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 another way, we would define another function, say, likers-of, so that (likers-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:

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

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 \(\langle - \) 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:

\[
\langle - \text{ head body...}
\]

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:

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

This can be read as:

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

\[
\text{if it can be proved that } x \text{ likes Lee and } x \text{ likes Kim.}
\]
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:

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

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:

\[
> \text{unify } '\ (\text{\texttt{f ?x}}) \ \Rightarrow \ ((?X \ . \ ?Y))
\]

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 unifier, which shows the structure that results from unifying two structures:

\[
\begin{align*}
&> \text{unify } '\ (a + a = 0) \ \Rightarrow \ ((?Y \ . \ 0) \ (?X \ . \ ?Y) \ (?A \ . \ ?X)) \\
&> \text{unifier } '\ (a + a = 0) \ \Rightarrow \ (0 + 0 = 0)
\end{align*}
\]

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:

```
> (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):

```
(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 nil bindings))
  "See if x and y match with given bindings."
  (cond ((eq bindings fail) fail)
        ((variable-p x) (uni ec-variable x y bindings))
        ((variable-p y) (uni ec-variable y x bindings))
        ((eql x y) 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."
  ;; 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)) \Rightarrow ((?y . 1) (?x . 2))
> (unify '?x '?y) \Rightarrow ((?x . ?y))
> (unify '(?x ?x) '(?y ?y)) \Rightarrow ((?y . ?y) (?x . ?y))

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

> (unify '(?x ?x ?x) '(?y ?y ?y))
> Trap 0x43622 (PDL-OVERFLOW REGULAR)
  The regular push-down list has overflowed.
  While in the function GET-BINDING \leftarrow UNIFY-VARIABLE \leftarrow 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 always succeed, so we can move the equality test in unify before the variable test. This assumes that equal variables are eq1, 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 nil bindings))
  "See if x and y match with given bindings."
  (cond ((eq bindings fail) fail)
        ((eql x y) bindings) ;*** moved this line
        ((variable-p x) (uni ec-variable x y bindings))
        ((variable-p y) (uni ec-variable y x bindings))
        ((and (consp x) (consp y))
         (unify (rest x) (rest y)
               (unify (first x) (first y) bindings)))
        (t fail)))
Here are some test cases:

> (unify '(@(x ?x) @(y ?y)) @(?(X . ?Y)))
> (unify '(@(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 (PD-Overflow Regular)
  >The regular push-down list has overflowed.
While in the function GET-BINDING \equiv UNIFY-VARIABLE \equiv 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. This 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 no-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's won'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 *occur* -check* is false, we get the following answers:

\[
\begin{align*}
> \text{unify ‘}(x \ ‘(f \ ?x)) & \Rightarrow ((\text{?X} \ F \ ?X)) \\
> \text{unify ‘}(\text{?x} \ ?y) ’(\text{f} \ ?y) \ (\text{f} \ ?x)) & \Rightarrow \\
& ((\text{?Y} \ F \ ?X) \ (\text{?X} \ F \ ?Y)) \\
> \text{unify ‘}(\text{?x} \ ?y \ ?z) ’((\text{?y} \ ?z) \ (\text{?x} \ ?z) \ (\text{?x} \ ?y)) & \Rightarrow \\
& ((\text{?z} \ ?X \ ?Y) \ (?Y \ ?X \ ?Z) \ (?X \ ?Y \ ?Z))
\end{align*}
\]

**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:

\[
\begin{align*}
\langle- \text{member ?item (?item . ?rest))} \\
\langle- \text{member ?item (?x . ?rest))} \text{ (member ?item ?rest)}
\end{align*}
\]

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:

\[
\begin{align*}
\text{(defun lisp-member \item \list)} \\
\text{\hspace{1em} (and \consp \list)} \\
\text{\hspace{2em} (or \eql \item \text{(first} \list))} \\
\text{\hspace{3em} \text{(lisp-member \item \text{(rest} \list)))}}
\end{align*}
\]

