- Synchronization involving control

Correctness of concurrent programs refers not only to getting the "right" answer but also to the rate of progress of processes. Processes can deadlock, each waiting for the other to proceed.

Chapter 12 deals with basic concepts, illustrated using the Ada programming language.

11

Logic Programming

> The programming language, Prolog, was born of a project aimed not at producing a programming language but at processing natural languages; in this case French. ··· It can be said that Prolog was the offspring of a successful marriage between natural language processing and automated theorem proving.
>
> – *Colmerauer and Roussel [1993], on "The Birth of Prolog."*

The concept of logic programming is linked historically to a language called Prolog, developed in 1972 and still the only widely available language of its kind. Prolog was first applied to natural language processing. It has since been used for specifying algorithms, searching databases, writing compilers, building expert systems — in short, for all the kinds of applications for which a language like Lisp might be used. Prolog is especially suited to applications involving pattern matching, backtrack searching, or incomplete information.

logic programs consist of facts and rules

Kowalski [1979b] illustrates the division of labor in logic programming by writing the informal equation

computation is deduction

algorithm = logic + control

Here logic refers to the facts and rules specifying what the algorithm does, and control refers to how the algorithm can be implemented by applying the rules in a particular order. This equation reflects a division of labor between us as programmers and a language for logic programming. We supply the logic part, and the programming language supplies the control.

Prolog has spawned numerous dialects, some with their own notions of control. The language in this chapter is Edinburgh Prolog, a de facto standard

dialect. Nonsyntactic differences between dialects can be illustrated by writing a family of equations

$$\text{algorithm}_D = \text{logic} + \text{control}_D$$

where D is a dialect and control_D represents its notion of control.

Control in Edinburgh Prolog proceeds from left to right; see Section 11.5 for details. The rule

$$P \text{ **if** } Q_1 \text{ **and** } Q_2 \text{ **and** } \cdots \text{ **and** } Q_k.$$

$k \geq 0$, can be read as

to deduce P,
    deduce Q_1;
    deduce Q_2;
    ...
    deduce Q_k;

This simple strategy is surprisingly versatile and flexible. Unfortunately, it sometimes gets stuck in infinite loops, and it can produce anomalies involving negation.

Prolog is a practical tool. It reduces logic programming to practice. However, it introduces a few impurities that are put into perspective if we distinguish between Prolog, the language, and logic programming, the concept. This chapter therefore begins with an informal discussion of the concept of logic programming.

11.1 COMPUTING WITH RELATIONS

relations treat arguments and results uniformly

Logic programming deals with relations rather than functions. It is based on the premise that programming with relations is more flexible than programming with functions, because relations treat arguments and results uniformly. Informally, relations have no sense of direction, no prejudice about who is computed from whom.

The running example in this section is a relation *append* on lists. Although Prolog itself is not introduced until Section 11.2, we anticipate its notation for lists. Lists are written between brackets [and], so [ ] is the empty list and $[b, c]$ is a list of two symbols b and c. If H is a symbol and T is a list, then $[H \mid T]$ is a list with head H and tail T. Hence

$$[a, b, c] = [a \mid [b, c]]$$

Relations

A concrete view of a *relation* is as a table with $n \geq 0$ columns and a possibly infinite set of rows. A tuple $(a_1, a_2, \ldots, a_n)$ is *in* a relation if a_i appears in column i, $1 \leq i \leq n$, of some row in the table for the relation.

Relation *append* is a set of tuples of the form (X, Y, Z), where Z consists of the elements of X followed by the elements of Y. A few of the tuples in *append* are as follows:

append

X	Y	Z
[ ]	[ ]	[ ]
$[a]$	[ ]	$[a]$
...	...	...
$[a, b]$	$[c, d]$	$[a, b, c, d]$
...	...	...

Relations are also called *predicates* because a relation name *rel* can be thought of as a test of the form

Is a given tuple in relation *rel*?

For example, $([a], [b], [a, b])$ is in relation *append*, but $([a], [b], [\,])$ is not.

The rest of this section uses pseudo-English to talk informally about relation *append*. A summary of this section appears in Fig. 11.1; it is keyed to an interactive Prolog session that will be discussed in Section 11.3.

Rules and Facts

Horn clauses lead to efficient implementations

Relations will be specified by *rules*, written in pseudocode as

$$P \text{ **if** } Q_1 \text{ **and** } Q_2 \text{ **and** } \cdots \text{ **and** } Q_k.$$

for $k \geq 0$.[1] Such rules are called *Horn clauses*, after Horn [1951], who studied them. Languages have tended to work with Horn clauses because Horn clauses lead to efficient implementations.

A *fact* is a special case of a rule, in which $k = 0$ and P holds without any conditions, written simply as

$$P.$$

[1] Looking ahead to Section 11.2, $P, Q_1, Q_2, \ldots, Q_k$ are terms. A *term* is either a constant or a variable or has the form $rel(T_1, T_2, \ldots, T_n)$, for $n \geq 0$, where *rel* is the name of a relation and $T_1, T_2, \ldots, T_n$ are terms. By convention, variable names begin with uppercase letters; constant and relation names begin with lowercase letters.

Rules

```
append([], Y, Y).                          append [ ] and Y to get Y.
append([H|X], Y, [H|Z]) :- append(X, Y, Z).  append [H | X] and Y to get [H | Z]
                                               if append X and Y to get Z
```

Queries

```
?- append([a,b], [c,d], [a,b,c,d]).        append [a, b] and [c, d] to get [a, b, c, d]?
   yes                                        Answer: yes
?- append([a,b], [c,d], Z).                append [a, b] and [c, d] to get Z?
   Z = [a,b,c,d]                              Answer: yes, when Z = [a, b, c, d]
?- append([a,b], Y, [a,b,c,d]).            append [a, b] and Y to get [a, b, c, d]?
   Y = [c,d]                                  Answer: yes, when Y = [c, d]
?- append(X, [c,d], [a,b,c,d]).            append X and [c, d] to get [a, b, c, d]?
   X = [a,b]                                  Answer: yes, when X = [a, b]
?- append(X, [d,c], [a,b,c,d]).
   no
```

Figure 11.1 A Prolog session with comments in pseudo-English.

The *append* relation is specified by two rules. The first is a fact stating that triples of the form ([ ], Y, Y) are in relation *append*. A pseudo-English statement of this fact is

append [ ] and Y to get Y.

The second rule for *append* is shown for completeness. It uses the notation [H | T] for a list with head H and tail T:

append [H | X] and Y to get [H | Z]
if append X and Y to get Z

It follows from this rule that

append [a, b] and [c, d] to get [a, b, c, d]
if append [b] and [c, d] to get [b, c, d]

Here $H = a$, $X = [b]$, $Y = [c, d]$, and $Z = [b, c, d]$. Note that [a | [b]] is the same list as [a, b] and that [a | [b, c, d]] is the same list as [a, b, c, d].

Queries

queries are yes/fail rather than yes/no

Logic programming is driven by queries about relations. The simplest queries ask whether a particular tuple belongs to a relation. The query

append [a, b] and [c, d] to get [a, b, c, d]?
Answer: yes (11.1)

asks whether the triple ([a, b], [c, d], [a, b, c, d]) belongs to relation *append*.

Horn clauses cannot represent negative information; that is, we cannot directly ask whether a tuple is not in a relation. Queries in this chapter will therefore have yes/fail answers rather than yes/no answers. The response "fail" indicates a failure to deduce a yes answer. Section 11.6 considers a limited form of negation based on failure.

Queries containing variables are much more interesting:

Is there a Z such that
append [a, b] and [c, d] to get Z?
Answer: yes, when Z = [a, b, c, d] (11.2)

What seems like a yes/fail query is really a request for suitable values for the variables in it. The query (11.2) is a request for a Z such that ([a b], [c, d], Z) is in the relation *append*.

A benefit of working with relations is that if we append X and Y to get Z, then any one of X, Y, and Z can be computed from the other two. This property motivates the earlier remark that relations are flexible because they have no prejudice about who is computed from whom. X can be computed from Y and Z:

Is there an X such that
append X and [c, d] to get [a, b, c, d]?
Answer: yes, when X = [a, b] (11.3)

Or Y can be computed from X and Z:

Is there a Y such that
append [a, b] and Y to get [a, b, c, d]?
Answer: yes, when Y = [c, d] (11.4)

Queries (11.1)–(11.4) illustrate several different ways of using the same relation *append*.

prefix and suffix can be defined in terms of append

New relations can be defined from old. In the following three rules for *prefix*, *suffix*, and *sublist*, the variables S, X, Y, and Z refer to portions of a list (see Fig. 11.2):

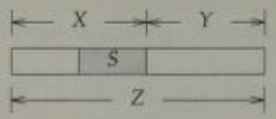


Figure 11.2 Variable names referring to portions of a list.

prefix X of Z
if for some Y, append X and Y to get Z.

suffix Y of Z
if for some X, append X and Y to get Z.

sublist S of Z
if for some X, prefix X of Z **and** suffix S of X.

11.2 INTRODUCTION TO PROLOG

This section introduces Prolog by considering relations on atomic objects. Data structures are considered in the next section; programs that benefit from Prolog's unique abilities appear in Section 11.4.

The examples in this section are motivated by the arrows or *links* in Fig. 11.3.

Terms

Facts, rules, and queries are specified using terms; see Fig. 11.4 for the basic syntax of Edinburgh Prolog.

A *simple term* is a *number*, a *variable* starting with an uppercase letter, or an *atom* standing for itself. Examples of simple terms are

```
0  1972  X  Source  lisp  algol60
```

Here `0` and `1972` are numbers, `X` and `Source` are variables, and `lisp` and `algol60` are atoms.

A *compound term* consists of an atom followed by a parenthesized sequence of subterms. The atom is called a *functor* and the subterms are called *arguments*. In

```
link(bcpl, c)
```

the functor is `link`, and the arguments are `bcpl` and `c`.

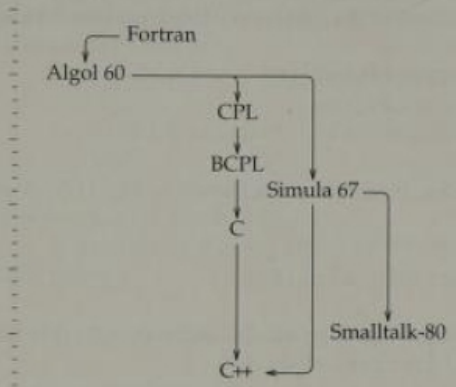


Figure 11.3 Links between languages.

