Chapter 8

Logic Programming

Although LISP has long been the language of choice for artificial intelligence (AI) research, other languages are increasingly common for some branches of AI. C and C++ have become quite popular. Some AI programs are meant to reason about the world, given some initial knowledge. Knowledge can be represented in property lists of LISP atoms, but it can also be stored as a set of rules and facts. One form of reasoning is to try to derive new facts or to prove or disprove conjectures from the current set of facts. Programs that follow this approach are called "inference engines."

In this chapter, I will present several languages intended for knowledge representation and inference engines. These logic languages tend to be **declarative**. Programs state goals and rules to achieve goals, but do not explicitly invoke those rules in order to achieve the goals. In contrast, both imperative and functional languages tend to be **procedural**; that is, programs are organized around control structures such as iteration and procedure invocation.

1 ◆ PROLOG

Prolog is a declarative programming language designed in 1972 by Philippe Roussel and Alain Colmerauer of the University of Aix-Marseille and Robert Kowalski at the University of Edinburgh. Prolog programs are related to computations in a formal logic. A programmer first provides a **database** of facts and rules of inference. Programs are formulated as assertions involving facts in the database. Programs are executed by proving or disproving a particular assertion.

1.1 Terms, Predicates, and Queries

Elementary values in Prolog are called **terms**. Terms are either constants (numbers like 43 and identifiers like parsley, starting with a lowercase letter), variables (identifiers like X, starting with an uppercase letter), or structures (identifiers starting with a lowercase letter, followed by parameters that are themselves terms, such as tasty(parsley)). The identifier that heads a structure (like tasty) is called a **functor**, based on its similarity in appearance to a function name. Figure 8.1 shows a sample term.

Figure 8.1

```
near(house, X, 22, distance(Y))
```

There are two constants (22 and house), two variables (X and Y), one unary functor (distance), one 4-ary functor (near), and two structures (distance(Y) and the whole term). This term has no inherent meaning; a program could use it to mean that house is within 22 miles of some object X, and that the actual distance is Y miles.

Programs are built out of facts, rules, and queries, which are all based on predicates. A **predicate** has the same form as a structure: a name in lowercase followed by parameters, which must be terms. Predicates represent a fact (actual or to be proven) relating the values of their parameters. I will often call the predicate name itself a predicate when there is no chance for confusion.

Although structures and predicates have parameters and otherwise look like function calls, this appearance is deceiving. Structures are used as patterns, and predicates are used to define rules and facts and to pose queries. Only in their role as queries are predicates at all like function calls.

A database is constructed out of facts and rules. To build a simple family-relation database, I will start with constants representing people: tom, dick, harry, jane, judy, and mary. I describe relationships among these people with binary predicates: fatherOf, motherOf, parentOf, grandparentOf, and siblingOf.

One of the hardest problems in reading Prolog programs is figuring out what predicates are supposed to mean. The predicate motherOf(mary,judy) could be taken to mean that Judy is the mother of Mary or that Mary is the mother of Judy; the proper interpretation is up to the programmer. I follow the convention that the first parameters represent the traditional inputs, and the last parameters represent outputs, although Prolog does not make this distinction. I therefore understand motherOf(mary,judy) to mean that the mother of Mary is Judy. The predicate motherOf(mary,judy) may be true, but motherOf(mary,tom) is very likely to be false.

My Prolog program begins by stating facts that define fundamental relations among the terms. **Facts** are predicates that are assumed true, such as those in Figure 8.2.

Figure 8.2

```
fatherOf(tom,dick) . /* read: "father of tom is dick" */          1
fatherOf(dick,harry) .                                            2
fatherOf(jane,harry) .                                            3
motherOf(tom,judy) .                                              4
motherOf(dick,mary) .                                             5
motherOf(jane,mary) .                                             6
```

The period is used to terminate facts and rules, which are allowed to cross line boundaries.

Given this database of facts, I can form queries. A **query** is a request to prove or disprove an assertion built of predicates. Interactive Prolog implementations expect that anything the user types in is a query. To input facts, the user must type a pseudoquery that causes Prolog to load the user's file of facts. My examples just show facts and queries together; I distinguish queries by prefixing them with the question symbol ?- , which is the usual prompt in an interactive Prolog session. Prolog will determine if the queried predicate is true or false and will reply Yes or No. Figure 8.3 shows an interactive example of queries (all examples in this chapter are in syntactically correct Prolog).

Figure 8.3

```
in:  ?- fatherOf(dick,harry) .                                    1
out: Yes                                                          2

in:  ?- fatherOf(harry,tom) .                                     3
out: No                                                           4
```

Any predicate that Prolog cannot prove true is assumed to be false. In logic, this rule is known as the **closed-world assumption**. When Prolog says No, it means "not as far as can be proven."

Queries can include variables, which are distinguished from constants by their initial capital letter. A variable that appears in a query acts like an unknown in an equation. Prolog tries to find an assignment to the variable that will make the predicate true. The assignment is then reported as the result of the query. In a sense, variables in queries are like result parameters, and the rest of the parameters are like value parameters. Consider Figure 8.4.

Figure 8.4

```
in:  ?- fatherOf(X,harry) .                                       1
out: X = dick ;                                                   2
     X = jane ;                                                   3
     No                                                           4
```

Line 1 presents a query with one variable, X, and one constant, harry. It asks for matches in the database to the given predicate name (fatherOf) that match the constant second parameter (harry); the first parameter is to be returned in the variable X. This query has two solutions. Prolog first presents the first (line 2). The user may request another solution by typing ; (at the end of line 2). When there are no more solutions, Prolog prints No. In the examples that follow, I omit the final No.

Variables may be placed in any number of parameters, as shown in Figure 8.5.

Figure 8.5

```
in:  ?- fatherOf(jane,X) .                                    1
out: X = harry                                                2

in:  ?- motherOf(X,Y) .                                       3
out: X = tom, Y = judy;                                       4
     X = dick, Y = mary;                                      5
     X = jane, Y = mary                                       6
```

A complex query is built from multiple predicates joined by , , which represents logical **and**. Each predicate is then called a **conjunct**. Consider Figure 8.6.

Figure 8.6

```
in:  ?- fatherOf(jane,X) , motherOf(jane,Y) .                 1
out: X = harry, Y = mary                                      2

in:  ?- fatherOf(tom,X) , fatherOf(X,harry)  .                3
out: X = dick                                                 4
```

The query in line 1 asks for both parents of jane; the query in line 3 asks for the person who is both the father of tom and the son of harry. If a variable appears more than once in a query, as X does in line 3, it must be replaced by the same solution in all its occurrences.

What makes Prolog particularly interesting as a programming language is that it allows us to write **rules** that define one predicate in terms of other predicates. A rule is of the form shown in Figure 8.7.

Figure 8.7

```
predicate1(param,param, ...) :-                               1
     predicate2(param,param, ...) , ... ,                     2
     predicateN(param,param, ...) .                           3
```

The predicate on the left is called the **head** of the rule; the predicates on the right form its **body**. A rule states that if the predicates in the body can all be proved (they have a simultaneous solution), then the head is true. You can read :- as **if**. Continuing Figure 8.2 (page 233), typical rules might include those of Figure 8.8.

Figure 8.8

```
/* grandmotherOf(X,GM) means the grandmother of X is GM */     1
grandmotherOf(X,GM) :- motherOf(M,GM) , motherOf(X,M) .       2
grandmotherOf(X,GM) :- motherOf(F,GM) , fatherOf(X,F) .       3

/* siblingOf(X,Y) means a sibling of X is Y */                4
siblingOf(X,Y) :- motherOf(X,M) , fatherOf(X,F) ,             5
     motherOf(Y,M) , fatherOf(Y,F) , not(X = Y) .             6
```

There are two ways in which GM can be a grandmother of X, so there are two alternative rules (lines 2–3). However, there is only one way for X and Y to be siblings, although it is fairly complicated (lines 5–6). This rule introduces variables M and F just to force X and Y to have the same parents.

Given these rules, I can pose the queries in Figure 8.9.

Figure 8.9

```
in:  ?- grandmotherOf(tom,X) .                                    1
out: X = mary                                                     2

in:  ?- siblingOf(X,Y) .                                          3
out: X = dick, Y = jane;                                          4
     X = jane, Y = dick                                           5
```

The query in line 3 generates two results because `siblingOf` is symmetric.

These examples begin to demonstrate the power of Prolog. The programmer states facts and rules, but queries don't specify which facts and rules to apply. This is why Prolog is called declarative.

Prolog attempts to satisfy a query by satisfying (that is, finding a way to prove) each conjunct of the query. Rules are applied as necessary by substituting the body of a rule for its head. This process isn't at all trivial, because more than one rule may apply to the same predicate. Each rule is tried in turn, which may lead to **backtracking**, in which alternative possibilities are tried (recursively) if a particular possibility fails. The way Prolog selects goals and subgoals during backtracking distinguish it from a more abstract language, LP (for logic programming), which selects goals and subgoals nondeterministically.

To demonstrate Prolog backtracking, I will return to the query `grandmotherOf(tom,X)`. The database is consulted to find either facts or rules with a `grandmotherOf` predicate as the head. (A fact is a rule with the given predicate as the head and `true` as the body.) The order of facts and rules is significant in Prolog (but not in LP); they are scanned from first to last. This ordering can affect the speed of a query and even determine if it will terminate, as you will see soon. In this case, no facts match, but two rules define `grandmother`. The first applicable rule is

```
grandmotherOf(X,GM) :- motherOf(M,GM) , motherOf(X,M) .
```

It is applicable because its head matches the query: they both use the binary predicate `grandmotherOf`. To avoid confusion, Prolog renames any variables in the rule that appear in the query. Since `X` appears in both, it is renamed in the rule, perhaps to `Y`. Prolog then binds the rule's `Y` (which is like a formal parameter) to the query's `tom` (which is like an actual parameter), and the rule's `GM` (a formal) to the query's `X` (an actual). The rule effectively becomes

```
grandmotherOf(tom,X) :- motherOf(M,X) , motherOf(tom,M) .
```

This matching, renaming, and binding is called **unification**.

There is a new subgoal: to prove

```
motherOf(M,X) , motherOf(tom,M) .
```

The only applicable rules are facts involving `motherOf`. These facts are unified in turn with the first conjunct, `motherOf(M,X)`. Each unification binds `M` and `X` (in both conjuncts). For each unification, the second conjunct, which is now fully bound, is matched against existing facts. No match succeeds, so

Prolog backtracks to the point where the first grandmotherOf rule was selected and tries the second rule instead. Unifying the query and the second grandmotherOf rule gives rise to the new subgoal

```
motherOf(F,X) , fatherOf(tom,F) .
```

This subgoal can be satisfied with F = dick and GM = mary. If the second rule had failed, the entire query would have failed, since no other rules apply.

