.

Components Are Accessed Implicitly

Let's consider an example using these definitions of lists: the append function described in Chapter 9 (Section 9.3). We will define this function as a predicate append such that append(X, Y, Z) means that Z is the result of appending X and Y. The approach is the same as we used in Chapter 9: First handle the primitive (null) list and then handle constructed lists. The clauses are

```
append( [], L, L).
append( X.L, M, X.N) :- append(L, M, N).
```

(We use the '.' abbreviation for cons.) The last clause can be read: "The result of appending the cons of X and L to M is the cons of X and N, where N results from appending L and M."

Let's consider a few steps in the execution of the goal

```
:- append( [a, b], [c, d], Ans).
```

When stripped of syntactic sugar, this is equivalent to

```
:- append( a.b.[], c.d.[], Ans).
```

The Prolog system will attempt to match this goal to the clauses for append. It does this by a process called *unification*, which means finding an assignment of values to the variables that makes the goal identical to the head of one of the clauses. In this case, the match will succeed on the second clause with X = a, L = b.[], M = c.d.[], and Ans = a.N. Hence, these variables are bound to these values (*instantiated*, in Prolog terminology), which leads to the subgoal

```
:- append( b.[], c.d.[], N).
```

This will eventually result in N being bound to b.c.d.[] (you will fill in the steps in an exercise). Then, since Ans = a.N, Ans will be bound to a.b.c.d.[]. The latter, when syntactically sugared on output, will be printed as

```
Ans = [a, b, c, d]
```

Compare the Prolog with the LISP definition of append:

```
(defun append (L M)
   (if (null L)
       M
       (cons (car L) (append (cdr L) M)) ))
```

They are similar in that they are both recursive definitions based on the null list. They are different in that the LISP definition uses car and cdr explicitly to break L into its components, whereas the Prolog definition does not use car and cdr at all. Instead, by matching a pattern such as X.L against a list such as a.b.[], it implicitly breaks the list into its components X = a and L = b.[].[5]

[5] Many Prologs allow [X|L] as an alternate notation for a list whose head is X and whose tail is L.

Frequently in Prolog selector functions are not needed to access the components of composite data structures; instead, the data structures can be matched against the appropriate constructor expressions. This is essentially asking the question: "What two things would I have to `cons` together to get `a.b.[]`?" Thus, "taking apart" is explained in terms of "putting together." Inverse operations are often thought of in this way. For example, we might explain 5 − 3 to a child by asking: "What number do I add to 3 to get 5?" This is the basis for the definition of `dif` that we showed on p. 451.

Although matching is a very simple way to select components of structures, it is not necessarily a very efficient way. Recall that LISP's `car` and `cdr` functions are implemented as simple pointer-following operations and thus are very efficient. In Prolog, component selection is accomplished by pattern matching, which is usually much less efficient. Some Prolog implementors have developed optimizations that make component selection in Prolog almost as fast as in LISP.

- ***Exercise 13-11:*** Fill in the rest of the steps of the `append` example above.
- ***Exercise 13-12:*** Define the `assoc` function (Chapter 9, Section 9.3) in Prolog.
- ***Exercise 13-13:*** Write a predicate `equal` analogous to the LISP `equal` function. That is, it determines if two lists are equal in all their elements and to arbitrary depth. *Hint:* This is *very* simple in Prolog.
- ***Exercise 13-14:*** Write clauses for a predicate `sum_red(L,S)` such that *S* is the sum-reduction of the list *L*. (Use the `sum` predicate already defined and assume numbers are represented in unary.)
- ***Exercise 13-15:*** Write clauses for a predicate `succ_map(L,M)` such that the list *M* is the result of taking the successor of every element of the list *L*.

Complex Structures Can Be Defined Directly

We have seen in the previous section show compound terms can be used to represent data structures. In particular, we have used terms of the form `cons(X, Y)` (or '*X.Y*') to represent lists. This is such a common use of compound terms that many Prolog programmers think of compound terms as records, similar to the records in Pascal and Ada. Thus, `cons(X, Y)` is not thought of as a function application; rather, it is considered a record of type `cons` with the fields *X* and *Y*. And, in fact, this is essentially the way many Prolog systems implement compound terms.

We know from our investigation of LISP that any structure we want can be defined with lists. In Prolog it is not necessary to do this since more complicated structures can be defined directly as terms or predicates. For example, suppose we want to write a program for performing symbolic differentiation of mathematical formulas. In most programming languages, it would be necessary to define data structures representing algebraic expression trees. In Prolog we can simply use compound terms. For example, the expression '$x^2 + bx + c$' could be represented by

```
plus( plus( sup(x, 2), times(b, x) ), c)
```

Many Prolog dialects interpret the arithmetic operators as functors for constructing compound terms, and in these dialects we could write

```
x↑2 + b*x + c
```

Again, we must emphasize that this is just syntactic sugar for a compound term similar to the one shown previously in prefix notation.[6]

Given the infix notation for compound terms, the symbolic differentiation program is easy to write and read (see Figure 13.2). Here, the terms represent themselves; there is no need for a separate data structure for the program to manipulate. The Prolog system essentially generates records of the form `plus(X, Y)`, `times(X, Y)`, and so on, but this is not a concern of the programmer. Thus, the Automation Principle is being obeyed.

■ ***Exercise 13-16:*** Trace the execution of the following goal:

```
:- d(x, x↑2 + b*x + c, Ans).
Ans = 2 * x↑(2-1) * 1 + (0 * x + b * 1) + 0
```

Notice that the expression is not simplified by these rules.

■ ***Exercise 13-17:*** Trace the execution of the goal

```
:- d(x, x + y, A).
```

■ ***Exercise 13-18:*** Trace the execution of the goal

```
:- d(t, 2*t + x, Rate).
```

■ ***Exercise 13-19:*** Trace the execution of the goal

```
:- d(y, y↑n / (x + y), Dy).
```

■ ***Exercise 13-20:*** Write a differentiation rule for negations. Assume "negative `U`" is written '`~U`'.

■ ***Exercise 13-21:*** Write a differentiation rule for the special case `C*U`, where `C` is atomic and `C` ≠ `X`.

```
d(X, U + V, DU + DV) :- d(X,U,DU), d(X,V,DV).
d(X, U - V, DU - DV) :- d(X,U,DU), d(X,V,DV).
d(X, U*V, DU*V + U*DV) :- d(X,U,DU), d(X,V,DV).
d(X, U/V, (DU*V - U*DV)/V↑2) :- d(X,U,DU), d(X,V,DV).
d(X, U↑C, C * U↑(C - 1) * DU) :- d(X,U,DU), atomic(C), C \= X.
d(X, X, 1).
d(X, C, 0) :- atomic(C), C \= X.
```

Figure 13.2 Symbolic Differentiation in Prolog

[6] Some Prolog dialects permit programmers to define their own prefix, infix, and postfix functor symbols.

Predicates Can Represent Structures Directly

We have said before that compound terms can often be replaced by predicates. For example, instead of having terms `plus(X,Y)`, `times(X,Y)`, etc. representing arithmetic expressions, we could have predicates `sum(X,Y,Z)`, `prod(X,Y,Z)`, etc. Let's consider the consequences of this in more detail.

Consider the compound term `(x + y) * (y + 1)` representing an arithmetic expression. This is a description of a data structure—a tree—as can be seen by writing it in prefix notation:

```
times(plus(x, y), plus(y, 1))
```

Thus, it is exactly analogous to a LISP structure such as

```
(times (plus x y) (plus y 1))
```

How can we express these same relationships in terms of predicates? In effect we need to describe a tree in terms of relationships among its nodes. We can get a hint of how to do this by recalling the kinship program in Figure 13.1. There we used the predicates `father` and `mother` to describe a *family tree.* We can describe the relationships in an *arithmetic expression tree* by predicates such as `sum` and `prod`.[7] For example, let `sum(X,Y,Z)` mean that the sum of *X* and *Y* is *Z*. Then the expression `(x + y) * (y + 1)` is described by the three facts

```
sum(x,y,t1).
sum(y,1,t2).
prod(t1,t2,t3).
```

since they say that the sum of `x` and `y` is `t1`, the sum of `y` and `1` is `t2`, and the product of `t1` and `t2` is `t3`. We can see the first difficulty with representing data structures as predicates: the loss of readability.

Suppose we write our differentiation program (Figure 13.2) so that it works with expressions represented as predicates. The sum and product rules would look like this:

```
d(X,W,Z)  :-  sum(U,V,W),  d(X,U,DU),  d(X,V,DV),  sum(DU,DV,Z).
d(X,W,Z)  :-  prod(U,V,W),  d(X,U,DU),  d(X,V,DV),
              prod(DU,V,A),  prod(U,DV,B),  sum(A,B,Z).
d(X,X,1).
d(X,C,0)  :-  atomic(C),  C \= X.
```

The sum rule does not look bad; it is quite readable: "The derivative with respect to `X` of `W` is `Z` if: `W` is the sum of `U` and `V`, and the derivative with respect to `X` of `U` is `DU`, and . . . ." The product rule is not so good though. The variables `A` and `B` get in the way; the only reason they are there is that we have to be able to name the intermediate nodes in the expression tree. So the predicate representation also makes the *rules* less readable. There are more subtle problems, however.

---

[7] Here we use `sum` to represent an algebraic relationship between terms. There is no relationship between this use of the predicate `sum` and its use to perform unary addition shown earlier.

Suppose we have the following fact in our database:

```
sum(x,1,z).
```

This says that the sum of x and 1 is z. Now suppose we enter the goal

```
:- d(x,z,D).
```

This asks the system to find a D such that the derivative with respect to x of z is D. Since z is the sum of x and 1, we expect D to be the sum of 1 and 0.

Will the goal be satisfied? It unifies with the head of the sum rule by the assignments X = x, W = z, Z = D, which leads to the subgoals

```
:- sum(U,V,z), d(x,U,DU), d(x,V,DV), sum(DU,DV,D).
```

The first subgoal is satisfied by U = x and V = 1, since we have in our database the fact sum(x,1,z). Hence, we get the subgoals

```
:- d(x,x,DU), d(x,1,DV), sum(DU,DV,D).
```

Now the first two subgoals here are satisfied by DU = 1 and DV = 0, so only one subgoal is left:

```
:- sum(1,0,D).
```

It seems that our program is done, but in fact this subgoal will *fail*, since there is no fact (or rule) of the form sum(1,0,D) in the database! Of course, if we had had the foresight to make an assertion such as

```
sum(1,0,a).
```

then the system would report the correct[8] answer D = a, but it is unreasonable to expect such foresight. Had the answer been more complicated, the situation would be even worse, since we would have had to anticipate the entire expression tree of the answer, else the system would not find it.

Prolog Models a Closed World

What is the source of this unintuitive behavior? The deductive rules of Prolog are based on a "closed world" assumption. This means that so far as Prolog is concerned, all that is true about the world is what can be proved on the basis of the facts and rules in its database. In many applications this is a reasonable assumption. For example, in our kinship program it is reasonable to assume that there are no people and no relationships among people other than those represented (explicitly or implicitly) in the database. It makes sense to assume that there is no object having a given property if one cannot be found in the database. The closed-world assumption is very reasonable in *object-oriented* kinds of applications, that is, in applications in which the objects and relationships in the computer model those in a real or imagined world.

