19

A Query Compiler

Some of the macros defined in the preceding chapter were large ones. To generate its expansion, `if-match` needed all the code in Figures 18.7 and 18.8, plus `destruc` from Figure 18.1. Macros of this size lead naturally to our last topic, embedded languages. If small macros are extensions to Lisp, large macros define sub-languages within it—possibly with their own syntax or control structure. We saw the beginning of this in `if-match`, which had its own distinct representation for variables.

A language implemented within Lisp is called an *embedded language.* Like "utility," the term is not a precisely defined one; `if-match` probably still counts as a utility, but it is getting close to the borderline.

An embedded language is not a like a language implemented by a traditional compiler or interpreter. It is implemented within some existing language, usually by transformation. There need be no barrier between the base language and the extension: it should be possible to intermingle the two freely. For the implementor, this can mean a huge saving of effort. You can embed just what you need, and for the rest, use the base language.

Transformation, in Lisp, suggests macros. To some extent, you could implement embedded languages with preprocessors. But preprocessors usually operate only on text, while macros take advantage of a unique property of Lisp: between the reader and the compiler, your Lisp program is represented as lists of Lisp objects. Transformations done at this stage can be much smarter.

The best-known example of an embedded language is CLOS, the Common Lisp Object System. If you wanted to make an object-oriented version of a conventional language, you would have to write a new compiler. Not so in Lisp. Tuning the

compiler will make CLOS run faster, but in principle the compiler doesn't have to be changed at all. The whole thing can be written in Lisp.

The remaining chapters give examples of embedded languages. This chapter describes how to embed in Lisp a program to answer queries on a database. (You will notice in this program a certain family resemblance to `if-match`.) The first sections describe how to write a system which interprets queries. This program is then reimplemented as a query compiler—in essence, as one big macro—making it both more efficient and better integrated with Lisp.

19.1 The Database

For our present purposes, the format of the database doesn't matter very much. Here, for the sake of convenience, we will store information in lists. For example, we will represent the fact that Joshua Reynolds was an English painter who lived from 1723 to 1792 by:

```
(painter reynolds joshua english)
(dates reynolds 1723 1792)
```

There is no canonical way of reducing information to lists. We could just as well have used one big list:

```
(painter reynolds joshua 1723 1792 english)
```

It is up to the user to decide how to organize database entries. The only restriction is that the entries (*facts*) will be indexed under their first element (the *predicate*). Within those bounds, any consistent form will do, although some forms might make for faster queries than others.

Any database system needs at least two operations: one for modifying the database, and one for examining it. The code shown in Figure 19.1 provides these operations in a basic form. A database is represented as a hash-table filled with lists of facts, hashed according to their predicate.

Although the database functions defined in Figure 19.1 support multiple databases, they all default to operations on `*default-db*`. As with packages in Common Lisp, programs which don't need multiple databases need not even mention them. In this chapter all the examples will just use the `*default-db*`.

We initialize the system by calling `clear-db`, which empties the current database. We can look up facts with a given predicate with `db-query`, and insert new facts into a database entry with `db-push`. As explained in Section 12.1, a macro which expands into an invertible reference will itself be invertible. Since `db-query` is defined this way, we can simply push new facts onto the `db-query` of their predicates. In Common Lisp, hash-table entries are initialized to `nil`

```
(defun make-db (&optional (size 100))
  (make-hash-table :size size))

(defvar *default-db* (make-db))

(defun clear-db (&optional (db *default-db*))
  (clrhash db))

(defmacro db-query (key &optional (db '*default-db*))
  `(gethash ,key ,db))

(defun db-push (key val &optional (db *default-db*))
  (push val (db-query key db)))

(defmacro fact (pred &rest args)
  `(progn (db-push ',pred ',args)
          ',args))
```

Figure 19.1: Basic database functions.

unless specified otherwise, so any key initially has an empty list associated with it. Finally, the macro `fact` adds a new fact to the database.

```
> (fact painter reynolds joshua english)
(REYNOLDS JOSHUA ENGLISH)
> (fact painter canale antonio venetian)
(CANALE ANTONIO VENETIAN)
> (db-query 'painter)
((CANALE ANTONIO VENETIAN)
 (REYNOLDS JOSHUA ENGLISH))
T
```

The `t` returned as the second value by `db-query` appears because `db-query` expands into a `gethash`, which returns as its second value a flag to distinguish between finding no entry and finding an entry whose value is `nil`.