The backtrack tree of Figure 8.10 shows these steps in more detail.

Figure 8.10

```
goal: grandmotherOf(tom,X)                                          1
     rule: motherOf(M,X), motherOf(tom,M)                           2
          fact: motherOf(tom,judy) [M = tom, X = judy]              3
               goal: motherOf(tom,tom)                              4
               fail                                                 5
          fact: motherOf(dick,mary) [M = dick, X = mary]            6
               goal: motherOf(tom,dick)                             7
               fail                                                 8
          fact: motherOf(jane,mary) [M = jane, X = mary             9
               goal: motherOf(tom,jane)                             10
               fail                                                 11
          fail                                                      12
     rule: motherOf(F,X), fatherOf(tom,F)                           13
          fact: motherOf(tom,judy) [F = tom, X = judy]              14
               goal: fatherOf(tom,tom)                              15
               fail                                                 16
          fact: motherOf(dick,mary) [F = dick, X = mary]            17
               goal: fatherOf(tom,dick)                             18
               succeed                                              19
          succeed; F = dick, X = mary                               20
     succeed; X = mary                                              21
```

I have bound all identifiers based on the unification steps so far. For example, in lines 2 and 13 I have changed GM to X.

The Prolog unification algorithm can be fooled by having it unify a variable with a term containing that same variable, as in Figure 8.11.

Figure 8.11

```
strange(X) :- X = strange(X) .                                      1

in:  ?- strange(Y) .                                                2
```

The variable Y in the query is unified with X in the rule in line 1, leading to Y = strange(Y). The = constraint causes Y to be unified with strange(Y), which represents a result, but one that cannot be displayed in a finite space. Prolog will try to print the nonsense result, which begins strange(strange(. It turns out to be relatively difficult to solve this "occur-check" problem and prevent such mistaken unification, so most Prolog implementations don't try. They do, however, handle the easier situation encountered in Figure 8.12.

Figure 8.12

```
yellow(green(X)) :- X=puce .                                    1

in:  ?- yellow(X) .                                             2
out: X = green(puce) .                                          3
```

The query lin line 2 matches the actual parameter X with the formal parameter green(X). Since one X is actual and the other formal, Prolog does not confuse them.

The backtracking algorithm can also be represented by a box model [Byrd 80]. Each predicate is represented as a box with two inputs and two outputs, as shown in Figure 8.13.

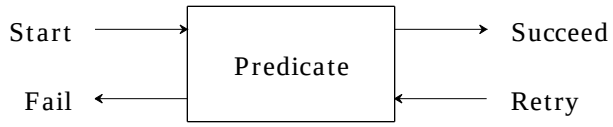


Figure 8.13 Box model of a predicate

The first invocation of a predicate enters from the left. If the predicate is satisfied, control continues out to the right. If a different solution is required, control reenters from the right. If there are no (more) solutions, control exits to the left.

The logical **and** of two predicates is formed by joining them together, as in Figure 8.14.

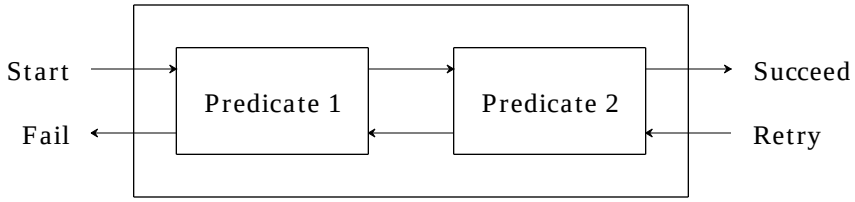


Figure 8.14 Logical **and** of two predicates

The logical **or** of two predicates is formed by a different combination, shown in Figure 8.15.

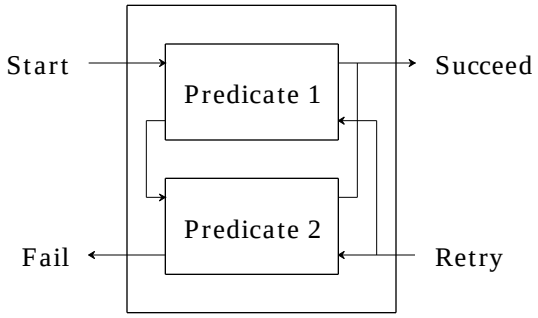
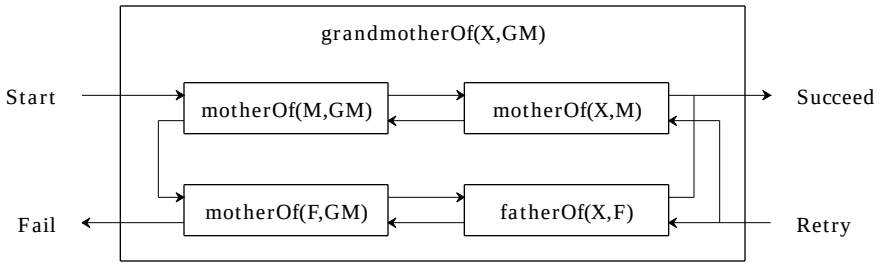


Figure 8.15 Logical **or** of two predicates

When control returns from the right in a retry attempt, it goes to whichever of the two predicates provided the most recent success.

The query grandmotherOf(tom,X) can be represented in Figure 8.16 in a box model:

Figure 8.16



The box model shows execution paths and what happens when a goal succeeds or fails. It doesn't show variable substitutions, however.

Backtracking examines the backtrack tree by depth-first search, that is, in a top-down order. The order of alternatives tried when more than one rule or fact can be applied can have a startling effect on the speed at which a query is satisfied (or found to be unsatisfiable). Assume that an evaluator must go through *n* levels of rules and facts to provide a solution for a query, and that at each level, it must choose between two alternatives. If the right decision is made at each point, the evaluator will operate in O(*n*) time, because only *n* decisions are made. However, if the wrong decision is made at each point, it is possible that $O(2^n)$ time will be required, because there are that many different settings for the decisions. As a result, Prolog programmers tend to sort rules and introduce other aids I will present shortly to help the evaluator make the decisions. For example, the rules for `grandmotherOf` in Figure 8.8 (page 234) can be improved by reordering the bodies, as shown in the exercises. Nonetheless, except in the case of recursive rules (in which an infinite expansion of rules can occur), the issue is one of speed, not correctness.

1.2 Separating Logic and Control

A well-known textbook on data structures by Niklaus Wirth is titled *Algorithms + Data Structures = Programs* [Wirth 76]. This equation defines the programming model implicit in modern procedural languages. The programming task is essentially to design data structures and the algorithms that manipulate them.

An interesting alternative view is presented by Robert Kowalski in "Algorithms = Logic + Control" [Kowalski 79]. This article proposes a declarative programming view. The programmer first specifies the logic of an algorithm. This component specifies what the result of the algorithm is to be. Then the control component is defined; it specifies how an evaluator may proceed to actually produce an answer. The logic component is essential because it defines what the programmer wants the program to generate. The control component may be optional, since it controls how fast an answer may be obtained. This represents a two-tiered approach in which you think first about getting the correct answer, and then about making the computation sufficiently fast.

Prolog supports this view of programming; its facts and rules are essentially the logic component of a program. The control component is largely hidden in Prolog's evaluator, although certain Prolog commands have been devised to aid an evaluator, as you will see shortly.

In theory, a language that cleanly separates logic and control has great advantages over conventional languages, which thoroughly intermix the definition of what is wanted and how to compute it. The logic component is the essential part of the program and often suffices to produce an answer. The control component allows efficiency issues to be addressed. It may be complex and detailed but can be ignored by most users of the program.

1.3 Axiomatic Data Types

One advantage of languages that support abstract data types is that the specification of an abstract data type can be separated from its implementation details. Prolog carries this idea further still: you can specify an abstract data type by axioms without any implementation at all. The axioms define the properties of the abstract data type, which is all the programmer really cares about. Of course, the evaluator must find some way to actually realize the operations of an abstract data type, but since Prolog is declarative, this isn't the programmer's concern!

I will show you how to define lists, a fundamental data structure of LISP, as seen in Chapter 5, based on terms, predicates, facts, and rules. I define the set of valid lists by indicating when `cons` generates a list, as in Figure 8.17.

Figure 8.17

```
isList(nil) .                                    1
isList(cons(_, T)) :- isList(T) .                2
```

Line 1 indicates that `nil` is a valid list, and line 2 shows that `cons` is a binary functor that builds new lists if the second parameter is a list. Because the body of this rule does not refer to the first parameter, I use the don't-care variable `_`. If `_` is used more than once in a rule or fact, each occurrence represents a different don't-care variable. If you want the same value to be forced in two different positions, you must use an explicit variable name. Line 2 shows that predicates can take parameters that are arbitrary terms, including structures. Although you may be tempted to treat `cons(_,T)` as a procedure call, it is only a structure. When it appears nested in the head of a rule, it is treated as a pattern to be matched to a query.

Predicates specifying the `car` and `cdr` functors are easy, as shown in Figure 8.18.

Figure 8.18

```
/* car(X,Y) means the car of X is Y */           1
car(cons(H,T),H) :- isList(T) .                  2

/* cdr(X,Y) means the cdr of X is Y */           3
cdr(cons(H,T),T) :- isList(T) .                  4
```

Once again, the functor `cons` is used in the head of rules as a pattern. Using these definitions, I can pose queries like those in Figure 8.19.

Figure 8.19

```
in:  ?- car(cons(2,cons(3,nil)),X) .                    1
out: X = 2                                              2

in:  ?- car(cons(a,b),X) .                              3
out: No                                                 4

in:  ?- car(cons(X,nil),2) .                            5
out: X = 2                                              6

in:  ?- car(cons(a,X),a) .                              7
out: X = nil;                                           8
     X = cons(_1, nil);                                 9
     X = cons(_2, cons(_1, nil));                       10
     ...                                                11
```

The first query (line 1) asks for the car of a particular valid list. The second (line 3) asks for the car of an invalid list. It fails because the rules only allow car to be applied to valid lists. The third query (line 5) inverts the question by asking what has to be consed to a list to obtain a car of 2. The fourth query (line 7) requests lists whose car is a. There are infinitely many answers. Prolog invents internal temporary names for unbound results, which I display as _1, _2, and so on (lines 9–10). I call such names **don't-care results**.

