24

Prolog

This chapter describes how to write Prolog as an embedded language. Chapter 19 showed how to write a program which answered complex queries on databases. Here we add one new ingredient: rules, which make it possible to infer facts from those already known. A set of rules defines a tree of implications. In order to use rules which would otherwise imply an unlimited number of facts, we will search this implication tree nondeterministically.

Prolog makes an excellent example of an embedded language. It combines three ingredients: pattern-matching, nondeterminism, and rules. Chapters 18 and 22 give us the first two independently. By building Prolog on top of the pattern-matching and nondeterministic choice operators we have already, we will have an example of a real, multi-layer bottom-up system. Figure 24.1 shows the layers of abstraction involved.

The secondary aim of this chapter is to study Prolog itself. For experienced programmers, the most convenient explanation of Prolog may be a sketch of its implementation. Writing Prolog in Lisp is particularly interesting, because it brings out the similarities between the two languages.

24.1 Concepts

Chapter 19 showed how to write a database system which would accept complex queries containing variables, and generate all the bindings which made the query true in the database. In the following example, (after calling `clear-db`) we assert two facts and then query the database:

Figure 24.1: Layers of abstraction.

```
> (fact painter reynolds)
(REYNOLDS)
> (fact painter gainsborough)
(GAINSBOROUGH)
> (with-answer (painter ?x)
    (print ?x))
GAINSBOROUGH
REYNOLDS
NIL
```

Conceptually, Prolog is the database program with the addition of *rules*, which make it possible to satisfy a query not just by looking it up in the database, but by inferring it from other known facts. For example, if we have a rule like:

If   `(hungry ?x)` and `(smells-of ?x turpentine)`
Then `(painter ?x)`

then the query `(painter ?x)` will be satisfied for `?x` = `raoul` when the database contains both `(hungry raoul)` and `(smells-of raoul turpentine)`, even if it doesn't contain `(painter raoul)`.

In Prolog, the if-part of a rule is called the *body*, and the then-part the *head*. (In logic, the names are *antecedent* and *consequent*, but it is just as well to have separate names, to emphasize that Prolog inference is not the same as logical implication.) When trying to establish bindings[1] for a query, the program looks first at the head of a rule. If the head matches the query that the program is trying to answer, the program will then try to establish bindings for the body of the rule. Bindings which satisfy the body will, by definition, satisfy the head.

The facts used in the body of the rule may in turn be inferred from other rules:

[1]Many of the concepts used in this chapter, including this sense of bindings, are explained in Section 18.4.

If   `(gaunt ?x)` or `(eats-ravenously ?x)`
Then `(hungry ?x)`

and rules may be recursive, as in:

If   `(surname ?f ?n)` and `(father ?f ?c)`
Then `(surname ?c ?n)`

Prolog will be able to establish bindings for a query if it can find some path through the rules which leads eventually to known facts. So it is essentially a search engine: it traverses the tree of logical implications formed by the rules, looking for a successful path.

Though rules and facts sound like distinct types of objects, they are conceptually interchangeable. Rules can be seen as virtual facts. If we want our database to reflect the discovery that big, fierce animals are rare, we could look for all the *x* such that there are facts `(species` *x*`)`, `(big` *x*`)`, and `(fierce` *x*`)`, and add a new fact `(rare` *x*`)`. However, by defining a rule to say

If   `(species ?x)` and `(big ?x)` and `(fierce ?x)`
Then `(rare ?x)`

we get the same effect, without actually having to add all the `(rare` *x*`)` to the database. We can even define rules which imply an infinite number of facts. Thus rules make the database smaller at the expense of extra processing when it comes time to answer questions.

Facts, meanwhile, are a degenerate case of rules. The effect of any fact *F* could be duplicated by a rule whose body was always true:

If   true
Then *F*

To simplify our implementation, we will take advantage of this principle and represent facts as bodyless rules.

24.2 An Interpreter

Section 18.4 showed two ways to define `if-match`. The first was simple but inefficient. Its successor was faster because it did much of its work at compile-time. We will follow a similar strategy here. In order to introduce some of the topics involved, we will begin with a simple interpreter. Later we will show how to write the same program much more efficiently.

```
(defmacro with-inference (query &body body)
 `(progn
    (setq *paths* nil)
    (=bind (binds) (prove-query ',(rep_ query) nil)
      (let ,(mapcar #'(lambda (v)
                        `(,v (fullbind ',v binds)))
                    (vars-in query #'atom))
        ,@body
        (fail)))))

(defun rep_ (x)
  (if (atom x)
      (if (eq x '_) (gensym "?") x)
      (cons (rep_ (car x)) (rep_ (cdr x)))))

(defun fullbind (x b)
  (cond ((varsym? x) (aif2 (binding x b)
                           (fullbind it b)
                           (gensym)))
        ((atom x) x)
        (t (cons (fullbind (car x) b)
                 (fullbind (cdr x) b)))))

(defun varsym? (x)
  (and (symbolp x) (eq (char (symbol-name x) 0) #\?)))
```

Figure 24.2: Toplevel macro.