---

[8] The answer is correct because the fact sum(1,0,a) asserts that 'a' is the sum of 1 and 0.

Where the closed-world assumption does not work well is in mathematical problems. In mathematics an object is generally take to exist if its existence does not contradict the axioms. Mathematically, it is natural to assume that an expression of the form '1 + 0' exists whenever one is needed. Its existence does not have to be explicitly asserted or provable from the hypotheses. Of course, we could explicitly axiomatize the existence of all arithmetic expressions we might need, but this seems unnatural and more trouble than it is worth.

In summary, it is possible to replace compound terms with predicates connecting their parts, but the closed-world assumption makes it hard to guarantee the right objects are there when we need them. Perhaps the conclusion we should draw is that predicates should be used for expressing object-oriented relationships and compound terms should be used for mathematical relationships.

Data Structures Are Inherently Abstract

We have discussed *abstract data types* in many places in this book: An abstract data type is a set of data values together with the operations defined on those data values. The primary value of abstract data types is that they are implementation independent, which makes them easier to maintain and more portable. Notice that in Prolog *all data types are inherently abstract.* This is so because the only way to describe a data type in Prolog is to give an abstract, algebraic description of the properties of its operations. The Prolog system decides on the proper representation of the data (which, however, might be quite inefficient compared to a custom-designed representation). We can view this as a nonprocedural approach to data structures.

A result of Prolog's approach to data types is a strict adherence to the Security Principle since the only operations defined on data values are those that the programmer has specifically described.

Should Infinite Terms Be Permitted?

Suppose we have the fact `equal(X,X)` in the database and that we type in the goal `:- equal(f(Y),Y)`. What will happen? The only way to satisfy this goal is to unify `equal(f(Y),Y)` with `equal(X,X)`. That is, we have to find bindings of `X` and `Y` that make these two relationships equal. The only way to do this is to bind `Y = f(Y)`, which makes the goal `:- equal(f(Y),f(Y))`, which obviously unifies with the fact `equal(X,X)` by the assignment `X = f(Y)`. Since the goal has been successfully satisfied, the Prolog system will print out the resulting bindings. We will see something like this:

```
Y = f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(
    f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(
    ....
```

The system will continue printing `'f('` until we stop it. What has happened?

The system is attempting to print out the value of `Y`. But the value of `Y` is `f(Y)`. So to print this it must print `'f('`, followed by the value of `Y`, followed by `')'`. But the value

of `Y` is `f(Y)`, and so on and so on. The variable `Y` is in effect bound to an infinite term, which we can see by substituting `f(Y)` for `Y` every time it occurs:

```
Y = f(Y)
  = f(f(Y))
  = f(f(f(Y)))
  :
  :
  = f(f(f(f(f(f(f(f(f(f(f(  ...  ))))))))))))
```

What are we to make of infinite terms? In the context of logic they are illegitimate, and if Prolog implemented the resolution algorithm correctly, it would not permit them. However, to prevent them would require checking to make sure that a variable is not bound to a term containing itself. This "occurs check" is relatively expensive; it takes quadratic time. Since unification is otherwise linear, implementing the check could significantly slow down the execution of Prolog programs. Hence, most Prologs do not make the "occurs check" and thus will behave as illustrated above. This is a lack of transparency.

Can this liability be turned into an asset? Or, as some would say, can this bug be turned into a feature? A self-embedding term is not necessarily meaningless. If f is a function, then the equation $Y = f(Y)$ asserts that Y is a *fixed point* of the function f. Now, some functions have fixed points and some do not. For example, the sine function has a fixed point at zero ($0 = \sin 0$); the logarithm function does not have a fixed point. Hence, in some contexts, the equation $Y = f(Y)$ may be perfectly meaningful.

But, you may object, an infinite term is an infinite data structure. Even if it makes sense mathematically, it cannot have any use in a program. As the example shows, the computer cannot even print it out. In fact, infinite terms such as these *are* representable on computers—by circularly linked lists. If we take a list $Y = (f\ X)$ and store into the left (`car`) component of its second cell a pointer back to the head of the list, then we will have a list satisfying the equation $Y = (f\ Y)$; see Figure 13.3. If we traverse this list by following the pointers, it will look for all the world like an infinite data structure, but it is represented in a finite amount of memory. In the case of the Prolog binding `Y = f(Y)`, the circular link is through the symbol table that binds `Y` to its value.

We have previously noted (Chapter 11, Section 11.2) that there is no agreement on whether circular structures should be permitted. There are certainly difficulties, both theoretical and practical (e.g., how do you print them?). On the other hand, Colmerauer and other computer scientists have argued that circular structures are the natural way to represent information in a variety of domains, including language processing and optimization.

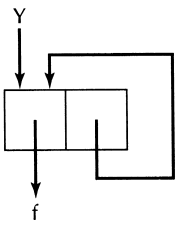

Figure 13.3 An Infinite Term as a Cyclic Structure

■ ***Exercise 13-22*:*** Discuss the usefulness of infinite terms. Show some situations in which they would be convenient. How would you handle these situations if infinite terms were prohibited? Should Prolog permit them or prohibit them?

13.4 DESIGN: CONTROL STRUCTURES

Logic Programming Separates Logic and Control

Logic programs do not have control structures in the usual sense.[9] In a conventional programming language control structures determine the order in which the actions comprising a program take place. This order is essential to the correctness of the program; indeed, it is unusual when the order of statements can be changed without altering the meaning of the program. Thus, the *logic* of a program is intimately related to its *control.*

Logic programming effects a much greater separation of logic and control. The order in which the clauses of a program are written has no effect on the meaning of the program. In other words, the logic of the program is determined by the logical interrelationships of the clauses, not by their physical relationship.

Control affects the order in which actions occur in time. The only actions that occur in the execution of a logic program are the generation and unification of subgoals. As we will see, this order can have a major effect on the efficiency of the program. Hence, in logic programming, issues of control affect only the performance of programs, not their meaning or correctness.

The separation of logic and control is an important application of the Orthogonality Principle. It means that a program can be developed in two distinct phases: logical analysis and control analysis. *Logical analysis* is entirely concerned with the correctness of the program, in other words, with producing the correct clauses to characterize the answer. *Control analysis* is entirely concerned with the efficiency of the program. It may be performed entirely by the logic programming system, or it may be partially under the control of the programmer. For example, the order in which the programmer writes clauses may affect the order in which subgoals are generated and hence the program's performance. Other systems allow the programmer to give the interpreter hints, such as that certain predicates are functions. In every case, there is a clear separation of logic—what the program does—from control—how it does it.

Top-Down Control Is Like Recursion

There are two principal ways in which subgoals can be generated: top-down and bottom-up. In top-down control, we start from the goal and attempt to reach the hypotheses; in bottom-up control, we start with the hypotheses and attempt to reach the goal.

---

[9] Note that in this section the term "logic program" refers to *pure* logic programs. We will see later that Prolog programs are not *pure* logic programs. Material in this section is based on the work of Robert Kowalski (1979).

To illustrate the difference between these, we will use a simple example: the generation of Fibonacci numbers. The Fibonacci numbers are the series

1, 1, 2, 3, 5, 8, 13, 21, ...

in which each number is the sum of the two previous numbers. Hence, if F_n represents the nth Fibonacci number, then

$F_0 = 1$
$F_1 = 1$
$F_n = F_{n-1} + F_{n-2}, \quad$ for $n > 1$

This can easily be expressed in a logic program in which the relationship `fib(N, F)` means that the Nth fibonacci number is F:[10]

```
fib(0, 1).
fib(1, 1).
fib(N, F) :- N = M + 1, M = K + 1, fib(M, G), fib(K, H),
             F = G + H, N > 1.
```

That is, the Nth Fibonacci number is F if there are natural numbers K and M, with $N = M + 1 > 1$, $M = K + 1$, such that the Mth Fibonacci number is G, the Kth Fibonacci number is H, and $F = G + H$.

Let's trace the top-down execution of the goal

```
:- fib(2, F).
```

To satisfy this goal, we must find a clause with which it can be *unified*, that is, a clause that can be made identical to the goal by some assignment of values to the variables of each. No assignment of values to `F` will make `fib(2,F)` equal to either of the first two clauses, so the only candidate for unification is the third clause. Clearly, the assignment `N = 2` will make `fib(N,F)` equal to `fib(2,F)`. Hence, to achieve the goal `fib(2,F)`, we must satisfy the subgoals

```
:- 2 = M + 1, M = K + 1, fib(M,G), fib(K,H), F = G + H, 2 > 1.
```

The subgoal `2 > 1` is satisfied, so it can be discarded, leaving

```
:- 2 = M + 1, M = K + 1, fib(M,G), fib(K,H), F = G + H.
```

The assignment `M = 1` allows the satisfaction of the subgoal `2 = M + 1`, so we get a new set of subgoals:

```
:- 1 = K + 1, fib(1,G), fib(K,H), F = G + H.
```

(Notice that the assignment `M = 1` has been made in all the subgoals.) Similarly, the assignment `K = 0` leads to the subgoals

```
:- fib(1,G), fib(0,H), F = G + H.
```

---

[10] Throughout the discussion of the `fib` program, we use equations of the form $Z = X + Y$ as abbreviations for relationships of the form `sum(X,Y,Z)`. The `sum` predicate was defined on p. 451.

We have now reached the hypotheses since the assignments `G = 1` and `H = 1` allow `fib(1,G)` to unify with `fib(1,1)` and `fib(0,H)` to unify with `fib(0,1)`. This leaves a single subgoal:

```
:- F = 1 + 1.
```

which is satisfied by the assignment `F = 2`, which is the answer sought.

Notice that the top-down execution of a logic program is very similar to the execution of a recursive procedure. The order in which the steps are taken is essentially the same as in this Pascal function:

```
function fib (N: integer): integer;
begin
   if N = 0 or N = 1 then fib := 1;
   else fib := fib(N - 1) + fib(N - 2)
end;
```

Notice that simple recursion is not a very efficient way of computing Fibonacci numbers. Suppose our goal had been

```
:- fib(10,F).
```

The top-down execution of this program would involve setting up the subgoals `fib(9,G)` and `fib(8,H)`. Suppose we began by attempting to satisfy `fib(8,H)`; this would require computing the eighth Fibonacci number. After this is accomplished, we would attempt to satisfy `fib(9,G)`, which would lead to the subgoals `fib(8,G')` and `fib(7,H')`. Now, if we followed the top-down discipline naively, satisfying the goal `fib(8,G')` would require us to recompute the eighth Fibonacci number again! This recomputation would take place at each stage of the recursion, which results in a very inefficient algorithm (an algorithm whose execution time is an exponential function of *N*). For this reason, some logic programming systems remember subgoals they have already satisfied so that they will not duplicate their satisfaction.