In this example, both car and cdr can be used to form queries. However, cons cannot be used that way; there are no rules with heads matching cons. In other words, cons is a functor, whereas car and cdr are predicate names.

Stacks are frequently used to illustrate abstract data types, so let me present an axiomatic definition of stacks in Figure 8.20.

Figure 8.20

```
isStack(nil) .                                          1
isStack(push(_,S)) :- isStack(S) .                      2
top(push(Elem,S),Elem) :- isStack(S) .                  3
pop(push(_,S),S) :-  isStack(S) .                       4

in:  isStack(push(a,push(b,nil))) .                     5
out: Yes                                                6

in:  pop(push(a,push(b,nil)),X) .                       7
out: X = push(b, nil)                                   8
```

I have used push as a functor, whereas top and pop are predicates. I leave making pop a functor as an exercise. The Prolog definition of stacks is similar to that of lists, which shouldn't be surprising, since lists are often used to implement stacks in conventional languages. The stacks defined above are heterogeneous. The exercises explore restricting stacks to hold only integers.

These examples show that Prolog can deal fairly directly with algebraic specification of data types, a field pioneered by Guttag [Guttag 77]. In the notation of algebraic specification, an integer stack looks like this:

Figure 8.21

```
type IntegerStack                                                    1

operations                                                           2
     create: → IntegerStack                                          3
     push: IntegerStack × integer → IntegerStack                     4
     pop: IntegerStack → IntegerStack                                5
     top: IntegerStack → integer                                     6

axioms                                                               7
     top(create) = error                                             8
     top(push(S,I)) = I                                              9
     pop(create) = error                                             10
     pop(push(S,I)) = S                                              11
```

Operations are first defined by naming their input and output types. In procedural languages, the operations are analogous to the specification part of an abstract data type. In Prolog, which has no need to distinguish input from output, the operations can be predicates or functors. Operations that result in IntegerStacks are called **constructors**; here, create, push, and pop are all constructors. Actually, pop is a special kind of constructor because it reduces the amount of information; such constructors are called **destructors**. Operations that do not result in the abstract data type are called **inspectors**; here, top is the only inspector. The axioms are simplification rules. It is not always easy to see what axioms are needed; a rule of thumb is that an axiom is needed for all combinations of inspectors and non-destructive constructors (lines 8 and 9) and all combinations of destructors and non-destructive constructors (lines 10 and 11). In Prolog, axioms are expressed as rules, which means that inspectors (top) and destructors (pop) will be predicates, whereas non-destructive constructors (push) will be functors.

An algebraic specification could in general be satisfied by many different models. The axioms equate such elements as create and pop(push(create,4)), which some models would keep distinct. If only those elements that the axioms equate are considered equal, the resulting algebra is called an **initial algebra**. If all elements are equated that cannot be distinguished by inspectors, the resulting algebra is called a **final algebra**. In the case of stacks, these two algebras are the same. In an algebraic specification of arrays, however, the order in which elements are assigned values does not affect what an inspector returns, so the final algebra is more appropriate than the initial algebra, which would distinguish arrays with the same elements that happen to have acquired the elements in a different order.

1.4 List Processing

Because lists are such a familiar and flexible data structure, Prolog provides a notation for the structures that represent lists. Predefining lists also makes their manipulation more efficient. Predefined arithmetic operators are provided for much the same reason. In Prolog, lists are delimited by brackets. The empty list is [], and [[a,b],[c,d,e]] is a list containing two sublists (with 2 and 3 elements respectively). The notation [H | T] is used to represent any list with car H and cdr T, as in Figure 8.22.

Figure 8.22

```
p([1,2,3,4]) .                                          1

in:  ?- p([X|Y]) .                                      2
out: X = 1, Y = [2,3,4]                                 3

in:  ?- p([_,_,X|Y]) .                                  4
out: X = 3, Y = [4]                                     5
```

Using this notation, Figure 8.23 defines the list operation append in a manner analogous to its definition in LISP (actually, append is often predefined).

Figure 8.23

```
/* append(A,B,C) means C is the list formed by          1
   appending element B to the end of list A */          2
append([],[],[]) .                                      3
append([],[H|T],[H|T]) .                                4
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3) .          5

in:  ?- append([1,2],[3,4],S) .                         6
out: S =  [1,2,3,4]                                     7
```

The correspondence to the LISP definition of append is almost exact, except that no explicit control structure is given. Instead, rules that characterize append are defined; they allow Prolog to recognize (or build) correctly appended lists. A LISP-like approach to list manipulation can be used to structure Prolog rules. Thus I could define a list-membership predicate memberOf (again, it is often predefined) as shown in Figure 8.24.

Figure 8.24

```
/* memberOf(X,L) means X is a member of list L */       1
memberOf(X,[X|_]) .                                     2
memberOf(X,[_|Y]) :- memberOf(X,Y) .                    3

in:  ?- memberOf(4,[1,2,3]) .                           4
out: No                                                 5
in:  ?- memberOf(4,[1,4,3]) .                           6
out: Yes                                                7
```

This definition is strongly reminiscent of LISP's tail recursion. However, Prolog's declarative nature allows definitions that are quite foreign to LISP's procedural nature. Consider the alternative definition of memberOf in Figure 8.25.

Figure 8.25

```
memberOf(X,L) :- append(_,[X|_],L) .
```

This definition says that X is a member of L exactly if there exists some list that can be appended to a list beginning with X to form list L.

Although both definitions of memberOf are correct, the first one will probably be more efficiently executed, because it is somewhat more procedural in flavor. In fact, Prolog definitions are often structured specifically to guide an evaluator toward a more efficient evaluation of a query. Sorting is a good example. The simplest abstract definition of a sort is a permutation of elements

that puts the elements in nondecreasing (or nonincreasing) order. This definition has a direct Prolog analogue, shown in Figure 8.26.

Figure 8.26

```
naiveSort(L1,L2) :- permutation(L1,L2) , inOrder(L2) .          1

permutation([],[]) .                                            2
permutation(L,[H|T]) :- append(V,[H|U],L) ,                     3
     append(V,U,W) , permutation(W,T) .                         4

inOrder([]) .                                                   5
inOrder([_]) .                                                  6
inOrder([A,B|T]) :- A =< B , inOrder([B|T]) .                   7
```

Since a list with n distinct elements has $n!$ permutations, the above definition may well lead to long and tedious searches in an effort to find a sorting of a list. An alternative is to define `inOrder` in a manner that leads to a more efficient evaluation sequence. For example, using the infamous bubble sort (shame on me!) as inspiration, I create the alternative definition in Figure 8.27.

Figure 8.27

```
bubbleSort(L,L) :- inOrder(L) .                                 1
bubbleSort(L1,L2) :- append(X,[A,B|Y],L1), A > B ,              2
     append(X,[B,A|Y],T), bubbleSort(T,L2) .                    3

inOrder([]) .                                                   4
inOrder([_]) .                                                  5
inOrder([A,B|T]) :- A =< B , inOrder([B|T]) .                   6
```

Actually, a trace of execution of `bubbleSort` will show that it always looks for and then swaps the first out-of-order pair in the list. There are $O(n^2)$ swaps, each of which requires $O(n)$ effort to discover by searching from the start of the list. The result is an $O(n^3)$ algorithm, which is worse than the $O(n^2)$ expected for bubble sort.

1.5 Difference Lists

List processing can be expensive. The `append` operation must step to the end of the first parameter in a recursive fashion before it can begin to construct the result. Prolog programmers have invented a programming trick called a **difference list** to alleviate this problem [Sterling 94]. Each list is represented in two pieces, which I will call `listextra` and `extra`. The actual list is `listextra` with `extra` removed from the end. What extra information to place at the end of a list is arbitrary and is based on convenience. For example, the list `[a,b]` can be represented in many ways, including `[a,b] []` and `[a,b,c,d] [c,d]`. In general, I can represent the list as `[a,b|Extra] [Extra]` and not specify what `Extra` might be. The `append` routine can now be written as a single, nonrecursive rule, as in Figure 8.28.

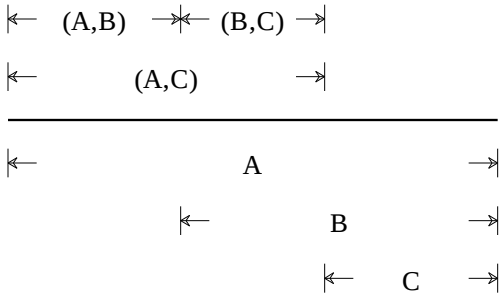
Figure 8.28

```
append(diff(A,B), diff(B,C), diff(A,C)) .                          1

in:  ?- append(diff([1,2|X],X), diff([3,4|Y],Y),                   2
          diff(ListExtra,Extra)) .                                 3
out: X = [3,4|_1],                                                 4
     Y = _1,                                                       5
     ListExtra = [1,2,3,4|_1],                                     6
     Extra = _1                                                    7
```

The append predicate defined in line 1 takes three parameters, each of which is a pattern representing a list in difference-list form. Figure 8.28 shows how line 1 represents appending two lists by explicitly showing A, B, and C. Lines 2–3 use this definition to append [1,2] and [3,4]. Each of these lists is represented with a variable (X and Y) to represent the extra parts. Lines 4–5 show how these variables get bound during match. The result is represented by (ListExtra,Extra), which (according to lines 6–7) is [1,2,3,4|_1] _1. So long as I am willing to use difference-list form, I have not needed to perform any recursion.

Figure 8.29 Difference lists



1.6 Arithmetic

In Prolog, the infix equality predicate = can be used for two purposes. In $\alpha = \beta$, if α and β are both constants, literals, structures, or bound variables, then the predicate succeeds or fails depending on whether or not the two operands are identical. Thus 1 = 1 is true, [1,2] = [] is false, and X = 2 is true if X has been bound to 2. This is the natural interpretation of the equality operator.

However, if α or β (or both) are unbound variables, then $\alpha = \beta$ succeeds and binds them together. So the same symbol acts both as an equality operator and as an assignment operator. Actually, the symbol introduces a constraint. Figure 8.30 illustrates this point.