A few extensions to the syntax of compound terms will be introduced as needed. Some operators can be written in infix as well as prefix notation; for example, the prefix notation `=(X,Y)` can equivalently be rewritten as `X=Y`.

The special variable "`_`" is a placeholder for an unnamed term. All occurrences of `_` are independent of each other.

Interacting with Prolog

A snapshot of an interactive session winds its way through the rest of this section. As in earlier chapters, system responses appear in italic letters. When started, the system responds with the "prompt" characters

```
?-
```

⟨*fact*⟩ ::= ⟨*term*⟩ .
⟨*rule*⟩ ::= ⟨*term*⟩ :- ⟨*terms*⟩ .
⟨*query*⟩ ::= ⟨*terms*⟩ .

⟨*term*⟩ ::= ⟨*number*⟩ | ⟨*atom*⟩ | ⟨*variable*⟩ | ⟨*atom*⟩ (⟨*terms*⟩)

⟨*terms*⟩ ::= ⟨*term*⟩ | ⟨*term*⟩ , ⟨*terms*⟩

Figure 11.4 Basic syntax of facts, rules, and queries in Edinburgh Prolog.

to indicate that a query is expected.

Prolog maintains a current database of rules

The *consult* construct reads in a file containing facts and rules, and adds its contents at the end of the current database of rules.[2] Thus,

```
?- consult(links).
   links consulted ···
   yes
```

reads file `links`; its contents are shown in Fig. 11.5. A sequence of facts starting with

```
link(fortran, algol60).
```

specifies a relation `link` on atoms. According to this fact, relation `link` contains the pair `(fortran, algol60)`.

Existential Queries

queries are answered with solutions

A query

$$\langle term \rangle_1 \ , \ \langle term \rangle_2 \ , \ \cdots \ , \ \langle term \rangle_k \ .$$

for $k \geq 1$, corresponds to the following pseudocode:

$\langle term \rangle_1$ **and** $\langle term \rangle_2$ **and** $\cdots$ **and** $\langle term \rangle_k$?

```
link(fortran,   algol60).
link(algol60,   cpl).
link(cpl,       bcpl).
link(bcpl,      c).
link(c,         cplusplus).
link(algol60,   simula67).
link(simula67,  cplusplus).
link(simula67,  smalltalk80).

path(L, L).
path(L, M) :- link(L, X), path(X, M).
```

Figure 11.5 Facts and rules in file `links`.

---

[2] Use *reconsult* to override rules in the database. Some implementations allow rules to be entered directly by consulting the special file name *user*.

Queries are also called *goals*. It is sometimes convenient to refer to the individual terms in a query as "subgoals." There is no formal distinction between a goal and a subgoal, however, just as there is no formal distinction between a term and a subterm.

Since there are no variables in the query

```
?- link(cpl,bcpl), link(bcpl,c).
   yes
```

the response is simply `yes`.

A variable in a query refers to the existence of some appropriate object. The query

```
?- link(algol60,L), link(L,M).
```

can therefore be read as

Are there L and M such that
link(*algol*60, L) **and** *link*(L, M)?

A *solution* to a query is a binding of variables to values that makes the query true.[3] A query with solutions is said to be *satisfiable*. The system responds with a solution to a satisfiable query:

```
?- link(algol60,L), link(L,M).
   L = cpl
   M = bcpl
```

We now have two choices:

- Type a carriage return. Prolog responds with `yes` to indicate that there might be more solutions. It then immediately prompts for the next query.
- Type a semicolon and a carriage return. Prolog responds with another solution, or with `no` to indicate that no further solutions can be found.

The semicolons in the following interaction keep asking for further solutions:

```
?- link(algol60,L), link(L,M).
   L = cpl
   M = bcpl ;
```

[3] Technically, all variables in a query are implicitly existentially quantified. With the quantifiers in place, this query becomes

$\exists L, M.$ *link*(*algol*60, L) **and** *link*(L, M)?

```
L = simula67
M = cplusplus ;

L = simula67
M = smalltalk80 ;

no
```

Variables can appear anywhere within a query. The query

```
?- link(L, bcpl).
```

asks for an object with a link to `bcpl`, and the query

```
?- link(bcpl, M).
```

asks for an object to which `bcpl` has a link.

Universal Facts and Rules

rules are Horn clauses

A rule

$$\langle term \rangle \quad :- \quad \langle term \rangle_1 \ , \ \langle term \rangle_2 \ , \ \cdots \ , \ \langle term \rangle_k \ .$$

for $k \geq 1$, corresponds to the following pseudocode:

$$\langle term \rangle \quad \textbf{if} \quad \langle term \rangle_1 \quad \textbf{and} \quad \langle term \rangle_2 \quad \textbf{and} \quad \cdots \quad \textbf{and} \quad \langle term \rangle_k .$$

The term to the left of the `:-` is called the *head* and the terms to the right of the `:-` are called *conditions*.

A fact is a special case of a rule. A fact has a head and no conditions.

The following fact and rule specify a relation `path`:

```
path(L, L).                                  (11.5)
path(L, M) :- link(L, X), path(X, M).        (11.6)
```

The idea is that a path consists of zero or more links. We take a path of zero links to be from `L` to itself. A path from `L` to `M` begins with a link to some `X` and continues along the path from `X` to `M`.

Any object can be substituted for a variable in the head of a rule. Fact (11.5) can therefore be read as

For all L,
  $path(L, L)$.

Rule (11.6) has a variable `X` that appears in the conditions but not in the head. Such variables stand for some object satisfying the conditions.[4] Rule (11.6) can therefore be read as

For all L and M,
  $path(L, M)$ **if**
    there exists X such that
      $link(L, X)$ **and** $path(X, M)$.

Negation as Failure

no means "I can't prove it"

Prolog answers `no` to a query if it fails to satisfy the query. The *negation as failure* assumption is tantamount to saying, "If I can't prove it, it must be false."

The facts in Fig. 11.5 say nothing about a link from `lisp` to `scheme`, hence the `no` answer in the following:

```
?- link(lisp,scheme).
no
```

Similarly, the `not` operator represents negation as failure rather than true logical negation. A query `not(P)` is treated as true if the system fails to deduce P. Negation as failure works for simple cases; more complex cases are dealt with in Section 11.6. The following example contains an application for `not`.

Example 11.1 Are there two languages L and M in Fig. 11.3 with links to the same language N? A first attempt at this query is

```
?- link(L,N), link(M,N).
L = fortran
N = algol60
M = fortran
```

Let us now add the requirement that variables `L` and `M` must have different values:

```
?- link(L,N), link(M,N), not(L=M).
L = c
N = cplusplus
M = simula67 ;
```

[4] Technically, all variables in facts and rules are implicitly universally quantified. With the quantifiers in place, the rule becomes

$\forall L, M, X.$ $path(L, M)$ **if** ($link(L, X)$ **and** $path(X, M)$)

Since X does not appear in its head, the rule is logically equivalent to

$\forall L, M.$ $path(L, M)$ **if** ($\exists X.$ $link(L, X)$ **and** $path(X, M)$)

```
L = simula67
N = cplusplus
M = c ;

no
```

subgoal order affects solutions

These two solutions are different because the individual variables have different values.

As a rule of thumb, `not` can be used to test known values or values that become known before `not` is applied. The reordered query

```
?- not(L=M), link(L,N), link(M,N).
   no
```

fails because the values of variables `L` and `M` are not known at the start of the query. Unknown values could be equal, so `not(L=M)` fails. □

Unification

How does Prolog solve equations of the following form

```
?- f(X,b) = f(a,Y).
   X = a
   Y = b
```

An *instance* of a term T is obtained by substituting subterms for one or more variables of T. The same subterm must be substituted for all occurrences of a variable.

Thus, `f(a,b)` is an instance of `f(X,b)` because it is obtained by substituting subterm `a` for variable `X` in `f(X,b)`. Similarly, `f(a,b)` is an instance of `f(a,Y)` because it is obtained by substituting subterm `b` for variable `Y` in `f(a,Y)`.

As another example, `g(a,a)` is an instance of `g(X,X)`, and so is `g(h(b),h(b))`. However, `g(a,b)` is not an instance of `g(X,X)` because we cannot substitute `a` for one occurrence of `X` and a different subterm `b` for the other occurrence of `X`.

computation is based on unification

Deduction in Prolog is based on the concept of unification; the two terms T_1 and T_2 *unify* if they have a common instance U. If a variable occurs in both T_1 and T_2, then the same subterm must be substituted for all occurrences of the variable in both T_1 and T_2.

Terms `f(X,b)` and `f(a,Y)` unify because they have a common instance `f(a,b)`.

Unification occurs implicitly when a rule is applied. Suppose that the relation `identity` is defined by the fact


```
identity(Z,Z).
```

Now unification is used to compute the response to the query

```
?- identity( f(X,b), f(a,Y) ).
   X = a
   Y = b
```

The response is computed by unifying `identity(Z,Z)` with

```
identity( f(X,b), f(a,Y) )
```

which leads to the unification of `Z` with `f(X,b)` and with `f(a,Y)`. In effect, `f(X,b)` is unified with `f(a,Y)`.

Arithmetic

The = operator stands for unification in Prolog, so

```
?- X = 2+3.
   X = 2+3
```

simply binds variable `X` to the term `2+3`.

The infix `is` operator evaluates an expression:

```
?- X is 2+3.
   X = 5
```

Since the `is` operator binds `X` to `5`, the query

```
?- X is 2+3, X = 5.
   X = 5
```

is satisfied. However,

```
?- X is 2+3, X = 2+3.
   no
```

fails because `2+3` does not unify with `5`. Term `2+3` is the application of operator `+` to arguments `2` and `3`, whereas `5` is simply the integer `5`. Hence, `2+3` does not unify with `5`.

11.3 DATA STRUCTURES IN PROLOG

Prolog supports several notations for writing Lisp-like lists. After reviewing the notations, we see that they are just syntactic sugar for ordinary terms; that is, they sweeten the syntax without adding any new capabilities.