- ***Exercise 13-23:*** Prove that the naive top-down computation of `fib` has exponential time complexity.
- ***Exercise 13-24:*** Trace (by hand) the top-down execution of the goal

  ```
  :- fib(3,A).
  ```

- ***Exercise 13-25:*** Show the top-down satisfaction of the goal

  ```
  :- grandparent(albert,victor).
  ```

- ***Exercise 13-26:*** Show the top-down satisfaction of the goal

  ```
  :- ancestor(X,cindy), sibling(X,jeffrey).
  ```

Bottom-Up Control Is Like Iteration

A different strategy for controlling the generation of subgoals is bottom-up, in which we attempt to reach the goal from the hypotheses. Consider the bottom-up satisfaction of the goal

```
:- fib(3,F).
```

We begin with the hypotheses, which are

```
fib(0,1), fib(1,1).
```

(For clarity, we will omit the hypotheses about integer arithmetic, which are built into the system.) These two hypotheses can be unified with the right-hand side of

```
fib(N,F) :- N = M + 1, M = K + 1, fib(M,G), fib(K,H),
            F = G + H, N > 1.
```

by the value assignments `K = 0, M = 1, N = 2, G = H = 1`, and `F = 2`. This allows us to conclude `fib(2,2)`, which, combined with the existing hypotheses, yields

```
fib(0,1), fib(1,1), fib(2,2).
```

The last two of these can again be unified with the right-hand side of the above-mentioned clause by the assignments `K = 1, M = 2, N = 3, G = 2, H = 1, F = 3` to yield the hypotheses

```
fib(0,1), fib(1,1), fib(2,2), fib(3,3).
```

The last of these hypotheses can be unified with the goal `fib(3,F)` by the assignment `F = 3`, which is the answer sought.

Notice that the order of operations resulting from a bottom-up execution is the same as in an iterative program. The bottom-up execution of this program is analogous to that of the Pascal function:

```
function fib (N: integer): integer;
  var n, Fn, Fm, Fk: integer;
begin
  Fn := Fm := Fk := 1;
  n := 1;
  while n < N do
    begin
      Fn := Fm + Fk;
      Fk := Fm;
      Fm := Fn;
      n  := n + 1
    end;
  fib := Fn;
end;
```

Notice that the bottom-up execution of the logic program is much more efficient than its naive top-down execution since the bottom-up execution does not recompute Fibonacci numbers needlessly. In fact, the time required for the bottom-up execution of this program is a linear function of its input (`N`).

Pure top-down and pure bottom-up are not the only ways of executing logic programs. We have already indicated a modification of top-down that remembers already satisfied subgoals. There are also various mixtures of top-down and bottom-up that work from both the goals and the hypotheses and attempt to meet in the middle. The details of these approaches

go beyond the scope of this book; the interested reader can find them discussed in books and courses that deal with artificial intelligence and logic programming.

■ ***Exercise 13-27:*** Show a bottom-up satisfaction of the goal

```
:- grandparent(albert,victor).
```

■ ***Exercise 13-28:*** Show a bottom-up satisfaction of the goal

```
:- ancestor(X,cindy), sibling(X,jeffrey).
```

Logic Programs Can Be Interpreted Procedurally

There is another way of looking at logic programs that makes them easier to compare with programs in conventional languages. In this interpretation, clauses are viewed as procedure definitions and relationships are viewed as procedure invocations. Consider a clause such as

```
fib(N,F) :- N = M + 1, M = K + 1, fib(M,G), fib(K,H),
            F = G + H, N > 1.
```

The head (left-hand side) of the clause is analogous to the head of a procedure declaration; it defines a template for invoking the procedure. The body (right-hand side) of the clause is analogous to the body of a procedure declaration; it is composed of a series of procedure calls. Multiple clauses that define the same procedure are analogous to the branches of a conditional in a conventional definition of a procedure (for example, compare the LISP and Prolog programs for `append` on p. 454). A goal is just the procedure call that starts a program going, and a hypothesis is just a procedure that returns without invoking any other procedures.

Consider the procedural interpretation of the goal

```
:- fib(3,F).
```

We are invoking the procedure `fib` with the input parameter `N = 3` and the output parameter `F` unbound. The first subgoal in the body of `fib`, `N = M + 1`, unifies `N` with `M + 1`, which results in `M` being bound to `2`. The second subgoal, `M = K + 1`, binds `K` to `1`. Next we have two recursive invocations of `fib`: `fib(2,G)` and `fib(1,H)`. The second of these is an invocation of the hypothesis

```
fib(1,1).
```

which is really just an abbreviation for

```
fib(1,1) :-.
```

Since this has no procedure invocations on the right, it immediately returns `H = 1`. When `fib(2, G)` returns its result `G = 2`, we will be able to return the result of the program, which is `F = 3`.

Procedure Invocations Can Be Executed in Any Order

One difference between the procedural interpretation of logic programs and procedures in other languages is immediately apparent: The statements in the body of a logic procedure

need not be executed in any particular order. That is, `fib(2,G)` can be executed before `fib(1,H)`, or vice versa. As we have noted before, this is very different from conventional languages in which the order in which things are done is essential to the meaning of the program. In a logic program, the procedure calls can be executed in any order or even concurrently. This makes logic programming languages a potential way to program multiprocessing and highly parallel computers.[11]

Backtracking Is Necessary and Frequent

In the examples that we have discussed, we have assumed that when there were several clauses defining a procedure, the correct clause was always selected by a procedure invocation. For example, the invocation `fib(1, G)` matches the head of both of these clauses:

```
fib(N,F) :- N = M + 1, M = K + 1, fib(M,G), fib(K,H),
            F = G + H, N > 1.
fib(1,1).
```

Suppose that we select the first clause for execution; this will set up the subgoals

```
:- 1 = M + 1, M = K + 1, fib(M,G), fib(K,H), F = G + H, 1 > 1.
```

Notice that the last subgoal, `1 > 1`, cannot be satisfied; therefore, there is no way to satisfy this set of subgoals. In other words, the invocation `fib(1,G)` fails when we select the first clause. Hence, the interpreter must *backtrack* to the last point where it made a choice [in this case, selecting the first of the two clauses that match `fib(1,G)`] and make a different choice. In this example, the only other choice is the hypothesis `fib(1,1)`, which will allow the goal to be satisfied.

It may be that the system tries all of the alternatives available for a procedure call and that it cannot satisfy the subgoal with any of them. If this occurs, the system must backtrack further, to an earlier choice, and try additional alternatives there. If the system runs out of all alternatives, the goal is unsatisfiable, which must be reported to the user.

The result of this process is that a logic programming system spends a lot of its time backtracking. In most languages and systems, backtracking is considered an unusual event that is usually connected with error recovery. Therefore, backtracking is commonly considered expensive. Since in logic programs backtracking is the rule rather than the exception, much of the challenge of the implementation of logic programming languages is the development of more efficient backtracking mechanisms. These mechanisms are beyond the scope of this book.

Input-Output Parameters Are Not Distinguished

There is another major difference between procedures in logic programming languages and those in other languages. We have seen that a goal such as

```
:- fib(3,F).
```

[11] Again, note that we are talking about *pure* logic programs; we discuss later the extent to which these observations apply to Prolog.

will result in binding `F = 3`; this tells us that the third Fibonacci number is 3. The system proves constructively that there is an `F` such that `fib(3,F)`. Consider instead this goal:

```
:- fib(N,3).
```

which asks the system to prove constructively that there is a number `N` such that `fib(N,3)`. This seems to be a request for the number *N* such that the *N*th Fibonacci number is 3. Will this work? Let's trace its execution.

The goal `fib(N,3)` sets up these subgoals:

```
:- N = M + 1, M = K + 1, fib(M,G), fib(K,H), 3 = G + H, N > 1.
```

To determine if these subgoals can be satisfied, the interpreter must try various value assignments to the variables. For example, it might try in order

```
G = 0, H = 0
G = 0, H = 1
G = 1, H = 0
G = 1, H = 1
G = 0, H = 2
    :
```

and so on, until it reaches `G = 1` and `H = 2`, since this is the first assignment that satisfies `3 = G + H`. This value assignment leads to the subgoals

```
:- N = M + 1, M = K + 1, fib(M,1), fib(K,2), N > 1.
```

The subgoal `fib(M,1)` unifies with the hypothesis `fib(1, 1)` by the assignment `M = 1`, which leads to

```
:- N = 1 + 1, 1 = K + 1, fib(K,2), N > 1.
```

Only the assignments `N = 2, K = 0` will allow satisfaction of `N = 1 + 1` and `1 = K + 1`, so we get the subgoal

```
:- fib(0,2).
```

This unifies with the head of

```
fib(N,F) :- N = M + 1, M = K + 1, fib(M,G), fib(K,H),
            F = G + H, N > 1.
```

yielding the subgoals:

```
:- 0 = M + 1, M = K + 1, fib(M,G), fib(K,H), 2 = G + H, 0 > 1.
```

Since the last subgoal (`0 > 1`) is not satisfiable, we must backtrack to the last choice and seek a new alternative.

In this case, the only choice was the selection of the values `G = 1, H = 2` to satisfy `F = G + H`. Suppose instead we continue enumerating values until we reach `G = 2, H = 1`; this yields the subgoals

```
:- N = M + 1, M = K + 1, fib(M,2), fib(K,1), N > 1.
```

But `fib(K,1)` unifies with the fact `fib(1,1)` by the assignment `K = 1`, so we get the subgoals

```
:- N = M + 1, M = 1 + 1, fib(M,2), N > 1.
```

This leads to the value assignments `M = 2, N = 3` and the subgoal

```
:- fib(2,2).
```

Since the second Fibonacci number is in fact 2, we can see that this subgoal is satisfiable (you will provide the details in an exercise). Therefore, the value assignment `N = 3` will be returned as the answer to

```
:- fib(N,3).
N = 3
```

Notice the remarkable thing that logic programming permits: Neither parameter of `fib` is inherently an input or an output. If either is supplied as an input, then the other can be computed as an output. It is the use of `fib` in each particular context that determines the function of its parameters in that context. One way of looking at this is that a logic program can be run either forward or backward as needed!

We summarize: In the goal `:- fib(N,F)`, if N is supplied as an input, then F can be computed as an output; if F is supplied as an input, then N can be computed as an output. Also, we have seen in the satisfaction of goals such as `fib(2,2)` that *both* F and N can be supplied as inputs. What happens if *neither* N nor F is supplied as an input? In this case, we are asking the system to prove constructively that there are N and F such that `fib(N,F)` is true. Of course, there are, and the system immediately finds `N = 0, F = 1`:

```
:- fib(N,F).
N = 0, F = 1
```

As usual we can type a semicolon to request the system to search for other solutions, and it finds them:

```
:- fib(N,F).
N = 0, F = 1;
N = 1, F = 1;
N = 2, F = 2;
N = 3, F = 3;
N = 4, F = 5;
N = 5, F = 8
yes
```

The response `yes` indicates that there may be additional solutions. Hence, our logic program `fib` can be used as a way of enumerating the Fibonacci numbers.

