4.4 Logic Programming

In this section, we introduce a declarative query language called `logic`, designed specifically for this text. It is based upon Prolog and the declarative language in Structure and Interpretation of Computer Programs. Data records are expressed as Scheme lists, and queries are expressed as Scheme values. The logic interpreter is a complete implementation that depends upon the Scheme project of the previous chapter.

4.4.1 Facts and Queries

Databases store records that represent facts in the system. The purpose of the query interpreter is to retrieve collections of facts drawn directly from database records, as well as to deduce new facts from the database using logical inference. A `fact` statement in the `logic` language consists of one or more lists following the keyword `fact`. A simple fact is a single list. A dog breeder with an interest in U.S. Presidents might record the genealogy of her collection of dogs using the `logic` language as follows:

```
(fact (parent abraham barack))
(fact (parent abraham clinton))
(fact (parent delano herbert))
(fact (parent fillmore abraham))
(fact (parent fillmore delano))
(fact (parent fillmore grover))
(fact (parent eisenhower fillmore))
```

Each fact is not a procedure application, as in a Scheme expression, but instead a *relation* that is declared. "The dog Abraham is the parent of Barack," declares the first fact. Relation types do not need to be defined in advance. Relations are not applied, but instead matched to queries.

A query also consists of one or more lists, but begins with the keyword `query`. A query may contain variables, which are symbols that begin with a question mark. Variables are matched to facts by the query interpreter:

```
(query (parent abraham ?child))
```

```
Success!
child: barack
child: clinton
```

The query interpreter responds with `Success!` to indicate that the query matches some fact. The following lines show substitutions of the variable `?child` that match the query to the facts in the database.

Compound facts. Facts may also contain variables as well as multiple sub-expressions. A multi-expression fact begins with a conclusion, followed by hypotheses. For the conclusion to be true, all of the hypotheses must be satisfied:

```
(fact <conclusion> <hypothesis0> <hypothesis1> ... <hypothesisN>)
```

For example, facts about children can be declared based on the facts about parents already in the database:

```
(fact (child ?c ?p) (parent ?p ?c))
```

The fact above can be read as: "`?c` is the child of `?p`, provided that `?p` is the parent of `?c`." A query can now refer to this fact:

```
(query (child ?child fillmore))
```

```
Success!
child: abraham
child: delano
child: grover
```

The query above requires the query interpreter to combine the fact that defines `child` with the various parent facts about `fillmore`. The user of the language does not need to know how this information is combined, but only that the result has a particular form. It is up to the query interpreter to prove that `(child abraham fillmore)` is true, given the available facts.

A query is not required to include variables; it may simply verify a fact:

```
(query (child herbert delano))
```

```
Success!
```

A query that does not match any facts will return failure:

```
(query (child eisenhower ?parent))
```

```
Failed.
```

Negation. We can check if some query does not match any fact by using the special keyword `not`:

```
(query (not <relation>))
```

This query succeeds if `<relation>` fails, and fails if `<relation>` succeeds. This idea is known as *negation as failure*.

```
(query (not (parent abraham clinton)))
```

```
Failed.
```

```
(query (not (parent abraham barack)))
```

```
Failed.
```

Sometimes, negation as failure may be counterintuitive to how one might expect negation to work. Think about the result of the following query:

```
(query (not (parent abraham ?who)))
```

Why does this query fail? Surely there are many symbols that could be bound to `?who` for which this should hold. However, the steps for negation indicate that we first inspect the relation `(parent abraham ?who)`. This relation succeeds, since `?who` can be bound to either `barack` or `clinton`. Because this relation succeeds, the negation of this relation must fail.

4.4.2 Recursive Facts

The `logic` language also allows recursive facts. That is, the conclusion of a fact may depend upon a hypothesis that contains the same symbols. For instance, the ancestor relation is

defined with two facts. Some `?a` is an ancestor of `?y` if it is a parent of `?y` or if it is the parent of an ancestor of `?y`:

```
(fact (ancestor ?a ?y) (parent ?a ?y))
(fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))
```

A single query can then list all ancestors of `herbert`:

```
(query (ancestor ?a herbert))
```

```
Success!
a: delano
a: fillmore
a: eisenhower
```

