15

Example: Inference

The next three chapters offer examples of substantial Lisp programs. These examples were chosen to illustrate the form that longer programs take, and also the kinds of problems for which Lisp is especially well-suited.

In this chapter we will write a program that makes inferences based on a collection of if-then rules. This is a classic example—not only in the sense that it often appears in textbooks, but also because it reflects the original idea of Lisp as a language for "symbolic computation." A lot of the earliest Lisp programs had the flavor of the example in this chapter.

15.1 The Aim

In this program, we're going to represent information in a familiar form: a list consisting of a predicate followed by zero or more arguments. To represent the fact that Donald is the parent of Nancy, we might say:

```
(parent donald nancy)
```

As well as facts, our program is going to represent rules that tell what can be inferred from the facts we already have. We will represent such rules as

(<- *head body*)

where *head* is the then-part and *body* is the if-part. Within the *head* and *body* we will represent variables as symbols beginning with question marks. So this rule

```
(<- (child ?x ?y) (parent ?y ?x))
```

says that if *y* is the parent of *x*, then *x* is the child of *y*; or more precisely, that we can prove any fact of the form `(child` *x y*`)` by proving `(parent` *y x*`)`.

It will be possible for the body (if-part) of a rule to be a complex expression, containing the logical operators `and`, `or`, and `not`. So if we want to represent the rule that if *x* is the parent of *y*, and *x* is male, then *x* is the father of *y*, we would write:

```
(<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
```

Rules may depend on facts implied by other rules. For example, the first rule we wrote was for proving facts of the form `(child` *x y*`)`. If we defined a rule

```
(<- (daughter ?x ?y) (and (child ?x ?y) (female ?x)))
```

then using it to prove `(daughter` *x y*`)` might cause the program to use the first rule to prove `(child` *x y*`)`.

The proof of an expression can continue back through any number of rules, so long as it eventually ends up on the solid ground of known facts. This process is sometimes called *backward chaining*. The *backward* comes from the fact that this kind of inference first considers the then-part, to see if the rule will be useful, before going on to prove the if-part. The *chaining* comes from the way that rules can depend on other rules, forming a chain (though in fact it's more like a tree) that leads from what we want to prove back to what we already know.°

15.2 Matching

In order to write our backward-chaining program, we are going to need a function to do pattern-matching: a function that can compare two lists, possibly containing variables, to see if there is some way of assigning values to the variables which makes the two equal. For example, if `?x` and `?y` are variables, then the two lists

```
(p ?x ?y c ?x)
(p  a  b c  a)
```

match when `?x` = `a` and `?y` = `b`, and the lists

```
(p ?x b ?y a)
(p ?y b  c a)
```

match when `?x` = `?y` = `c`.

Figure 15.1 contains a function called `match`. It takes two trees, and if they can be made to match, it returns an assoc-list showing how:

```
(defun match (x y &optional binds)
  (cond
   ((eql x y) (values binds t))
   ((assoc x binds) (match (binding x binds) y binds))
   ((assoc y binds) (match x (binding y binds) binds))
   ((var? x) (values (cons (cons x y) binds) t))
   ((var? y) (values (cons (cons y x) binds) t))
   (t
    (when (and (consp x) (consp y))
      (multiple-value-bind (b2 yes)
                           (match (car x) (car y) binds)
        (and yes (match (cdr x) (cdr y) b2)))))))

(defun var? (x)
  (and (symbolp x)
       (eql (char (symbol-name x) 0) #\?)))

(defun binding (x binds)
  (let ((b (assoc x binds)))
    (if b
        (or (binding (cdr b) binds)
            (cdr b)))))
```

Figure 15.1: Matching function.

```
> (match '(p a b c a) '(p ?x ?y c ?x))
((?Y . B) (?X . A))
T
> (match '(p ?x b ?y a) '(p ?y b c a))
((?Y . C) (?X . ?Y))
T
> (match '(a b c) '(a a a))
NIL
```

As `match` compares its arguments element by element, it builds up assignments of values to variables, called *bindings*, in the parameter `binds`. If the match is successful, `match` returns the bindings generated; otherwise, it returns `nil`. Since not all successful matches generate any bindings, `match`, like `gethash`, returns a second value to show that the match succeeded:

```
> (match '(p ?x) '(p ?x))
NIL
T
```

When `match` returns `nil` and `t` as above, it indicates a successful match that yielded no bindings. In English, the match algorithm works as follows:

1. If x and y are `eql` they match; otherwise,
2. If x is a variable that has a binding, they match if it matches y; otherwise,
3. If y is a variable that has a binding, they match if it matches x; otherwise,
4. If x is a variable (without a binding), they match and thereby establish a binding for it; otherwise,
5. If y is a variable (without a binding), they match and thereby establish a binding for it; otherwise,
6. They match if they are both conses, and the cars match, and the cdrs match with the bindings generated thereby.