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

\[
\begin{align*}
\langle- \text{member ?item ?list} \\
\text{\hspace{1em} (= ?list (?item . ?rest))}
\end{align*}
\]

\[\text{\textsuperscript{2}Actually, this is more like the Lisp \texttt{find} than the Lisp \texttt{member}. In this chapter we have adopted the traditional Prolog definition of \texttt{member}.}\]
\begin{verbatim}
\(<-\ (\text{member } ?\text{item } ?\text{list})
\n\quad (= ?\text{list} (\text{?first . } ?\text{rest}))
\n\quad (\text{member } ?\text{item } ?\text{rest}))
\end{verbatim}

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

\begin{verbatim}
\(<-\ (\text{member } ?\text{item } ?\text{list})
\n\quad (= ?\text{list} (\text{?first . } ?\text{rest}))
\n\quad (\text{or } (= ?\text{item } ?\text{first})
\n\quad (\text{member } ?\text{item } ?\text{rest})))
\end{verbatim}

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 ?-., and that the definition of member has been entered. Then we would see:

\begin{verbatim}
> (?- (\text{member } 2 \text{ (1 2 3)}))
Yes;

> (?- (\text{member } 2 \text{ (1 2 3 2 1)}))
Yes;
Yes;
\end{verbatim}

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:

\begin{verbatim}
> (?- (\text{member } ?x \text{ (1 2 3)}))
?x = 1;
?x = 2;
?x = 3;
\end{verbatim}

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.)
Now we turn to the implementation of the Prolog interpreter, as summarized in figure 11.1. The first implementation choice is the representation of rules and facts. We will build a single uniform data base of clauses, without distinguishing rules from facts. The simplest representation of clauses is as a cons cell holding the head and the body. For facts, the body will be empty.

;;; Clauses are represented as (head . body) cons cells
(defun clause-head (clause) (first clause))
(defun clause-body (clause) (rest clause))

The next question is how to index the clauses. Recall the procedural interpretation of a clause: when we want to prove the head, we can do it by proving the body. This suggests that clauses should be indexed in terms of their heads. Each clause will be stored on the property list of the predicate of the head of the clause. Since the data base is now distributed across the property list of various symbols, we represent the entire data base as a list of symbols stored as the value of *db-predicates*.

;;; Clauses are stored on the predicate's plist
(defun get-clauses (pred) (get pred 'clauses))
(defun predicate (relation) (first relation))

(defvar *db-predicates* nil
"A list of all predicates stored in the database."
)

Now we need a way of adding a new clause. The work is split up into the macro <-, which provides the user interface, and a function, add-clause, that does the work. It is worth defining a macro to add clauses because in effect we are defining a new language: Prolog-In-Lisp. This language has only two syntactic constructs: the <- macro to add clauses, and the ?- macro to make queries.

(defmacro <- (&rest clause)
"Add a clause to the data base."
'(add-clause ,.clause))

(defun add-clause (clause)
"Add a clause to the data base, indexed by head's predicate."
;;; The predicate must be a non-variable symbol.
(let ((pred (predicate (clause-head clause))))
  (assert (and (symbolp pred) (not (variable-p pred)))
  (pushnew pred *db-predicates*)
  (setf (get pred 'clauses)
    (nconc (get-clauses pred) (list clause)))
  pred))

Now all we need is a way to remove clauses, and the data base will be complete.
### Top-Level Macros
- Add a clause to the data base.
- Prove a query and print answer(s).

### Special Variables
- *db-predicates*
  - A list of all predicates.
- *occurs-check*
  - Should we check for circular unifications?

### Data Types
- **clause**
  - Consists of a head and a body.
- **variable**
  - A symbol starting with a ?.

### Major Functions
- **add-clause**
  - Add a clause to the data base.
- **prove**
  - Return a list of possible solutions to goal.
- **prove-al**
  - Return a list of solutions to the conjunction of goals.
- **top-level-prove**
  - Prove the goals, and print variables readably.

### Auxiliary Functions
- **get-clauses**
  - Find all the clauses for a predicate.
- **predicate**
  - Pick out the predicate from a relation.
- **clear-db**
  - Remove all clauses (for all predicates) from the data base.
- **clear-predicate**
  - Remove the clauses for a single predicate.
- **rename-variables**
  - Replace all variables in x with new ones.
- **unique-find-anywhere-if**
  - Find all unique leaves satisfying predicate.
- **show-prolog-solutions**
  - Print the variables in each of the solutions.
- **show-prolog-vars**
  - Print each variable with its binding.
- **variables-in**
  - Return a list of all the variables in an expression.