Compound queries. A query may have multiple subexpressions, in which case all must be satisfied simultaneously by an assignment of symbols to variables. If a variable appears more than once in a query, then it must take the same value in each context. The following query finds ancestors of both `herbert` and `barack`:

```
(query (ancestor ?a barack) (ancestor ?a herbert))
```

```
Success!
a: fillmore
a: eisenhower
```

Recursive facts may require long chains of inference to match queries to existing facts in a database. For instance, to prove the fact `(ancestor fillmore herbert)`, we must prove each of the following facts in succession:

```
(parent delano herbert)       ; (1), a simple fact
(ancestor delano herbert)     ; (2), from (1) and the 1st ancestor fact
(parent fillmore delano)      ; (3), a simple fact
(ancestor fillmore herbert)   ; (4), from (2), (3), & the 2nd ancestor fact
```

In this way, a single fact can imply a large number of additional facts, or even infinitely many, as long as the query interpreter is able to discover them.

Hierarchical facts. Thus far, each fact and query expression has been a list of symbols. In addition, fact and query lists can contain lists, providing a way to represent hierarchical data. The color of each dog may be stored along with the name an additional record:

```
(fact (dog (name abraham) (color white)))
(fact (dog (name barack) (color tan)))
(fact (dog (name clinton) (color white)))
(fact (dog (name delano) (color white)))
(fact (dog (name eisenhower) (color tan)))
(fact (dog (name fillmore) (color brown)))
(fact (dog (name grover) (color tan)))
(fact (dog (name herbert) (color brown)))
```

Queries can articulate the full structure of hierarchical facts, or they can match variables to whole lists:

```
(query (dog (name clinton) (color ?color)))
```

```
Success!
color: white
```

```
(query (dog (name clinton) ?info))
```

```
Success!
info: (color white)
```

Much of the power of a database lies in the ability of the query interpreter to join together multiple kinds of facts in a single query. The following query finds all pairs of dogs for which one is the ancestor of the other and they share a color:

```
(query (dog (name ?name) (color ?color))
       (ancestor ?ancestor ?name)
       (dog (name ?ancestor) (color ?color)))
```

```
Success!
name: barack color: tan ancestor: eisenhower
name: clinton color: white ancestor: abraham
name: grover color: tan ancestor: eisenhower
name: herbert color: brown ancestor: fillmore
```

Variables can refer to lists in hierarchical records, but also using dot notation. A variable following a dot matches the rest of the list of a fact. Dotted lists can appear in either facts or queries. The following example constructs pedigrees of dogs by listing their chain of ancestry. Young `barack` follows a venerable line of presidential pups:

```
(fact (pedigree ?name) (dog (name ?name) . ?details))
(fact (pedigree ?child ?parent . ?rest)
      (parent ?parent ?child)
      (pedigree ?parent . ?rest))

(query (pedigree barack . ?lineage))
```

```
Success!
lineage: ()
lineage: (abraham)
lineage: (abraham fillmore)
lineage: (abraham fillmore eisenhower)
```

Declarative or logical programming can express relationships among facts with remarkable efficiency. For example, if we wish to express that two lists can append to form a longer list with the elements of the first, followed by the elements of the second, we state two rules. First, a base case declares that appending an empty list to any list gives that list:

```
(fact (append-to-form () ?x ?x))
```

Second, a recursive fact declares that a list with first element `?a` and rest `?r` appends to a list `?y` to form a list with first element `?a` and some appended rest `?z`. For this relation to hold, it must be the case that `?r` and `?y` append to form `?z`:

```
(fact (append-to-form (?a . ?r) ?y (?a . ?z)) (append-to-form ?r ?y ?z))
```

Using these two facts, the query interpreter can compute the result of appending any two lists together:

```
(query (append-to-form (a b c) (d e) ?result))
```

```
Success!
result: (a b c d e)
```

In addition, it can compute all possible pairs of lists `?left` and `?right` that can append to form the list `(a b c d e)`:

```
(query (append-to-form ?left ?right (a b c d e)))
```

```
Success!
left: () right: (a b c d e)
left: (a) right: (b c d e)
left: (a b) right: (c d e)
```

```
left: (a b c) right: (d e)
left: (a b c d) right: (e)
left: (a b c d e) right: ()
```

Although it may appear that our query interpreter is quite intelligent, we will see that it finds these combinations through one simple operation repeated many times: that of matching two lists that contain variables in an environment.

Continue: 4.5 Unification
