Lisp is the major language for AI work, but it is by no means the only one. The other strong contender is Prolog, whose name derives from “programming in logic.” The idea behind logic programming is that the programmer should state the relationships that describe a problem and its solution. These relationships act as constraints on the algorithms that can solve the problem, but the system itself, rather than the programmer, is responsible for the details of the algorithm. The tension between the “programming” and “logic” will be covered in chapter 14, but for now it is safe to say that Prolog is an approximation to the ideal goal of logic programming. Prolog has arrived at a comfortable niche between a traditional programming language and a logical specification language. It relies on three important ideas:

1. Actually, programmation en logique, since it was invented by a French group (see page 382).
Prolog encourages the use of a single uniform data base. Good compilers provide efficient access to this data base, reducing the need for vectors, hash tables, property lists, and other data structures that the Lisp programmer must deal with in detail. Because it is based on the idea of a data base, Prolog is relational, while Lisp (and most languages) are functional. In Prolog we would represent a fact like "the population of San Francisco is 750,000" as a relation. In Lisp, we would be inclined to write a function, population, which takes a city as input and returns a number. Relations are more flexible; they can be used not only to find the population of San Francisco but also, say, to find the cities with populations over 500,000.

Prolog provides logic variables instead of "normal" variables. A logic variable is bound by unification rather than by assignment. Once bound, a logic variable can never change. Thus, they are more like the variables of mathematics. The existence of logic variables and unification allow the logic programmer to state equations that constrain the problem (as in mathematics), without having to state an order of evaluation (as with assignment statements).

Prolog provides automatic backtracking. In Lisp each function call returns a single value (unless the programmer makes special arrangements to have it return multiple values, or a list of values). In Prolog, each query leads to a search for relations in the data base that satisfy the query. If there are several, they are considered one at a time. If a query involves multiple relations, as in "what city has a population over 500,000 and is a state capital?," Prolog will go through the population relation to find a city with a population over 500,000. For each one it finds, it then checks the capital relation to see if the city is a capital. If it is, Prolog prints the city; otherwise it backtracks, trying to find another city in the population relation. So Prolog frees the programmer from worrying about both how data is stored and how it is searched. For some problems, the naive automatic search will be too inefficient, and the programmer will have to restate the problem. But the ideal is that Prolog programs state constraints on the solution, without spelling out in detail how the solutions are achieved.

This chapter serves two purposes: it alerts the reader to the possibility of writing certain programs in Prolog rather than Lisp, and it presents implementations of the three important Prolog ideas, so that they may be used (independently or together) within Lisp programs. Prolog represents an interesting, different way of looking at the programming process. For that reason it is worth knowing. In subsequent chapters we will see several useful applications of the Prolog approach.
11.1 Idea 1: A Uniform Data Base

The first important Prolog idea should be familiar to readers of this book: manipulating a stored data base of assertions. In Prolog the assertions are called clauses, and they can be divided into two types: facts, which state a relationship that holds between some objects, and rules, which are used to state contingent facts. Here are representations of two facts about the population of San Francisco and the capital of California. The relations are population and capital, and the objects that participate in these relations are SF, 750000, Sacramento, and CA:

(population SF 750000)
(capital Sacramento CA)

We are using Lisp syntax, because we want a Prolog interpreter that can be imbedded in Lisp. The actual Prolog notation would be population(sf,750000). Here are some facts pertaining to the likes relation:

(likes Kim Robin)
(likes Sandy Lee)
(likes Sandy Kim)
(likes Robin cats)

These facts could be interpreted as meaning that Kim likes Robin, Sandy likes both Lee and Kim, and Robin likes cats. We need some way of telling Lisp that these are to be interpreted as Prolog facts, not a Lisp function call. We will use the macro <- to mark facts. Think of this as an assignment arrow which adds a fact to the data base:

(- (likes Kim Robin))
(- (likes Sandy Lee))
(- (likes Sandy Kim))
(- (likes Robin cats))