Such enumeration could be required, for example, during backtracking. Suppose we wanted to search for an N such that $N + F_N = 13$. We could enter the goal

```
:- fib(N,F), N + F = 13.
N = 5, F = 8
```

The subgoal `fib(N,F)` is initially satisfied by `N = 0, F = 1`, but this assignment does not satisfy `N + F = 13`. Therefore, we backtrack to `fib(N,F)` to find another (N,F) pair satisfying this relationship. Eventually this backtracking produces the pair `N = 5, F = 8`, which satisfies the goal.

- ***Exercise 13-29:*** Show that the goal `:- fib(2,2)` can be satisfied.
- ***Exercise 13-30:*** Trace in detail the satisfaction of the goal `:- fib(N,F)` from `N = 0` through `N = 3`.
- ***Exercise 13-31:*** Trace in detail the satisfaction of

```
:- fib(N,F), N + F = 9.
```

- ***Exercise 13-32:*** Explain what happens when we enter the goal

```
:- fib(N,F), N + F = 10.
```

Suggest ways of avoiding the difficulty.

Prolog Uses Depth-First Search

In this section we have been discussing logic programming's separation of logic and control. With this separation the programmer worries about the logical relationships in the program, while the system worries about implementing a proper control strategy to execute the program. Recall that avoiding the procedural "how to" issues is the whole point of nonprocedural programming. We will see that this separation holds only for *pure* logic programming languages, that is, for logic programming languages that implement some complete deductive algorithm (such as J. Alan Robinson's *resolution algorithm*). Unfortunately, it does *not* hold for Prolog.

Prolog abandons the separation of logic and control by *specifying* the control regime to be used, rather than leaving it up to the Prolog system. The Prolog language is *defined* to use a depth-first search strategy. This design decision has many consequences, both good and bad, that we discuss in the remainder of this section. Before investigating these, however, we must say exactly what we mean by *depth-first search.*

The easiest way to understand Prolog's control strategy is to remember that it does everything in a specific order: first to last (i.e., top to bottom, left to right). Thus, if we have the subgoals

```
:- ancestor(X,cindy), sibling(X,jeffrey).
```

Prolog will attempt to satisfy them in the order written. Hence, the first subgoal is `ancestor(X,cindy)`. To satisfy this it will try the clauses for `ancestor` *in the order in which they were entered into the database.* Thus, it starts with

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

Replacing `ancestor(X,cindy)` by `parent(X,cindy)` leads to the subgoals

```
:- parent(X,cindy), sibling(X,jeffrey).
```

Notice that the new subgoal, `parent(X,cindy)` is put at the *beginning* of the list of subgoals; this leads to a *depth-first* search order.

Again, Prolog starts with the first subgoal on the list of subgoals; in this case, it is `parent(X,cindy)`. There are two clauses for `parent` and the first, `parent(X,Y) :- father(X,Y)`, leads to the subgoals

```
:- father(X,cindy), sibling(X,jeffrey).
```

The subgoal `father(X,cindy)` unifies with the fact `father(george,cindy)` by the assignment `X = george`, so we get the subgoal

```
:- sibling(george,jeffrey).
```

And so on. A complete trace (such as produced by a Prolog system) is shown in Figure 13.4. (Note that `Exit` means successful satisfaction of a goal and `Fail` means unsuccessful satisfaction of a goal. This is common Prolog terminology.)

If there had been a failure in the satisfaction of `ancestor(X,cindy)`, the system would have backtracked, trying additional alternatives in order. It would not move on to the subgoal `sibling(X,jeffrey)` until an `X` satisfying the first subgoal had been found.

```
:- ancestor(X,cindy), sibling(X,jeffrey).
Invocation  Depth  Event  Subgoal
    (1)       1    Call:  ancestor(X,cindy)
    (2)       2    Call:  parent(X,cindy)
    (3)       3    Call:  father(X,cindy)
    (3)       3    Exit:  father(george,cindy)
    (2)       2    Exit:  parent(george,cindy)
    (1)       1    Exit:  ancestor(george,cindy)
    (4)       1    Call:  sibling(george,jeffrey)
    (5)       2    Call:  mother(M,george)
    (5)       2    Exit:  mother(alice,george)
    (6)       2    Call:  mother(alice,jeffrey)
    (6)       2    Exit:  mother(alice,jeffrey)
    (7)       2    Call:  father(F,george)
    (7)       2    Exit:  father(albert,george)
    (8)       2    Call:  father(albert,jeffrey)
    (8)       2    Exit:  father(albert,jeffrey)
    (9)       2    Call:  not george=jeffrey
   (10)       3    Call:  george=jeffrey
   (10)       3    Fail:  george=jeffrey
    (9)       2    Exit:  not george=jeffrey
    (4)       1    Exit:  sibling(george,jeffrey)
X = george
yes
```

Figure 13.4 Trace of Prolog Program Execution

Furthermore, if this X had not satisfied `sibling(X,jeffrey)`, then the system would have been forced to backtrack to find other X's satisfying `ancestor(X,cindy)`. The search for these other Xs would pick up where it had left off when it found `X = george`.

■ ***Exercise 13-33:*** Given the definition of sum on p. 451, trace the Prolog execution of the goal

```
:- sum(succ(succ(0)), succ(succ(succ(0))), A).
```

■ ***Exercise 13-34:*** Given the same definition of sum and the definition of `dif` in the same place, trace the Prolog execution of

```
:- dif(succ(succ(succ(0))), What, succ(0)).
```

■ ***Exercise 13-35:*** Trace the Prolog execution of the goal

```
:- d(x, (2 * x) * (x + 1), Deriv).
```

■ ***Exercise 13-36:*** Describe algorithmically a breadth-first search strategy for logic programs.

Prolog Loses Many of the Advantages of Logic Programming

Consider again our Fibonacci program; it is not legal Prolog as it stands, but it is close. The difficulty is with the arithmetic equations such as `N = M + 1` and `F = G + H`. Recall that in most Prolog dialects an expression such as `M + 1` denotes a compound term equivalent to `plus(M,1)`; it does not call for addition to be performed. Furthermore, the '=' sign denotes a pattern-matching (unification) operation. Thus, `N = M + 1` is a request to unify N and the compound term `M + 1`. Now this was not our intention; for `fib` to work correctly, `N = M + 1` must be interpreted as the assertion that N and `M + 1` are *numerically* equal.

There is a simple (but inefficient) way out of this difficulty. Recall our definition of unary addition:

```
sum(M, 0, M).
sum(M, succ(N), succ(S)) :- sum(M, N, S).
```

Thus, if *M*, *N*, and *S* are compound terms representing (natural) numbers in unary notation (i.e., applications of succ to 0), then sum(*M,N,S*) asserts that *S* is *numerically* equal to *M* + *N*. Given this definition of sum, our Fibonacci program can be written in Prolog:

```
fib(0, succ(0)).
fib(succ(0), succ(0)).
fib(N,F) :- sum(M, succ(0), N), sum(K, succ(0), M),
            fib(M,G), fib(K,H), sum(G,H,F).
```

Notice that we have had to write '1' as '`succ(0)`'. As far as Prolog is concerned, '1' and '`succ(0)`' are just terms; there is no inherent connection between them. When these changes

have been made, our definition of `fib` becomes a legal Prolog program that will behave just as discussed earlier in this section (i.e., any combination of its arguments may be bound or unbound).

There are several difficulties with this. First, it is very inconvenient to express numbers in unary notation. Finding out the fifth Fibonacci number looks like this:

```
:- fib(succ(succ(succ(succ(succ(0))))), A).
A = succ(succ(succ(succ(succ(succ(succ(succ(0))))))))
```

There are ways we can get around this in Prolog (such as writing a converter between unary and regular numbers), but there is a more fundamental problem.

Doing arithmetic by unification on terms representing numbers in unary notation is an extremely inefficient way of doing precisely what computers are built to do efficiently: arithmetic. As noted earlier, all Prologs provide some means for making use of the computer's built-in arithmetic capabilities. In many Prologs this takes the form of the built-in predicate '`is`'. Suppose *N* is a variable and *E* is an arithmetic expression all of whose variables are bound (*instantiated*, in Prolog terminology). Then the subgoal '*N* `is` *E*' has the following effect: The expression *E* is evaluated to yield a numeric value *V*. If *N* is unbound, then *N* becomes bound to *V*; if *N* is already bound, then its bound value is compared with *V*. Thus, if *N* is unbound, then '*N* `is` *E*' behaves much like an assignment, '*N* := *E*'. If *N* is bound, it behaves more like a comparison, '*N* = *E*'. In either case, however, the expression *E* is evaluated using the arithmetic capabilities of the computer.

The obvious way to improve the efficiency of our Fibonacci program is to translate the `sum` relations into `is` relations:

```
fib(N,F) :- N is M + 1, M is K + 1, fib(M,G), fib(K,H),
            F is G + H, N > 1.
```

This looks very much like our original formulation. Unfortunately, it will not work.

Consider the goal `:- fib(2,A)`. In evaluating the subgoals from left to right, we begin with `N is M + 1`. However, `M` is unbound, so the arithmetic expression cannot be evaluated, and the program aborts. Taking our clue from imperative languages, we might try to solve this problem by rearranging the subgoals like this:

```
fib(N,F) :- M is N - 1, K is M - 1, fib(M,G), fib(K,H),
            F is G + H, N > 1.
```

This will in fact satisfy the goal `:- fib(2,A)` correctly.

Exercise 13-37: Trace in detail the depth-first satisfaction of this goal.

Now, however, suppose that we try the goal `:- fib(A,3)`. In this case, `N` is unbound, so the subgoal '`M is N - 1`' fails. We have lost the ability to execute `fib` "backward." Perhaps another order will work. Perhaps we can make use of backtracking's ability to enumerate (*N,F*) pairs satisfying `fib(`*N,F*`)` in order to avoid the ordering dependencies inherent in '`is`'. For example, we can program

```
fib(N,F) :- fib(K,H), M is K + 1, fib(M,G),
            N is M + 1, F is G + H, N > 1.
```

Suppose now we execute the goal `:- fib(3,A)`. The first subgoal is `fib(K,H)`. Since `K` and `H` are not bound, this subgoal will be satisfied by `K = 0, H = 1`. This in turn leads to `M` being bound to `1` and the subgoal `fib(1,G)`, which is satisfied with `G = 1` and the fact `fib(1,1)`. Now, however, the subgoal '`N is M + 1`' compares 3 (the value of `N`) with 2 (the value of `M + 1`), and so fails. This causes a backtrack to seek another value of `G` satisfying `fib(1,G)`. Since both the facts have been tried, we now take `N = 1` in the rule headed `fib(N,F)`. As before `fib(K,H)` is satisfied by `K = 0, H = 1`, so we set up the subgoal `fib(1,G)`. But this is precisely the subgoal we were already trying to satisfy. The `fib` predicate has been called recursively with precisely the same arguments as a call already active; we are in an infinite loop! (A trace is shown in Figure 13.5.) Once again we have failed to produce a correct version of `fib` (one that works with every possible combination of arguments).