Figure 8.30

```
set(A,B) :- A=B .                                          1

in:  ?- set(1,2) .                                         2
out: No                                                    3

in:  ?- set(X,2) .                                         4
out: X = 2                                                 5

in:  ?- set(3,X) .                                         6
out: X = 3                                                 7

in:  ?- set(Y,Z) .                                         8
out: Y = _1, Z = _1                                        9
```

For free (unbound) variables, = constrains the value of the variable, and this binds it, as shown in lines 4 and 6. These values hold only for the duration of the search for solutions, not afterward. Line 9 shows that Y and Z have been bound together, both to the same don't-care result.

For programmers accustomed to procedural programming, using = to bind variables is familiar, but in Prolog a few pitfalls await the unwary. For example, you will probably be surprised that the query (1+1) = 2 results in No. In fact, in Prolog 1+1 doesn't equal 2, because 1+1 is taken as an abbreviation for the structure +(1,1). This structure isn't the same as the integer 2, so the negative response is justified. Prolog doesn't automatically evaluate arithmetic expressions.

To force arithmetic expressions to be evaluated, Prolog provides the **is** operator, which effects assignment. It first evaluates its second operand as an arithmetic expression (it must not have any unbound variables), then tests for equality, and (if necessary) binds the free variable in the first operand. However, **is** will not invert expressions to bind free variables. Consider Figure 8.31.

Figure 8.31

```
in:  ?- 1 is 2*2-3 .                                       1
out: Yes                                                   2

in:  ?- X is 2*2 .                                         3
out: X = 4                                                 4

in:  ?- 4 is X*3-7 .                                       5
out: Unbound variable in arithmetic expression.            6
```

As lines 5–6 show, Prolog avoids the complexity of solving arbitrary equations. (Metafont can solve linear equations, and mathematics languages like Mathematica, discussed in Chapter 9, can handle a wide variety of equations.) Unfortunately, this restriction violates the symmetry between inputs and outputs found in other Prolog constructs.

1.7 Termination Issues

If free variables in queries may be bound only to a finite set of values, a Prolog evaluator should be able to prove or disprove any query. However, Prolog allows recursive definitions (such as the ones I showed earlier for lists and stacks) that imply infinite domains, as well as primitive objects (such as integers) with infinite domains. Not all queries will necessarily terminate. Prolog specifies that the order in which rules and facts are specified determines the order in which an evaluator attempts to apply them in proofs. Bad orders can sometimes lead to an infinite recursion. Suppose that I define the `isList` predicate as in Figure 8.32.

Figure 8.32

```
isList(cons(H,T)) :- isList(T) .                    1
isList(nil) .                                       2
```

A query like `isList(nil)` works fine, but `isList(X)` runs into a real snag. The top-down evaluator will set `X = cons(H,T)` and try to prove `isList(T)`. To do this, it will set `T = cons(H′,T′)`[1] and try to prove `isList(T′)`, and so forth. Eventually, the evaluator runs out of stack space. Putting the fact `isList(nil)` before the rule solves the problem.

Other problems may arise because of an inadequate set of rules. For example, I might provide a definition for odd integers and ask if any odd integer is equal to 2, as in Figure 8.33.

Figure 8.33

```
in:  odd(1) .                                       1
     odd(N) :- odd(M), N is M + 2 .                 2
     ?- odd(2) .                                    3
out: [does not terminate]                           4
```

The evaluator never finishes the query in line 3. It keeps generating odd numbers (1, 3, ...) to match `M` in line 2, but none of them satisfies `2 is M + 2`. It doesn't know that after considering 1, all succeeding odd numbers will be greater than 2 and hence not equal to 2. The query is false, but Prolog has no mechanism to prove it!

1.8 Resolution Proof Techniques

Prolog is designed to be amenable to a particular class of automatic proof techniques termed "resolution techniques." **Resolution** is a general inference rule shown in Figure 8.34.

[1] All variable names in a rule are local to that rule; hence, recursive applications cause no naming conflicts.

Figure 8.34

if $A_1 \wedge \cdots \wedge A_n \Rightarrow B_1 \vee \cdots \vee B_m \vee C$ 1
and $D_1 \wedge \cdots \wedge D_p \wedge C \Rightarrow E_1 \vee \cdots \vee E_q$ 2

then $A_1 \wedge \cdots \wedge A_n \wedge D_1 \wedge \cdots \wedge D_p \Rightarrow B_1 \vee \cdots \vee B_m \vee E_1 \vee \cdots \vee E_q$ 3

That is, if a term C appears on the right-hand side of one implication and on the left-hand side of a second implication, it can be removed, and the two rules can be joined. If C contains any free (that is, unbound) variables, these must be unified (that is, matched). This resolution operation doesn't appear to lead to any great simplification. Fortunately, Prolog limits the form of rules of inference to "Horn clauses," which are those that have only one term on the right-hand side of an implication (the head of a rule) and only '$\wedge$' as a connective. That is, Prolog rules of the form

```
A or B :- X, ... .
```

aren't allowed. For Prolog, the resolution rule takes the form shown in Figure 8.35.

Figure 8.35

if $A_1 \wedge \cdots \wedge A_n \Rightarrow C$ 1
and $D_1 \wedge \cdots \wedge D_p \wedge C \Rightarrow E$ 2

then $A_1 \wedge \cdots \wedge A_n \wedge D_1 \wedge \cdots \wedge D_p \Rightarrow E$ 3

This rule is the basis of the substitution technique employed earlier in top-down evaluation. Still, this substitution appears to make things more, rather than less complex. However, a Prolog fact `F` can be viewed as an implication of the form `true` $\Rightarrow$ `F`. When a fact is resolved, resolution in effect replaces a term with `true`, and since `true` $\wedge X \equiv X$, this substitution does lead to a simplification.

Resolution is interesting in that it doesn't actually try to prove a query directly from known facts and rules. If

$$A_1 \wedge \cdots \wedge A_n \Rightarrow B$$

then

$$A_1 \wedge \cdots \wedge A_n \wedge \neg B \Rightarrow \texttt{false}\ .$$

That is, if B is implied from known rules and facts, then $\neg B$ must lead to a contradiction. If a resolution theorem-prover is asked to prove B, it introduces $\neg B$ by introducing the implication $B \Rightarrow$ `false`. It then manipulates implications by resolution, trying to establish the implication `true` $\Rightarrow$ `false`.

Resolution theorem-provers use this unintuitive approach because resolution is "refutation complete"; that is, if a set of rules and facts are contradictory (that is, inconsistent), then resolution will always be able to conclude that `true` $\Rightarrow$ `false`. It doesn't guarantee that an evaluator will not pursue useless paths of unbounded length. Rather it says that if a finite resolution path exists, a smart enough evaluator will find it.

1.9 Control Aspects

So far I have emphasized the logic component of Prolog. The language also contains features that exercise control over the evaluation process. Such features in general compromise the otherwise declarative nature of Prolog. (Prolog also contains I/O, testing, and debugging features that are not purely declarative.)

The most frequently used control operator is **cut** (represented by ! in Prolog syntax). **Cut** terminates backtracking within a rule. (**Cut** is similar to **fence** in SNOBOL, described in Chapter 9. SNOBOL patterns use a backtrack mechanism that is very similar to the one Prolog uses.) In particular, if **cut** is encountered, all alternatives prior to **cut** in the rule are frozen, as shown in the box model in Figure 8.36.

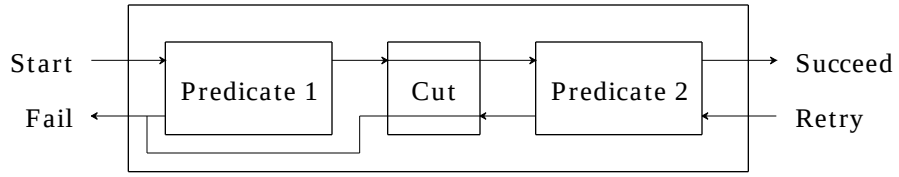


Figure 8.36 `Cut` operator

Prolog will not try alternative matches to earlier conjuncts of the rule's body. In fact, it will not try alternative rules to the rule that contains **cut**. Consider Figure 8.37.

Figure 8.37

```
even(X) :- X=2 , X>0, !, X < 0 .                    1
even(X) :- X=10 .                                   2

in:  even(E) .                                      3
out: No                                             4
```

The query in line 3 matches the rule in line 1. The first conjunct binds `X` to 2, the second succeeds, but the last conjunct fails. The **cut** prevents not only a reevaluation of the first conjuncts (which wouldn't find anything new in any case) but also any attempt to use the rule of line 2, which would succeed. Without the **cut** operation, Prolog would report that `X = 10`.

Cut can be useful in cases where once one rule is found to match, it is unnecessary to try other rules. For example, recall the list membership predicate, `memberOf`, shown in Figure 8.38.

Figure 8.38

```
memberOf(X,[X|_]) .                                 1
memberOf(X,[_|Y]) :- memberOf(X,Y) .                2
```

Since an element may appear in a list more than once, if a goal containing a successful `memberOf` conjunct later fails, Prolog might try to resatisfy it by looking for an alternative match to `memberOf`. In general this search will be futile, since once an item is known to be in a list, backtracking can't establish anything new. Thus, we have Figure 8.39.

Figure 8.39

```
memberOf(X,[X|_]) :- ! .                                    1
memberOf(X,[_|Y]) :- memberOf(X,Y) .                        2
```

This code implements our observation that once membership is established, further backtracking should be avoided. Unfortunately, **cut** changes the meaning of the memberOf rule. If the program uses this rule not to verify that X is a member of some list L but to find a list L with X as a member, **cut** will force X to be the first element in L, which prevents other perfectly good lists from being formed.

Another control operator in Prolog is **fail**, which always fails; that is, it can never be proven true. (Again, SNOBOL has an analogous **fail** pattern.) A program may use **fail** to state that a goal can't be established. For example, given a predicate male(X), I might create the rule in Figure 8.40.

Figure 8.40

```
grandmotherOf(X,GM) :- male(GM) , fail .
```

This rule states that if GM is male, then GM can't be anyone's grandmother. Interestingly, this rule doesn't achieve its intended purpose. The problem is that backtracking takes effect, saying (in effect) that if this rule doesn't work, maybe some other rule will. To avoid backtracking, **cut** must added, as in Figure 8.41.

Figure 8.41

```
grandmotherOf(GM,X) :- male(GM) , ! , fail .
```

Cut is often used in conjunction with **fail** when backtracking is to be suppressed. Another use for **fail** is to create a sort of **while** loop, as in Figure 8.42.