One of the major differences between Prolog and Lisp hinges on the difference between relations and functions. In Lisp, we would define a function likes, so that (likes 'Sandy) would return the list (Lee Kim). If we wanted to access the information the other way, we would define another function, say, likes-of, so that (likes-of 'Lee) returns (Sandy). In Prolog, we have a single likes relation instead of multiple functions. This single relation can be used as if it were multiple functions by posing different queries. For example, the query (likes Sandy ?who) succeeds with ?who bound to Lee or Kim, and the query (likes ?who Lee) succeeds with ?who bound to Sandy.
The second type of clause in a Prolog database is the rule. Rules state contingent facts. For example, we can represent the rule that Sandy likes anyone who likes cats as follows:

\[ \text{\texttt{\textless - (\text{likes Sandy ?x} \text{ likes ?x cats})}} \]

This can be read in two ways. Viewed as a logical assertion, it is read, "For any x, Sandy likes x if x likes cats." This is a declarative interpretation. Viewed as a piece of a Prolog program, it is read, "If you ever want to show that Sandy likes some x, one way to do it is to show that x likes cats." This is a procedural interpretation. It is called a backward-chaining interpretation, because one reasons backward from the goal (Sandy likes x) to the premises (x likes cats). The symbol \textless - is appropriate for both interpretations: it is an arrow indicating logical implication, and it points backwards to indicate backward chaining.

It is possible to give more than one procedural interpretation to a declarative form. (We did that in chapter 1, where grammar rules were used to generate both strings of words and parse trees.) The rule above could have been interpreted procedurally as "If you ever find out that some x likes cats, then conclude that Sandy likes x." This would be forward chaining: reasoning from a premise to a conclusion. It turns out that Prolog does backward chaining exclusively. Many expert systems use forward chaining exclusively, and some systems use a mixture of the two.

The leftmost expression in a clause is called the head, and the remaining ones are called the body. In this view, a fact is just a rule that has no body; that is, a fact is true no matter what. In general, then, the form of a clause is:

\[ \text{\texttt{\textless - head body...}} \]

A clause asserts that the head is true only if all the goals in the body are true. For example, the following clause says that Kim likes anyone who likes both Lee and Kim:

\[ \text{\texttt{\textless - (\text{likes Kim ?x} \text{ likes ?x Lee}) (\text{likes ?x Kim})}} \]

This can be read as:

\[ For \text{\text{ any x, deduce that Kim likes x}} \]

\[ if \text{\text{ it can be proved that x likes Lee and x likes Kim.}} \]
11.2 Idea 2: Unification of Logic Variables

Unification is a straightforward extension of the idea of pattern matching. The pattern-matching functions we have seen so far have always matched a pattern (an expression containing variables) against a constant expression (one with no variables). In unification, two patterns, each of which can contain variables, are matched against each other. Here's an example of the difference between pattern matching and unification:

\[
\begin{align*}
& (\text{pat-match } '(?x + ?y) '(2 + 1)) \Rightarrow ((?y . 1) (?x . 2)) \\
& (\text{unify } '(?x + 1) '(2 + ?y)) \Rightarrow ((?y . 1) (?x . 2))
\end{align*}
\]

Within the unification framework, variables (such as \( ?x \) and \( ?y \) above) are called logic variables. Like normal variables, a logic variable can be assigned a value, or it can be unbound. The difference is that a logic variable can never be altered. Once it is assigned a value, it keeps that value. Any attempt to unify it with a different value leads to failure. It is possible to unify a variable with the same value more than once, just as it was possible to do a pattern match of \( (?x + ?x) \) with \( (2 + 2) \).

The difference between simple pattern matching and unification is that unification allows two variables to be matched against each other. The two variables remain unbound, but they become equivalent. If either variable is subsequently bound to a value, then both variables adopt that value. The following example equates the variables \(?x\) and \(?y\) by binding \(?x\) to \(?y\):

\[
(\text{unify } '(?x) '(?y)) \Rightarrow ((?x . ?y))
\]

Unification can be used to do some sophisticated reasoning. For example, if we have two equations, \( a + a = 0 \) and \( x + y = y \), and if we know that these two equations unify, then we can conclude that \( a, x, \) and \( y \) are all 0. The version of unify we will define shows this result by binding \(?y\) to 0, \(?x\) to \(?y\), and \(?a\) to \(?x\). We will also define the function unify, which shows the structure that results from unifying two structures.

\[
\begin{align*}
& (\text{unify } '(?a + ?a = 0) '(?x + ?y = ?y)) \Rightarrow \\
& \quad ((?y . 0) (?x . ?y) (?a . ?x)) \\
& (\text{unifier } '(?a + ?a = 0) '(?x + ?y = ?y)) \Rightarrow (0 + 0 = 0)
\end{align*}
\]