The view of terms as data carries over from lists to other data structures, particularly trees.

Lists in Prolog

list notation is a way of writing terms

The simplest way of writing a list is to enumerate its elements. The list consisting of the three atoms `a`, `b`, and `c` can be written as

```
[a, b, c]
```

The empty list is written as `[]`.

We can also specify an initial sequence of elements and a trailing list, separated by `|`. The list `[a,b,c]` can also be written as

```
[a, b, c | []]
[a, b | [c]]
[a | [b, c]]
```

A special case of this notation is a list with head `H` and tail `T`, written as `[H|T]`. The head is the first element of a list, and the tail is the list consisting of the remaining elements.

Unification can be used to extract the components of a list, so explicit operators for extracting the head and tail are not needed. The solution of the query

```
?- [H|T] = [a,b,c].
   H = a
   T = [b,c]
```

binds variable `H` to the head and variable `T` to the tail of list `[a,b,c]`. The query

```
?- [a|T] = [H, b, c].
   T = [b,c]
   H = a
```

illustrates Prolog's ability to deal with partially specified terms. The term `[a|T]` is a partial specification of a list with head `a` and unknown tail denoted by variable `T`. Similarly, `[H,b,c]` is a partial specification of a list with unknown head `H` and tail `[b,c]`. In order for these two specifications to unify, `H` must denote `a` and `T` must denote `[b,c]`.

Example 11.2 The append relation on lists is defined by the following rules:

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

These rules are Prolog counterparts of the pseudo-English rules in Fig. 11.1. In words, the result of appending the empty list `[]` and a list `Y` is `Y`. If the result of appending `X` and `Y` is `Z`, then the result of appending `[H|X]` and `Y` is `[H|Z]`.

The responses to the following queries show that the rules for `append` can be used to compute any one of the arguments from the other two:

```
?- append([a,b], [c,d], Z).
   Z = [a,b,c,d]

?- append([a,b], Y, [a,b,c,d]).
   Y = [c,d]

?- append(X, [c,d], [a,b,c,d]).
   X = [a,b]
```

The following query shows that inconsistent arguments are rejected

```
?- append(X, [d,c], [a,b,c,d]).
   no
```

Difference lists, to be discussed in Section 11.4, lead to a faster implementation of append. □

Terms as Data

The connection between lists and terms is as follows. `[H|T]` is syntactic sugar for the term `.(H,T)`:

```
?- .(H, T) = [a,b,c].
   H = a
   T = [b,c]
```

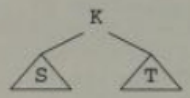
Thus, the dot operator or functor "`.`" corresponds to cons in Lisp, and lists are terms. The term for the list `[a,b,c]` is

```
.(a, .(b, .(c, [])))
```

There is a one-to-one correspondence between trees and terms. That is, any tree can be written as a term and any term can be drawn as a tree. Any data structure that can be simulated using trees can therefore be simulated using terms. Using an example from Section 9.5, binary trees can be written as terms, using an atom `leaf` for a leaf and a functor `nonleaf` with two arguments for a nonleaf node, as in Fig. 11.6.

Example 11.3 A *binary search tree* is either empty, or it consists of a node with two binary search trees as subtrees. Each node holds an integer (see Fig. 11.7). The elements in a binary search tree are arranged so that smaller elements appear in the left subtree of a node and larger elements appear in the right subtree.

Let atom `empty` represent an empty binary search tree and let a term `node(K,S,T)` represent a tree



with an integer value `K` at the root, left subtree `S`, and right subtree `T`.

The rules in Fig. 11.8 define a relation `member` to test whether an integer appears at some node in a tree. The two arguments of `member` are an integer and a tree. The fact

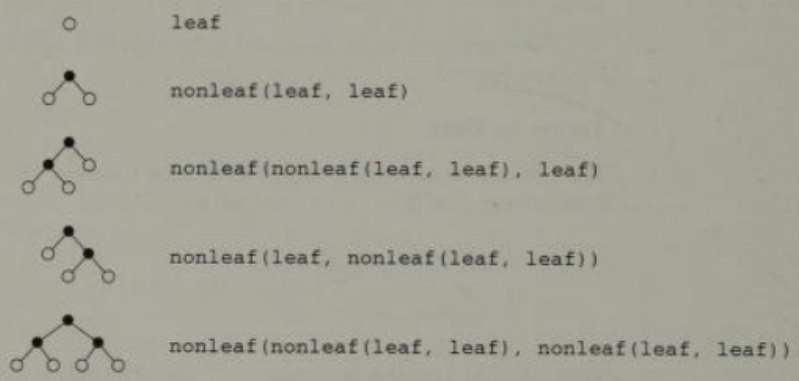


Figure 11.6 Terms for representing binary trees.

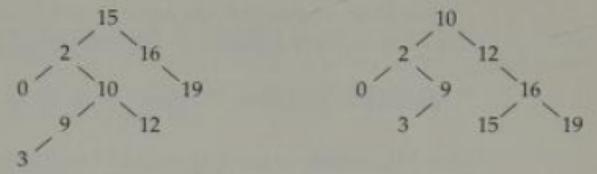


Figure 11.7 Two binary search trees.

```
member(K, node(K,_,_)).
```

can be interpreted as saying that `K` appears in a tree if it appears at the root. Each occurrence of the special variable _ is a placeholder for a distinct unnamed term, so `node(K,_,_)` represents a binary search tree with `K` at the root and some unnamed left and right subtrees. We can emphasize that `member` is a relation on pairs consisting of an integer `K` and a tree `U` by restating the preceding rule as

```
member(K, U) :- U = node(N,S,T), K = N.
```

The rule

```
member(K, node(N,S,_)) :- K < N, member(K, S).
```

can be interpreted as

K is in a tree *node*(*N*, *S*, _) **if**
K < *N* **and** *K* is in the left subtree *S*.

It is left as an exercise to define a relation `insert` corresponding to

insert *K* into *S* to get *T*.

```
member(K, node(K,_,_)).
member(K, node(N,S,_)) :- K < N, member(K, S).
member(K, node(N,_,T)) :- K > N, member(K, T).
```

Figure 11.8 Relation `member` on binary search trees.

and a relation `delete` to remove an integer from a tree. □

The atom `empty` and the functor `node` in Example 11.3 simulate the value constructors *empty* and *node* in the following ML datatype declaration:

datatype *searchtree* = *empty* | *node* **of** **int** * *searchtree* * *searchtree*;

Other ML datatypes can be simulated similarly.

Beyond trees, variables in Prolog allow terms to represent data structures with sharing. The term `node(K,S,S)` represents a graph

Although this chapter does not explore the possibility, terms can also represent graphs with cycles.

11.4 PROGRAMMING TECHNIQUES

The programming techniques in this section exploit the strengths of Prolog—namely, backtracking and unification. Backtracking allows a solution to be found if one exists. Unification allows variables to be used as placeholders for data to be filled in later.

Careful use of the techniques in this section can lead to efficient programs. The programs in this section rely on the left-to-right evaluation of subgoals in Prolog. Therefore, variants of the programs may not work, for reasons that will become clear when control in Prolog is discussed in Section 11.5.

Guess and Verify

the guess subgoal generates solutions, the verify subgoal chooses satisfiable ones

A *guess-and-verify* query has the form

Is there an *S* such that
guess(*S*) **and** *verify*(*S*)?

where *guess*(*S*) and *verify*(*S*) are subgoals. Prolog responds to such a query by generating solutions to *guess*(*S*) until a solution satisfying *verify*(*S*) is found. Such queries are also called *generate-and-test* queries.

Similarly, a *guess-and-verify* rule has the following form:

conclusion(···) **if** *guess*(···, *S*, ···) **and** *verify*(···, *S*, ···)

Example 11.4 The guess-and-verify rule in this example is as follows:


```
overlap(X, Y) :- member(M, X), member(M, Y).
```

In words, two lists `X` and `Y` overlap if there is some `M` that is a member of both `X` and `Y`. The first goal `member(M,X)` guesses an `M` from list `X`, and the second goal `member(M,Y)` verifies that `M` also appears in list `Y`.

The rules for `member` are

```
member(M, [M |_]).
member(M, [_ |T]) :- member(M, T).
```

The first rule says that `M` is a member of a list with head `M`. The second rule says that `M` is a member of a list if `M` is a member of its tail `T`.

To see why

```
?- overlap([a,b,c,d], [1,2,c,d]).
   yes
```

produces a *yes* response, consider the query

```
?- member(M, [a,b,c,d]), member(M, [1,2,c,d]).
```

The first goal in this query generates solutions and the second goal tests to see whether they are acceptable. The solutions generated by the first goal are

```
?- member(M, [a,b,c,d]).
   M = a ;
   M = b ;
   M = c ;
   M = d ;
   no
```

The first two are not acceptable, but the third is

```
?- member(a, [1,2,c,d]).
   no
?- member(b, [1,2,c,d]).
   no
?- member(c, [1,2,c,d]).
   yes
```

□

Hint. Since computation in Prolog proceeds from left to right, the order of the subgoals in a guess-and-verify query can affect efficiency. Choose the subgoal with fewer solutions as the guess goal.

As an extreme example of the effect of goal order on efficiency, consider the following two queries:

```
?- X = [1,2,3], member(a,X).
   no

?- member(a,X), X = [1,2,3].
   [ infinite computation ]
```

Relation `member` is as in Example 11.4. In

```
?- X = [1,2,3], member(a,X).
   no
```

the guess goal `X=[1,2,3]` has just one solution, and this solution does not satisfy `member(a,X)` because `a` is not in the list `[1,2,3]`. On the other hand, in

```
?- member(a,X), X = [1,2,3].
   [ infinite computation ]
```

the guess goal `member(a,X)` has an infinite number of solutions:

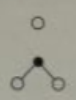
```
X = [a|_] ;          a can be the first element of X
X = [_,a,|_] ;       a can be the second element of X
X = [_,_,a,|_] ;     a can be the third element of X
  ...
```

none of which binds `X` to the list `[1,2,3]`. Hence, an infinite computation ensues as Prolog tries the solutions in turn, looking for one that satisfies `X=[1,2,3]`.

Variables as Placeholders in Terms

So far in this chapter, variables have been used in rules and queries but not in terms representing objects. Terms containing variables can be used to simulate modifiable data structures; the variables serve as placeholders for subterms to be filled in later. Such terms will be used in Example 11.5 to implement queues efficiently.