Here is an example illustrating, in order, each of the six cases:

```
> (match '(p ?v  b ?x  d (?z ?z))
         '(p  a ?w  c ?y ( e  e))
         '((?v . a) (?w . b)))
((?Z . E) (?Y . D) (?X . C) (?V . A) (?W . B))
T
```

To find the value (if there is one) associated with a variable in a list of bindings, `match` calls `binding`. This function has to be recursive, because matching can build up binding lists in which a variable is only indirectly associated with its value: `?x` might be bound to `a` in virtue of the list containing both `(?x . ?y)` and `(?y . a)`.

```
> (match '(?x a) '(?y ?y))
((?Y . A) (?X . ?Y))
T
```

By matching `?x` with `?y` and then `?y` with `a`, we establish indirectly that `?x` must be `a`.

```
(defvar *rules* (make-hash-table))

(defmacro <- (con &optional ant)
  `(length (push (cons (cdr ',con) ',ant)
                 (gethash (car ',con) *rules*))))
```

Figure 15.2: Defining rules.

15.3 Answering Queries

Now that the concept of bindings has been introduced, we can say more precisely what our program will do: it will take an expression, possibly containing variables, and return all the bindings that make it true given the facts and rules that we have. For example, if we have just the fact

```
(parent donald nancy)
```

and we ask the program to prove

```
(parent ?x ?y)
```

it should return something like

```
(((?x . donald) (?y . nancy)))
```

which says that there is exactly one way for the expression to be true: if `?x` is `donald` and `?y` is `nancy`.

Now that we have a matching function we are already a good part of the way to our destination. Figure 15.2 contains the code for defining rules. The rules are going to be contained in a hash table called `*rules*`, hashed according to the predicate in the head. This imposes the restriction that we can't use variables in the predicate position. We could eliminate this restriction by keeping all such rules in a separate list, but then to prove something we would have to match it against every one.

We will use the same macro, `<-`, to define both facts and rules. A fact will be represented as a rule with a head but no body. This is consistent with our definition of rules. A rule says that you can prove the head by proving the body, so a rule with no body means that you don't have to prove anything to prove the head. Here are two familiar examples:

```
> (<- (parent donald nancy))
1
> (<- (child ?x ?y) (parent ?y ?x))
1
```

```
(defun prove (expr &optional binds)
  (case (car expr)
    (and (prove-and (reverse (cdr expr)) binds))
    (or  (prove-or (cdr expr) binds))
    (not (prove-not (cadr expr) binds))
    (t   (prove-simple (car expr) (cdr expr) binds))))

(defun prove-simple (pred args binds)
  (mapcan #'(lambda (r)
              (multiple-value-bind (b2 yes)
                                   (match args (car r)
                                          binds)
                (when yes
                  (if (cdr r)
                      (prove (cdr r) b2)
                      (list b2)))))
          (mapcar #'change-vars
                  (gethash pred *rules*))))

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

(defun vars-in (expr)
  (if (atom expr)
      (if (var? expr) (list expr))
      (union (vars-in (car expr))
             (vars-in (cdr expr)))))
```

Figure 15.3: Inference.

Calls to `<-` return the number of rules now stored under a given predicate; wrapping the `push` in a call to `length` saves us from seeing a big return value at the toplevel.

Figure 15.3 contains most of the code we need for inference. The function `prove` is the pivot on which inference turns. It takes an expression and an optional list of bindings. If the expression doesn't contain logical operators, it calls `prove-simple`, and it is here that chaining takes place. This function works by looking at all the rules with the right predicate, and trying to match the head of each with the fact it is trying to prove. For each head that matches, it calls prove on the body, with the new bindings generated by the match. The lists of bindings returned by each call to prove are then collected by mapcan and returned:

```
> (prove-simple 'parent '(donald nancy) nil)
(NIL)
> (prove-simple 'child '(?x ?y) nil)
(((#:?6 . NANCY) (#:?5 . DONALD) (?Y . #:?5) (?X . #:?6)))
```

Both of the return values above indicate that there is one way to prove what we asked about. (A failed proof would return nil.) The first example generated one empty set of bindings, and the second generated one set of bindings in which ?x and ?y were (indirectly) bound to nancy and donald.

Incidentally, we see here a good example of the point made on page 23. Because our program is written in a functional style, we can test each function interactively.

What about those gensyms in the second return value? If we are going to use rules containing variables, we need to avoid the possibility of two rules accidentally containing the same variable. If we define two rules as follows

```
(<- (child ?x ?y) (parent ?y ?x))

(<- (daughter ?y ?x) (and (child ?y ?x) (female ?y)))
```

then we mean that for *any* x and y, x is the child of y if y is the parent of x, and for *any* x and y, y is the daughter of x if y is the child of x and female. The relationship of the variables within each rule is significant, but the fact that the two rules happen to use the same variables is entirely coincidental.