Figures 24.2–24.4 contain the code for a simple Prolog interpreter. It accepts the same queries as the query interpreter of Section 19.3, but uses rules instead of the database to generate bindings. The query interpreter was invoked through a macro called `with-answer`. The interface to the Prolog interpreter will be through a similar macro, called `with-inference`. Like `with-answer`, `with-inference` is given a query and a series of Lisp expressions. Variables in the query are symbols beginning with a question mark:

```
(with-inference (painter ?x)
  (print ?x))
```

A call to `with-inference` expands into code that will evaluate the Lisp expressions for each set of bindings generated by the query. The call above, for example, will print each *x* for which it is possible to infer `(painter x)`. ○

Figure 24.2 shows the definition of `with-inference`, together with the function it calls to retrieve bindings. One notable difference between `with-answer` and `with-inference` is that the former simply collected all the valid bindings. The new program searches nondeterministically. We see this in the definition of `with-inference`: instead of expanding into a loop, it expands into code which will return one set of bindings, followed by a `fail` to restart the search. This gives us iteration implicitly, as in:

```
> (choose-bind x '(0 1 2 3 4 5 6 7 8 9)
    (princ x)
    (if (= x 6) x (fail)))
0123456
6
```

The function `fullbind` points to another difference between `with-answer` and `with-inference`. Tracing back through a series of rules can build up binding lists in which the binding of a variable is a list of other variables. To make use of the results of a query we now need a recursive function for retrieving bindings. This is the purpose of `fullbind`:

```
> (setq b '((?x . (?y . ?z)) (?y . foo) (?z . nil)))
((?X ?Y . ?Z) (?Y . FOO) (?Z))
> (values (binding '?x b))
(?Y . ?Z)
> (fullbind '?x b)
(FOO)
```

Bindings for the query are generated by a call to `prove-query` in the expansion of `with-inference`. Figure 24.3 shows the definition of this function and the functions it calls. This code is structurally isomorphic to the query interpreter described in Section 19.3. Both programs use the same functions for matching, but where the query interpreter used mapping or iteration, the Prolog interpreter uses equivalent *choose*s.

Using nondeterministic search instead of iteration does make the interpretation of negated queries a bit more complex. Given a query like

```
(not (painter ?x))
```

the query interpreter could just try to establish bindings for `(painter ?x)`, returning `nil` if any were found. With nondeterministic search we have to be more careful: we don't want the interpretation of `(painter ?x)` to fail back outside the scope of the `not`, nor do we want it to leave saved paths that might

```
(=defun prove-query (expr binds)
  (case (car expr)
    (and  (prove-and (cdr expr) binds))
    (or   (prove-or (cdr expr) binds))
    (not  (prove-not (cadr expr) binds))
    (t    (prove-simple expr binds))))

(=defun prove-and (clauses binds)
  (if (null clauses)
      (=values binds)
      (=bind (binds) (prove-query (car clauses) binds)
        (prove-and (cdr clauses) binds))))

(=defun prove-or (clauses binds)
  (choose-bind c clauses
    (prove-query c binds)))

(=defun prove-not (expr binds)
  (let ((save-paths *paths*))
    (setq *paths* nil)
    (choose (=bind (b) (prove-query expr binds)
              (setq *paths* save-paths)
              (fail))
            (progn
              (setq *paths* save-paths)
              (=values binds)))))

(=defun prove-simple (query binds)
  (choose-bind r *rlist*
    (implies r query binds)))
```

Figure 24.3: Interpretation of queries.

be restarted later. So now the test for `(painter ?x)` is done with a temporarily empty list of saved states, and the old list is restored on the way out.

Another difference between this program and the query interpreter is in the interpretation of simple patterns—expressions such as `(painter ?x)` which consist just of a predicate and some arguments. When the query interpreter generated bindings for a simple pattern, it called `lookup` (page 251). Now, instead of calling `lookup`, we have to get any bindings implied by the rules.

```
(defvar *rlist* nil)

(defmacro <- (con &rest ant)
  (let ((ant (if (= (length ant) 1)
                 (car ant)
                 `(and ,@ant))))
    `(length (conc1f *rlist* (rep_ (cons ',ant ',con))))))

(=defun implies (r query binds)
  (let ((r2 (change-vars r)))
    (aif2 (match query (cdr r2) binds)
          (prove-query (car r2) it)
          (fail))))

(defun change-vars (r)
  (sublis (mapcar #'(lambda (v)
                      (cons v (symb '? (gensym))))
                  (vars-in r #'atom))
          r))
```

Figure 24.4: Code involving rules.

```
⟨rule⟩       : (<- ⟨sentence⟩ ⟨query⟩)
⟨query⟩      : (not ⟨query⟩)
             : (and ⟨query⟩*)
             : (or ⟨query⟩*)
             : ⟨sentence⟩
⟨sentence⟩   : (⟨symbol⟩ ⟨argument⟩*)
⟨argument⟩   : ⟨variable⟩
             : ⟨symbol⟩
             : ⟨number⟩
⟨variable⟩   : ?⟨symbol⟩
```

Figure 24.5: Syntax of rules.

Code for defining and using rules is shown in Figure 24.4. The rules are kept in a global list, `*rlist*`. Each rule is represented as a dotted pair of body and head. At the time a rule is defined, all the underscores are replaced with unique variables.