Recall the use of terms to represent binary trees in Section 11.3:



`leaf`

`nonleaf(leaf, leaf)`

The terms `leaf` and `nonleaf(leaf,leaf)` are completely specified. By contrast, the list `[a,b|X]` containing a variable `X` is partially specified because we do not yet know what `X` represents. This list `[a,b|X]` has `a` as its first element and `b` as its second, and it has a variable `X` representing the rest of the list. If `X` is subsequently unified with `[]`, then `[a,b|X]` will represent `[a,b]`, but if `X` is unified with `[c]`, then `[a,b|X]` will represent `[a,b,c]`, and so on, for other possible values for `X`.

an open list can be extended through its end marker

An *open list* is a list ending in a variable, referred to as the *end marker variable* of the list. An empty open list consists of just an end-marker variable. A list is *closed* if it is not open.

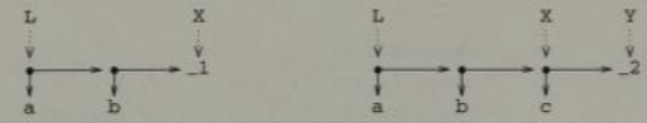
Internally, Prolog uses machine-generated variables, written with a leading underscore _ followed by an integer. In the following interaction, the machine-generated variable `_1` corresponds to the end marker `X`:

```
?- L = [a,b|X].
   L = [a,b|_1]
   X = _1
```

Prolog generates fresh variables each time it responds to a query or applies a rule.

An open list can be modified by unifying its end marker. The following query extends `L` into a new open list with end marker `Y` (see Fig. 11.9):

```
?- L = [a,b|X], X = [c,Y].
   L = [a,b,c|_2]
   X = [c|_2]
   Y = _2
```



(a) Before `X` is bound. (b) After `X = [c|Y]`.

Figure 11.9 Extending an open list by unifying its end marker.

Unification of an end-marker variable is akin to an assignment to that variable. In Fig. 11.9, list `L` changes from `[a,b|_1]` to `[a,b,c|_2]` when `_1` unifies with `[c|_2]`.

An advantage of working with open lists is that the end of a list can be accessed quickly, in constant time, through its end marker. The following example uses open lists to implement queues.

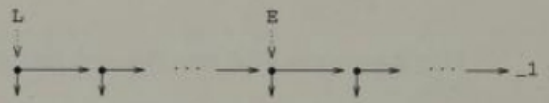
Example 11.5 This example discusses the rules in Fig. 11.10 for manipulating queues. The relation `enter(a,Q,R)` is described informally by

> When element `a` enters queue `Q`, we get queue `R`.

Similarly, `leave(a,Q,R)` is described by

> When element `a` leaves queue `Q`, we get queue `R`.

When a queue is created, it is represented by a term of the form `q(L,E)`, where `L` is an open list with end marker `E`. Subsequent operations will, in general, extend the list `L`, as in the following diagram:



Therefore, `L` in `q(L,E)` represents an open list, `E` represents some suffix of `L`, and the contents of the queue `q(L,E)` are the elements of `L` that are not in `E`.

The implementation of queues is illustrated in Fig. 11.11, which shows the effect of the query

```
?- setup(Q), enter(a,Q,R), enter(b,R,S),
       leave(X,S,T), leave(Y,T,U), wrapup(U).
```

```
setup(q(X,X)).
enter(A, q(X,Y), q(X,Z)) :- Y = [A|Z].
leave(A, q(X,Z), q(Y,Z)) :- X = [A|Y].
wrapup(q([],[])).
```

Figure 11.10 Rules for manipulating queues.

The first goal `setup(Q)` creates an empty queue:

```
?- setup(Q).
   Q = q(_1,_1)
```

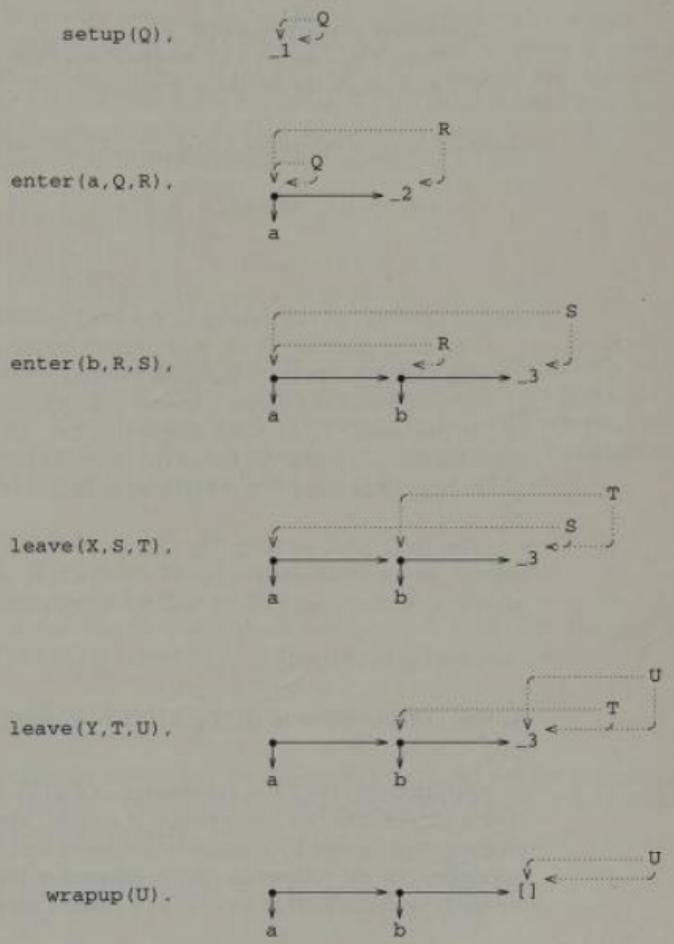


Figure 11.11 Operations on a queue.

In Fig. 11.11, a queue `q(L,E)` is marked by dotted arrows to `L` and `E`. The arrows from `Q` therefore go to the empty open list `_1` with end marker `_1`.

Now consider the second goal `enter(a,Q,R)`. The rule

```
enter(A, q(X,Y), q(X,Z)) :- Y = [A|Z].
```

can be read as

> To enter A into a queue $q(X, Y)$,
> bind Y to a list $[A \mid Z]$, where Z is a fresh end marker,
> and return the resulting queue $q(X, Z)$.

After `setup(Q)` initializes `Q` to `q(_1,_1)`, the second goal in

```
?- setup(Q), enter(a,Q,R).
   Q = ···
   R = q([a|_2],_2)
```

unifies `_1` with `[a|_2]`, where `_2` is a fresh end marker.

The third goal `enter(b,R,S)` enters `b` into `q([a|_2],_2)` by unifying `_2` with `[b|_3]`, where `_3` is a fresh variable.

When an element leaves a queue `q(L,E)`, the resulting queue has the tail of `L` in place of `L`. Note in the diagram to the right of `leave(X,S,T)` that the open list for queue `T` is the tail of the open list for `S`. Similarly, in the diagram to the right of `leave(Y,T,U)`, the open list for `U` is the tail of the open list for `T`.

The final goal `wrapup(U)` checks that the enter and leave operations leave `U` in an initial state `q(L,E)`, where `L` is an empty open list with end marker `E`. Otherwise, `q(L,E)` will not unify with `q([],[])` in the fact

```
wrapup(q([],[])).
```

In Fig. 11.11, `U` refers to `q(_3,_3)` and `wrapup(q(_3,_3))` unifies `_3` with `[]`. □

Surprisingly, the rules for queues in Fig. 11.10 support "deficit" queues, which an element can leave before it enters. More precisely, an unspecified element represented by a variable can leave and be later filled in by an enter operation. In the following query, element `X` leaves the initial queue before we learn from the goal `enter(a,R,S)` that `X` represents `a`:

```
?- setup(Q), leave(X, Q, R), enter(a, R, S), wrapup(S).
   Q = q([a],[a])
   X = a
   R = q([],[a])
   S = q([],[])
```


Difference Lists

a difference list consists of a list and its suffix

Applications that use lists can be adapted to use open lists instead. Open-list versions require more care, but they can be more efficient. Care is needed because an open list changes when its end marker is unified (see again Fig. 11.9). Difference lists are a technique for coping with such changes.

A *difference list* is made up of two lists `L` and `E`, where `E` unifies with a suffix of `L`. The contents of the difference list consist of the elements that are in `L` but not in `E`. We write this difference list as `dl(L,E)`. The lists `L` and `E` can be either open or closed. They are typically open.

Examples of difference lists with contents `[a,b]` are

```
dl([a,b], []).
dl([a,b,c], [c]).
dl([a,b|E], E).
dl([a,b,c|F], [c|F]).
```

In effect, the variables in `dl(L,E)` allow us to refer directly to the endpoints of its contents. The append operation on difference lists can be implemented in constant time using a nonrecursive rule. An informal reading of the following rule is that if `X` extends from `L` up to `M` and `Y` extends from `M` to `N`, then `Z`, the result of appending `X` and `Y`, extends from `L` to `N`:

```
append_dl(X, Y, Z) :-
     X = dl(L,M), Y = dl(M,N), Z = dl(L,N).
```

We close this section with an example that leads into the discussion of control in Prolog in Section 11.5. It is tempting to define a rule

```
contents(X, dl(L,E)) :- append(X, E, L).
```

to formalize the notion that the contents of `dl(L,E)` are the elements that are in `L` but not in `E`. The following queries confirm that each of the following difference lists has contents `[a,b]`:

```
?- contents([a,b], dl([a,b,c], [c])).
   yes
?- contents([a,b], dl([a,b,c|F], [c|F])).
   F = _1
   yes
```

An attempt to ask for the contents of `dl([a,b|E],E)` leads to a response that will be explained in Section 11.5:

```
?- contents(X, dl([a,b|E], E)).
   X = []
   E = [a,b,a,b,a,b,a,b,a,b,a,b,a,b,a,b, ···
```

11.5 CONTROL IN PROLOG

goal order and rule order are significant

In the informal equation

algorithm = logic + control

"logic" refers to the rules and queries in a logic program and "control" refers to how a language computes a response to a query. The pseudocode in Fig. 11.12 is an overview of control in Prolog.