### Previously Defined Constants
- **fail**
  - An indication that unification has failed.
- **no-bindings**
  - A successful unification with no variables.

### Previously Defined Functions
- **unify**
  - Return bindings that unify two expressions (section 11.2).
- **unify-variable**
  - Unify a variable against an expression.
- **occurs-check**
  - See if a particular variable occurs inside an expression.
- **subst-bindings**
  - Substitute bindings into an expression.
- **get-binding**
  - Get the (var . val) binding for a variable.
- **lookup**
  - Get the value for a variable.
- **extend-bindings**
  - Add a new variable/value pair to a binding list.
- **variable-p**
  - Is the argument a variable?
- **reuse-cons**
  - Like cons, except will reuse an old value if possible.

---

Figure 11.1: Glossary for the Prolog Interpreter
(defun clear-db ()
  "Remove all clauses (for all predicates) from the database."
  (mapc #'clear-predicate *db-predicates*))

(defun clear-predicate (predicate)
  "Remove the clauses for a single predicate."
  (setf (get predicate 'clauses) nil))

A data base is useless without a way of getting data out, as well as putting it in. The
function prove will be used to prove that a given goal either matches a fact that is in
the database directly or can be derived from the rules. To prove a goal, first find all
the candidate clauses for that goal. For each candidate, check if the goal unifies with
the head of the clause. If it does, try to prove all the goals in the body of the clause.
For facts, there will be no goals in the body, so success will be immediate. For rules,
the goals in the body need to be proved one at a time, making sure that bindings from
the previous step are maintained. The implementation is straightforward:

(defun prove (goal bindings)
  "Return a list of possible solutions to goal."
  (mapcan *'(lambda (clause)
            (let ((new-clause (rename-variables clause)))
              (prove-all (clause-body new-clause)
                          (unify goal (clause-head new-clause) bindings)))
               (get-clauses (predicate goal)))))

(defun prove-all (goals bindings)
  "Return a list of solutions to the conjunction of goals."
  (cond ((eq bindings fail) fail)
        ((null goals) (list bindings))
        (t (mapcan *'(lambda (goal-solution)
                       (prove-all (rest goals) goal-solution))
                      (prove (first goals) bindings))))

The tricky part is that we need some way of distinguishing a variable \(?x\) in one
clause from another variable \(?x\) in another clause. Otherwise, a variable used in two
different clauses in the course of a proof would have to take on the same value in
each clause, which would be a mistake. Just as arguments to a function can have
different values in different recursive calls to the function, so the variables in a clause
are allowed to take on different values in different recursive uses. The easiest way to
keep variables distinct is just to rename all variables in each clause before it is used.
The function rename-variables does this.\(^3\)

\(^3\)See exercise 11.12 for an alternative approach.
(defun rename-variables (x)
  "Replace all variables in x with new ones."
  (sublis (mapcar #"(lambda (var) (cons var (gensym (string var))))
    (variables-in x))
  x))

Rename-variables makes use of gensym, a function that generates a new symbol each
time it is called. The symbol is not interned in any package, which means that there
is no danger of a programmer typing a symbol of the same name. The predicate
variables-in and its auxiliary function are defined here:

(defun variables-in (exp)
  "Return a list of all the variables in EXP."
  (unique-find-anywhere-if #\'variable-p exp))

(defun unique-find-anywhere-if (predicate tree
  &optional found-so-far)
  "Return a list of leaves of tree satisfying predicate,
with duplicates removed."
  (if (atom tree)
    (if (funcall predicate tree)
      (adjoin tree found-so-far)
      found-so-far)
    (unique-find-anywhere-if
      predicate
      (first tree)
      (unique-find-anywhere-if predicate (rest tree)
        found-so-far))))

Finally, we need a nice interface to the proving functions. We will use `?-` as a macro
to introduce a query. The query might as well allow a conjunction of goals, so `?-` will
call prove-all. Together, `<-` and `?-` define the complete syntax of our Prolog-In-Lisp
language.