The definition of `<-` follows three conventions often used in programs of this type:

1. New rules are added to the end rather than the front of the list, so that they will be applied in the order that they were defined.
2. Rules are expressed head first, since that's the order in which the program examines them.
3. Multiple expressions in the body are within an implicit `and`.

The outermost call to `length` in the expansion of `<-` is simply to avoid printing a huge list when `<-` is called from the toplevel.

The syntax of rules is given in Figure 24.5. The head of a rule must be a pattern for a fact: a list of a predicate followed by zero or more arguments. The body may be any query that could be handled by the query interpreter of Chapter 19. Here is the rule from earlier in this chapter:

```
(<- (painter ?x) (and (hungry ?x)
                      (smells-of ?x turpentine)))
```

or just

```
(<- (painter ?x) (hungry ?x)
                 (smells-of ?x turpentine))
```

As in the query interpreter, arguments like `turpentine` do not get evaluated, so they don't have to be quoted.

When `prove-simple` is asked to generate bindings for a query, it nondeterministically chooses a rule and sends both rule and query to `implies`. The latter function then tries to match the query with the head of the rule. If the match succeeds, `implies` will call `prove-query` to establish bindings for the body. Thus we recursively search the tree of implications.

The function `change-vars` replaces all the variables in a rule with fresh ones. An `?x` used in one rule is meant to be independent of one used in another. In order to avoid conflicts with existing bindings, `change-vars` is called each time a rule is used.

For the convenience of the user, it is possible to use `_`(underscore) as a wildcard variable in rules. When a rule is defined, the function `rep_` is called to change each underscore into a real variable. Underscores can also be used in the queries given to `with-inference`.

24.3 Rules

This section shows how to write rules for our Prolog. To start with, here are the two rules from Section 24.1:

```
(<- (painter ?x) (hungry ?x)
                 (smells-of ?x turpentine))

(<- (hungry ?x) (or (gaunt ?x) (eats-ravenously ?x)))
```

If we also assert the following facts:

```
(<- (gaunt raoul))
(<- (smells-of raoul turpentine))
(<- (painter rubens))
```

Then we will get the bindings they generate according to the order in which they were defined:

```
> (with-inference (painter ?x)
    (print ?x))
RAOUL
RUBENS
@
```

The `with-inference` macro has exactly the same restrictions on variable binding as `with-answer`. (See Section 19.4.)

We can write rules which imply that facts of a given form are true for all possible bindings. This happens, for example, when some variable occurs in the head of a rule but not in the body. The rule

```
(<- (eats ?x ?f) (glutton ?x))
```

Says that if `?x` is a glutton, then `?x` eats everything. Because `?f` doesn't occur in the body, we can prove any fact of the form `(eats ?x` *y*`)` simply by establishing a binding for `?x`. If we make a query with a literal value as the second argument to `eats`,

```
> (<- (glutton hubert))
7
> (with-inference (eats ?x spinach)
    (print ?x))
HUBERT
@
```

then any literal value will work. When we give a variable as the second argument:

```
> (with-inference (eats ?x ?y)
    (print (list ?x ?y)))
(HUBERT #:G229)
@
```

we get a gensym back. Returning a gensym as the binding of a variable in the query is a way of signifying that *any* value would be true there. Programs can be written explicitly to take advantage of this convention:

```
> (progn
    (<- (eats monster bad-children))
    (<- (eats warhol candy)))
9
> (with-inference (eats ?x ?y)
    (format t "~A eats ~A.~%"
            ?x
            (if (gensym? ?y) 'everything ?y)))
HUBERT eats EVERYTHING.
MONSTER eats BAD-CHILDREN.
WARHOL eats CANDY.
@
```

Finally, if we want to specify that facts of a certain form will be true for *any* arguments, we make the body a conjunction with no arguments. The expression `(and)` will always behave as a true fact. In the macro `<-` (Figure 24.4), the body defaults to `(and)`, so for such rules we can simply omit the body:

```
> (<- (identical ?x ?x))
10
> (with-inference (identical a ?x)
    (print ?x))
A
@
```

For readers with some knowledge of Prolog, Figure 24.6 shows the translation from Prolog syntax into that of our program. The traditional first Prolog program is `append`, which would be written as at the end of Figure 24.6. In an instance of appending, two shorter lists are joined together to form a single larger one. Any two of these lists define what the third should be. The Lisp function `append` takes the two shorter lists as arguments and returns the longer one. Prolog `append` is more general; the two rules in Figure 24.6 define a program which, given any two of the lists involved, can find the third.

Our syntax differs from traditional Prolog syntax as follows:

1. Variables are represented by symbols beginning with question marks instead of capital letters. Common Lisp is not case-sensitive by default, so it would be more trouble than it's worth to use capitals.
2. `[ ]` becomes `nil`.
3. Expressions of the form `[`*x* `|` *y*`]` become `(`*x* `.` *y*`)`.
4. Expressions of the form `[`*x*`,` *y*`, ...]` become `(`*x* *y* `...)`.
5. Predicates are moved inside parentheses, and no commas separate arguments: *pred*`(`*x*`,` *y*`, ...)` becomes `(`*pred x y* `...)`.