Figure 8.42

```
while(X) :- cond(X) , body(X) , fail .                      1
cond(X) :- memberOf(X, [1,2,3,7]) .                         2
body(X) :- write(X) , write(' ') .                          3

in:  while(_) .                                             4
out: 1 2 3 7                                                5
     No                                                     6
```

Other more complex Prolog operators exist. For example, **assert** and **retract** can be used to add or remove facts and rules from the database while evaluation is in progress. These operators can be used to allow a program to learn, but since they have the flavor of self-modifying code, they are potentially very dangerous.

1.10 An Example of Control Programming

To illustrate some of the subtleties of programming in Prolog, I will consider the canonical (mis-)example of recursive programming, factorial. The obvious definition appears in Figure 8.43.

Figure 8.43

```
fact(N,F) :- N >= 0, N =< 1 , F = 1 .                          1
fact(N,F) :- N > 1 , M is N-1 , fact(M,G) , F is N*G .         2
```

This definition, though correct, is not entirely satisfactory. The problem is that fact can be used in a surprisingly large number of ways, depending on what values are bound (that is, input parameters) and what values are to be computed (that is, output parameters). Possible combinations that should be handled are the following:

1. Both N and F are bound: Fact should succeed or fail depending on whether or not N! = F.
2. N is bound, but F is free: F should be correctly computed. Attempts to resatisfy fact to obtain a different value for F should fail (since N! is unique for bound N).
3. F is bound, but N is free: If an integer N exists such that N! = F, it should be computed; else fact should fail. Attempts to resatisfy fact to obtain a different value for N should fail, except when F=1.
4. Both N and F are free: The initial solution fact(1,1) should be found. Attempts to resatisfy fact to obtain different values for N and F should succeed, producing monotonically increasing (N,F) pairs.

The program in Figure 8.43 works when N is bound, but not when it is free. This isn't too surprising, since the definition I chose is the one used in languages that assume that N must be bound. The problem is that my definition gives no clue as to how to choose a value for N when it is free. I can take a step toward a better solution by dealing with the simpler case in which N is bound.[2] The predeclared predicates bound and free (in some implementations nonvar and var) can be used to determine whether a parameter is bound or not. That is, bound(X) is true whenever X is bound (even temporarily, by resolution during backtrack) to a value. Consider Figure 8.44.

Figure 8.44

```
fact(0,1) .                                                    1
fact(N,F) :- bound(N) , N > 0 , M is N-1 ,                     2
    fact(M, G) , F is N*G .                                    3
fact(N,F) :- free(N) , bound(F) , fact(M,G) , N is M+1 ,       4
    F2 is N*G , F =< F2 , ! , F = F2 .                         5
fact(N,F) :-  free(N) , free(F) , fact(M,G) , N is M+1 ,       6
     F is N*G .                                                7
```

The rule in line 1 defines the base case for factorial. The rule in line 2 covers the case where N is bound and greater than 0. **Cut** is used to prohibit backtracking, since N! is single-valued. The rule in line 3 covers the case where N is bound and greater than 1. It recursively computes or verifies N!. The case in which N is free but F is bound is more interesting. The rule in lines 4–5 covers this case. It computes factorial pairs (using the rule in lines 6–7) until the factorial value is ≥ the bound value of F. At this point it cuts the backtracking. If the computed factorial value F2 is equal to F, it succeeds, and N is correctly bound. Otherwise it fails, indicating that F isn't a factorial value.

[2] This solution was provided by Bill Pugh.

Finally, the general case in which both N and F are free is considered in lines 1 and 6–7. Prolog first matches the base case of fact(0,0), then generates the next solution, fact(1,1), from it, using the rule in lines 6–7. Successive solutions are obtained by building upon the most recently discovered factorial pair.

1.11 Negation

If a query cannot be satisfied by any binding of its free variables, Prolog considers it false. Prolog has a built-in higher-order predicate not that tests for unsatisfiability. (It is higher-order in that it takes a predicate, not a term, as its parameter.) Consider Figure 8.45.

Figure 8.45

```
motherOf(nora, fatima) .                                    1

in:  ?- not(motherOf(nora, jaleh)) .                        2
out: Yes                                                    3

in:  ?- not(motherOf(nora, fatima)) .                       4
out: No                                                     5
```

Line 1 introduces a new fact. Line 2 tests to see if a particular fact is unknown. Because it is, the response is Yes. Line 4 tests to see if a known fact is unknown; it elicits No.

Under the closed-world assumption that facts that cannot be proved are false, the facts and rules known to the program constitute the entire world; no new facts or rules from "outside" will be added that might render a previously unprovable conclusion true. The closed-world assumption is an example of **nonmonotonic reasoning**, which is a property of a logic in which adding information (in the form of facts and rules) can reduce the number of conclusions that can be proved.

It is only safe to use not with all parameters bound. Otherwise, unexpected results may occur, as in Figure 8.46.

Figure 8.46

```
motherOf(nora, fatima) .                                    1

in:  ?- not(motherOf(X,jaleh)) .                            2
out: X=_1                                                   3

in:  ?- not(motherOf(_,jaleh)) .                            4
out: Yes                                                    5

in:  ?- not(motherOf(X, fatima)) .                          6
out: No                                                     7

in:  ?- not(motherOf(nora, Y)) .                            8
out: No                                                     9

in:  ?- not(motherOf(X, fatima)), X=jaleh .                 10
out: No                                                     11
```

```
in:  ?- X=jaleh, not(motherOf(X,fatima)) .                    12
out: X=jaleh                                                  13
```

In line 2, since no facts match motherOf(X,jaleh), any substitution for X serves to satisfy its negation. Prolog returns a don't-care result. Line 4 asks the same query, without expecting a binding; it replaces the free variable X with the don't-care pattern _ . The result, Yes, shows that any substitution works. The next two queries produce surprising results. Lines 6 and 8 present queries where the free variable could be set to a known constant to make the query fail (X could be nora in line 5, and Y could be fatima in line 7); all other settings allow the query to succeed. If Prolog implemented **constructive negation**, it would be able to report X ≠ nora in line 7 and Y ≠ fatima in line 9. But most implementations of Prolog do not provide constructive negation. Prolog can't even represent the answers by a single don't-care result, such as Y=_1, because at least one value is *not* part of the answer. Instead, Prolog gives up and fails. Line 10 tries to suggest a reasonable result: X = jaleh. However, Prolog binds variables from left to right in a query, and X is unbound within the not predicate. Line 12 succeeds in binding X by reordering the query. In short, unbound variables inside not only give the expected result if either all bindings succeed (as in line 2) or no bindings succeed. Intermediate possibilities just lead to failure.

A different problem is shown by Figure 8.47.

Figure 8.47

```
blue(sky) .                                                   1

in:  not(not(blue(X)) .                                       2
out: X = _1                                                   3
```

The query in line 2 begins by unifying blue(X) with blue(sky), binding X to sky. This unification succeeds. The first not therefore fails, causing the binding of X to be lost. The second not therefore succeeds, but X has no binding, so it is presented as a don't-care result.

Negation in rules can also lead to anomalies, as shown in the rule in Figure 8.48.

Figure 8.48

```
wise(X) :- not(wise(X)) .
```

This rule appears to be a conundrum. A person is wise if that person is not wise. In symbolic logic, there is a solution to this puzzle: everyone is wise. The derivation in Figure 8.49 demonstrates this result.

Figure 8.49

```
¬ wise(X) => wise(X)                                          1
¬ ¬ wise(X) ∨ wise(X)                                         2
wise(X) ∨ wise(X)                                             3
wise(X)                                                       4
```

However, Prolog enters an unterminated loop when a query such as wise(murali) is presented. In general, there is no completely accurate way to handle logical negation, and most Prolog implementations don't do very

well with it.

1.12 Other Evaluation Orders

Prolog programmers must sort rules to avoid infinite loops. They must also sort the conjuncts within rules for the sake of efficiency. The naiveSort example in Figure 8.26 (page 243) would fail to terminate if line 1 were written as in Figure 8.50.

Figure 8.50

```
naiveSort(L1,L2) :- inOrder(L2) , permutation(L1,L2) .
```

The Prolog interpreter builds more and more fanciful values of L2 that have nothing at all to do with L1 and fails on each one. Prolog programmers learn to build rules so that the first conjunct generates potential solutions, and the remaining conjuncts test them for acceptability. If the generator builds too many unacceptable results, the rule will be very inefficient.

The fact that rule and conjunct order is so crucial to efficiency detracts from the declarative nature of Prolog. It would be nice if the rules merely stated the desired result, and if the implementation were able to dynamically sort the rules and conjuncts to generate the result efficiently.

One proposal for a different evaluation strategy is found in Specint [Darlington 90]. A static version of the idea, called "sideways information passing," appears in Datalog. The idea is to reorder the conjuncts as they are satisfied, so that attention is directed to the first conjunct that has not yet been satisfied. As each conjunct is satisfied, it is rotated to the *end* of the list of conjuncts; it may be retested (and resatisfied) later if other conjuncts fail in the meantime. The programmer can supply hints for each predicate that suggest what parameters will satisfy that predicate. Predefined predicates have their own hints. For example, Figure 8.51 gives a slightly different version of naiveSort.

Figure 8.51

```
naiveSort(L1,L2) :- permutation(L1,L2) , inOrder(L2) .          1

permutation(X,Y) :- X = Y                                       2
permutation(X|Z,Y) :- delete(X,Y,T) , permutation(Z,T)          3

inOrder([]) .                                                   4
inOrder([_]) .                                                  5
inOrder([A,B|T]) :- A =< B , inOrder([B|T]) .                   6
```

To evaluate naiveSort([1,3,2],result), the evaluator first tries to satisfy the first conjunct of line 1. This conjunct brings it to line 2 to find an acceptable permutation Y of X = [1,3,2]. By default, permutation will first try the empty list for Y. It fails, because it satisfies neither line 2 nor line 3. However, the equality test of line 2 has a default hint: set Y to X. Now permutation(X,Y) is satisfied, so the Specint evaluator moves to the inOrder conjunct of line 1, bringing it to line 6. In line 6, A is 1, B is 3, and T is [2]. The first conjunct succeeds, and inOrder is called recursively on [3,2]. In the recursive call of line 6, A is 3, B is 2, and T is []. The first conjunct fails. The hint for satisfying =< is to interchange the two values. Now line 6 suc-

ceeds (after a brief further recursion), and the new values are backed up to the first instance of line 6. Now A is 1, B is 2, and T is [3]. This instance rechecks the first conjunct. It was previously satisfied, but values have changed. Luckily, the new values still satisfy this conjunct. Evaluation returns to line 1. Now L2 is [1,2,3], and the second conjunct is satisfied. The first conjunct is rechecked. After several instantiations of line 3, this check is satisfied. Specint ends up with something like insertion sort, using quadratic time, instead of Prolog's exponential-time evaluation.