We will not continue this exercise; the point has been made. Suffice it to say that we have come a long way from the goal of nonprocedural programming: telling the computer *what* you want, not *how* to get it. In Prolog, not only do we have to specify in the correct order the steps that must be taken to reach the goal, but we also have to cope with a complicated, expensive, and unintuitive control strategy. The control strategy is anything but transparent.

```
:- fib(3,A).
Invocation   Depth   Event   Subgoal
-----------------------------------
    (1)        1     Call:   fib(3,F)
    (2)        2     Call:   fib(K,H)
    (2)        2     Exit:   fib(0,1)
    (3)        2     Call:   M is 0 + 1
    (3)        2     Exit:   1 is 0 + 1
    (4)        2     Call:   fib(1,G)
    (4)        2     Exit:   fib(1,1)
    (5)        2     Call:   3 is 1 + 1
    (5)        2     Fail:   3 is 1 + 1
    (4)        2     Redo:   fib(1,G)
    (6)        3     Call:   fib(K',H')
    (6)        3     Exit:   fib(0,1)
    (7)        3     Call:   M' is 0 + 1
    (7)        3     Exit:   1 is 0 + 1
    (8)        3     Call:   fib(1,G')
    (8)        3     Exit:   fib(1,1)
    (9)        3     Call:   1 is 1 + 1
    (9)        3     Fail:   1 is 1 + 1
    (8)        3     Redo:   fib(1,G')
   (10)        4     Call:   fib(K",H")
    .          .       .       .
    .          .       .       .
    .          .       .       .
```

Figure 13.5 Trace of Incorrect Definition of 'fib'

Notice that all our versions of `fib` were *logically* the same; reordering the subgoals affects only the control properties. Unfortunately, in Prolog, as opposed to *pure* logic programming, the control properties can determine the correctness of the program. In pure logic programming, on the other hand, the correctness of the program depends only on the logic of the algorithm. Control is an issue of efficiency, which can be dealt with after the program is correct.

Concern for Efficiency Has Motivated Prolog's Search Strategy

Given the unintuitive consequences of Prolog's control regime, we should ask ourselves why the designers have decided to specify a specific strategy at all, let alone the specific one they chose. The first reason is that a depth-first search may be considerably more efficient than other search strategies, such as a breadth-first search. Since the search goes *deep* through the search tree, the information that must be stored is that gathered along a single path down the tree. On the other hand, a breadth-first search is in effect pursuing all search paths in parallel (this is why it is guaranteed to find a solution if there is one), so it must retain the information for all these paths. For this reason, a breadth-first search tends to need exponentially more space than a depth-first search. That is certainly a good reason for going depth-first.

The trouble with depth-first is that the search tree may be infinitely deep. That is, although some branches may lead to a solution in a finite number of steps, others are infinitely long and never reach a solution. If the search has the bad luck to stumble into one of these "black holes" before it finds a solution, then it will never come out.

Since there is no general procedure by which the Prolog system can determine whether or not it has started off on an endless search, the burden has been put on the programmer. The language specifies the precise order in which the search is done. The programmer then has the responsibility of arranging the clauses and subgoals in the program so that it will terminate when used in the intended way (but not necessary in unintended ways, as we saw before).

There is another reason that Prolog specifies its search order, which comes from the fact that Prolog is used for programming that is not, strictly speaking, logical. For example, most Prologs have "special" predicates for performing input and output. Thus, a subgoal involving the predicate `write` succeeds exactly once but causes the arguments to be written out as a side effect. Conversely, a subgoal involving `read` has the side effect of reading a term from a file, which is then unified with the argument. A new term is read every time the system backtracks to the `read`.

Predicates such as these allow programmers to use conventional programming idioms in their Prolog programs. Obviously, however, they would be of little value if they were not executed at predictable times. Hence, the desire to do things such as input-output dictates a defined search order.

■ ***Exercise 13-38:*** Explain the consequence of the following Prolog definition of `ancestor`:

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

■ ***Exercise 13-39:*** Write Prolog clauses for a predicate `unary` that relates natural numbers to their unary representations. For example,

```
:- unary(N,U).
N = 0, U = 0;
N = 1, U = succ(0);
N = 2, U = succ(succ(0));
N = 3, U = succ(succ(succ(0)))
```

■ ***Exercise 13-40:*** Why can't a Prolog (or any other) software system decide in general whether a particular search path will terminate?

■ ***Exercise 13-41*:*** Can you suggest a way of handling input-output in logic programs that does not depend on a specific search order? *Hint:* How can the organization of information on the input-output media be specified *logically*?

Assert and Retract Permit Nonmonotonic Reasoning

One facility that depends crucially on Prolog having a defined search order is the ability to *assert* and *retract* facts and rules. Satisfying a subgoal of the form `retract(`*C*`)` causes the first clause matching the term *C* to be removed from the database. Resatisfaction (e.g., during backtracking) may cause additional matching clauses to be removed.[12] Satisfying a subgoal of the form `asserta(`*C*`)` or `assertz(`*C*`)` causes the term *C*, interpreted as a clause, to be inserted into the database (at its beginning for `asserta` or end for `assertz`: remember that in Prolog the order of the clauses affects the logic). Thus, the Prolog database is an *updatable* database.

While it seems perfectly reasonable for a database to be updatable, it is not so obvious what assertions and retractions mean in the context of *logic* programming. In predicate logic (on which Prolog is based), a proposition is either true or false; it cannot be true now and false later, or vice versa. We say that predicate logic is *monotonic* (mono = single, tonos = tension), that is, the set of known truths never decreases. Applying the deductive rules of a monotonic logic can increase the set of propositions known to be true, but never decrease it. Deduction never causes things known to be true to cease being true (or things known to be false to cease being false).

It certainly seems reasonable for logic to be monotonic: If we could prove something yesterday, it seems we ought to be able to prove it today. Yet there are situations in which a *nonmonotonic* logic is just what is needed. Consider a robot vehicle planning its path across a terrain. It will decide which path is best on the basis of a variety of information sources, including general rules and particular facts. Some of these facts may turn out to be false. For example, a road that was "known" to connect two points is now found to be closed. Or what looked like solid ground from afar is found to be a swamp on closer inspection. In other words, things that were thought to be true turn out to be false, and vice versa. Therefore, conclusions that were drawn on the basis of these hypotheses (such as the best route) are

[12] Some Prolog systems do not permit resatisfaction of `retracts`.

found not to hold. The robot vehicle must be able to revise its conclusions on the basis of this new information.

In an application such as this, propositions are not simply true or false; they are true or false *at a particular time* in the program's execution. Thus, it becomes important for programmers to know the order in which things are done in their program. They must pay attention to the *temporal* as well as the *logical* relationships.

■ ***Exercise 13-42*:*** Suggest a way in which nonmonotonic reasoning could be handled in a *pure* monotonic logic programming language. *Hint:* How can you express logically the temporality of facts?

Cuts Are Used to Guarantee Termination

Consider the following straightforward Prolog definition of factorial:

```
fac(0,1).
fac(N,F) :- N1 is N - 1, fac(N1,R1), F is R1*N.
```

The following goal gives the expected answer:

```
:- fac(4,A).
A = 24
```

Unfortunately, this (obvious) definition of `fac` has a hidden trap. Let's consider what happens if we try to satisfy the following (unsatisfiable) goal:

```
:- fac(4,A), A = 10.
```

As before, the subgoal `fac(4,A)` succeeds with the instantiation `A = 24`. If we number the `fac` clauses (1) and (2), then we can represent the series of choices by which this solution was reached:

```
(2), (2), (2), (2), (1)
```

That is, we had to take the second clause for `N` = 4, 3, 2, 1, but for `N` = 0 we took the first clause.

Now consider what happens when we try to satisfy the subgoal `A = 10`. Since `A` is bound to 24, this subgoal fails, which causes Prolog to backtrack to the last choice it made. In effect, it will try to find another `A` for which `fac(4,A)` is true. Of course, the Prolog system has no way of knowing that each number has only one factorial, so it will search for another one even though it does not exist. It does this by trying to find another clause to unify with the last subgoal it satisfied. In this case, that subgoal was `fib(0,R)`, which was satisfied by clause (1) of the definition of `fac`. Hence, Prolog searches for another clause to unify with `fib(0,R)` and it finds clause (2), which unifies by the assignment `N = 0`. This sets up the subgoals

```
:- N1 is 0 - 1, fac(N1,R1), F is R1*1.
```

Execution of the 'is' immediately leads to the subgoals

```
:- fac(-1,R1), F is R1*1.
```

The system is now off looking for a factorial for −1, and, of course, it will not find one. The system is in an infinite loop, although a simple `no` is the answer we expect.

What was the cause of this difficulty? We can see that once the system had found `A = 24` as a solution of `fib(4,A)`, it never should have looked for another solution; a number has at most one factorial. It would be useful if there were some way to say to Prolog, "You have found all the solutions there are; do not bother trying to find any others."

Prolog provides just such a mechanism; it is called a *cut* (presumably because it "cuts off" the search for alternatives). In this case, we would like to say, "Once you have found the solution `A = 1` for `fib(0,A)`, do not look for any others." This is expressed in Prolog:

```
fac(0,1) :- !.
fac(N,F) :- N1 is N - 1, fac(N1,R1), F is R1*N.
```

The symbol '`!`' is the "cut." It is a predicate that always succeeds, but past which you can never backtrack. Hence, having once found the solution `A = 1` for `fib(0,A)` we will not attempt to find another solution. Now we get the expected response if we execute

```
:- fac(4,A), A = 10.
no
```

This example illustrates one of the common uses of a "cut": signaling that the right solution has been found.

Unfortunately, our problems with `fac` are still not solved. Suppose we enter the goal

```
:- fac(0,0).
```

(asking if the factorial of zero is zero). The system will again go into an infinite loop. Since `fib(0,0)` does not unify with `fib(0,1)`, the head of the first clause, the system will go on to the second clause, and we are off on the same goose chase. There are several solutions to this problem. Since our definition works only if `N` is instantiated, perhaps the simplest is to add conditions on `N`:

```
fac(0,1) :- !.
fac(N,F) :- N > 0, N1 is N - 1, fac(N1,R1), F is R1*N.
```

■ ***Exercise 13-43:*** Rewrite the above definition of `fac` so that the factorial of negative numbers is `infinity`. (Of course, `infinity` is just an atom like `nil` or `albert`.)

Cuts Are Used to Control Execution Order

Prolog's defined search order, in addition to "impure" features such as the "cut," have encouraged many programmers to adopt a very procedural Prolog programming style. Here is an example:

```
squares_to(N) :- asserta(current(1)),
                 repeat,
                 one_step,
                 current(N).
```

```
one_step :-     current(K),
                Sq is K*K,
                write(Sq), nl,
                NewK is K + 1,
                retract(current(K)),
                asserta(current(NewK)),
                !.
```

The predicate `repeat`, which is built into most Prologs, is defined by the clauses

```
repeat.
repeat :- repeat.
```

Viewed logically, these clauses are curious indeed. They mean something like "repeat is true; also repeat is true *if* repeat is true." To understand their significance it is necessary to *trace* the execution of `squares_to`. Suppose we enter the goal `:- squares_to(5).` We will see this:

```
:- squares_to(5).
1
4
9
16
yes
```

These are the squares of the numbers from 1 up to but not including 5. Consider how the Prolog system goes about trying to satisfy this goal.