Thus the Prolog definition of `append`:

```
append([ ], Xs, Xs).
append([X | Xs], Ys, [X | Zs]) <- append(Xs, Ys, Zs).
```

becomes:

```
(<- (append nil ?xs ?xs))
(<- (append (?x . ?xs) ?ys (?x . ?zs))
    (append ?xs ?ys ?zs))
```

Figure 24.6: Prolog syntax equivalence.

```
> (with-inference (append ?x (c d) (a b c d))
    (format t "Left: ~A~%" ?x))
Left: (A B)
@
> (with-inference (append (a b) ?x (a b c d))
    (format t "Right: ~A~%" ?x))
Right: (C D)
@
> (with-inference (append (a b) (c d) ?x)
    (format t "Whole: ~A~%" ?x))
Whole: (A B C D)
@
```

Not only that, but given only the last list, it can find all the possibilities for the first two:

```
> (with-inference (append ?x ?y (a b c))
    (format t "Left: ~A  Right: ~A~%" ?x ?y))
Left: NIL  Right: (A B C)
Left: (A)  Right: (B C)
Left: (A B)  Right: (C)
Left: (A B C)  Right: NIL
@
```

The case of `append` points to a great difference between Prolog and other languages. A collection of Prolog rules does not have to yield a specific value. It can instead yield *constraints*, which, when combined with constraints generated by other parts of the program, yield a specific value. For example, if we define `member` thus:

```
(<- (member ?x (?x . ?rest)))
(<- (member ?x (_ . ?rest)) (member ?x ?rest))
```

then we can use it to test for list membership, as we would use the Lisp function `member`:

```
> (with-inference (member a (a b)) (print t))
T
@
```

but we can also use it to establish a constraint of membership, which, combined with other constraints, yields a specific list. If we also have a predicate `cara`

```
(<- (cara (a _)))
```

which is true of any two-element list whose car is `a`, then between that and `member` we have enough constraint for Prolog to construct a definite answer:

```
> (with-inference (and (cara ?lst) (member b ?lst))
    (print ?lst))
(A B)
@
```

This is a rather trivial example, but bigger programs can be constructed on the same principle. Whenever we want to program by combining partial solutions, Prolog may be useful. Indeed, a surprising variety of problems can be expressed in such terms: Figure 24.14, for example, shows a sorting algorithm expressed as a collection of constraints on the solution.

24.4 The Need for Nondeterminism

Chapter 22 explained the relation between deterministic and nondeterministic search. A deterministic search program could take a query and generate all the solutions which satisfied it. A nondeterministic search program will use *choose* to generate solutions one at a time, and if more are needed, will call *fail* to restart the search.

When we have rules which all yield finite sets of bindings, and we want all of them at once, there is no reason to prefer nondeterministic search. The difference between the two strategies becomes apparent when we have queries which would generate an infinite number of bindings, of which we want a finite subset. For example, the rules

```
(<- (all-elements ?x nil))
(<- (all-elements ?x (?x . ?rest))
    (all-elements ?x ?rest))
```

imply all the facts of the form `(all-elements` *x y*`)`, where every member of *y* is equal to *x*. Without backtracking we could handle queries like:

```
(all-elements a  (a a a))
(all-elements a  (a a b))
(all-elements ?x (a a a))
```

However, the query `(all-elements a ?x)` is satisfied for an infinite number of possible `?x`: `nil`, `(a)`, `(a a)`, and so on. If we try to generate answers for this query by iteration, the iteration will never terminate. Even if we only wanted one of the answers, we would never get a result from an implementation which had to generate all the bindings for the query before it could begin to iterate through the Lisp expressions following it.

This is why `with-inference` interleaves the generation of bindings with the evaluation of its body. Where queries could lead to an infinite number of answers, the only successful approach will be to generate answers one at a time, and return to pick up new ones by restarting the suspended search. Because it uses *choose* and *fail*, our program can handle this case:

```
> (block nil
    (with-inference (all-elements a ?x)
      (if (= (length ?x) 3)
          (return ?x)
          (princ ?x))))
NIL(A)(A A)
(A A A)
```

Like any other Prolog implementation, ours simulates nondeterminism by doing depth-first search with backtracking. In theory, “logic programs” run under true nondeterminism. In fact, Prolog implementations always use depth-first search. Far from being inconvenienced by this choice, typical Prolog programs depend on it. In a truly nondeterministic world, the query

```
(and (all-elements a ?x) (length ?x 3))
```

has an answer, but it takes you arbitrarily long to find out what it is.

Not only does Prolog use the depth-first implementation of nondeterminism, it uses a version equivalent to that defined on page 293. As explained there, this implementation is not always guaranteed to terminate. So Prolog programmers must take deliberate steps to avoid loops in the search space. For example, if we had defined `member` in the reverse order

```
(<- (member ?x (_ . ?rest)) (member ?x ?rest))
(<- (member ?x (?x . ?rest)))
```

then logically it would have the same meaning, but as a Prolog program it would have a different effect. The original definition of `member` would yield an infinite stream of answers in response to the query `(member 'a ?x)`, but the reversed definition will yield an infinite recursion, and no answers.