(defmacro ?- (&rest goals) `(prove-all `goals no-bindings))

Now we can enter all the clauses given in the prior example:

(<- (likes Kim Robin))
(<- (likes Sandy Lee))
(<- (likes Sandy Kim))
(<- (likes Robin cats))
(<- (likes Sandy ?x) (likes ?x cats))
(<- (likes Kim ?x) (likes ?x Lee) (likes ?x Kim))
(<- (likes ?x ?x))
To ask whom Sandy likes, we would use:

```
> (?- (likes Sandy ?who))
((?WHO . LEE))
((?WHO . KIM))
((?X2856 . ROBIN) (?WHO . ?X2856))
((?X2860 . CATS) (?X2857 CATS) (?X2856 SANDY) (?WHO . ?X2856))
((?X2865 . CATS) (?X2856 ?X2865) (?WHO . ?X2856))
((?WHO . SANDY) (?X2867 . SANDY))
```

Perhaps surprisingly, there are six answers. The first two answers are Lee and Kim, because of the facts. The next three stem from the clause that Sandy likes everyone who likes cats. First, Robin is an answer because of the fact that Robin likes cats. To see that Robin is the answer, we have to unravel the bindings: ?who is bound to ?x2856, which is in turn bound to Robin.

Now we’re in for some surprises: Sandy is listed, because of the following reasoning: (1) Sandy likes anyone/thing who likes cats, (2) cats like cats because everyone likes themself, (3) therefore Sandy likes cats, and (4) therefore Sandy likes Sandy. Cats is an answer because of step (2), and finally, Sandy is an answer again, because of the clause about liking oneself. Notice that the result of the query is a list of solutions, where each solution corresponds to a different way of proving the query true. Sandy appears twice because there are two different ways of showing that Sandy likes Sandy. The order in which solutions appear is determined by the order of the search. Prolog searches for solutions in a top-down, left-to-right fashion. The clauses are searched from the top down, so the first clauses entered are the first ones tried. Within a clause, the body is searched left to right. In using the (likes Kim ?x) clause, Prolog would first try to find an x who likes Lee, and then see if x likes Kim.

The output from prove-all is not very pretty. We can fix that by defining a new function, top-level-prove, which calls prove-all as before, but then passes the list of solutions to show-prolog-solutions, which prints them in a more readable format. Note that show-prolog-solutions returns no values: (values). This means the read-eval-print loop will not print anything when (values) is the result of a top-level call.

```lisp
(defun top-level-prove (goals)
  "Prove the goals, and print variables readable."
  (show-prolog-solutions
   (variables-in goals)
   (prove-all goals no-bindings)))
```

```
defmacro ?- (&rest goals)
  '(top-level-prove ,goals))
```
(defun show-prolog-solutions (vars solutions)  
  "Print the variables in each of the solutions."
  (if (null solutions)  
      (format t """"""&No.""")  
      (mapc #""""(lambda (solution) (show-prolog-vars vars solution))"""  
        solutions))  
  (values))

(defun show-prolog-vars (vars bindings)  
  "Print each variable with its binding."
  (if (null vars)  
      (format t """"""&Yes""")  
      (doalist (var vars)  
        (format t """"""&-a = ""a" var"""  
          (subst-bindings bindings var)))  
      (princ ":"')))  

Now let's try some queries:

> (?- (likes Sandy ?who))  
?WHO = LEE;  
?WHO = KIM;  
?WHO = ROBIN;  
?WHO = SANDY;  
?WHO = CATS;  
?WHO = SANDY;  

> (?- (likes ?who Sandy))  
?WHO = SANDY;  
?WHO = KIM;  
?WHO = SANDY;  

> (?- (likes Robin Lee))  
No.

The first query asks again whom Sandy likes, and the second asks who likes Sandy.  
The third asks for confirmation of a fact. The answer is "no," because there are no  
clauses or facts that say Robin likes Lee.  Here's another example, a list of pairs of  
people who are in a mutual liking relation. The last answer has an uninstantiated  
variable, indicating that everyone likes themselves.
> (?- (likes ?x ?y) (likes ?y ?x))
> ?Y = KIM
> ?X = SANDY;
> ?Y = SANDY
> ?X = SANDY;
> ?Y = SANDY
> ?X = SANDY;
> ?Y = SANDY
> ?X = KIM;
> ?Y = SANDY
> ?X = SANDY;
> ?Y = ?X3251
> ?X = ?X3251;

It makes sense in Prolog to ask open-ended queries like "what lists is 2 a member of?" or even "what items are elements of what lists?"