To satisfy `:- squares_to(5)`, the system must satisfy *in order* the subgoals

```
:- asserta(current(1)), repeat, one_step, current(5).
```

The `asserta` enters the fact `current(1)` in the database and succeeds. The `repeat` subgoal also succeeds because the fact `repeat` is in the database. Next we come to the subgoal `one_step`; to satisfy this we must satisfy the subgoals given by the clause for `one_step`. The first of these simply binds `K` to `1` (since we just put `current(1)` in the database). The second binds `Sq` to `K` squared, which is `1` in this case. The following `write` prints the value 1, which is followed by `nl`, a built-in predicate to go to a new line. The next three subgoals retract `current(1)` and assert `current(2)`. The "cut" means that we will not try to resatisfy `one_step`. Since `one_step` has been satisfied, we return to `squares_to` and attempt to satisfy `current(5)`, but since `current(2)` is the only relevant fact in the database, this attempt fails.

Now we must backtrack. Since `one_step` has been "cut," we cannot try it again, so we go back to `repeat` and attempt to resatisfy it. There is another clause for `repeat`, namely, the rule `repeat :- repeat`, so `repeat` can be satisfied if we can satisfy—`repeat`! Since `repeat` is a fact, the rule can be applied, and `repeat` can be satisfied *in a different way* from the first time. Hence, we can return to the `squares_to` rule and satisfy again `one_step`. This time satisfaction of `one_step`

prints 4, which is 2 squared. In the process it retracts `current(2)` and asserts `current(3)`. However, since `current(5)` still fails, we backtrack again to the `repeat`. And so we continue, repeating `one_step` until finally `current(5)` has been asserted, at which point `squares_to(5)` is satisfied and our program stops. See Figure 13.6.

How far we have come from logic programming! And how far we have come from *nonprocedural* programming! This program is certainly as procedural as any we might write in Pascal or LISP. The predicate `current` is in effect a variable, and the asserts and retracts just implement assignments to this variable. The `repeat` predicate, as its name implies, has

```
:- squares_to(5).
Invocation  Depth  Event  Subgoal
(1)         1      Call:  squares_to(5)
(2)         2      Call:  asserta(current(1))
(2)         2      Exit:  asserta(current(1))
(3)         2      Call:  repeat
(3)         2      Exit:  repeat
(4)         2      Call:  one_step
(5)         3      Call:  current(K)
(5)         3      Exit:  current(1)
(6)         3      Call:  Sq is 1*1
(6)         3      Exit:  1 is 1*1
(7)         3      Call:  write(1)
(7)         3      Exit:  write(1)
(8)         3      Call:  nl
(8)         3      Exit:  nl
(9)         3      Call:  NewK is 1 + 1
(9)         3      Exit:  2 is 1 + 1
(10)        3      Call:  retract(current(1))
(10)        3      Exit:  retract(current(1))
(11)        3      Call:  asserta(current(2))
(11)        3      Exit:  asserta(current(2))
(4)         2      Exit:  one_step
(12)        2      Call:  current(5)
(12)        2      Fail:  current(5)
(3)         2      Redo:  repeat
(3)         2      Exit:  repeat
(13)        2      Call:  one_step
 .          .      .      .
 .          .      .      .
 .          .      .      .
```

Figure 13.6 Trace of Procedural Prolog Program

the effect of beginning a loop, which is returned to by the backtracking mechanism. The Pascal equivalent is simply

```
procedure SquaresTo (N: integer);
  var current: integer;
begin
  current := 1;
  repeat
    writeln(current*current);
    current := current + 1;
  until current = N;
end;
```

Certainly the programmer's intent is more clearly expressed in the Pascal program. Further, we will be very confused if we attempt to attach any logical significance to the clauses in the Prolog program. Here we have an example of the *ampliative* and *reductive* aspects of logic-programming technology (Section 1.4). The ampliative aspects are clear enough: the ability to express logical relationships directly. However, there is a corresponding reduction of control over the sequence of operations, which decreases efficiency and complicates temporal programming.

- ***Exercise 13-44*:*** Do you prefer the Pascal or Prolog program? Explain why.
- ***Exercise 13-45*:*** Discuss procedural programming in Prolog. Should it be encouraged or discouraged? What benefits does it have over procedural programming in conventional languages? How could it be discouraged, given Prolog's depth-first search order?

Prolog Indirectly Supports Higher-Order Rules

If we wanted to add all the elements of a list, we could use clauses such as these:

```
sum_red([], 0).
sum_red(X.L, S) :- sum_red(L, T), S is X + T.
```

This is a "sum reduction" such as we studied in Chapter 10. Other reductions would follow a similar pattern. This should lead us to ask if there is a way in Prolog of defining a general reduction operation such as we defined in LISP (p. 352).

Let P be a predicate corresponding to a binary operation F. Thus, $P(X, Y, Z)$ if and only if $Z = F(X, Y)$. We would like to define a predicate `red` such that `red(P, I, L, S)` means that S is the result of doing an F reduction (starting from the initial value I) of the list L. That is,

$$S = F(L_1, F(L_2, ..., F(L_{n-1}, F(L_n, I)) ...))$$

The natural way to do this is simply to make `P` and `I` parameters to the predicate:

```
red(P, I, [], I).
red(P, I, X.L, S) :- red(P, I, L, T), P(X, T, S).
```

Unfortunately, this rule would not be legal in most logic programming languages. The reason is that the `P` parameter refers to a predicate, which makes it a *higher-order rule.* In general, logic programming is restricted to *first-order logic*, that is, to logic in which the variables may be bound to individuals (terms), but not higher-order objects (predicates).

Although most Prolog systems do not permit higher-order rules, some do allow the same effect to be achieved in a more roundabout way. In these systems `red` could be defined:

```
red (P, I, [], I).
red(P, I, X.L, S) :- red(P, I, L, T), Q =.. [P, X, T, S], call(Q).
```

The strange looking predicate '`=..`' binds `Q` to the result of converting the list `[P,X,T,S]` into the internal representation of the subgoal `P(X,T,S)`. The subgoal `call(Q)` then causes the system to attempt to satisfy the subgoal represented by the structure `Q`. In effect `call` passes its argument to the Prolog interpreter (in this sense it is like LISP `eval`).

The '`=..`' predicate depends on the fact that the internal representation of subgoals (code) is very similar to the internal representation of compound terms (data). We saw an analogous situation in LISP, where the program structures and data structures are represented the same way: *S*-expressions. This is especially clear if we write the list `[P,X,T,S]` in *S*-expression notation: `(P X T S)`.

In LISP both code and data are represented as first-order objects. In Prolog, on the other hand, we need an operator to convert a first-order object, such as the term `[P,X,T,S]`, into a higher-order object, such as the relationship `P(X,T,S)`. Hence, Prolog provides limited access to higher-order clauses by way of their representation as first-order terms.

Why doesn't Prolog simply permit higher-order clauses, rather than requiring this detour through first-order representations? The answer is simple, but fundamental. Robinson's resolution algorithm is *refutation complete* for *first-order* predicate logic. This means that if the goal is derivable from the clauses in the database, then the resolution algorithm will find a derivation.[13] Unfortunately, Gödel's incompleteness theorem tells us that neither resolution nor any other algorithm is refutation complete for *higher-order* predicate logic.

■ ***Exercise 13-46:*** Write a predicate `map` that maps a function across a list. That is, if $P(X, Y)$ if and only if $Y = F(X)$, then `map(P,L,M)` returns a list M that is the result of applying F to each element of L.

■ ***Exercise 13-47*:*** Discuss the following language design issue: Is it better to permit higher-order clauses, knowing that a complete deductive procedure does not exist, or to restrict programs to first-order clauses, which in effect prevents people from writing nonterminating programs? Does your answer change if you assume a depth-first strategy as in Prolog?

Negation Is Interpreted as Unsatisfiability

The problem of negation in logic programming languages illustrates some further applications of higher-order rules. What is the problem of negation? Suppose we want to define a

[13] Note that we are talking about the resolution algorithm. Since Prolog implements only a subset of resolution, it may not find a solution even if one exists. We have seen several examples of this.

predicate `can_marry(X,Y)` that means that *X* can legally marry *Y*. For the sake of the example, we assume that *X* can marry *Y* provided that they are not more closely related than (first) cousins. Thus, *X* can marry *Y* if (1) *X* and *Y* are not the same individual, (2) *X* and *Y* are not siblings, and (3) *X* and *Y* are not cousins. How can we express this in Prolog?

Expressing that *X* and *Y* are not the same is easy since most Prologs have a built-in not-equal predicate '`\=`' (but see below). Thus, we might begin

```
can_marry(X,Y) :- X \= Y, nonsibling(X,Y), noncousin(X,Y).
```

We face difficulties, however, when we try to define `nonsibling`. Our rule for `sibling` says that two individuals are siblings when they are not the same and they have the same mother and father. Hence, we can negate this to find out the ways in which two individuals could be *non*siblings. Thus, *X* and *Y* are not siblings if (1) they are the same, or (2) they have different mothers, or (3) they have different fathers:

```
nonsibling(X,Y) :- X = Y.
nonsibling(X,Y) :- mother(M1,X), mother(M2,Y), M1 \= M2.
nonsibling(X,Y) :- father(F1,X), father(F2,Y), F1 \= F2.
```

It would seem that this does it, but suppose we try

```
:- nonsibling(albert, alice).
no
```

The system tells us that it is *not* the case that Albert and Alice are nonsiblings, that is, that Albert *is* a sibling of Alice. Yet if we look at Figure 13.1 we can see that Albert and Alice do not have common parents. We can also see the cause of the problem: So far as the program is concerned, Albert and Alice do not have parents at all! Of course, this does not mean that they do not *in fact* have parents, but only that our database must stop somewhere: You cannot record the entire state of the world in a computer. Yet the "closed-world" semantics of Prolog acts as though anything not recorded in the database is not true. As far as our program is concerned, Albert and Alice are parentless. Hence, to complete our definition of `nonsibling` we must take care of the possibility that either *X* or *Y* does not have a parent:

```
nonsibling(X,Y) :- no_parent(X).
nonsibling(X,Y) :- no_parent(Y).
```

Now we face another difficulty: How do we express the fact that *X* does not have a parent? The trouble is that there is no positive fact that expresses the *absence* of a parent. Of course, we could add facts such as `no_parent(albert)` to the database, but that does not solve the fundamental problem, to which we now turn.

We have seen that the predicate `nonsibling` is definable if we are willing to put in enough negative facts of the form `no_parent(X)`. A similar approach works for `noncousin`. The trouble is that this general approach to defining negative predicates—considering all the possible ways the corresponding positive predicate could fail to hold—is tedious at best and error-prone at worst. We would like to take a more direct approach: define `nonsibling(X,Y)` to hold in exactly those cases in which `sibling(X,Y)` does not hold. Unfortunately, with Horn clauses such as

C :– P_1, P_2, ..., P_n

we can say only that C holds if the premises P_1, P_2, ..., P_n *do* hold; we are unable to conclude anything from the fact that they *do not* hold.