24.5 New Implementation

In this section we will see another instance of a familiar pattern. In Section 18.4, we found after writing the initial version that `if-match` could be made much faster. By taking advantage of information known at compile-time, we were able to write a new version which did less work at runtime. We saw the same phenomenon on a larger scale in Chapter 19. Our query interpreter was replaced by an equivalent but faster version. The same thing is about to happen to our Prolog interpreter.

Figures 24.7, 24.8, and 24.10 define Prolog in a different way. The macro `with-inference` used to be just the interface to a Prolog interpreter. Now it is most of the program. The new program has the same general shape as the old one, but of the functions defined in Figure 24.8, only `prove` is called at runtime. The others are called by `with-inference` in order to generate its expansion.

Figure 24.7 shows the new definition of `with-inference`. As in `if-match` or `with-answer`, pattern variables are initially bound to gensyms to indicate that they haven’t yet been assigned real values by matching. Thus the function `varsym?`, which `match` and `fullbind` use to detect variables, has to be changed to look for gensyms.

```
(defmacro with-inference (query &rest body)
  (let ((vars (vars-in query #'simple?)) (gb (gensym)))
    `(with-gensyms ,vars
       (setq *paths* nil)
       (=bind (,gb) ,(gen-query (rep_ query))
         (let ,(mapcar #'(lambda (v)
                           `(,v (fullbind ,v ,gb)))
                       vars)
           ,@body)
         (fail)))))

(defun varsym? (x)
  (and (symbolp x) (not (symbol-package x))))
```

Figure 24.7: New toplevel macro.

To generate the code to establish bindings for the query, `with-inference` calls `gen-query` (Figure 24.8). The first thing `gen-query` does is look to see whether its first argument is a complex query beginning with an operator like `and` or `or`. This process continues recursively until it reaches simple queries, which are expanded into calls to `prove`. In the original implementation, such logical structure was analyzed at runtime. A complex expression occurring in the body of a rule had to be analyzed anew each time the rule was used. This is wasteful because the logical structure of rules and queries is known beforehand. The new implementation decomposes complex expressions at compile-time.

As in the previous implementation, a `with-inference` expression expands into code which iterates through the Lisp code following the query with the pattern variables bound to successive values established by the rules. The expansion of `with-inference` concludes with a `fail`, which will restart any saved states.

The remaining functions in Figure 24.8 generate expansions for complex queries—queries joined together by operators like `and`, `or`, and `not`. If we have a query like

```
(and (big ?x) (red ?x))
```

then we want the Lisp code to be evaluated only with those `?x` for which both conjuncts can be proved. So to generate the expansion of an `and`, we nest the expansion of the second conjunct within that of the first. When `(big ?x)` succeeds we try `(red ?x)`, and if that succeeds, we evaluate the Lisp expressions. So the whole expression expands as in Figure 24.9.

```
(defun gen-query (expr &optional binds)
  (case (car expr)
    (and (gen-and (cdr expr) binds))
    (or  (gen-or  (cdr expr) binds))
    (not (gen-not (cadr expr) binds))
    (t   `(prove (list ',(car expr)
                       ,@(mapcar #'form (cdr expr)))
                 ,binds))))

(defun gen-and (clauses binds)
  (if (null clauses)
      `(=values ,binds)
      (let ((gb (gensym)))
        `(=bind (,gb) ,(gen-query (car clauses) binds)
           ,(gen-and (cdr clauses) gb)))))

(defun gen-or (clauses binds)
  `(choose
     ,@(mapcar #'(lambda (c) (gen-query c binds))
               clauses)))

(defun gen-not (expr binds)
  (let ((gpaths (gensym)))
    `(let ((,gpaths *paths*))
       (setq *paths* nil)
       (choose (=bind (b) ,(gen-query expr binds)
                 (setq *paths* ,gpaths)
                 (fail))
               (progn
                 (setq *paths* ,gpaths)
                 (=values ,binds))))))

(=defun prove (query binds)
   (choose-bind r *rules* (=funcall r query binds)))

(defun form (pat)
  (if (simple? pat)
      pat
      `(cons ,(form (car pat)) ,(form (cdr pat)))))
```

Figure 24.8: Compilation of queries.

```
(with-inference (and (big ?x) (red ?x))
  (print ?x))
```

expands into:

```
(with-gensyms (?x)
  (setq *paths* nil)
  (=bind (#:g1) (=bind (#:g2) (prove (list 'big ?x) nil)
                  (=bind (#:g3) (prove (list 'red ?x) #:g2)
                    (=values #:g3)))
     (let ((?x (fullbind ?x #:g1)))
       (print ?x))
     (fail)))
```

Figure 24.9: Expansion of a conjunction.

An `and` means nesting; an `or` means a *choose*. Given a query like

```
(or (big ?x) (red ?x))
```

we want the Lisp expressions to be evaluated for values of `?x` established by either subquery. The function `gen-or` expands into a `choose` over the `gen-query` of each of the arguments. As for `not`, `gen-not` is almost identical to `prove-not` (Figure 24.3).

Figure 24.10 shows the code for defining rules. Rules are translated directly into Lisp code generated by `rule-fn`. Since `<-` now expands rules into Lisp code, compiling a file full of rule definitions will cause rules to be compiled functions.