Control in Prolog is characterized by two decisions in Fig. 11.12:

1. *Goal order*. Choose the leftmost subgoal.
2. *Rule order*. Select the first applicable rule.

The response to a query is affected both by goal order within the query and by rule order within the database of facts and rules.

Example 11.6 The examples in this section use the rules in Fig. 11.13.

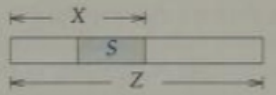
A sublist S of Z

```
start with a query as the current goal;
while the current goal is nonempty do
    choose the leftmost subgoal;
    if a rule applies to the subgoal then
        select the first applicable rule;
        form a new current goal
    else
        backtrack
    end if
end while;
succeed
```

Figure 11.12 Control in Prolog.


```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X,Y,Z).
prefix(X,Z) :- append(X,Y,Z).
suffix(Y,Z) :- append(X,Y,Z).
appen2([H|X], Y, [H|Z]) :- appen2(X,Y,Z).
appen2([], Y, Y).
```

Figure 11.13 Database of rules for examples in Section 11.5.



can be specified in the following seemingly equivalent ways:

prefix X of Z **and** suffix S of X.
suffix S of X **and** prefix X of Z.

The corresponding Prolog queries usually produce the same responses. Their responses differ, however, if S is not a sublist of Z:

```
?- prefix(X,[a,b,c]), suffix([e],X).
   no
?- suffix([e],X), prefix(X,[a,b,c]).
   [ infinite computation ]
```

We look closely at the suffix-prefix goal order in this section.

Rule order can also make a difference. New solutions are produced on demand for

```
?- append(X, [c], Z).
   X = []
   Z = [c] ;

   X = [_1]
   Z = [_1,c] ;

   X = [_1,_2]
   Z = [_1,_2,c]

   yes
```

(Recall that the `yes` means there might be more solutions.)

Relation `appen2` in Fig. 11.13 has the same rules as `append`, but they are written in the opposite order. The response

```
?- appen2(X, [c], Z).
   [ infinite computation ]
```

is also explained in this section. □

Unification and Substitutions

definition of unification

Since unification is central to control in Prolog, we now define it more formally than we did on page 436.

A *substitution* is a function from variables to terms. Let us write a substitution as a set of elements of the form $X \to T$, where variable X is mapped to term T. Unless stated otherwise, if a substitution maps X to T, then variable X does not occur in term T. The following is an example of a substitution: $\{V \to [b,c],\ Y \to [a,b,c]\}$.

$T\sigma$ is a standard notation for the result of applying substitution σ to term T. The result of applying a substitution to a term is given by

$$\begin{aligned} X\sigma &= U && \text{if } X \to U \text{ is in } \sigma \\ X\sigma &= X && \text{otherwise, for variable } X \\ (f(T_1, T_2))\sigma &= f(U_1, U_2) && \text{if } T_1\sigma = U_1,\ T_2\sigma = U_2 \end{aligned}$$

This definition generalizes to functors f with $k \geq 0$ arguments. In words, if σ contains $X \to U$, then the result of applying σ to variable X is U; otherwise $X\sigma$ is simply X. The result of applying σ to a term $f(T_1, \ldots, T_k)$, for $k \geq 0$ is obtained by applying σ to each subterm. For example,

$$\begin{aligned} Y\ \{V \to [b,c],\ Y \to [a,b,c]\} &= [a,b,c] \\ Z\ \{V \to [b,c],\ Y \to [a,b,c]\} &= Z \\ (\mathit{append}([\,],Y,Y)\ \{V \to [b,c],\ Y \to [a,b,c]\} &= \mathit{append}([\,],[a,b,c],[a,b,c]) \end{aligned}$$

A term U is an *instance* of T, if $U = T\sigma$, for some substitution σ. Terms T_1 and T_2 unify if $T_1\sigma$ and $T_2\sigma$ are identical for some substitution σ; we call σ a *unifier* of T_1 and T_2. Substitution σ is the *most general unifier* of T_1 and T_2, if for all other unifiers σ', $T_1\sigma'$ is an instance of $T_1\sigma$.[5]

[5] In the definition of most general unifier, it is enough to say that $T_1\sigma'$ is an instance of $T_1\sigma$. Since σ and σ' are both unifiers, $T_2\sigma'$ is automatically an instance of $T_2\sigma$, because $T_1\sigma = T_2\sigma$ and $T_1\sigma' = T_2\sigma'$.

The terms $\mathit{append}([\,], Y, Y)$ and $\mathit{append}([\,], [a|V], [a,b,c])$ unify because they have a common instance $\mathit{append}([\,], [a,b,c], [a,b,c])$. Their most general unifier is the substitution $\{V \to [b,c],\ Y \to [a,b,c]\}$.

Applying a Rule to a Goal

if a rule applies, then unification refines the current goal

The pseudocode in Fig. 11.14 restates control in Prolog in terms of substitutions and unifiers. We explore it by first considering an example that succeeds without backtracking. Backtracking is then described in terms of a tree representation of a computation.

In Fig. 11.14, a rule A :- $B_1, \ldots, B_n$ *applies* to a subgoal G if its head A unifies with G. Variables in the rule are renamed before unification to keep them distinct from variables in the subgoal.

Example 11.7 The response to the query

```
?- suffix([a],L), prefix(L,[a,b,c]).
   L = [a]
```

is computed without backtracking, as in Fig. 11.15. Initially, the current goal consists of the two subgoals in this query:

```
suffix([a],L), prefix(L,[a,b,c])        (11.7)
```

Choose the leftmost subgoal `suffix([a],L)`. Select the only rule that applies to this subgoal. Rename the variables in the rule to keep them distinct

---

start with a query as the current goal;
while the current goal is nonempty **do**
  let the current goal be $G_1, \ldots, G_k$, where $k \geq 1$;
  choose the leftmost subgoal G_1;
  if a rule applies to G_1 **then**
    select the first such rule A :- $B_1, \ldots, B_j$, where $j \geq 0$;
    let σ be the most general unifier of G_1 and A;
    the current goal becomes $B_1\sigma, \ldots, B_j\sigma, G_2\sigma, \ldots, G_k\sigma$
  else
    backtrack
  end if
end while;
succeed

Figure 11.14 Control in Prolog: A restatement of Fig. 11.12.

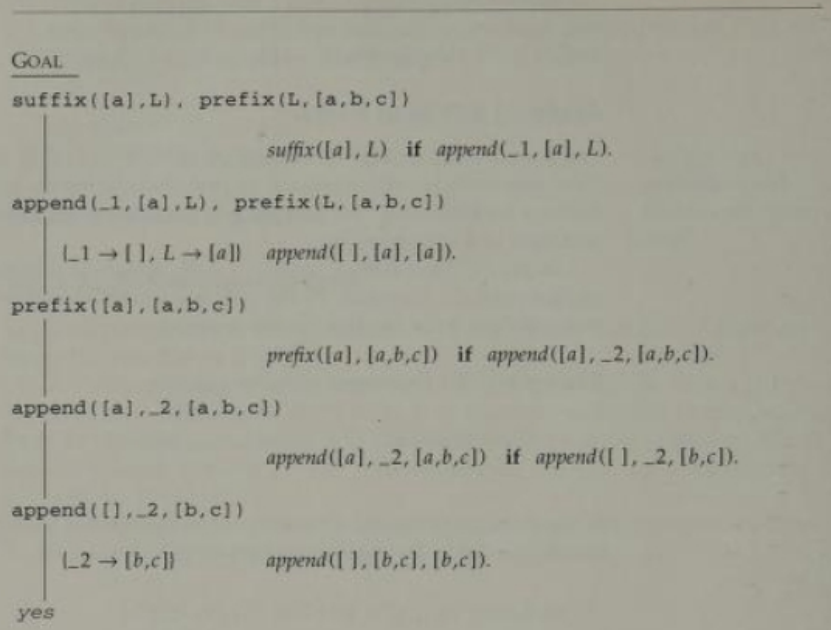


Figure 11.15 A computation that succeeds without backtracking.

from any variables in the goal. With `X'`, `Y'`, and `Z'` as the renamed variables, the rule for `suffix` becomes

```
suffix(Y', Z') :- append(X', Y', Z').
```

The substitution {`Y'` → `[a]`, `Z'` → `L`} unifies the rule head with the chosen subgoal. The rule as it applies to the subgoal `suffix([a],L)` is given by the pseudocode:

suffix([*a*], *L*) **if** *append*(_1, [*a*], *L*)

Here, [*a*] takes the place of `Y'`, *L* takes the place of `Z'`, and a Prolog-like name _1 takes the place of `X'`.

Replace the subgoal `suffix([a],L)` in (11.7) by the condition `append(_1,[a],L)` to get the new current goal:

```
append(_1,[a],L), prefix(L,[a,b,c])        (11.8)
```


The fact `append([],Y'',Y'')` applies to the new leftmost subgoal `append(_1,[a],L)` because `[]` unifies _1 and `Y''` unifies with both `[a]` and `L`. Since a fact consists of a head and no conditions, the new current goal is

```
prefix([a],[a,b,c])        (11.9)
```

Note that `[a]` has been substituted for variable `L`.

The rest of the computation can be seen from Fig. 11.15. □

Prolog Search Trees

if no rule applies, backtrack

The chain of goals in Fig. 11.15 generalizes into *Prolog search trees*, which depict computations that explore all possible solutions to a goal. Nodes in a Prolog search tree represent goals. A node has a child for each rule that applies to the leftmost subgoal at the node. The order of the children is the same as the rule order in the database of rules.[6]

A Prolog computation explores a Prolog search tree in a "depth-first" manner. It starts at the root and explores the subtrees at the children of each node from left to right. The computation produces a *yes* response each time it reaches a node in the subtree with an empty goal.

Example 11.8 A portion of the Prolog search tree for the query

```
?- suffix([b], L), prefix(L, [a,b,c]).
   L = [a,b]
```

appears in Fig. 11.16.