To avoid getting carried away by the power of unification, it is a good idea to take stock of exactly what unification provides. It does provide a way of stating that variables are equal to other variables or expressions. It does not provide a way of automatically solving equations or applying constraints other than equality. The following example
makes it clear that unification treats the symbol + only as an uninterpreted atom, not as the addition operator:

\[
> \text{(unifier '(?a + ?a = 2) (x + ?y = ?y)) ⇒ (2 + 2 = 2)}
\]

Before developing the code for unify, we repeat here the code taken from the pattern-matching utility (chapter 6):

```lisp
(defconstant fail nil "Indicates pat-match failure")
(defconstant no-bindings "((t . t))
  "Indicates pat-match success, with no variables.")
(defun variable-p (x)
  "Is x a variable (a symbol beginning with '?')?"
  (and (symbolp x) (equal (char (symbol-name x) 0) #
     )))
(defun get-binding (var bindings)
  "Find a (variable . value) pair in a binding list."
  (assoc var bindings))
(defun binding-val (binding)
  "Get the value part of a single binding."
  (cdr binding))
(defun lookup (var bindings)
  "Get the value part (for var) from a binding list."
  (binding-val (get-binding var bindings)))
(defun extend-bindings (var val bindings)
  "Add a (var . value) pair to a binding list."
  (cons (cons var val)
    ;; Once we add a "real" binding,
    ;; we can get rid of the dummy no-bindings
    (if (and (eq bindings no-bindings))
      nil
    bindings)))
(defun match-variable (var input bindings)
  "Does VAR match INPUT? Uses (or updates) and returns bindings."
  (let ((binding (get-binding var bindings))
    (cond ((not binding) (extend-bindings var input bindings))
      ((equal input (binding-val binding)) bindings)
    (t fail))))
```

The unify function follows; it is identical to pat-match (as defined on page 180) except for the addition of the line marked ***. The function unify-variable also follows match-variable closely:
(defun unify (x y &optional (bindings 'no-bindings))
  "See if x and y match with given bindings."
  (cond ((eq bindings fail) fail)
        ((variable-p x) (unify-variable x y bindings))
        ((variable-p y) (unify-variable y x bindings))
        ((eq x y) bindings)
        ((and (consp x) (consp y))
         (extend-bindings var x bindings)
         (unify (rest x) (first y) (first y) bindings))
        (t fail)))

(defun unify-variable (var x bindings)
  "Unify var with x, using (and maybe extending) bindings."
  ;; Warning - buggy version
  (if (get-binding var bindings)
      (unify (lookup var bindings) x bindings)
      (extend-bindings var x bindings)))

Unfortunately, this definition is not quite right. It handles simple examples:

> (unify '((?x + 1) '(2 + ?y)) ((?y . 1) (?x . 2))
> (unify '?x '(?y) ((?x . ?y))
> (unify '((?x ?x) '(?y ?y)) ((?y . ?y) (?x . ?y))

but there are several pathological cases that it can’t contend with:

> (unify '((?x ?x ?x) '(?y ?y ?y))
> Trap #a043622 (PDL-OVERFLOW REGULAR)
  The regular push-down list has overflowed.
  While in the function GET-BINDING = UNIFY-VARIABLE = UNIFY

The problem here is that once ?y gets bound to itself, the call to unify inside
unify-variable leads to an infinite loop. But matching ?y against itself must al-
ways succeed, so we can move the equality test in unify before the variable test. This
assumes that equal variables are eql, a valid assumption for variables implemented
as symbols (but be careful if you ever decide to implement variables some other way).

(defun unify (x y &optional (bindings 'no-bindings))
  "See if x and y match with given bindings."
  (cond ((eq bindings fail) fail)
        ((eq x y) bindings) ;; moved this line
        ((variable-p x) (unify-variable x y bindings))
        ((variable-p y) (unify-variable y x bindings))
        ((and (consp x) (consp y))
         (unify (rest x) (rest y))
         (unify (first x) (first y) bindings)))
Here are some test cases:

> (unify '((?x ?x) (?y ?y)) ) ⇒ ((?X . ?Y))
> (unify '((?x ?x ?x) (?y ?y ?y)) ) ⇒ ((?X . ?Y))
> (unify '((?x ?y) (?y ?x)) ) ⇒ ((?Y . ?X) (?X . ?Y))
> (unify '((?x ?y a) (?y ?x ?x)) )
>>Trap #043622 (PDV-OVERFLOW REGULAR)

The regular push-down list has overflowed.