When a rule-function is sent a pattern, it tries to match it with the head of the rule it represents. If the match succeeds, the rule-function will then try to establish bindings for the body. This task is essentially the same as that done by `with-inference`, and in fact `rule-fn` ends by calling `gen-query`. The rule-function eventually returns the bindings established for the variables occurring in the head of the rule.

24.6 Adding Prolog Features

The code already presented can run most "pure" Prolog programs. The final step is to add extras like cuts, arithmetic, and I/O.

Putting a *cut* in a Prolog rule causes the search tree to be pruned. Ordinarily, when our program encounters a `fail`, it backtracks to the last choice point. The

```
(defvar *rules* nil)

(defmacro <- (con &rest ant)
  (let ((ant (if (= (length ant) 1)
                 (car ant)
                 `(and ,@ant))))
    `(length (conc1f *rules*
                     ,(rule-fn (rep_ ant) (rep_ con))))))

(defun rule-fn (ant con)
  (with-gensyms (val win fact binds)
    `(=lambda (,fact ,binds)
       (with-gensyms ,(vars-in (list ant con) #'simple?)
         (multiple-value-bind
             (,val ,win)
             (match ,fact
                    (list ',(car con)
                          ,@(mapcar #'form (cdr con)))
                    ,binds)
           (if ,win
               ,(gen-query ant val)
               (fail)))))))
```

Figure 24.10: Code for defining rules.

implementation of *choose* in Section 22.4 stores choice points in the global variable `*paths*`. Calling `fail` restarts the search at the most recent choice point, which is the car of `*paths*`. Cuts introduce a new complication. When the program encounters a `cut`, it will throw away some of the most recent choice points stored on `*paths*`—specifically, all those stored since the last call to `prove`.

The effect is to make rules mutually exclusive. We can use cuts to get the effect of a case statement in Prolog programs. For example, if we define `minimum` this way:

```
(<- (minimum ?x ?y ?x) (lisp (<= ?x ?y)))
(<- (minimum ?x ?y ?y) (lisp (> ?x ?y)))
```

it will work correctly, but inefficiently. Given the query

```
(minimum 1 2 ?x)
```

Prolog will immediately establish that `?x` = 1 from the first rule. A human would stop here, but the program will waste time looking for more answers from the second rule, because it has been given no indication that the two rules are mutually exclusive. On the average, this version of `minimum` will do 50% more work than it needs to. We can fix the problem by adding a cut after the first test:

```
(<- (minimum ?x ?y ?x) (lisp (<= ?x ?y)) (cut))
(<- (minimum ?x ?y ?y))
```

Now when Prolog has finished with the first rule, it will fail all the way out of the query instead of moving on to the next rule.

It is trivially easy to modify our program to handle cuts. On each call to `prove`, the current state of `*paths*` is passed as a parameter. If the program encounters a cut, it just sets `*paths*` back to the old value passed in the parameter. Figures 24.11 and 24.12 show the code which has to be modified to handle cuts. (Changed lines are marked with semicolons. Not all the changes are due to cuts.)

Cuts which merely make a program more efficient are called *green cuts.* The cut in `minimum` was a green cut. Cuts which make a program behave differently are called *red cuts.* For example, if we define the predicate `artist` as follows:

```
(<- (artist ?x) (sculptor ?x) (cut))
(<- (artist ?x) (painter ?x))
```

the result is that, if there are any sculptors, then the query can end there. If there are no sculptors then painters get to be considered as artists:

```
> (progn (<- (painter 'klee))
         (<- (painter 'soutine)))
4
> (with-inference (artist ?x)
    (print ?x))
KLEE
SOUTINE
@
```

But if there are sculptors, the cut stops inference with the first rule:

```
> (<- (sculptor 'hepworth))
5
> (with-inference (artist ?x)
    (print ?x))
HEPWORTH
@
```

```
(defun rule-fn (ant con)
  (with-gensyms (val win fact binds paths)                        ;
    `(=lambda (,fact ,binds ,paths)                               ;
       (with-gensyms ,(vars-in (list ant con) #'simple?)
         (multiple-value-bind
             (,val ,win)
             (match ,fact
                    (list ',(car con)
                          ,@(mapcar #'form (cdr con)))
                    ,binds)
           (if ,win
               ,(gen-query ant val paths)                         ;
               (fail)))))))

(defmacro with-inference (query &rest body)
  (let ((vars (vars-in query #'simple?)) (gb (gensym)))
    `(with-gensyms ,vars
       (setq *paths* nil)
       (=bind (,gb) ,(gen-query (rep_ query) nil '*paths*) ;
         (let ,(mapcar #'(lambda (v)
                           `(,v (fullbind ,v ,gb)))
                       vars)
           ,@body)
         (fail)))))

(defun gen-query (expr binds paths)                               ;
  (case (car expr)
    (and  (gen-and (cdr expr) binds paths))                       ;
    (or   (gen-or  (cdr expr) binds paths))                       ;
    (not  (gen-not (cadr expr) binds paths))                      ;
    (lisp (gen-lisp (cadr expr) binds))                           ;
    (is   (gen-is (cadr expr) (third expr) binds))                ;
    (cut  `(progn (setq *paths* ,paths)                           ;
                  (=values ,binds)))                              ;
    (t    `(prove (list ',(car expr)
                        ,@(mapcar #'form (cdr expr)))
                  ,binds *paths*))))                              ;

(=defun prove (query binds paths)                                 ;
   (choose-bind r *rules*
     (=funcall r query binds paths)))                             ;
```