19.2 Pattern-Matching Queries

Calling `db-query` is not a very flexible way of looking at the contents of the database. Usually the user wants to ask questions which depend on more than just the first element of a fact. A *query language* is a language for expressing

⟨query⟩ : (⟨symbol⟩ ⟨argument⟩*)
: (not ⟨query⟩)
: (and ⟨query⟩*)
: (or ⟨query⟩*)
⟨argument⟩ : ?⟨symbol⟩
: ⟨symbol⟩
: ⟨number⟩

Figure 19.2: Syntax of queries.

more complicated questions. In a typical query language, the user can ask for all the values which satisfy some combination of restrictions—for example, the last names of all the painters born in 1697.

Our program will provide a declarative query language. In a declarative query language, the user specifies the constraints which answers must satisfy, and leaves it to the system to figure out how to generate them. This way of expressing queries is close to the form people use in everyday conversation. With our program, we will be able to express the sample query by asking for all the *x* such that there is a fact of the form `(painter` *x* `...)`, and a fact of the form `(dates` *x* `1697 ...)`. We will be able to refer to all the painters born in 1697 by writing:

```
(and (painter ?x ?y ?z)
     (dates ?x 1697 ?w))
```

As well as accepting simple queries consisting of a predicate and some arguments, our program will be able to answer arbitrarily complex queries joined together by logical operators like `and` and `or`. The syntax of the query language is shown in Figure 19.2.

Since facts are indexed under their predicates, variables cannot appear in the predicate position. If you were willing to give up the benefits of indexing, you could get around this restriction by always using the same predicate, and making the first argument the *de facto* predicate.

Like many such systems, this program has a skeptic's notion of truth: some facts are known, and everything else is false. The `not` operator succeeds if the fact in question is not present in the database. To a degree, you could represent explicit falsity by the *Wayne's World* method:

```
(edible motor-oil not)
```

However, the `not` operator wouldn't treat these facts differently from any others.

In programming languages there is a fundamental distinction between interpreted and compiled programs. In this chapter we examine the same question with respect to queries. A query interpreter accepts a query and uses it to generate answers from the database. A query compiler accepts a query and generates a *program* which, when run, yields the same result. The following sections describe a query interpreter and then a query compiler.

19.3 A Query Interpreter

To implement a declarative query language we will use the pattern-matching utilities defined in Section 18.4. The functions shown in Figure 19.3 interpret queries of the form shown in Figure 19.2. The central function in this code is `interpret-query`, which recursively works through the structure of a complex query, generating bindings in the process. The evaluation of complex queries proceeds left-to-right, as in Common Lisp itself.

When the recursion gets down to patterns for facts, `interpret-query` calls `lookup`. This is where the pattern-matching occurs. The function `lookup` takes a pattern consisting of a predicate and a list of arguments, and returns a list of all the bindings which make the pattern match some fact in the database. It gets all the database entries for the predicate, and calls `match` (page 239) to compare each of them against the pattern. Each successful match returns a list of bindings, and `lookup` in turn returns a list of all these lists.

```
> (lookup 'painter '(?x ?y english))
(((?Y . JOSHUA) (?X . REYNOLDS)))
```

These results are then filtered or combined depending on the surrounding logical operators. The final result is returned as a list of sets of bindings. Given the assertions shown in Figure 19.4, here is the example from earlier in this chapter:

```
> (interpret-query '(and (painter ?x ?y ?z)
                         (dates ?x 1697 ?w)))
(((?W . 1768) (?Z . VENETIAN) (?Y . ANTONIO) (?X . CANALE))
 ((?W . 1772) (?Z . ENGLISH) (?Y . WILLIAM) (?X . HOGARTH)))
```

As a general rule, queries can be combined and nested without restriction. In a few cases there are subtle restrictions on the syntax of queries, but these are best dealt with after looking at some examples of how this code is used.

The macro `with-answer` provides a clean way of using the query interpreter within Lisp programs. It takes as its first argument any legal query; the rest of the arguments are treated as a body of code. A `with-answer` expands into

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

(defun interpret-query (expr &optional binds)
  (case (car expr)
    (and  (interpret-and (reverse (cdr expr)) binds))
    (or   (interpret-or (cdr expr) binds))
    (not  (interpret-not (cadr expr) binds))
    (t    (lookup (car expr) (cdr expr) binds))))

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

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

(defun interpret-not (clause binds)
  (if (interpret-query clause binds)
      nil
      (list binds)))

(defun lookup (pred args &optional binds)
  (mapcan #'(lambda (x)
              (aif2 (match x args binds) (list it)))
          (db-query pred)))
```

Figure 19.3: Query interpreter.