While in the function GET-BINDING ⇐ UNIFY-VARIABLE ⇐ UNIFY

We have pushed off the problem but not solved it. Allowing both (?Y . ?X) and
(?X . ?Y) in the same binding list is as bad as allowing (?Y . ?Y). To avoid the
problem, the policy should be never to deal with bound variables, but rather with
their values, as specified in the binding list. The function unify-variable fails to
implement this policy. It does have a check that gets the binding for var when it is a
bound variable, but it should also have a check that gets the value of x, when x is a
bound variable:

(defun unify-variable (var x bindings)
   "Unify var with x, using (and maybe extending) bindings."
   (cond ((get-binding var bindings)
       (unify (lookup var bindings) x bindings))
       ((and (variable-p x) (get-binding x bindings)) ;***
       (unify var (lookup x bindings) bindings)) ;***
       (t (extend-bindings var x bindings))))

Here are some more test cases:

> (unify '((?x ?y) '(?y ?x)) ) ⇒ ((?X . ?Y))
> (unify '((?x ?y a) '(?y ?x ?x)) ) ⇒ ((?Y . A) (?X . ?Y))

It seems the problem is solved. Now let's try a new problem:

(unify '?x '(f ?x)) ⇒ ((?X F ?X))

Here ((?X F ?X)) really means ((?X . ((F ?X))))), so ?X is bound to (F ?X). This
represents a circular, infinite unification. Some versions of Prolog, notably Prolog II
(Giannesini et al. 1986), provide an interpretation for such structures, but it is tricky
to define the semantics of infinite structures.
The easiest way to deal with such infinite structures is just to ban them. Thus, ban can be realized by modifying the unifier so that it fails whenever there is an attempt to unify a variable with a structure containing that variable. This is known in unification circles as the occurs check. In practice the problem rarely shows up, and since it can add a lot of computational complexity, most Prolog systems have ignored the occurs check. This means that these systems can potentially produce unsound answers. In the final version of unify following, a variable is provided to allow the user to turn occurs checking on or off.

(defparameter *occurs-check* t "Should we do the occurs check?"

(defun unify (x y &optional (bindings nil-bindings))
  "See if x and y match with given bindings."
  (cond ((eq bindings fail) fail)
        ((eql x y) bindings)
        ((variable-p x) (unify-variable x y bindings))
        ((variable-p y) (unify-variable y x bindings))
        ((and (consp x) (consp y))
           (unify (rest x) (rest y))
           (unify (first x) (first y) bindings)))
        (t fail)))

(defun unify-variable (var x bindings)
  "Unify var with x, using (and maybe extending) bindings."
  (cond ((get-binding var bindings)
        (unify (lookup var bindings) x bindings))
        ((and (variable-p x) (get-binding x bindings))
        (unify var (lookup x bindings) bindings))
        ((and *occurs-check* (occurs-check var x bindings))
        fail)
        (t (extend-bindings var x bindings))))

(defun occurs-check (var x bindings)
  "Does var occur anywhere inside x?"
  (cond ((eq var x) t)
        ((and (variable-p x) (get-binding x bindings))
        (occurs-check var (lookup x bindings) bindings))
        ((consp x) (or (occurs-check var (first x) bindings)
        (occurs-check var (rest x) bindings)))
        (t nil)))

Now we consider how unify will be used. In particular, one thing we want is a function for substituting a binding list into an expression. We originally chose association lists as the implementation of bindings because of the availability of the function subst. Ironically, subst isn't work any more, because variables can be bound to other variables, which are in turn bound to expressions. The function subst-bindings acts like subst, except that it substitutes recursive bindings.
(defun subst-bindings (bindings x)
   "Substitute the value of variables in bindings into x, taking recursively bound variables into account."
   (cond ((eq bindings fail) fail)
         ((eq bindings no-bindings) x)
         ((and (variable-p x) (get-binding x bindings))
          (subst-bindings bindings (lookup x bindings)))
         ((atom x) x)
         (t (reuse-cons (subst-bindings bindings (car x))
                   (subst-bindings bindings (cdr x))
                   x))))