(?- (member 2 ?list))
(?- (member ?item ?list))

These queries are valid Prolog and will return solutions, but there will be an infinite number of them. Since our interpreter collects all the solutions into a single list before showing any of them, we will never get to see the solutions. The next section shows how to write a new interpreter that fixes this problem.

Exercise 11.1[m] The representation of relations has been a list whose first element is a symbol. However, for relations with no arguments, some people prefer to write (<- p q r) rather than (- (p) (q) (r)). Make changes so that either form is acceptable.

Exercise 11.2[m] Some people find the <- notation difficult to read. Define macros rule and fact so that we can write:

(fact (likes Robin cats))
(rule (likes Sandy ?x) if (likes ?x cats))
11.3 Idea 3: Automatic Backtracking

The Prolog interpreter implemented in the last section solves problems by returning a list of all possible solutions. We'll call this a batch approach, because the answers are retrieved in one uninterrupted batch of processing. Sometimes that is just what you want, but other times a single solution will do. In real Prolog, solutions are presented one at a time, as they are found. After each solution is printed, the user has the option of asking for more solutions, or stopping. This is an incremental approach. The incremental approach will be faster when the desired solution is one of the first out of many alternatives. The incremental approach will even work when there is an infinite number of solutions. And if that is not enough, the incremental approach can be implemented so that it searches depth-first. This means that at any point it will require less storage space than the batch approach, which must keep all solutions in memory at once.

In this section we implement an incremental Prolog interpreter. One approach would be to modify the interpreter of the last section to use pipes rather than lists. With pipes, unnecessary computation is delayed, and even infinite lists can be expressed in a finite amount of time and space. We could change to pipes simply by changing the mapcan in prove and prove-all to mappend-pipe (page 286). The books by Winston and Horn (1988) and by Abelson and Sussman (1985) take this approach. We take a different one.

The first step is a version of prove and prove-all that return a single solution rather than a list of all possible solutions. This should be reminiscent of achieve and achieve-all from gfs (chapter 4). Unlike gfs, recursive subgoals and clobbered sibling goals are not checked for. However, prove is required to search systematically through all solutions, so it is passed an additional parameter: a list of other goals to achieve after achieving the first goal. This is equivalent to passing a continuation to prove. The result is that if prove ever succeeds, it means the entire top-level goal has succeeded. If it fails, it just means the program is backtracking and trying another sequence of choices. Note that prove relies on the fact that fail is nil, because of the way it uses some.

(defun prove-all (goals bindings)
   "Find a solution to the conjunction of goals."
   (cond ((eq bindings fail) fail)
         ((null goals) bindings)
         (t (prove (first goals) bindings (rest goals))))
)

(defun prove (goal bindings other-goals)
   "Return a list of possible solutions to goal."
   (some '#'(lambda (clause)
             (let ((new-clause (rename-variables clause)))
               (prove-all
                (append (clause-body new-clause) other-goals)
                (append (goal bindings))
                (rest other-goals))))
)
(unify goal (clause-head new-clause) bindings)))
(get-clauses (predicate goal)))

If prove does succeed, it means a solution has been found. If we want more solutions, we need some way of making the process fail, so that it will backtrack and try again. One way to do that is to extend every query with a goal that will print out the variables, and ask the user if the computation should be continued. If the user says yes, then the goal fails, and backtracking starts. If the user says no, the goal succeeds, and since it is the final goal, the computation ends. This requires a brand new type of goal: one that is not matched against the data base, but rather causes some procedure to take action. In Prolog, such procedures are called primitives, because they are built-in to the language, and new ones may not be defined by the user. The user may, of course, define nonprimitive procedures that call upon the primitives.