If we used these rules as written, they would not work that way. If we tried to prove that a was b's daughter, matching against the head of the second rule would leave ?y bound to a and ?x to b. We could not then match the head of the first rule with these bindings:

```
> (match '(child ?y ?x)
         '(child ?x ?y)
         '((?y . a) (?x . b)))
NIL
```

To ensure that the variables in a rule imply only something about the relations of arguments within that rule, we replace all the variables in a rule with gensyms. This is the purpose of the function change-vars. A gensym could not possibly turn up as a variable in another rule. But because rules can be recursive, we also have to guard against the possibility of a rule clashing with itself, so change-vars has to be called not just when a rule is defined, but each time it is used.

```
(defun prove-and (clauses binds)
  (if (null clauses)
      (list binds)
      (mapcan #'(lambda (b)
                  (prove (car clauses) b))
              (prove-and (cdr clauses) binds))))

(defun prove-or (clauses binds)
  (mapcan #'(lambda (c) (prove c binds))
          clauses))

(defun prove-not (clause binds)
  (unless (prove clause binds)
    (list binds)))
```

Figure 15.4: Logical operators.

```
(defmacro with-answer (query &body body)
  (let ((binds (gensym)))
    `(dolist (,binds (prove ',query))
       (let ,(mapcar #'(lambda (v)
                         `(,v (binding ',v ,binds)))
                     (vars-in query))
         ,@body))))
```

Figure 15.5: Interface macro.

Now all that remains is to define the functions that prove complex expressions. These are shown in Figure 15.4. Handling an `or` or `not` expression is particularly simple. In the former case we collect all the bindings returned by each of the expressions within the `or`. In the latter case, we return the current bindings iff the expression within the `not` yields none.

The function `prove-and` is only a little more complicated. It works like a filter, proving the first expression for each set of bindings that can be established for the remaining expressions. This would cause the expressions within the `and` to be considered in reverse order, except that the call to `prove-and` within `prove` reverses them to compensate.

Now we have a working program, but it's not very user-friendly. It's a nuisance to have to decipher the lists of bindings returned by `prove`—and

```
(with-answer (p ?x ?y)
  (f ?x ?y))
```

is macroexpanded into:

```
(dolist (#:g1 (prove '(p ?x ?y)))
  (let ((?x (binding '?x #:g1))
        (?y (binding '?y #:g1)))
    (f ?x ?y)))
```

Figure 15.6: Expansion of a call to `with-answer`.

they only get longer as the rules get more complex. Figure 15.5 contains a macro that will make our program more pleasant to use: a `with-answer` expression will take a query (not evaluated) and a body of expressions, and will evaluate its body once for each set of bindings generated by the query, with each pattern variable bound to the value it has in the bindings.

```
> (with-answer (parent ?x ?y)
    (format t "~A is the parent of ~A.~%" ?x ?y))
DONALD is the parent of NANCY.
NIL
```

This macro does the work of deciphering the bindings for us, and gives us a convenient way of using `prove` in programs. Figure 15.6 shows what an expansion looks like, and Figure 15.7 shows some examples of it in use.

15.4 Analysis

It may seem as if the code we've written in this chapter is simply the natural way to implement such a program. In fact it is grossly inefficient. What we've done here, essentially, is to write an interpreter. We could implement the same program as a compiler.

Here is a sketch of how it would be done. The basic idea would be to pack the whole program into the macros `<-` and `with-answer`, and make them do at macro-expansion time most of the work the program now does at run-time. (The germ of this idea is visible in `avg`, on page 170.) Instead of representing rules as lists, we would represent them as functions, and instead of having functions like `prove` and `prove-and` to interpret expressions at run-time, we would have corresponding functions to transform expressions into code. The expressions are available at the time a rule is defined. Why wait until it

If we do a (clrhash *rules*) and then define the following rules and facts,

```
(<- (parent donald nancy))
(<- (parent donald debbie))
(<- (male donald))
(<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
(<- (= ?x ?x))
(<- (sibling ?x ?y) (and (parent ?z ?x)
                         (parent ?z ?y)
                         (not (= ?x ?y))))
```

we will be able to make inferences like the following:

```
> (with-answer (father ?x ?y)
    (format t "~A is the father of ~A.~%" ?x ?y))
DONALD is the father of DEBBIE.
DONALD is the father of NANCY.
NIL
> (with-answer (sibling ?x ?y)
    (format t "~A is the sibling of ~A.~%" ?x ?y))
DEBBIE is the sibling of NANCY.
NANCY is the sibling of DEBBIE.
NIL
```

Figure 15.7: The program in use.

is used in order to analyze them? The same goes for `with-answer`, which would call the same functions as `<-` to generate its expansion.

This sounds like it would be a lot more complicated than the program we wrote in this chapter, but in fact it would probably only be about two or three times as long. Readers who would like to learn about such techniques should see *On Lisp* or *Paradigms of Artificial Intelligence Programming*, which contain several examples of programs written in this style.