Now let's try unify on some examples:

> (unify '((?x ?y a) '((?y ?x ?x))) => (((?y . A) (?x . ?y)))
> (unify '(?x '(f ?x)) => NIL
> (unify '((?x ?y) '(((f ?y) (f ?x))) => NIL
> (unify '((?x ?y ?z) '(((?y ?z) (?x ?z) (?x ?y))) => NIL
> (unify 'a 'a) => ((T . T))

Finally, the function unifier calls unify and substitutes the resulting binding list into one of the arguments. The choice of x is arbitrary; an equal result would come from substituting the binding list into y.

(defun unifier (x y)
   "Return something that unifies with both x and y (or fail)."
   (subst-bindings (unify x y) x))

Here are some examples of unifier:

> (unifier '((?x ?y a) '((?y ?x ?x))) => (A A A)
> (unifier '((?a * ?x ^ 2) + (?b * ?x) + ?c)
      '(@(z + (4 * 5) + 3)) =>
    (((?a + 5 ^ 2) + (4 * 5) + 3)
When *occurs-check* is false, we get the following answers:

```prolog
> (unify '(?x '(f ?x))) ⇒ ((?X F ?X))
> (unify '((?x ?y) '((f ?y) (f ?x))) ⇒
  ((?Y F ?X) (?X F ?Y))
> (unify '((?x ?y ?z) '((?y ?z) (?x ?z) (?x ?y))) ⇒
```

**Programming with Prolog**

The amazing thing about Prolog clauses is that they can be used to express relations that we would normally think of as "programs," not "data." For example, we can define the member relation, which holds between an item and a list that contains that item. More precisely, an item is a member of a list if it is either the first element of the list or a member of the rest of the list. This definition can be translated into Prolog almost verbatim:

```prolog
(\(- (member ?item (?item . ?rest)))
(\(- (member ?item (?x . ?rest)) (member ?item ?rest))
```

Of course, we can write a similar definition in Lisp. The most visible difference is that Prolog allows us to put patterns in the head of a clause, so we don’t need recognizers like consp or accessors like first and rest. Otherwise, the Lisp definition is similar:

```lisp
(defun lisp-member (item list)
  (and (consp list)
       (or (eql item (first list))
          (lisp-member item (rest list))))
```

If we wrote the Prolog code without taking advantage of the pattern feature, it would look more like the Lisp version:

```prolog
(\(- (member ?item ?list)
    (= ?list (?item . ?rest)))
```

\[2\] Actually, this is more like the Lisp find than the Lisp member. In this chapter we have adopted the traditional Prolog definition of member.
(\(\neg\) (member ?item ?list)
  (= ?list (?x . ?rest))
  (member ?item ?rest))

If we define or in Prolog, we would write a version that is clearly just a syntactic variant of the Lisp version.

(\(\neg\) (member ?item ?list)
  (= ?list (?first . ?rest))
  (or (= ?item ?first)
      (member ?item ?rest)))

Let's see how the Prolog version of member works. Imagine that we have a Prolog interpreter that can be given a query using the macro \(\neg\), and that the definition of member has been entered. Then we would see:

> (\(\neg\) (member 2 (1 2 3)))
Yes;

> (\(\neg\) (member 2 (1 2 3 2 1)))
Yes;
Yes;

The answer to the first query is “yes” because 2 is a member of the rest of the list. In the second query the answer is “yes” twice, because 2 appears in the list twice. This is a little surprising to Lisp programmers, but there still seems to be a fairly close correspondence between Prolog’s and Lisp’s member. However, there are things that the Prolog member can do that Lisp cannot:

> (\(\neg\) (member ?x (1 2 3)))
?x = 1;
?x = 2;
?x = 3;

Here member is used not as a predicate but as a generator of elements in a list. While Lisp functions always map from a specified input (or inputs) to a specified output, Prolog relations can be used in several ways. For member, we see that the first argument, ?x, can be either an input or an output, depending on the goal that is specified. This power to use a single specification as a function going in several different directions is a very flexible feature of Prolog. (Unfortunately, while it works very well for simple relations like member, in practice it does not work well for large programs. It is very difficult to, say, design a compiler and automatically have it work as a disassembler as well.)