There is only one rule for `suffix`, so the root has only one child. The approach of Example 11.7 yields the following goal at the child of the root:

```
append(_1,[b],L), prefix(L,[a,b,c])        (11.10)
```

Both of the rules for `append` apply to the leftmost subgoal in (11.10). The node for (11.10) therefore has two children. Prolog tries rules in the order they appear in the database of rules. It therefore unifies `append(_1,[b],L)` with `append([],Y',Y')` to get the substitution

{`_1` → `[]`, `L` → `[b]`}

[6] The definition of Prolog search trees presupposes that rules are applied to the leftmost subgoal. A more general definition of search trees would allow any subgoal to be chosen as the subgoal to which rules are applied. Furthermore, a more general definition would allow rules to be selected in any order.

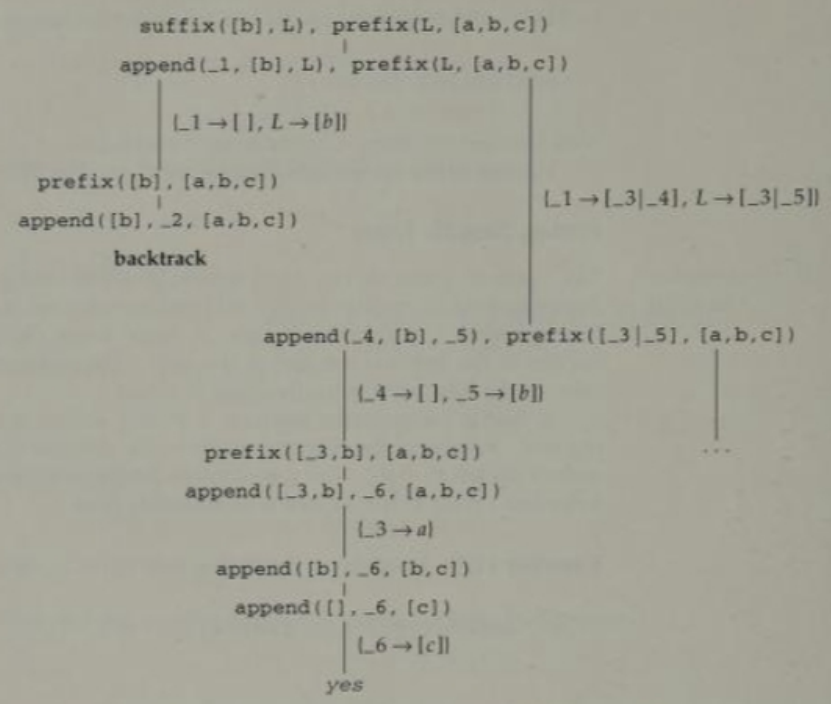


Figure 11.16 Portion of a Prolog search tree leading to a *yes* response.

Since the first rule for append is a fact, it has no conditions, so the new goal formed from (11.10) is

```
prefix([b],[a,b,c])                                    (11.11)
```

There is only one rule for `prefix`, and it leads to a new goal,

```
append([b],_2,[a,b,c])                                 (11.12)
```

Now we have a problem. This subgoal does not unify with either of the rules for append. Prolog therefore backtracks to the nearest goal with an untried rule. Such a goal is (11.10), presented again here:


```
append(_1,[b],L), prefix(L,[a,b,c])                    (11.10)
```

The first rule for append did not lead to success, so Prolog tries the second one. A suitable instance of the second rule is

```
append([_3|_4], [b], [_3|_5]) :- append(_4,[b],_5).
```

A new goal is formed from (11.10) using the substitution

```
{_1 → [_3|_4], L → [_3|_5]}
```

The new goal is

```
append(_4,[b],_5), prefix([_3|_5],[a,b,c])             (11.13)
```

The computation now proceeds without backtracking to a *yes* response in Fig. 11.16. □

Figure 11.17 presents a final restatement of control in Prolog. Procedure *visit* calls itself recursively to try new subgoals. In terms of search trees, a recursive call corresponds to visiting one of the children of a node. The recursion can stop in one of two ways:

1. The goal G is empty and **succeed** is reached.
2. No rule applies to the leftmost subgoal of G and the activation ends.

Backtracking corresponds to the latter case, in which no rule applies to G and control returns to the caller.

Goal Order Changes Solutions

rules are applied to the leftmost subgoal

The order of subgoals within a query affects the Prolog search tree for a query. The reason is that rules are always applied to the leftmost subgoal.

From Example 11.7 or from Fig. 11.15, the solution `L=[a]` to the following query is produced without backtracking. Note, however, that an infinite computation ensues if we ask for another solution.

```
?- suffix([a],L), prefix(L,[a,b,c]).
   L = [a] ;
   [ infinite computation ]
```

The leftmost subgoal

```
?- suffix([a],L).
   L = [a] ;
```

```
procedure visit(G);
begin
    if the current goal G is nonempty then
        let G be G_1, ..., G_k, where k ≥ 1;
        choose the leftmost subgoal G_1;
        for i := 1 to the number of rules do
            let rule i be A :- B_1, ..., B_j, where j ≥ 0;
            if rule i applies to G_1 then
                let σ be the most general unifier of G_1 and A;
                let G' be B_1σ, ..., B_jσ, G_2σ, ..., G_kσ;
                visit(G')
            end if
        end for
    else
        succeed
    end if
    { backtrack by returning to caller }
end visit
```

Figure 11.17 Control in Prolog as a recursive procedure.

```
L = [_1,a] ;
L = [_1,_2,a] ;
  ...
```

has an infinite number of solutions, only the first of which satisfies `prefix(L,[a,b,c])`.

In other words, the Prolog search tree for the goal

```
suffix([a],L), prefix(L,[a,b,c])
```

has exactly one *yes* node that is reached without backtracking as in Fig. 11.15. A futile search through the rest of the infinite tree ensues if we ask for a further solution.

By contrast, the reordered query

```
?- prefix(X,[a,b,c]), suffix([a],X).
   L = [a] ;
   no
```

has a finite Prolog search tree. The portion leading up to the only *yes* node appears in Fig. 11.18. The same solution `L=[a]` is reached without backtracking in Fig. 11.15 and with backtracking in Fig. 11.18. Hence, a change in goal order leads to a change in the Prolog search tree.

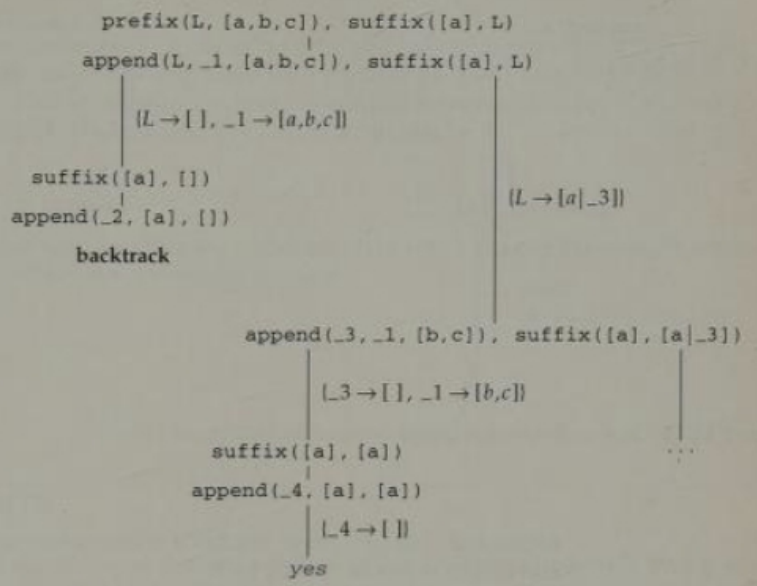


Figure 11.18 The effect of changing goal order; compare with Fig. 11.15.

Rule Order Affects the Search for Solutions

rules are applied in order

The rule order in the database of rules determines the order of the children of a node in a Prolog search tree. Rule order therefore changes the order in which solutions are reached by a Prolog computation.

The Prolog search trees in Fig. 11.19 are based on the following rule order:

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X,Y,Z).

appen2([H|X], Y, [H|Z]) :- appen2(X,Y,Z).
appen2([], Y, Y).
```

The search tree for `appen2(X,[c],Z)` is a mirror image of the search tree for `append(X,[c],Z)`. Unfortunately, the Prolog computation for

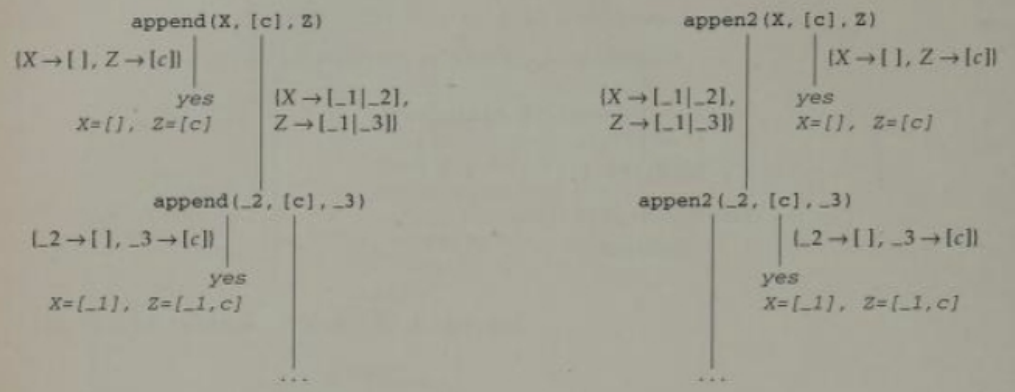


Figure 11.19 Rule order determines the order of the children of a node.

`appen2(X,[c],Z)` never reaches a solution because it keeps going deeper and deeper down an infinite path:

```
?- appen2(X, [c], Z).
   [ infinite computation ]
```

The computation for `append(X,[c],Z)`, on the other hand, goes from one solution to the next:

```
?- append(X, [c], Z).
   X = []
   Z = [c] ;

   X = [_1]
   Z = [_1,c] ;
    ...
```

The Occurs-Check Problem

occurs checks prevent cycles

In the name of efficiency, Prolog neglects to check whether a variable X occurs in a term T before it unifies X with T; such checks are called *occurs checks*. When X does indeed occur in T, then unification of X and T can lead to a nonterminating computation. For example, consider

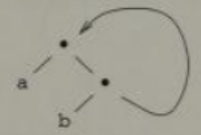