Figure 24.11: Adding support for new operators.

```
(defun gen-and (clauses binds paths)                                  ;
  (if (null clauses)
      `(=values ,binds)
      (let ((gb (gensym)))
       `(=bind (,gb) ,(gen-query (car clauses) binds paths);
          ,(gen-and (cdr clauses) gb paths)))))                       ;

(defun gen-or (clauses binds paths)                                   ;
  `(choose
     ,@(mapcar #'(lambda (c) (gen-query c binds paths))           ;
               clauses)))

(defun gen-not (expr binds paths)                                     ;
  (let ((gpaths (gensym)))
    `(let ((,gpaths *paths*))
       (setq *paths* nil)
       (choose (=bind (b) ,(gen-query expr binds paths)           ;
                 (setq *paths* ,gpaths)
                 (fail))
               (progn
                 (setq *paths* ,gpaths)
                 (=values ,binds))))))

(defmacro with-binds (binds expr)
  `(let ,(mapcar #'(lambda (v) `(,v (fullbind ,v ,binds)))
                 (vars-in expr))
     ,expr))

(defun gen-lisp (expr binds)
  `(if (with-binds ,binds ,expr)
       (=values ,binds)
       (fail)))

(defun gen-is (expr1 expr2 binds)
  `(aif2 (match ,expr1 (with-binds ,binds ,expr2) ,binds)
         (=values it)
         (fail)))
```

Figure 24.12: Adding support for new operators.

```
⟨rule⟩      : (<- ⟨sentence⟩ ⟨query⟩)
⟨query⟩     : (not ⟨query⟩)
            : (and ⟨query⟩*)
            : (lisp ⟨lisp expression⟩)
            : (is ⟨variable⟩ ⟨lisp expression⟩)
            : (cut)
            : (fail)
            : ⟨sentence⟩
⟨sentence⟩  : (⟨symbol⟩ ⟨argument⟩*)
⟨argument⟩  : ⟨variable⟩
            : ⟨lisp expression⟩
⟨variable⟩  : ?⟨symbol⟩
```

Figure 24.13: New syntax of rules.

The cut is sometimes used in conjunction with the Prolog fail operator. Our function `fail` does exactly the same thing. Putting a cut in a rule makes it like a one-way street: once you enter, you're committed to using only that rule. Putting a cut-fail combination in a rule makes it like a one-way street in a dangerous neighborhood: once you enter, you're committed to leaving with nothing. A typical example is in the implementation of not-equal:

```
(<- (not-equal ?x ?x) (cut) (fail))
(<- (not-equal ?x ?y))
```

The first rule here is a trap for impostors. If we're trying to prove a fact of the form `(not-equal 1 1)`, it will match with the head of the first rule and thus be doomed. The query `(not-equal 1 2)`, on the other hand, will not match the head of the first rule, and will go on to the second, where it succeeds:

```
> (with-inference (not-equal 'a 'a)
    (print t))
@
> (with-inference (not-equal '(a a) '(a b))
    (print t))
T
@
```

The code shown in Figures 24.11 and 24.12 also gives our program arithmetic, I/O, and the Prolog `is` operator. Figure 24.13 shows the complete syntax of rules and queries.

We add arithmetic (and more) by including a trapdoor to Lisp. Now in addition to operators like `and` and `or`, we have the `lisp` operator. This may be followed by any Lisp expression, which will be evaluated with the variables within it bound to the bindings established for them by the query. If the expression evaluates to `nil`, then the `lisp` expression as a whole is equivalent to a `(fail)`; otherwise it is equivalent to `(and)`.

As an example of the use of the `lisp` operator, consider the Prolog definition of `ordered`, which is true of lists whose elements are arranged in ascending order:

```
(<- (ordered (?x)))
(<- (ordered (?x ?y . ?ys))
    (lisp (<= ?x ?y))
    (ordered (?y . ?ys)))
```

In English, a list of one element is ordered, and a list of two or more elements is ordered if the first element of the list is less than or equal to the second, and the list from the second element on is ordered.

```
> (with-inference (ordered '(1 2 3))
    (print t))
T
@
> (with-inference (ordered '(1 3 2))
    (print t))
@
```

By means of the `lisp` operator we can provide other features offered by typical Prolog implementations. Prolog I/O predicates can be duplicated by putting Lisp I/O calls within `lisp` expressions. The Prolog `assert`, which as a side-effect defines new rules, can be duplicated by calling the `<-` macro within `lisp` expressions.