Standard Prolog evaluation starts at the goal and moves to subgoals; this approach is called **top-down evaluation**. Another evaluation order that has been proposed is **bottom-up** evaluation. In its pure form, bottom-up evaluation would mean starting with facts and deriving consequences, both direct and indirect. But this sort of undirected evaluation is unlikely to tend toward the desired goal. Luckily, bottom-up evaluation can be implemented in a more directed fashion. Given a query, some preprocessing based on the top-down tree can lead to insight concerning a reasonable ordering of the conjuncts in the bodies of rules. This insight is based on sideways information passing, which determines what information is passed in variables between the conjuncts. The result is a transformed program that can be executed bottom-up. The bottom-up approach leads to certain simplifications. In particular, the unification algorithm is not needed if every variable that appears in the head of the rule also appears in its body. This restriction does not seem unreasonable. Avoiding unification can be essential in some domains. In particular, strings can be introduced into Prolog with matching rules that match "abcd" to A + B, matching any initial substring to A and the rest to B. Unification is intractable in this setting.

1.13 Constraint-Logic Programming (CLP)

A small extension to Prolog's evaluation mechanism simplifies programs like factorial in Figure 8.44 (page 250). This extension, called constraint-logic programming, or CLP, lets identifiers have a **constrained** status, which lies between bound and free [Fruhwirth 92]. CLP(*R*) is Prolog with constraints expressed with respect to real numbers.

A conjunct in CLP(*R*) such as X < 5 is merely checked if X is bound, but if X is free or constrained, this conjunct introduces a constraint on X. If the new constraint, in combination with previous constraints, makes a variable unsatisfiable, the evaluator must backtrack. The power of this idea can be seen in Figure 8.52.

Figure 8.52

```
in:  Nice(X, Y) :- X = 6 , Y < 5 .                    1
     ?- Nice(A,B) .                                   2
out: A = 6, B < 5                                     3

in:  ?- Nice(A,B) , B > 7 .                           4
out: No                                               5
```

Line 2 is only satisfied by a restricted set of values; the output shows the applicable constraints. When I add a conflicting constraint in line 4, no results can be found. Figure 8.53 is a factorial predicate.

Figure 8.53

```
fact(N,F) :- N <= 1 , F = 1 .                                    1
fact(N,F) :- N >  1 , M = N-1 , fact(M,G) , F = N*G .            2
```

Line 2 does not use the **is** predicate, because my intent is to introduce a constraint, not to perform arithmetic. The query `fact(X,Y)` elicits the following solutions:

```
X ≤ 1, Y = 1
X = Y, 1 < Y ≤ 2.
2 < X ≤ 3, Y = X*(X-1)
3 < X ≤ 4, Y = X*(X-1)*(X-2)
4 < X ≤ 5, Y = X*(X-1)*(X-2)*(X-3)
...
```

If we add the constraint that all numbers are integers, not arbitrary reals, then these values reduce to exact solutions. Figure 8.54 computes Fibonacci numbers.

Figure 8.54

```
fib(0,1).                                                        1
fib(1,1).                                                        2
fib(N, X1 + X2) :- N > 1 , fib(N - 1, X1) , fib(N - 2, X2) .     3
```

This program works correctly no matter whether the parameters are free or bound. Figure 8.55 shows how to multiply two complex numbers.

Figure 8.55

```
complexMultiply(c(R1, I1), c(R2, I2), c(R3, I3)) :-              1
     R3 = R1 * R2 - I1 * I2 ,                                    2
     I3 = R1 * I2 + R2 * I1 .                                    3
```

The functor `c` is used as a pattern. We can give CLP(*R*) queries about `complexMultiply`, as in Figure 8.56.

Figure 8.56

```
in:  ?- complexMultiply(X, c(2,2), c(0,4))                       1
out: X = c(1,1)                                                  2

in:  ?- complexMultiply(X, Y, c(0,0))                            3
out: X = c(A,B), Y = c(C,D), A*C = B*D, A*D = -B*C                4
```

There are rare occasions when a term `someFunctor(3 + 4)` is meant to be different from the term `someFunctor(4 + 3)`. CLP(*R*) allows the programmer to prevent evaluation of the `+` operator in such cases by means of a quoting mechanism.

As a final example of constraint-logic programming, consider the program of Figure 8.57, which finds the circuits that can be built from two available resistors in series and a single voltage source so that the drop in voltage across the second resistor is between 14.5 and 16.25 volts [Jaffar 92]:

Figure 8.57

```
/* available resistors */                                          1
     Resistor(10) .                                               2
     Resistor(14) .                                               3
     Resistor(27) .                                               4
     Resistor(60) .                                               5
     Resistor(100) .                                              6
/* available voltage sources */                                    7
     Voltage(10) .                                                8
     Voltage(20) .                                                9
/* electrical law */                                               10
     Ohm(Voltage, Amperage, Resistance) :-                        11
          Voltage = Amperage * Resistance .                       12
/* query about our circuit */                                     13
?- 14.5 < V2, V2 < 16.25 , /* voltage constraints */              14
     Resistor(R1) , Resistor(R2), /* choice of resistors */       15
     Voltage(V) , /* choice of voltages */                        16
     Ohm(V1,A1,R1) , Ohm(V2,A2,R2), /* electrical laws */        17
     A1 = A2, V = V1 + V2 . /* series circuit */                  18
```

An evaluator might choose to solve linear constraints before nonlinear ones in order to achieve the three solutions.

1.14 Metaprogramming

Chapter 4 shows how a LISP interpreter may be written in LISP. Prolog provides a similar ability. As with LISP, the trick is to make the language homoiconic, that is, to be able to treat programs as data. Programs are just sets of rules (a fact is a rule with a body of `true`). The body of a rule is a comma-separated list of predicates. In addition to the bracket-delimited lists shown earlier, Prolog also accepts simple comma-separated lists.[3] The head of a rule is also a predicate, and the `:-` separator can be treated as an infix binary predicate.

So a program is a set of predicates, and Prolog provides a way to inspect, introduce, and delete the predicates that currently make up the program, that is, to treat the program as data. The `clause` predicate is used for inspecting rules, as in Figure 8.58.

Figure 8.58

```
grandmotherOf(X,GM) :- motherOf(M,GM) , motherOf(X,M) .          1
grandmotherOf(X,GM) :- motherOf(F,GM) , fatherOf(X,F) .          2
```

[3] Comma-separated lists are the underlying concept. The list `a,b,c` is equivalent to `a,(b,c)`. The bracketed list `[H | T]` is syntactic sugar for the predicate `.(H,T)`, where the dot is a binary cons functor.

```
in:  ?- clause(grandmotherOf(A,B),Y).                    3
out: A = _1,                                             4
     B = _2,                                             5
     Y = motherOf(_3 _2), motherOf(_1, _3);              6

     A = _1,                                             7
     B = _2,                                             8
     Y = motherOf(_3, _2), fatherOf(_1, _3)              9

motherOf(janos,hette) .                                  10

in:  ?- clause(MotherOf(X,Y),Z) .                        11
out: X = janos,                                          12
     Y = hette,                                          13
     Z = true                                            14
```

Lines 1–2 reintroduce the `grandmotherOf` predicate that I used before. Line 3 asks for all rules that have a left-hand side matching `grandmotherOf(A,B)`. This line treats `grandmotherOf(A,B)` as a structure, not a predicate. Prolog finds the two results shown, which are expressed in terms of don't-care results. The result `Y` treats `motherOf` as a functor, not a predicate name. This ability to interchange the treatment of structures and predicates is essential in making Prolog homoiconic, because structures are data, whereas predicates are program. Facts, such as the one shown in line 10, are also discovered by `clause`; the second parameter to `clause` matches `true` for facts.

The `clause` predicate can be used to build an evaluation predicate `eval`, as in Figure 8.59.

Figure 8.59

```
eval(true).                                              1
eval((A,B)) :- eval(A), eval(B).                         2
eval(A) :- clause(A,B), eval(B).                         3
```

This set of rules defines standard Prolog evaluation order. Line 1 is the base case. Line 2 indicates how to evaluate a list of conjuncts. (The parenthesized list notation `(A,B)` matches any list with at least two elements; the first matches `A`, and the rest match `B`. This alternative list notation is not interchangeable with `[A | B]`; it is a historical relic.) Line 3 shows how to evaluate a single conjunct that happens to match the left-hand side of some rule or fact. (Some implementations of Prolog have a more generous implementation of `clause`; for these, I would need to introduce **`cut`** at the start of the body of line 2.)

The fact that you can write your own evaluator means that you can override the standard interpretation of conjuncts. Building new evaluators is called **metaprogramming**. The ordinary Prolog evaluator remains available to metaprograms as the `call` predicate. The exercises pursue these ideas.

So far, I have concentrated on the static aspect of Prolog, which treats rules as an unchangeable set of givens. Prolog also allows rules to be introduced and deleted during the execution of queries by using `assert` and `retract`, as in Figure 8.60.

Figure 8.60

```
allow(X) :- assert(person(zul)) ,                                    1
     assert(person(Y) :- Y=kealoha), person(X).                      2
deny(X) :- retract(person(X)).                                       3

in:  ?- person(X) .                                                  4
out: No                                                              5

in:  allow(fred) .                                                   6
out: No                                                              7

in:  ?- person(X) .                                                  8
out: X = zul;                                                        9
     X = kealoha                                                     10

in:  ?- deny(beruria) .                                              11
out: No                                                              12

in:  ?- person(X) .                                                  13
out: X = zul;                                                        14
     X = kealoha                                                     15

in:  ?- deny(zul) .                                                  16
out: Yes                                                             17

in:  ?- person(X) .                                                  18
out: X = kealoha                                                     19

in:  ?- deny(kealoha) .                                              20
out: No                                                              21

in:  ?- retract(person(X) :- Y) .                                    22
out: X = _1,                                                         23
     Y = _1 = kealoha                                                24

in:  ?- person(X) .                                                  25
out: No                                                              26
```

Lines 1–3 introduce rules that, when evaluated, cause facts and rules to be introduced and deleted. Lines 4–5 show that at the start, nobody is known to be a `person`. The query in line 6 fails, but it still manages to execute two `assert` conjuncts, which introduce new rules. It is valid to introduce duplicate rules; Prolog does not automatically check for or remove duplicates. Lines 8–10 prove that new rules have been introduced. Evaluating the `deny` predicate in line 11 tries to retract a rule that is not present; it fails. However, `deny(zul)` (line 16) succeeds and does retract a rule. Line 20 tries to retract the fact `person(kealoha)`. However, this predicate does not correspond to any fact, even though it is currently derivable that `kealoha` is a `person`. The way to remove that derivation is by retracting the rule. Line 22 retracts the first rule whose head is `person(X)`. There is only one such rule, and `retract` succeeds in removing it.