```
?- append([], E, [a,b|E]).
   E = [a,b,a,b,a,b,a,b,a,b,a,b,a,b,a,b, ···
```

For `append([],E,[a,b|E])` to unify with `append([],Y,Y)`, variable `Y` must unify with both `E` and with the term `[a,b|E]` containing `E`.

Prolog neglects to check whether `E` occurs within `[a,b|E]`. When we attempt to substitute `[a,b|E]` for `E`, we get

```
E = [a,b|E] = [a,b,a,b|E] = [a,b,a,b,a,b|E] = ···
```

Some variants of Prolog construct cyclic terms like the following if a variable is unified with a term containing it:



11.6 CUTS

cuts are procedural

Informally, a cut prunes or "cuts out" an unexplored part of a Prolog search tree. Cuts can therefore be used to make a computation more efficient by eliminating futile searching and backtracking. Cuts can also be used to implement a form of negation, something Horn clauses cannot do.

Cuts are controversial because they are impure. As we saw in Section 11.5, control in Prolog sometimes sends it into an infinite loop that could be avoided by choosing a different order of evaluation. Pure logic is order independent, so Prolog is an approximation of pure logic. Cuts make Prolog depart further from pure logic, to the point where a Prolog program must be read procedurally; that is, it must be read in terms of its computation.

A *cut*, written as `!`, appears as a condition within a rule. When a rule

$$B :- C_1, \ldots, C_{j-1}, !, C_{j+1}, \ldots, C_k$$

is applied during a computation, the cut tells control to backtrack past $C_{j-1}, \ldots, C_1$, B, without considering any remaining rules for them. We explore the implications of this remark before considering programming applications of cuts.

A Cut as the First Condition

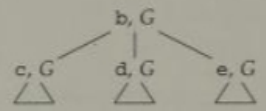
Consider rules of the form $B :- !, C$, in which a cut appears as the first condition. If the goal C fails, then control backtracks past B without considering

any remaining rules for B. Thus, the cut has the effect of making B fail if C fails.

To see the effect of a cut on a Prolog search tree, consider the following rules for `b`:

```
b :- c.
b :- d.
b :- e.
```

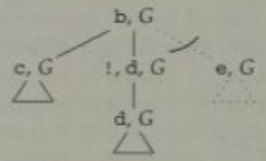
Since there are three rules for `b`, any node with `b` as the first subgoal has three children, one for each rule. Suppose that the condition at a node is `b`, G, where G represents some additional subgoals. Then the subtree rooted at the node has the following form:



Now, suppose a cut is inserted in the second rule, changing it to

```
b :- !, d.
```

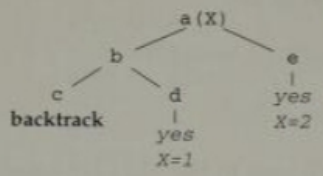
This cut eliminates the rule `b:-e` from ever being considered. The new Prolog search tree is as follows. (The dotted part is shown only for comparison; it is not part of the new search tree.)



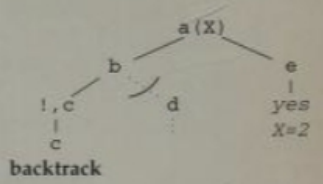
In more detail, a cut as the first subgoal in `!, d`, G is satisfied immediately, leaving `d`, G as the new goal. During backtracking, however, the cut has the side effect of eliminating the third rule `b:-e` from consideration.

Example 11.9 The following database of rules is designed specifically for the Prolog search tree in Fig. 11.20(a):

```
a(1) :- b.
a(2) :- e.
b :- c.
b :- d.
```

(a)



(b)

Figure 11.20 The effect of a cut.

```
d.
e.
```

The query `a(X)` has two solutions:

```
?- a(X).
   X = 1 ;
   X = 2 ;
   no
```

If the rule `b:-c` is changed to `b:-!,c` by inserting a cut as the first condition, the Prolog search tree changes to the one in Fig. 11.20(b). The query `a(X)` then has just one solution:

```
?- a(X).
   X = 2 ;
   no
```

□

The Effect of a Cut

cuts restrict backtracking

As mentioned earlier, when a rule

$$B :- C_1, \ldots, C_{j-1}, !, C_{j+1}, \ldots, C_k$$

is applied during a computation, the cut tells control to backtrack past $C_{j-1}, \ldots, C_1$, B, without considering any remaining rules for them.

The following example considers the effect of inserting a cut in the middle of a guess-and-verify rule. As discussed in Section 11.4, the right side of a guess-and-verify rule has the form *guess*(*S*), *verify*(*S*), where *guess*(*S*) gener-

ates potential solutions until one satisfying *verify*(S) is found. The effect of inserting a cut between them, as in

conclusion(S) :- *guess*(S), !, *verify*(S)

is to eliminate all but the first guess.

Example 11.10 The search trees in Fig. 11.21 are based on the rules in Fig. 11.22. The computation depicted by Fig. 11.21(a) begins with the succession of goals:

```
a(Z)              starting goal
b(Z)              from a(X) :- b(X).
g(Z), v(Z)        from b(X) :- g(X), v(X).
```

The subgoal `g(Z)` generates values `1`, `2`, and `3` for `Z`. Each of these values leads to a solution of the original goal `a(Z)`.

The fourth solution *Z=4* in Fig. 11.21(a) is obtained by backtracking and trying the alternative rule for `b`:

```
b(X) :- X=4, v(X).
```

We get to the final solution *Z=5* by backtracking all the way back to the original goal `a(Z)` and trying the rule

```
a(X) :- f(X).
```

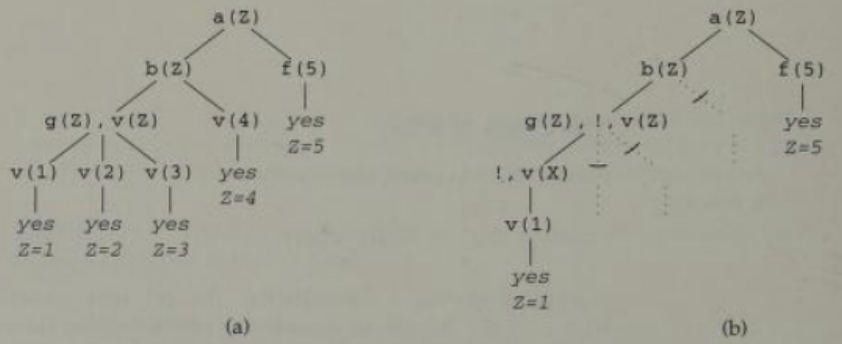


Figure 11.21 Solutions eliminated by a cut.


```
a(X) :- b(X).
a(X) :- f(X).
b(X) :- g(X), v(X).
b(X) :- X = 4, v(X).
g(1).
g(2).
g(3).
v(X).
f(5).
```

(a)

```
a(X) :- b(X).
a(X) :- f(X).
b(X) :- g(X), !, v(X).
b(X) :- X = 4, v(X).
g(1).
g(2).
g(3).
v(X).
f(5).
```

(b)

Figure 11.22 Insertion of a cut in the third rule.

Now, consider the database of Fig. 11.22(b), where a cut appears in the first rule for `b`:

```
b(X) :- g(X), !, v(X).
```

The insertion of this cut changes the Prolog search tree in Fig. 11.21(a) into the tree in Fig. 11.21(b). This cut tells control to backtrack past `g(X)` and `b(X)` without considering any remaining rules for `g` and `b`, as in Fig. 11.21(b). The goal `a(Z)` now has just two solutions. □

Cuts in Prolog are frequently misunderstood, perhaps with good reason. From the search tree in Fig. 11.21(b), the query `a(Z)` has two solutions:

```
?- a(Z).
   Z = 1 ;
   Z = 5 ;
   no
```

We might expect from these responses that `a(2)`, `a(3)`, and `a(4)` are unsatisfiable. In fact, the search trees in Fig. 11.23 reach a *yes* response for each of `a(2)`, `a(3)`, and `a(4)`.

The queries `a(2)` and `a(3)` lead to a *yes* response without backtracking, as in Fig. 11.23(a-b). Since no backtracking is needed, the cut does not prevent the computation from reaching a *yes*.

Finally consider the query `a(4)`. The computation in Fig. 11.23(c) never reaches the cut, because `g(4)` is unsatisfiable. The cut therefore has no effect, and the computation reaches a *yes*.

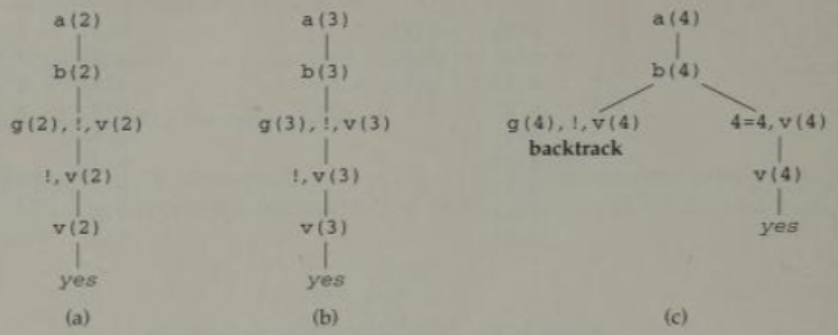


Figure 11.23 Prolog search trees based on the rules in Fig. 11.22(b).

Programming Applications of Cuts

green cuts skip fruitless searches

A relatively benign use of cuts is to prune parts of a Prolog search tree that cannot possibly reach a solution. Such cuts have been called *green* cuts; they make a program efficient without changing its solutions. Cuts that are not green are called *red*.

By restricting backtracking, cuts reduce the memory requirements of a program. Without cuts, every single rule application and unification has to be recorded until the overall computation succeeds or backtracking occurs.

Example 11.11 For an example of green cuts, consider the following rules for binary search trees, from Fig. 11.8:

```
member(K, node(K,_,_)).
member(K, node(N,S,_)) :- K < N, member(K, S).
member(K, node(N,_,T)) :- K > N, member(K, T).
```

These three rules are mutually exclusive because only one of `K=N`, `K<N`, and `K>N` can be true at a time. The cut in the second rule,

```
member(K, node(K,_,_)).
member(K, node(N,S,_)) :- K < N, !, member(K, S).
member(K, node(N,_,T)) :- K > N, member(K, T).
```

is therefore a green cut. It is reached only if `K<N`, so the third rule cannot possibly apply if the cut is reached. The cut makes the program more efficient in the following case: `K<N` but `member(K,S)` fails because `K` is not in the tree.

Without the cut, Prolog will backtrack and try the third rule, only to fail on the test `K>N` — there could be many such futile tests during a single unsuccessful search. □

Variants of the `lookup` relation in the following example have been used in compilers written in Prolog. The compilers use binary search trees. For simplicity, however, the example uses linear search to look for a key in a list of key-value pairs.[7]

Example 11.12 The only difference between the rules for `lookup` and `install` is a cut in the first rule for `lookup`:

```
lookup(K,V, [(K,W)|_]) :- !, V = W.
lookup(K,V, [_|Z]) :- lookup(K,V,Z).

install(K,V, [(K,W)|_]) :- V = W.
install(K,V, [_|Z]) :- install(K,V,Z).
```

The `lookup` relation can be used to enter information into a table of key-value pairs. The table is maintained as an open list, ending in a variable.