The `is` operator offers a form of assignment. It takes two arguments, a pattern and a Lisp expression, and tries to match the pattern with the result returned by the expression. If the match fails, then the program calls `fail`; otherwise it proceeds with the new bindings. Thus, the expression `(is ?x 1)` has the effect of setting `?x` to `1`, or more precisely, insisting that `?x` be `1`. We need `is` to *calculate*—for example, to calculate factorials:

```
(<- (factorial 0 1))
(<- (factorial ?n ?f)
    (lisp (> ?n 0))
    (is ?n1 (- ?n 1))
    (factorial ?n1 ?f1)
    (is ?f (* ?n ?f1)))
```

We use this definition by making a query with a number *n* as the first argument and a variable as the second:

```
> (with-inference (factorial 8 ?x)
    (print ?x))
40320
@
```

Note that the variables used in a `lisp` expression, or in the second argument to `is`, must have established bindings for the expression to return a value. This restriction holds in any Prolog. For example, the query:

```
(with-inference (factorial ?x 120)                     ; wrong
  (print ?x))
```

won't work with this definition of `factorial`, because `?n` will be unknown when the `lisp` expression is evaluated. So not all Prolog programs are like `append`: many insist, like `factorial`, that certain of their arguments be real values.

24.7 Examples

○ This final section shows how to write some example Prolog programs in our implementation. The rules in Figure 24.14 define quicksort. These rules imply facts of the form `(quicksort` *x y*`)`, where *x* is a list and *y* is a list of the same elements sorted in ascending order. Variables may appear in the second argument position:

```
> (with-inference (quicksort '(3 2 1) ?x)
    (print ?x))
(1 2 3)
@
```

An I/O loop is a test for our Prolog, because it makes use of the `cut`, `lisp`, and `is` operators. The code is shown in Figure 24.15. These rules should be invoked by trying to prove `(echo)`, with no arguments. That query will match the first rule, which will bind `?x` to the result returned by `read`, and then try to establish `(echo ?x)`. The new query can match either of the second two rules. If `?x` = `done`, then the query will terminate in the second rule. Otherwise the query will only match the third rule, which prints the value read, and starts the process over again.

```
(setq *rules* nil)

(<- (append nil ?ys ?ys))
(<- (append (?x . ?xs) ?ys (?x . ?zs))
    (append ?xs ?ys ?zs))

(<- (quicksort (?x . ?xs) ?ys)
    (partition ?xs ?x ?littles ?bigs)
    (quicksort ?littles ?ls)
    (quicksort ?bigs ?bs)
    (append ?ls (?x . ?bs) ?ys))
(<- (quicksort nil nil))

(<- (partition (?x . ?xs) ?y (?x . ?ls) ?bs)
    (lisp (<= ?x ?y))
    (partition ?xs ?y ?ls ?bs))
(<- (partition (?x . ?xs) ?y ?ls (?x . ?bs))
    (lisp (> ?x ?y))
    (partition ?xs ?y ?ls ?bs))
(<- (partition nil ?y nil nil))
```

Figure 24.14: Quicksort.

```
(<- (echo)
    (is ?x (read))
    (echo ?x))
(<- (echo 'done)
    (cut))
(<- (echo ?x)
    (lisp (prog1 t (format t "~A~%" ?x)))
    (is ?y (read))
    (cut)
    (echo ?y))
```

Figure 24.15: An I/O loop in Prolog.

Collectively, the rules define a program that will continue to echo what you type, until you type `done`:

```
> (with-inference (echo))
hi
HI
ho
HO
done
@
```

Programs like this are difficult to read because they subvert the abstract model of Prolog. It might be easier to understand `echo` if we look at a literal Lisp translation

```
(defun echo (&rest args)
  (cond ((null args) (echo (read)))
        ((eq (car args) 'done) nil)
        (t (format t "~A~%" (car args))
           (echo (read)))))
```

which in idiomatic Common Lisp would be:

```
(defun echo (&optional (arg (read)))
  (unless (eq arg 'done)
    (format t "~A~%" arg)
    (echo)))
```

24.8 The Senses of Compile

The word "compile" has several senses. In the most general sense, to compile is to transform some abstract description of a program into lower-level code. The program described in this chapter is certainly a compiler in this sense, because it translates rules into Lisp functions.

In a more specific sense, to compile is to transform a program into machine language. Good Common Lisps compile functions into native machine code. As mentioned on page 25, if code which generates closures is compiled, it will yield compiled closures. Thus the program described here is a compiler in the stricter sense as well. In a good Lisp, our Prolog programs will get translated into machine language.

However, the program described here *is still not a Prolog compiler.* For programming languages there is a still more specific sense of "compile," and merely generating machine code is not enough to satisfy this definition. A compiler for a programming language must optimize as well as translate. For example, if a Lisp compiler is asked to compile an expression like

```
(+ x (+ 2 5))
```

it should be smart enough to realize that there is no reason to wait until runtime to evaluate `(+ 2 5)`. The program can be optimized by replacing it with `7`, and instead compiling

```
(+ x 7)
```

In our program, all the compiling is done by the Lisp compiler, and it is looking for Lisp optimizations, not Prolog optimizations. Its optimizations will be valid ones, but too low-level. The Lisp compiler doesn't know that the code it's compiling is meant to represent rules. While a real Prolog compiler would be looking for rules that could be transformed into loops, our program is looking for expressions that yield constants, or closures that could be allocated on the stack.

Embedded languages allow you to make the most of available abstractions, but they are not magic. If you want to travel all the way from a very abstract representation to fast machine code, someone still has to tell the computer how to do it. In this chapter we travelled a good part of that distance with surprisingly little code, but that is not the same as writing a true Prolog compiler.