```
(clear-db)
(fact painter hogarth william english)
(fact painter canale antonio venetian)
(fact painter reynolds joshua english)
(fact dates hogarth 1697 1772)
(fact dates canale 1697 1768)
(fact dates reynolds 1723 1792)
```

Figure 19.4: Assertion of sample facts.

code which collects all the sets of bindings generated by the query, then iterates through the body with the variables in the query bound as specified by each set of bindings. Variables which appear in the query of a `with-answer` can (usually) be used within its body. When the query is successful but contains no variables, `with-answer` evaluates the body of code just once.

With the database as defined in Figure 19.4, Figure 19.5 shows some sample queries, accompanied by English translations. Because pattern-matching is done with `match`, it is possible to use the underscore as a wild-card in patterns.

To keep these examples short, the code within the bodies of the queries does nothing more than print some result. In general, the body of a `with-answer` can consist of any Lisp expressions.

19.4 Restrictions on Binding

There are some restrictions on which variables will be bound by a query. For example, why should the query

```
(not (painter ?x ?y ?z))
```

assign any bindings to `?x` and `?y` at all? There are an infinite number of combinations of `?x` and `?y` which are *not* the name of some painter. Thus we add the following restriction: the `not` operator will filter out bindings which are already generated, as in

```
(and (painter ?x ?y ?z) (not (dates ?x 1772 ?d)))
```

but you cannot expect it to generate bindings all by itself. We have to generate sets of bindings by looking for painters before we can screen out the ones not born in 1772. If we had put the clauses in the reverse order:

```
(and (not (dates ?x 1772 ?d)) (painter ?x ?y ?z))     ; wrong
```

The first name and nationality of every painter called Hogarth.

```
> (with-answer (painter hogarth ?x ?y)
    (princ (list ?x ?y)))
(WILLIAM ENGLISH)
NIL
```

The last name of every painter born in 1697. (Our original example.)

```
> (with-answer (and (painter ?x _ _)
                    (dates ?x 1697 _))
    (princ (list ?x)))
(CANALE)(HOGARTH)
NIL
```

The last name and year of birth of everyone who died in 1772 or 1792.

```
> (with-answer (or (dates ?x ?y 1772)
                   (dates ?x ?y 1792))
    (princ (list ?x ?y)))
(HOGARTH 1697)(REYNOLDS 1723)
NIL
```

The last name of every English painter not born the same year as a Venetian one.

```
> (with-answer (and (painter ?x _ english)
                    (dates ?x ?b _)
                    (not (and (painter ?x2 _ venetian)
                              (dates ?x2 ?b _))))
    (princ ?x))
REYNOLDS
NIL
```

Figure 19.5: The query interpreter in use.

then we would get `nil` as the result if there were *any* painters born in 1772. Even in the first example, we shouldn't expect to be able to use the value of `?d` within the body of a `with-answer` expression.

Also, expressions of the form `(or` $q_1 \ldots q_n$`)` are only guaranteed to generate real bindings for variables which appear in all of the q_i. If a `with-answer` contained the query

```
(or (painter ?x ?y ?z) (dates ?x ?b ?d))
```

you could expect to use the binding of `?x`, because no matter which of the subqueries succeeds, it will generate a binding for `?x`. But neither `?y` nor `?b` is guaranteed to get a binding from the query, though one or the other will. Pattern variables not bound by the query will be `nil` for that iteration.

19.5 A Query Compiler

The code in Figure 19.3 does what we want, but inefficiently. It analyzes the structure of the query at runtime, though it is known at compile-time. And it conses up lists to hold variable bindings, when we could use the variables to hold their own values. Both of these problems can be solved by defining `with-answer` in a different way.

Figure 19.6 defines a new version of `with-answer`. The new version continues a trend which began with `avg` (page 182), and continued with `if-match` (page 242): it does at compile-time much of the work that the old version did at runtime. The code in Figure 19.6 bears a superficial resemblance to that in Figure 19.3, but none of these functions are called at runtime. Instead of generating bindings, they generate code, which becomes part of the expansion of `with-answer`. At runtime this code will generate all the bindings which satisfy the query according to the current state of the database.

In effect, this program is one big macro. Figure 19.7 shows the macroexpansion of a `with-answer`. Most of the work is done by `pat-match` (page 242), which is itself a macro. Now the only new functions needed at runtime are the basic database functions shown in Figure 19.1.

When `with-answer` is called from the toplevel, query compilation has little advantage. The code representing the query is generated, evaluated, then thrown away. But when a `with-answer` expression appears within a Lisp program, the code representing the query becomes part of its macroexpansion. So when the containing program is compiled, the code for all the queries will be compiled inline in the process.

Although the primary advantage of the new approach is speed, it also makes `with-answer` expressions better integrated with the code in which they appear. This shows in two specific improvements. First, the arguments within the query now get evaluated, so we can say:

```
> (setq my-favorite-year 1723)
1723
> (with-answer (dates ?x my-favorite-year ?d)
    (format t "~A was born in my favorite year.~%" ?x))
REYNOLDS was born in my favorite year.
NIL
```

```
(defmacro with-answer (query &body body)
  `(with-gensyms ,(vars-in query #'simple?)
     ,(compile-query query `(progn ,@body))))

(defun compile-query (q body)
  (case (car q)
    (and  (compile-and (cdr q) body))
    (or   (compile-or  (cdr q) body))
    (not  (compile-not (cadr q) body))
    (lisp `(if ,(cadr q) ,body))
    (t    (compile-simple q body))))

(defun compile-simple (q body)
  (let ((fact (gensym)))
    `(dolist (,fact (db-query ',(car q)))
       (pat-match ,(cdr q) ,fact ,body nil))))

(defun compile-and (clauses body)
  (if (null clauses)
      body
      (compile-query (car clauses)
                     (compile-and (cdr clauses) body))))

(defun compile-or (clauses body)
  (if (null clauses)
      nil
      (let ((gbod (gensym))
            (vars (vars-in body #'simple?)))
        `(labels ((,gbod ,vars ,body))
           ,@(mapcar #'(lambda (cl)
                         (compile-query cl `(,gbod ,@vars)))
                     clauses)))))

(defun compile-not (q body)
  (let ((tag (gensym)))
    `(if (block ,tag
           ,(compile-query q `(return-from ,tag nil))
           t)
         ,body)))
```

Figure 19.6: Query compiler.

```
(with-answer (painter ?x ?y ?z)
  (format t "~A ~A is a painter.~%" ?y ?x))
```

is expanded by the query interpreter into:

```
(dolist (#:g1 (interpret-query '(painter ?x ?y ?z)))
  (let ((?x (binding '?x #:g1))
        (?y (binding '?y #:g1))
        (?z (binding '?z #:g1)))
    (format t "~A ~A is a painter.~%" ?y ?x)))
```

and by the query compiler into:

```
(with-gensyms (?x ?y ?z)
  (dolist (#:g1 (db-query 'painter))
    (pat-match (?x ?y ?z) #:g1
         (progn
           (format t "~A ~A is a painter.~%" ?y ?x))
         nil)))
```

Figure 19.7: Two expansions of the same query.

This could have been done in the query interpreter, but only at the cost of calling `eval` explicitly. And even then, it wouldn't have been possible to refer to lexical variables in the query arguments.

Since arguments within queries are now evaluated, any literal argument (e.g. `english`) that doesn't evaluate to itself should now be quoted. (See Figure 19.8.)

The second advantage of the new approach is that it is now much easier to include normal Lisp expressions within queries. The query compiler adds a `lisp` operator, which may be followed by any Lisp expression. Like the `not` operator, it cannot generate bindings by itself, but it will screen out bindings for which the expression returns `nil`. The `lisp` operator is useful for getting at built-in predicates like `>`:

```
> (with-answer (and (dates ?x ?b ?d)
                    (lisp (> (- ?d ?b) 70)))
    (format t "~A lived over 70 years.~%" ?x))
CANALE lived over 70 years.
HOGARTH lived over 70 years.
NIL
```

A well-implemented embedded language can have a seamless interface with the

The first name and nationality of every painter called Hogarth.

```
> (with-answer (painter 'hogarth ?x ?y)
    (princ (list ?x ?y)))
(WILLIAM ENGLISH)
NIL
```

The last name of every English painter not born in the same year as a Venetian painter.

```
> (with-answer (and (painter ?x _ 'english)
                    (dates ?x ?b _)
                    (not (and (painter ?x2 _ 'venetian)
                              (dates ?x2 ?b _))))
    (princ ?x))
REYNOLDS
NIL
```

The last name and year of death of every painter who died between 1770 and 1800 exclusive.

```
> (with-answer (and (painter ?x _ _)
                    (dates ?x _ ?d)
                    (lisp (< 1770 ?d 1800)))
    (princ (list ?x ?d)))
(REYNOLDS 1792)(HOGARTH 1772)
NIL
```

Figure 19.8: The query compiler in use.

base language on both sides.

Aside from these two additions—the evaluation of arguments and the new `lisp` operator—the query language supported by the query compiler is identical to that supported by the interpreter. Figure 19.8 shows examples of the results generated by the query compiler with the database as defined in Figure 19.4.

Section 17.2 gave two reasons why it is better to compile an expression than feed it, as a list, to `eval`. The former is faster, and allows the expression to be evaluated in the surrounding lexical context. The advantages of query compilation are exactly analogous. Work that used to be done at runtime is now done at compile-time. And because the queries are compiled as a piece with the surrounding Lisp code, they can take advantage of the lexical context.