```
?- lookup(p,72,D).
   D = [(p,72)|_1] ;
   no
```

(Recall that we type a semicolon to ask for more solutions. The *no* response means that there are no more solutions.) An attempt to enter two different values for the same key ends in failure:

```
?- lookup(p,72,D), lookup(p,73,D).
   no
```

The following query uses relation `lookup` both to enter pairs and to look up a value:

```
?- lookup(1,58,D), lookup(p,72,D), lookup(p,Y,D).
   D = [(1,58),(p,72)|_1]
   Y = 72 ;

   no
```

Lacking the cut, relation `install` behaves quite differently. The following query has an infinite number of solutions:

[7] The simplified syntax of terms in Section 11.2 does not allow terms of the form `(K,V)`, but Prolog does. The term `(K,V)` is a pair with first component `K` and second component `V`.

```
?- install(p,72,D).
   [(p,72)|_1] ;
   [_2,(p,72)|_3] ;
   [_2,_4,(p,72)|_5]

   yes
```

Relation `install` also allows two different values to be entered for the same key:

```
?- install(p,72,D), install(p,73,D).
   [(p,72),(p,73)|_1] ;
```

The role of the cut can be seen more clearly from the following equivalent version of the above rules for `lookup`:

```
lookup(K,V,L) :- L = [(K,W)|_], !, V = W.
lookup(K,V,L) :- L = [_|Z], lookup(K,V,Z).
```

If `D` is a fresh variable, we get the following succession of goals:

```
lookup(p,72,D)                    starting goal
D=[(p,_1)|_2], !, 72=_1           apply first rule
!, 72=_1                          D unifies
72=_1
yes                               72 unifies with _1
```

The cut prevents further solutions by eliminating consideration of the second rule for `lookup`.

Now consider the query

```
?- lookup(p,72,D), lookup(p,73,D).
   no
```

As we just saw, the subgoal `lookup(p,72,D)` leaves `D` bound to a list `[(p,72)|_2]`. We therefore get the following succession of goals:

```
lookup(p,73,[(p,72)|_2])
[(p,72)|_2]=[(p,_3)|_4], !, 73=_3      apply first rule
!, 73=72                               the lists unify
73=72
```

The cut forces failure by preventing consideration of the second rule for `lookup`. □

Negation as Failure

The `not` operator in Prolog is implemented by the rules

```
not(X) :- X, !, fail.
not(_).
```

Informally, the first rule attempts to satisfy the argument `X` of `not`. If the goal `X` succeeds, then the cut and `fail` are reached. The construct `fail` forces failure and the cut prevents consideration of the second rule. On the other hand, if the goal `X` fails, then the second rule succeeds, because _ unifies with any term.

These rules for `not` explain the difference between the responses to

```
?- X = 2, not(X = 1).
   X = 2
```

and

```
?- not(X = 1), X = 2.
   no
```

In the first of these two queries, the subgoal `X=2` unifies `X` with 2, as shown in Fig. 11.24(a), leaving `not(2=1)` as the current goal. The first rule for `not` yields the new goal

```
2=1, !, fail
```

Since `2=1` fails, the cut is not reached; hence, the cut is not shown in Fig. 11.24(a). Then, the second rule for `not` is tried and the goal `not(2=1)` succeeds.

The search tree for the second query appears in Fig. 11.24(b). The first rule for `not` yields the current goal

```
X=1, !, fail, X=2.
```

The subgoal `X=1` succeeds, the cut is satisfied, and `fail` is reached. The cut eliminates consideration of the other rule for `not` so the entire computation fails. Note that the subgoal `X=2` is never reached.

In general, it is safe to apply `not` to a term without variables because such terms have no variables to be changed by unification.

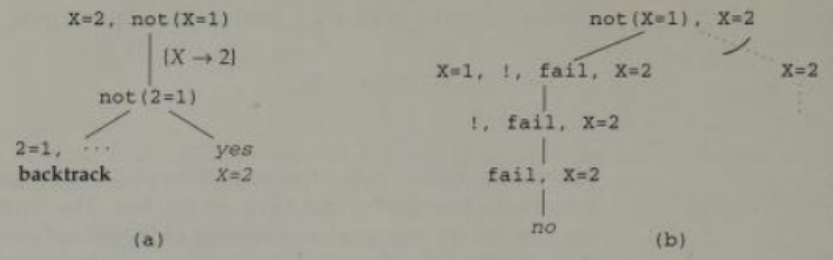


Figure 11.24 Prolog search trees illustrating negation as failure.

Exercises

11.1 Given the relations

`father(X,Y)`	`X` is the father of `Y`
`mother(X,Y)`	`X` is the mother of `Y`
`female(X)`	`X` is female
`male(X)`	`X` is male

define relations for the following:

a. Sibling
b. Sister
c. Grandson
d. First cousin
e. Descendant

11.2 Using only the `append` relation, formulate queries to determine the following:

a. The third element of a list
b. The last element of a list
c. All but the last element of a list
d. Whether a list is a concatenation of three copies of the same sublist
e. Whether a list `Y` is formed by inserting an element `A` somewhere in a list `X`

11.3 Define relations to determine the following:

a. Is a list a permutation of another list?
b. Does a list have an even number of elements?
c. Is a list formed by merging two lists?
d. Is a list a palindrome; that is, does it read the same from left to right as it does from right to left?

11.4 Define relations corresponding to the following operations on lists:

a. Remove the second and succeeding adjacent duplicates.
b. Leave only the elements that do not have an adjacent duplicate.
c. Leave only one copy of the elements that have adjacent duplicates.

11.5 Complete Example 11.3 by defining relations `insert` and `delete` on binary search trees.

11.6 Arithmetic can be performed in Prolog by using subgoals of the form

⟨*variable*⟩ `is` ⟨*expression*⟩

This subgoal succeeds if the result of evaluating the expression unifies with the variable, as in

```
?- X is 2, Y is X+1.
   X = 2
   Y = 3
```

An error occurs if the expression after `is` contains any unbound variables, as in

```
?- Y is X+1, X is 2.
   Error
```

a. Define a relation corresponding to the factorial function.
b. Define a relation corresponding to a tail-recursive version of the factorial function.

11.7 Draw Prolog search trees for the query

```
?- reverse([a,b,c,d],W).
```

where `reverse` is defined by the rules:

a.
```
reverse([],[]).
reverse([A|X],Z) :- reverse(X,Y), append(Y,[A],Z).
```

b.
```
reverse(X,Z) :- rev(X,[],Z).
rev([],Y,Y).
rev([A|X],Y,Z) :- rev(X,[A|Y],Z).
```

11.8 Consider a relation `member` defined by the rules

```
member(M, [M|_]).
member(M, [_|T]) :- member(M, T).
```

Draw the portion of the Prolog search tree that corresponds to the responses in the following:

a.
```
?- member(b, [a,b,c]).
   yes
```

b.
```
?- member(d, [a,b,c]).
   no
```

c.
```
?- member(b, X).
   X = [b|_] ;
   X = [_,b|_] ;
   X = [_,_,b|_]
   yes
```

11.9 With the relation `member` as in Exercise 11.8, draw Prolog search trees for the following queries:

a. `X = [1,2,3], member(a,X).`

b. `member(a,X), X = [1,2,3].`

11.10 Except for the cut in the first rule, the following rules are the same as those in Section 11.5:

```
append([], Y, Y) :- !.
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
prefix(X,Z) :- append(X,Y,Z).
suffix(Y,Z) :- append(X,Y,Z).
```

Compare the Prolog search trees for the following queries with the trees in Fig. 11.15 and 11.16:

a. `suffix([a],L), prefix(L,[a,b,c]).`

b. `suffix([b],L), prefix(L,[a,b,c]).`

Bibliographic Notes

Prolog, from *programmation en logique*, is the name of a programming language developed by Alain Colmerauer and Phillipe Roussel in 1972. Companion articles by Kowalski [1988] and Cohen [1988] provide a glimpse of its early history. The development of the language was influenced by W-grammars (van Wijngaarden et al. [1975]), the description language for Algol 68, and by Robinson's [1965] resolution principle for mechanical theorem proving.

Kowalski notes, "Looking back on our early discoveries, I value most the discovery that computation could be subsumed by deduction." His early examples included "computationally efficient axioms for such recursive predicates as addition and factorial." He continues, "For [Colmerauer], the Horn clause definition of appending lists was much more characteristic of the importance of logic programming."

Cohen offers reasons why Prolog developed slowly, relative to Lisp: (1) the lack of interesting examples illustrating the expressive power of the language, (2) the lack of adequate implementations, and (3) the availability of Lisp. He adds, "It is fair to say that the subsequent interpreters and compilers developed by Warren played a major role in the acceptance of Prolog." Warren [1980] describes how Prolog itself can be used for compiler writing.

More information on Prolog programming techniques can be found in textbooks such as Sterling and Shapiro [1994] and Clocksin and Mellish [1987]. Example 11.5 and Exercise 11.7 are motivated by examples in Sterling and Shapiro [1994]. Cohen [1988] cites difference lists as Colmerauer's valuable contribution to Roussel's original Prolog interpreter; Clark and Tärnlund [1977] is a published reference for difference lists.

Cuts were introduced by Colmerauer to conserve memory space. See Clark [1978] for a treatment of negation-as-failure.

For a discussion of logic programming in general, see Kowalski [1979a] or Hogger [1984].