The problem may be summarized as follows. The facts in a Horn clause program state a number of relationships that hold among a number of individuals. The rules allow a number of additional facts to be deduced from the given facts. Thus, from the fact that certain relationships hold, we can conclude that further relationships hold, and these are the *only* conclusions we can draw. So we can conclude that relationships *do* hold, but there is no way to conclude that they *do not* hold. From positive relationships we can deduce only positive relationships. Unfortunately, in the case of `nonsibling` and `noncousin` we need to deduce negative relationships. The only way to conclude negative relationships is to put in facts with negative import, as we did with `no_parent`. Unfortunately, this approach often leads to a knowledge explosion, since the number of ways something can fail to be true is often much greater than the number of ways it can succeed at being true.

Solving these difficulties with negative predicates requires going beyond pure Horn clauses. One way to do this is with the "cut-fail" combination. This is a Prolog idiom that combines the "cut" with a predicate `fail`, which always fails. Although it is built into most Prologs, the `fail` predicate is easy to define: We simply avoid putting the fact `fail` into the database!

To illustrate this idiom we define `nonsibling`. We want to guarantee that `nonsibling(X,Y)` fails when `sibling(X,Y)` succeeds, but allow it to succeed otherwise. The rules are

```
nonsibling(X,Y) :- sibling(X,Y), !, fail.
nonsibling(X,Y).
```

Logically, these rules are absolute harsh, but they work given Prolog's procedural interpretation.

To see this, observe that in attempting to satisfy `nonsibling(jeffrey, george)` we must first try to satisfy `sibling(jeffrey, george)`. Since this succeeds we execute the "cut" (which always succeeds) and then move on to the `fail` (which always fails). Since the "cut" has prevented backtracking, the entire clause fails. The resulting nonsatisfiability of `nonsibling(jeffrey, george)` correctly reflects the fact that Jeffrey and George are siblings.

Now consider the goal `nonsibling(albert, alice)`. The system attempts to satisfy the first clause for `nonsibling`, but this requires satisfying the subgoal `sibling(albert, alice)`. Since this subgoal cannot be satisfied the system backtracks (which is allowed since we have not reached the "cut"), and tries the second `nonsibling` clause. This immediately succeeds, so the system correctly deduces that Alice and Albert are not siblings.

This situation is common enough that most Prolog dialects provide a `not` predicate defined so that `not(C)` succeeds if C fails, and vice versa. It can be defined within Prolog by means of the "cut-fail" combination and higher-order features:

```
not(C) :- call(C), !, fail.
not(C).
```

Given `not` it is easy to define `nonsibling`:

```
nonsibling(X,Y) :- not(sibling(X,Y)).
```

we can say only that C holds if the premises P_1, P_2, ..., P_n *do* hold; we are unable to conclude anything from the fact that they *do not* hold.

The problem may be summarized as follows. The facts in a Horn clause program state a number of relationships that hold among a number of individuals. The rules allow a number of additional facts to be deduced from the given facts. Thus, from the fact that certain relationships hold, we can conclude that further relationships hold, and these are the *only* conclusions we can draw. So we can conclude that relationships *do* hold, but there is no way to conclude that they *do not* hold. From positive relationships we can deduce only positive relationships. Unfortunately, in the case of `nonsibling` and `noncousin` we need to deduce negative relationships. The only way to conclude negative relationships is to put in facts with negative import, as we did with `no_parent`. Unfortunately, this approach often leads to a knowledge explosion, since the number of ways something can fail to be true is often much greater than the number of ways it can succeed at being true.

Solving these difficulties with negative predicates requires going beyond pure Horn clauses. One way to do this is with the "cut-fail" combination. This is a Prolog idiom that combines the "cut" with a predicate `fail`, which always fails. Although it is built into most Prologs, the `fail` predicate is easy to define: We simply avoid putting the fact `fail` into the database!

To illustrate this idiom we define `nonsibling`. We want to guarantee that `nonsibling(X,Y)` fails when `sibling(X,Y)` succeeds, but allow it to succeed otherwise. The rules are

```
nonsibling(X,Y) :- sibling(X,Y), !, fail.
nonsibling(X,Y).
```

Logically, these rules are absolute harsh, but they work given Prolog's procedural interpretation.

To see this, observe that in attempting to satisfy `nonsibling(jeffrey, george)` we must first try to satisfy `sibling(jeffrey, george)`. Since this succeeds we execute the "cut" (which always succeeds) and then move on to the `fail` (which always fails). Since the "cut" has prevented backtracking, the entire clause fails. The resulting nonsatisfiability of `nonsibling(jeffrey, george)` correctly reflects the fact that Jeffrey and George are siblings.

Now consider the goal `nonsibling(albert, alice)`. The system attempts to satisfy the first clause for `nonsibling`, but this requires satisfying the subgoal `sibling(albert, alice)`. Since this subgoal cannot be satisfied the system backtracks (which is allowed since we have not reached the "cut"), and tries the second `nonsibling` clause. This immediately succeeds, so the system correctly deduces that Alice and Albert are not siblings.

This situation is common enough that most Prolog dialects provide a `not` predicate defined so that `not(C)` succeeds if C fails, and vice versa. It can be defined within Prolog by means of the "cut-fail" combination and higher-order features:

```
not(C) :- call(C), !, fail.
not(C).
```

Given `not` it is easy to define `nonsibling`:

```
nonsibling(X,Y) :- not(sibling(X,Y)).
```

That is, `nonsibling(X,Y)` is satisfiable if `sibling(X,Y)` is not satisfiable, and vice versa. In fact, the simplest solution is probably to dispense with the negative predicates altogether, and simply define `can_marry` using `not`:

```
can_marry(X,Y) :- X \= Y, not(sibling(X,Y)), not(cousin(X,Y)).
```

This is certainly much more readable than our original definition. Unfortunately, `not` contains some hidden traps.

■ ***Exercise 13-48:*** Trace the execution of the goal

```
:- can_marry(george, mary).
```

■ ***Exercise 13-49:*** Explain in detail the operation of the above definition of `not`, both when it succeeds and when it fails.

Difficulties with Negation as Unsatisfiability

Although the preceding definition of `not` seems straightforward enough, it has some dark corners that we now explore. Consider the following goal, which enumerates all of the ways of appending two lists to get `[a,b,c,d]`:

```
:- append(X,Y,[a,b,c,d]).
X = [], Y = [a,b,c,d];
X = [a], Y = [b,c,d];
X = [a,b], Y = [c,d];
X = [a,b,c], Y = [d];
X = [a,b,c,d], Y = [];
no
```

Each semicolon causes the system to backtrack and find additional variable instantiations that satisfy the goal. Now, consider the following goal:

```
:- not(not(append(X,Y,[a,b,c,d]))).
```

In logic two `not`s cancel each other, so we would expect this goal to mean the same as the original goal. This is not the case, however, as we can see by tracing the execution: The outer `not` sets up the subgoal `not(append(X,Y,[a,b,c,d]))`, which in turn sets up the subgoal `append(X,Y,[a,b,c,d])`. The latter we know is satisfiable by `X = []` and `Y = [a,b,c,d]`. Since the `append` succeeds, it causes, through the "cut-fail" combination, the failure of the inner `not`. Now, we know that whenever a subgoal fails, we must backtrack, so any bindings made in the attempt to satisfy that subgoal are unbound. Hence, `X` and `Y` return to their unbound state. Finally, since the inner `not` failed, the outer `not` succeeds, and so the goal is satisfied, but with `X` and `Y` unbound! We will see something like this:

```
:- not(not(append(X,Y,[a,b,c,d]))).
X = _0, Y = _1;
no
```

Here '_0' and '_1' are symbols used by the Prolog system to indicate that the variables X and Y are unbound. Thus, although a double negation has no effect logically, it may have a major effect on the solutions found by a Prolog program. Once again logic has been inadequate to explain the outcome of a Prolog program. Rather, we have had to appeal to a *procedural* understanding of its execution.

What is the cause of this unintuitive behavior? We have been fooled by the name `not` into expecting this operation to behave like logical negation. In fact, `not` in Prolog does not mean the same thing as 'not' in logic. In Prolog `not` means "not satisfiable." In logic there is an important difference between proving that something is false and not being able to prove that it is true. Yet, in effect, the Prolog `not` predicate identifies these two concepts. This illustrates again how far Prolog is from logic programming, due to a loss of logical transparency.

Prolog Provides Term Equality

Notice that in the kinship program (Figure 13.1), if it were not for the condition `X \= Y` in the definition of `sibling`, people would be considered siblings of themselves. In fact, they would be considered their own cousins! In common usage, we say that *X* is a sibling of *Y* if *X* and *Y* have the same parents *and X* is not the same person as *Y*. Hence, it is our intention that `X \= Y` means that `X` and `Y` are not the same person.

What, exactly, does the predicate '`\=`' mean? By investigating this question we will discover some interesting issues in logic programming. Formally, not-equals can be defined in terms of equals:

```
X \= Y :- not(X = Y).
```

Thus *X* `\=` *Y* is satisfiable just when *X* = *Y* is not satisfiable. This reduces the question of the meaning of '`\=`' to the meaning of '='.

There are many notions of equality. With all of these we would expect `albert = albert` to be true, but some of the other consequences are not so obvious. For example, Leibniz' *Principle of the Identity of Indiscernibles* says that two things are identical if all of their properties are the same. This seems plausible enough: If there is no property by which the two things can be discerned, then what basis is there for the claim that there are *two* things? On the other hand, the principle has some surprising consequences when applied to logic programs. For example, since in our database in Figure 13.1 `cindy` and `victor` participate in exactly the same relationships, Leibniz' principle tells us that they are identical. Similarly, `jeffrey` and `george` are identical.

This is clearly unsatisfactory. It is our intention that `cindy` and `victor` correspond to distinct persons (Cindy and Victor), who no doubt differ in many of their properties, although they happen to coincide in those properties represented in our database. Thus, Leibniz' principle, no matter what its virtues in the real world, cannot be applied in the simulated world of the computer. We need a stronger notion of equality.

There is another notion of equality, *term equality*, that is implicit in the Prolog system. We can define this equality predicate by a single fact:

```
X = X.
```

By the usual rules of unification, the above clause will be satisfied only if both its arguments are the same term. Thus, `cindy = cindy` will unify with the above fact. On the other hand, two different terms cannot both be unified with `X`. Hence, `cindy = victor` will not be satisfiable. In effect, two atomic terms are equal if and only if they are denoted by the same string of characters.[14] This seems to be just what we expect. Every object mentioned in our program is distinct; each has its own identity. This is a reasonable expectation whenever we are doing *object-oriented* programming, that is, whenever the objects in the computer are simulating objects in the real world. In these situations it is natural to consider two objects to be distinct *unless* they are specified to be the same.