All asserted facts and rules are added to the top-level environment. In fact, Prolog rules do not follow scope; there is only one environment. A program can simulate dynamic scope for rules, however, by introducing new

rules in the first conjunct of a body and retracting those rules in the last conjunct. The programmer must make sure that the rules are retracted in case the intervening conjuncts fail. Also, the programmer must be careful not to introduce rules with heads that already exist in the environment; such introduced rules will be added to, not replace, existing rules, and retraction might accidentally remove more rules than are intended.

2 ◆ GÖDEL

Gödel, developed by P. M. Hill, is intended to be the successor of Prolog [Hill 94]. It borrows much of Prolog's form, but attempts to address many of the problems and deficiencies of Prolog. It provides modules to make the language suitable for large projects, has a strong type system, permits enhanced logical forms, and has more consistent search-pruning operators. Gödel also directly supports integers, floats, rational numbers, strings, lists, and sets.

2.1 Program Structure

A program in Gödel consists of at least one module. Each module is divided into several parts, most of which are optional.

- **module** names the module. Modules can also be declared **closed** (which means the implementation is not provided, and may well be in a different language), **export** (all definitions are exported to other modules), or **local** (all definitions are private).
- **import** lists the modules whose declarations are imported.
- **base** declares types.
- **constructor** declares type constructors.
- **constant** declares constants.
- **function** lists functions and declares their type. (These are like functors in Prolog.)
- **predicate** lists predicates and defines their type.
- **delay** lists conditions for controlling evaluation of predicates.
- **proposition** lists propositions.
- Finally, there is a list of rules.

For example, the module in Figure 8.61 calculates factorials.

Figure 8.61

```
module Factorial.                                          1
import Integers.                                           2
predicate Fact : Integer * Integer.                        3

Fact(0,1).                                                 4
Fact(1,1).                                                 5
Fact(n,f) <- n > 1 & Fact(n-1,g) & f = g * n.              6
```

The module is named `Factorial` and imports types, functions, and predicates from the (library) module `Integers`. It has one predicate, `Fact`, which has two integer parameters. Three rules define `Fact` (lines 4–6). The program is executed by supplying a query, as in Figure 8.62.

Figure 8.62

```
in:  <- Fact(4,x).                                              1
out: x = 24                                                     2
```

2.2 Types

Figure 8.63 illustrates construction of programmer-defined types.

Figure 8.63

```
module M1.                                                      1
base Day, ListOfDay.                                            2
constant                                                        3
     Nil : ListOfDay;                                           4
     Monday, Tuesday, Wednesday, Thursday, Friday,              5
          Saturday, Sunday : Day.                               6
function Cons : Day * ListOfDay -> ListOfDay.                   7
predicate Append : ListOfDay * ListOfDay * ListOfDay.           8

/* Append(a,b,c) means list a appended to list b;              9
     results in list c. */                                      10
Append(Nil,x,x).                                                11
Append(Cons(u,x),y,Cons(u,z)) <- Append(x,y,z).                  12
```

Day and ListOfDay (line 2) are the only types of this program. Cons (line 7) is not a pattern symbol, as it would be in Prolog, but rather a function. Every constant, function, proposition, and predicate of the language must be declared, but variable types are inferred, as in ML. Constructors can be used in type declarations. They may be applied to the ground types defined in the **base** clause to create new types. This process can be recursively applied to make an infinite number of types. I can improve this module by making the concept of list polymorphic, as in Figure 8.64.

Figure 8.64

```
module M2.                                                      1
base Day, Person.                                               2
constructor List/1.                                             3
constant                                                        4
     Nil : List('a);                                            5
     Monday, Tuesday, Wednesday, Thursday, Friday,              6
          Saturday, Sunday : Day;                               7
     Fred, Barney, Wilma, Betty : Person.                       8
function Cons : 'a * List('a) -> List('a).                      9
predicate Append : List('a) * List('a) * List('a).              10

Append(Nil,x,x).                                                11
Append(Cons(u,x),y,Cons(u,z)) <- Append(x,y,z).                  12
```

The constructor List (line 3) is followed by an integer indicating its arity. The identifier 'a in lines 5, 9, and 10 is a type identifier. The types for this program are Day, Person, List(Day), List(Person), List(List(Day)), and so forth.

LISP-like lists form such a common structure in declarative programming that Gödel, like Prolog, predeclares the List constructor, the Cons function,

and the Nil constant. The constructors for lists are the same as in Prolog; the list Cons(Fred,Cons(Bill,x)) can be written as [Fred, Bill | x].

2.3 Logic Programming

Unlike Prolog programs, Gödel programs are not limited to Horn clauses. The following quantifiers and connectives are allowed.

Symbol	Meaning
&	conjunction (**and**)
\/	disjunction (**or**)
~	negation (**not**)
<-	implication
->	right implication
<->	equivalence
all	universal quantifier
some	existential quantifier

The quantifiers have two parameters, a list of variables and the body, as in Figure 8.65.

Figure 8.65

```
module Inclusion.                                                    1
import Lists.                                                        2
predicate IncludedIn : List(a) * List(a)                             3
     -- IncludedIn(a,b) means list a is included in list b.          4

IncludedIn(x,y) <- all [z] (MemberOf(z,y) <- MemberOf(z,x)).         5
     -- MemberOf(a,b) means element a is a member of list b.         6
```

The rule in line 5 indicates that list x is included in list y if all members of x are also members of y. This example also illustrates some of the use of modules. The predicate MemberOf (used in line 5) is declared in the imported module Lists.

Queries are quite simple to write. For example, assume that a module has been declared with the classic family relationship predicates and facts FatherOf, MotherOf, ParentOf, AncestorOf, and so forth. Then the query "Does everyone who has a mother also have a father?" can be written, as in Figure 8.66.

Figure 8.66

```
<- all [x]                                                           1
     (some [z] FatherOf(x,z) <- some [y] MotherOf(x,y))).            2
```

In practice, **some** is seldom used, because Gödel provides _ as a don't-care pattern. The above query can be written more simply as in Figure 8.67.

Figure 8.67

```
<- all [x] (FatherOf(x,_) <- MotherOf(x,_))
```

The quantifier **some** may also be used in queries to restrict the display of results to the variables of interest to the user. For example, the query in Figure 8.68

Figure 8.68

```
<- some [y] (ParentOf(x,y) & ParentOf(y,Jane)).
```

will display the value to which x is bound, but not y. The same query can be written using the colon notation (: is read "such that"), as in Figure 8.69.

Figure 8.69

```
<- x : ParentOf(x,y) & ParentOf(y,Jane).
```

2.4 Conditionals

Gödel allows the use of conditional statements primarily as a concession to computational efficiency. The structure **if** condition **then** formula is logically equivalent to condition -> formula. The semantics of conditionals differ procedurally from implications, however. Unlike implications, the evaluation of a conditional waits until the condition has no free variables.

If the condition and the formula share local variables, the form in Figure 8.70 is used.

Figure 8.70

```
if some [r1, ..., rn] condition then formula
```

This form is equivalent to that in Figure 8.71.

Figure 8.71

```
(some [r1, ..., rn] (condition & formula)) \/                    1
~ some [r1, ..., rn] condition.                                  2
```

The **if** construct is defined similarly, but oddly enough, the rule for resolving the dangling-**else** problem is contrary to standard convention.

The module for defining LISP-like association lists in Figure 8.72 illustrates conditionals.

Figure 8.72

```
module AssocList.                                                1
import Strings.                                                  2
base PairType.                                                   3
function Pair : Integer * String -> PairType.                    4
predicate Lookup : Integer * String * List(PairType)             5
     * List(PairType).                                           6
```

```
Lookup(key, value, assoc_list, new_assoc_list) <-                    7
     if some [v]                                                     8
          MemberOf(Pair(key,v), assoc_list)                          9
     then                                                            10
          value = v &                                                11
          new_assoc_list = assoc_list                                12
     else                                                            13
          new_assoc_list = [Pair(key,value) | assoc_list].           14
```

2.5 Control

Logic programming in its pure form allows the parameters of predicates to be arbitrarily bound or unbound. As you saw in Prolog, it is often difficult (and unnecessary) to write rules that cover all cases. Gödel uses something like Prolog's bound predicate, but it enhances it with a control structure that delays evaluation of predicates until certain conditions are met. Predicates in a conjunction can be processed like coroutines. For example, the definition of Permutation in Figure 8.73 might be placed in the Lists module.

Figure 8.73

```
predicate Permutation : List(a) * List(a).                           1
     -- Permutation(a,b) means list a is                             2
     -- a permutation of list b                                      3
delay Permutation(x,y) until bound(x) \/ bound(y).                   4

Permutation([],[]).                                                  5
Permutation([x | y],[u | v]) <-                                      6
     Delete(u,[x | y],z) & Permutation(z,v).                         7
     -- Delete(a,b,c) means deleting element a                       8
     -- from list b gives list c                                     9
```

The **delay** construct in line 4 causes Permutation to pause until one of its parameters is bound. If it is invoked with both parameters unbound and no other predicates can be explored to bind one of the parameters, as in Figure 8.74, Permutation will fail. (This behavior is nonmonotonic.)

Figure 8.74

```
in:  <- Permutation(x,y).                                            1
out: No                                                              2

in:  <- Permutation(x,y) & x = [1,2].                                3
out: x = [1,2], y = [1,2];                                           4
     x = [1,2], y = [2,1]                                            5
```

In line 1, neither parameter is bound, so the query fails. In line 3, evaluation of Permutation delays until the second conjunct is evaluated. That conjunct binds x to a value, so now Permutation may be invoked.

In order to build a sort program similar to naiveSort in Figure 8.26 (page 243), I will introduce in Figure 8.75 a Sorted predicate for the Lists module.

Figure 8.75

```
predicate Sorted : List(integer).                                    1
delay                                                                2
     Sorted([]) until true;                                          3
     Sorted([_]) until true;                                         4
     Sorted([x,[y|_]) until bound(x) & bound(y).                     5

Sorted([]).                                                          6
Sorted([_]).                                                         7
Sorted([x,y|z]) <- x =< y & Sorted([y|z]).                           8
```

The **delay** construct in line 2 takes multiple patterns, each of which may be treated differently. If Sorted is invoked with a list of length 0 or 1, it will not delay, even if the list element is not bound (lines 3–4). For more complex lists, Sorted will be delayed until the first two elements of the list are both bound. I can now use Sorted and Permutation to define a Sort module, as in Figure 8.76.

Figure 8.76

```
module Sort.                                                         1
import Lists. -- brings in Sorted and Permutation                    2
predicate SlowSort : List(Integer) * List(Integer).                  3

SlowSort(x,y) <- Sorted(y) & Permutation(x, y).                      4
     -- SlowSort(a,b) means list a is sorted to produce b.           5
```

SlowSort works if either x or y (or both) is bound. If y is bound, Sorted will not be delayed. If Sorted succeeds, Permutation will either verify that x is a permutation of y if x is bound or find all values of x that are permutations of y. If y is not bound and x is, then Sorted(y) will delay, and Permutation will instantiate y to a permutation of x. The entire permutation need not be produced before Sorted continues; only the first two elements of the permutation are necessary. Perhaps these elements are enough to show that y is not sorted, and this permutation may be abandoned. If the first two elements of y are in order, Sorted makes a recursive call, which may again be delayed. At this point, Permutation will continue producing more elements of the permutation. If Sorted fails at any point, Permutation will backtrack to an alternative permutation. In effect, **delay** gives Gödel lazy evaluation, which makes SlowSort much faster than Prolog's equivalent naiveSort in Figure 8.26 (page 243). It is still a very poor sorting method, however.

The **delay** construct also helps the programmer to guarantee termination. Consider the definition of Delete in Figure 8.77, which is needed to define Permutation in Figure 8.73 (page 263).

Figure 8.77

```
-- Delete(a,b,c) means deleting element a                            1
-- from list b gives list c                                          2
predicate Delete : a * List(a) * List(a).                            3
delay Delete(_,y,z) until bound(y) \/ bound(z).                      4

Delete(x,[x|y],y).                                                   5
Delete(x,[y|z],[y|w]) <- Delete(x,z,w).                              6
```

Without the **delay** in line 4, the query Permutation(x, [1, 2, 3]) would first

produce the answer x = [1, 2, 3] and then go into an infinite loop. With the **delay** present, the six possible permutations are produced, and then Permutation fails.

Gödel also allows the programmer to prune the backtrack search tree in a fashion similar to Prolog's **cut**, but in a more consistent fashion. The simplest version of pruning is **bar commit**. **Bar commit** works like conjunction but with the added meaning that only one solution will be found for the formula in its scope (to its left); all branches arising from other statements for the same predicate are pruned. The order of statement evaluation is not specified, so **bar commit** lacks the sequential property of **cut**. For example, a Partition predicate to be used in Quicksort is shown in Figure 8.78.

Figure 8.78

```
-- Partition(a,b,c,d) means list a partitioned about                    1
-- element b results in lists c and d.                                  2
predicate Partition : List(Integer) * Integer *                         3
     List(Integer) * List(Integer)                                      4
delay                                                                   5
     Partition([],_,_,_) until true;                                    6
     Partition([u|_],y,_,_) until bound(u) & bound(y).                  7

Partition([],y,[],[]) <- bar.                                           8
Partition([x|xs],y,[x|ls],rs) <- x =< y bar                             9
     Partition(xs,y,ls,rs).                                             10
Partition([x|xs],y,ls,[x|rs]) <- x > y bar                              11
     Partition(xs,y,ls,rs).                                             12
```

In this case, I use **bar commit** (I denote it just by **bar** in lines 8–11) because the statements in the definition of Partition (lines 8–12) are mutually exclusive, so it prunes useless computation. It is possible to use **bar commit** to prune answers as well.

Gödel also provides **singleton commit**. A formula enclosed in curly brackets { and } will only produce one answer. For example, the query {Permutation([1,2,3], x)} will produce only one of the permutations instead of all six.

The **delay** construct can prevent premature **bar commit**s that could lead to unexpected failures. Figure 8.79 shows how to code the Delete predicate differently.

Figure 8.79

```
-- Delete(a,b,c) means deleting element a                               1
-- from list b gives list c                                             2
predicate Delete : Integer * List(Integer) * List(Integer).             3
delay Delete(x,[u|_],_) until bound(x) & bound(u).                      4

Delete(x,[x|y],y) <- bar .                                              5
Delete(x,[y|z],[y|w]) <- x ~= y bar Delete(x,z,w).                      6
```

If the program did not delay Delete until x was bound, the query Delete(x, [1, 2, 3], y) & x = 2 could commit with x bound to 1, by line 5. Then x = 2 would fail, causing the entire query to fail, because the **bar** in line 6 prevents backtracking for another binding to x.

3 ◆ FINAL COMMENTS

In some ways, Prolog is a hybrid of three ideas: LISP data structures, recursive pattern matching as in SNOBOL, and resolution theorem-proving. As a programming language, Prolog lacks mechanisms for structuring programs and has no type facilities. It is hard to read Prolog programs, because the order of parameters in predicates is not always obvious. This is a problem in other languages, but it seems especially severe in Prolog. For all its shortcomings, Prolog is widely used, especially in Europe, for artificial intelligence rule-based programs. It has been successfully used in such applications as genetic sequence analysis, circuit design, and stock-market analysis.

Enhancements of Prolog to give it an understanding of constraints and to organize search differently reduce the difficulty of writing clear and efficient programs. Various constraint-based extensions to Prolog have been developed, including CLP(Σ*), which understands regular expressions [Walinsky 89], and versions that deal with strings in general [Rajasekar 94].

Concurrent logic programming is an active research topic. All the conjuncts in the body of a rule can be evaluated simultaneously, with bindings of common variables communicated as they arise between otherwise independent evaluators. This technique is called **and**-parallelism. Similarly, multiple rules whose heads match a goal can be evaluated simultaneously; this technique is called **or**-parallelism. Research topics include the ramifications of using shared and distributed memory, how to manage bindings for variables, how much parallelism can be discovered by the compiler in ordinary Prolog programs, and how the programmer can assist that task in extensions to Prolog. One such extension, called Guarded Horn Clauses, allows guard predicates, much like the guards in Ada's **select** statement (discussed in Chapter 7), to restrict the rules that are to be considered during concurrent evaluation. Much of the literature on concurrent logic programming has been surveyed by Shapiro [Shapiro 89].

Gödel manages to blend Prolog with strong typing, some type polymorphism, and modularization, while increasing the range of logical operators. It also provides lazy evaluation, which makes some naive programs far more efficient. However, it is a much more complex language; one of Prolog's advantages is its relative simplicity.

There are other languages specifically intended for knowledge-based reasoning. In particular, OPS5 shares with Prolog and Gödel the concept of rules, facts, and queries [Brownston 86]. It is based on an inference engine, which repeatedly (1) determines which rules match existing facts, (2) selects one of those rules based on some strategy, and then (3) applies the actions specified in the selected rule, usually adding to or altering the set of facts. Step 1 can be extremely costly, but step 3 can propagate changes to a data structure to make step 1 reasonably efficient.

EXERCISES

Review Exercises

8.1 Figure 8.10 (page 236) shows the backtrack tree for the query `grandmotherOf(tom,X)`. Show the backtrack tree with the rules for `grandmotherOf` in Figure 8.8 (page 234) reordered as in Figure 8.80.

Figure 8.80

```
grandmotherOf(X,GM) :- motherOf(X,M) , motherOf(M,GM) .     1
grandmotherOf(X,GM) :- fatherOf(X,F) , motherOf(F,GM) .     2
```

8.2 It appears that the rules on lines 3 and 4 of Figure 8.23 (page 242) can be replaced by the single rule `append([],X,X)`. Is this so?

8.3 What is the result of the query in Figure 8.81?

Figure 8.81

```
in:  ?- append([1,2],X,[1,2,3,4]) .
```

8.4 Modify the `eval` rules in Figure 8.59 (page 257) so that bodies are interpreted from right to left, that is, with the last conjunct first.

8.5 Design a functor `fraction` with two parameters (the numerator and denominator) and predicate `lessThan` that takes two fractions and is satisfied if the first fraction is less than the second. The `lessThan` predicate does not need to be defined for unbound parameters.

Challenge Exercises

8.6 Does Prolog have static or dynamic scope rules for formal parameters?

8.7 Are predicates in Prolog first-, second-, or third-class values? How about predicate names, functors, and terms?

8.8 Show how to build a stack containing {1,2,3} and to verify that it is a stack, using the definitions of Figure 8.20 (page 240).

8.9 In Figure 8.20 (page 240), I defined nonhomogeneous stacks. Show how the existence of a built-in `integer` predicate allows you to define integer stacks.

8.10 In Figure 8.20 (page 240), `pop` is a predicate name. Rewrite this example so that `pop` is a functor.

8.11 In Figure 8.26 (page 243), how many solutions are there to `naiveSort([11,2,11],S)`?

8.12 In Figure 8.26 (page 243), how many solutions are there to `naiveSort(S,[1,2])`?

8.13 As an alternative to `naiveSort` and `bubbleSort`, encode `insertionSort` in Prolog. Make sure your program works correctly in all four cases of `insertionSort`(α,β), whether α or β is a constant or a variable.

8.14 In Figure 8.33 (page 246), modify the definition of odd so that odd(2) evaluates to No instead of leading to an infinite computation.

8.15 In Chapter 2, a CLU program is shown for generating all binary trees with *n* nodes. Write a Prolog program that accomplishes the same task.

8.16 Prolog's **cut** operator is not quite the same as SNOBOL's **fence**, which only freezes alternatives selected within the current body, but does not prohibit the evaluator from trying other rules whose heads match. How can we achieve SNOBOL semantics in Prolog?

8.17 Modify the eval rules in Figure 8.59 (page 257) so that eval takes an additional parameter, which matches a tree that shows the subgoals that succeed leading to each result.

8.18 What is the complexity of SlowSort in Figure 8.76 (page 264) when x is bound and y is free?

8.19 Write SlowSort from Figure 8.76 (page 264) using CLU iterators to achieve the lazy evaluation needed.