What about equality of compound terms? This is just a recursive extension of atomic equality. That is, two compound terms are equal if they have the same functor and the same number of arguments, and if the corresponding arguments are equal. This is in effect the `equal` function of LISP (see Chapter 10, Section 10.1). It is a very intuitive notion of equality since terms are equal just when they are written the same way.

Other Equivalence Relations Cannot Be Defined

Term equality is not always the kind of equality that is needed. Suppose we were defining finite sets, including various operations such as membership, intersection, and union. A straightforward approach is to represent the null set by the atomic term `empty` and nonnull sets by compound terms of the form `adjoin(X,S)`. The intended interpretation of `adjoin(X,S)` is $\{X\} \cup S$. Thus, the set {a,b,c} is represented by the term

```
adjoin(a, adjoin(b, adjoin(c, empty)))
```

Many of the set operations are now easy to define; for example, membership:

```
member(X, adjoin(X, S)).
member(X, adjoin(Y, S)) :- member(X, S).
```

We have to be careful, however, when we come to equality between sets.

Suppose we are implementing an inventory control program and we have a predicate `parts(X,S)` that means that S is the set of parts required to build product X. Further suppose we need a predicate `same_parts(X,Y)` that means that X and Y require the same set of parts. An obvious (but incorrect) way of defining `same_parts` is

```
same_parts(X, Y) :- parts(X, S), parts(Y, S).
```

The problem will be clearer if we write this rule in the equivalent form:

```
same_parts(X, Y) :- parts(X, S), parts(Y, T), S = T.
```

We are using *term equality* to compare the sets of parts required for `X` and `Y`. What is wrong with that?

You know that sets are considered identical if they have the same elements—regardless

[14] This is essentially the notion of equality of atoms used in LISP (see Section 9.3).

of the number of times elements appear or the order in which they are listed. Thus, these all are descriptions of the same set:

```
{a,b,c}, {c,a,b}, {a, b, b, c, a}
```

Analogously the following compound terms all represent the same set:

```
adjoin(a, adjoin(b, adjoin(c, empty)))
adjoin(c, adjoin(a, adjoin(b, empty)))
adjoin(a, adjoin(b, adjoin(b, adjoin(c, adjoin(a, empty)))))
```

Clearly, though, they are different terms, and so will not be considered the same under term equality. Thus, our definition of `same_parts` will not work because it requires the sets of parts to be written in the same way; the sets of parts will be considered different if they happen to be written in a different order. Hence, the term equality test is too strong.

The obvious solution is to define a `set_equal` predicate that explicitly axiomatizes the fact that the order of set elements does not matter. We use a typical axiomatization of set equality.[15] First we state that all null sets are equal:

```
set_equal(empty, empty).
```

To express the fact that element multiplicities do not matter we write

```
set_equal(adjoin(X, adjoin(X, S)), adjoin(X, S)).
```

To express the fact that element order does not matter we write

```
set_equal(adjoin(X, adjoin(Y, S)), adjoin(Y, adjoin(X, S))).
```

These are all facts; we also need rules that specify the standard properties of equivalence relations:

```
set_equal(X,X).
set_equal(X,Y) :- set_equal(Y,X).
set_equal(X,Z) :- set_equal(X,Y), set_equal(Y,Z).
```

Unfortunately, although these are perfectly correct logically, they will throw Prolog into an infinite loop. The reason is Prolog's depth-first search order; a breadth-first search, though expensive, would terminate.

In the case of sets, there is an alternative, more procedural way of defining equality (you will work it out in an exercise), but the fact remains that in Prolog one cannot generally define equivalence relations in terms of their logical properties. Again, Prolog forces us to program procedurally rather than logically.

■ ***Exercise 13-50:*** Show an example goal involving `set_equal` that leads to an infinite loop.

■ ***Exercise 13-51*:*** If we wish to allow sets as members of sets, then the above definition of `set_equal` is not even *logically* correct. Identify the logical problem and solve it.

---

[15] Our set equality axioms are based on Manna and Waldinger (1985).

- ***Exercise 13-52:*** Trace the goal

```
:- member(c,adjoin(a,adjoin(b,adjoin(c,empty)))).
```

- ***Exercise 13-53:*** Trace the goal

```
:- member(z,adjoin(a,adjoin(b,adjoin(c,empty)))).
```

- ***Exercise 13-54:*** Explain why we do not need a clause beginning `member(X,empty) :- ....`
- ***Exercise 13-55:*** Define set intersection.
- ***Exercise 13-56:*** Define set union.
- ***Exercise 13-57:*** Define set difference.
- ***Exercise 13-58:*** Define the (improper) subset predicate.
- ***Exercise 13-59:*** Define set equality in terms of the subset predicate.
- ***Exercise 13-60:*** Define the proper subset predicate in terms of the subset predicate and the equality predicate.
- ***Exercise 13-61*:*** Recall our discussion of the Name and Structural Type Equivalence Rules (Chapter 5, Section 5.3). Define each in Prolog.

Difficulties with Inequality

Even term inequality is not without its difficulties. Suppose we have defined the following predicate for determining if two different individuals have the same father:

```
same_father(X,Y) :- X \= Y, father(F,X), father(F,Y).
```

Now suppose we enter the goal

```
:- same_father(cindy,A).
no
```

The correct answer is `A = victor`. What went wrong? To understand we have to trace our program. In this case, the goal unifies with the `same_father` clause by the assignments `X = cindy`, `Y = A`. This sets up the subgoal `cindy \= A`. Now recall that in most Prologs '`\=`' behaves as though it were defined

```
X \= Y :- not(X = Y).
```

Thus, to satisfy `cindy \= A` we set up the subgoal `cindy = A`. The latter is satisfiable since `A` is an unbound variable, and an unbound variable unifies with anything. Since `cindy = A` succeeds, `cindy \= A` must fail. Hence, our goal, `same_father(cindy,A)`, also must fail.

The problem is that '`\=`' is defined in terms of `not` and, as we saw before, `not` is un-

satisfiability, not logical negation. Hence, we must continually remind ourselves that '\=' does not mean "not equal"; rather, it means "not unifiable with," which is quite a different thing.

Is there any solution to these difficulties? Some dialects of Prolog (e.g., the Marseilles Prolog II system) have attempted to solve this problem by building in a much more sophisticated notion of inequality. In effect `cindy \= A` is retained as a *constraint* (sort of a negative variable binding) until such time as `A` becomes bound.[16] When this occurs the term bound to `A` can be compared with `cindy` by term equality: If they are different terms, we continue, else we backtrack. This interpretation of inequality seems to have more intuitive behavior than that found in most Prolog systems.

■ ***Exercise 13-62:*** The necessity of having to trace one's program in order to understand it is a violation of one of our principles. Name the principle and explain the nature of the violation.

13.5 EVALUATION AND EPILOG

Logic Programs Are Self-Documenting

It is too soon to be able to evaluate logic programming; there is too little experience in its use. However, it has many promising characteristics, at least in its pure form. In this section we review these characteristics, both in the case of *pure* (Horn clause) logic programming and in the case of the logic-oriented language Prolog.

One of the benefits of logic programming is its high level and application orientation. Since programs are described in terms of predicates and individuals of the problem domain, programs are transparent and almost self-documenting. This is apparent in the kinship (Figure 13.1) and symbolic differentiation (Figure 13.2) programs. This characteristic promotes clear, rapid, accurate programming.

Pure Logic Programming Separates Logic and Control

Pure logic programming's separation of logic and control issues simplifies program development by permitting the correctness and performance of a program to be addressed as separate problems. One important result of this separation is that correctness proofs are greatly simplified because they have to deal only with the logic component of a program. It also means that programs can be optimized with confidence since changes in the control discipline cannot affect the correctness of a program.

This separation is an important application of the Orthogonality Principle. The knowledge that is used in solving a problem (the logic) is clearly separated from the way it is used (the control).

[16] The actual Prolog II semantics is more complicated than this.

Prolog Falls Short of Logic Programming

Prolog wears its control strategy on its sleeve. Prolog programmers must be intimately aware of it in every clause they write. This influences the style of Prolog programming, and so many Prolog programmers go far beyond mere awareness of Prolog's control strategy, and make explicit use of its idiosyncrasies. The resulting programs have little to do with logic. On the contrary, the necessity of reasoning about the order of events as Prolog works its way forward and backward through the program rivals in difficulty the temporal reasoning required with the most imperative of conventional programming languages.

Researchers in logic programming are well aware of the problems described above. As Clocksin and Mellish (1984) say, "Among the highest priorities of workers in this area is to develop a practical system that does not need the 'cut' and has a version of **not** that exactly corresponds to the logical notion of negation." Hence, we may hope that future logic-oriented languages will retain the efficiency of Prolog while preserving the benefits of pure logic programming.

Implementation Techniques Are Improving

Even though Prolog's deviation from pure logic programming is motivated by performance considerations, it is still far from being an efficient programming language. This characteristic limits its use to two kinds of applications: (1) those in which performance is not important and (2) those that are so complicated that they might have been unimplementable in a conventional language. Prolog is often unacceptable for the vast middle ground of moderate-sized programs with moderate performance requirements.

However, new methods of interpreting and compiling Prolog programs are being invented. Some current compilers produce code comparable in efficiency to LISP. While this still leaves a lot to be desired, it does point the way to Prolog systems whose performance will compete with conventional languages.

Prolog Is a Step toward Nonprocedural Programming

Perhaps the most important characteristic of pure logic programs is that they are examples of nonprocedural programming. People have discussed the possibility of nonprocedural programming for many years, but pure logic programs are the first that can really lay claim to the title. Thus, they prove the possibility of nonprocedural programming.

Unfortunately Prolog falls short of this standard. Its depth-first search strategy requires programmers to think in terms of the operations that must be performed and their proper ordering in time. Indeed this reasoning is more difficult than in conventional languages due to Prolog's more complicated control regime. Perhaps the conclusion we may draw is that while Prolog was a step in the right direction, there is still much important work to be done before nonprocedural programming will be practical.

EXERCISES

1*. Read and critique at least one paper from the "Proceedings of a Symposium on Very High Level Languages" (*SIGPLAN Notices 9*, 4, April 1974).

2. Implement association lists in Prolog.

3*. Define a Prolog data structure for representing Prolog programs. Write an editor for these data structures in Prolog.

4*. Does Prolog have data types? Discuss the notion of typing in Prolog.

5*. Read and critique Kowalski's "Algorithm = Logic + Control" (Kowalski, 1979).

6*. Discuss how the procedural interpretation of Prolog could be used as the basis of a Prolog implementation.

7*. Describe how pure logic programs could be executed in parallel to increase their performance. How could Prolog be modified to permit parallel execution?

8*. Suggest architectural features that could improve the performance of Prolog programs.

9*. Evaluate Prolog's syntax.

10*. Compare and contrast function-oriented programming (e.g., LISP), object-oriented programming (e.g., Smalltalk), and logic-oriented programming (e.g., Prolog).

11*. Attack or defend this statement: Prolog makes all other programming languages obsolete.