In our implementation, primitives will be represented as Lisp functions. A predicate can be represented either as a list of clauses (as it has been so far) or as a single primitive. Here is a version of prove that calls primitives when appropriate:

(defun prove (goal bindings other-goals)
  "Return a list of possible solutions to goal."
  (let ((clauses (get-clauses (predicate goal))))
    (if (listp clauses)
        (some  
          #'(lambda (clause)
              (let ((new-clause (rename-variables clause)))
                (prove-all
                 (append (clause-body new-clause) other-goals)
                 (unify goal (clause-head new-clause) bindings))))
            clauses)
      ;; The predicate's "clauses" can be an atom:  
      ;; a primitive function to call 
      (funcall clauses (rest goal) bindings other-goals))))

Here is the version of top-level-prove that adds the primitive goal show-prolog-vars to the end of the list of goals. Note that this version need not call show-prolog-solutions itself, since the printing will be handled by the primitive for show-prolog-vars.

(defun top-level-prove (goals)
  (prove-all '('(goals (show-prolog-vars .#(variables-in goals)))
        no-bindings)
    (format t ""&No."
    (values))

Here we define the primitive show-prolog-vars. All primitives must be functions of
three arguments: a list of arguments to the primitive relation (here a list of variables to show), a binding list for these arguments, and a list of pending goals. A primitive should either return fail or call prove-all to continue.

(defun show-prolog-vars (vars bindings other-goals)
  "Print each variable with its binding. Then ask the user if more solutions are desired."
  (if (null vars)
      (format t "&Yes")
      (dolist (var vars)
        (format t "&" a = "a" var
                   (subst-bindings bindings var))))
    (if (continue-p)
        fail
        (prove-all other-goals bindings))

Since primitives are represented as entries on the clauses property of predicate symbols, we have to register show-prolog-vars as a primitive like this:

(setf (get 'show-prolog-vars 'clauses) 'show-prolog-vars)

Finally, the Lisp predicate continue-p asks the user if he or she wants to see more solutions:

(defun continue-p ()
  "Ask user if we should continue looking for solutions."
  (case (read-char)
    (\; t)
    (\; nil)
    (\newline (continue-p))
    (otherwise
      (format t "Type \; to see more or . to stop")
      (continue-p))))

This version works just as well as the previous version on finite problems. The only difference is that the user, not the system, types the semicolons. The advantage is that we can now use the system on infinite problems as well. First, we'll ask what lists 2 is a member of:
The answers mean that 2 is a member of any list that starts with 2, or whose second element is 2, or whose third element is 2, and so on. The infinite computation was halted when the user typed a period rather than a semicolon. The "no" now means that there are no more answers to be printed; it will appear if there are no answers at all, if the user types a period, or if all the answers have been printed.

We can ask even more abstract queries. The answer to the next query says that an item is an element of a list when it is the the first element, or the second, or the third, or the fourth, and so on.

Now let's add the definition of the relation length:

\[
\begin{align*}
\text{(length () 0)} \\
\text{(length (?x . ?y) (1+ ?n)) (length ?y ?n))}
\end{align*}
\]

Here are some queries showing that length can be used to find the second argument, the first, or both:

\[
\begin{align*}
\text{> (?- (length (\textquoteleft a \textquoteleft b \textquoteleft c \textquoteleft d) ?n))}
\text{?N = (1+ (1+ (1+ (1+ 0)))));}
\text{No.}
\text{> (?- (length ?list (1+ (1+ 0))))}
\text{?LIST = (?X3869 ?X3872);}
\text{No.}
\end{align*}
\]
The next two queries show the two lists of length two with a as a member. Both queries give the correct answer, a two-element list that either starts or ends with a. However, the behavior after generating these two solutions is quite different.

```
> (?- (length ?list ?n))
?list = NIL
?n = 0;
?list = (?x3918)
?n = (1+ 0);
?list = (?x3918 ?x3921)
?n = (1+ (1+ 0)).
No.
```

In the first query, length only generates one possible solution, the list with two unbound elements. member takes this solution and instantiates either the first or the second element to a.

In the second query, member keeps generating potential solutions. The first two partial solutions, where a is the first or second member of a list of unknown length, are extended by length to yield the solutions where the list has length two. After that, member keeps generating longer and longer lists, which length keeps rejecting. It is implicit in the definition of member that subsequent solutions will be longer, but because that is not explicitly known, they are all generated anyway and then explicitly tested and rejected by length.

This example reveals the limitations of Prolog as a pure logic-programming language. It turns out the user must be concerned not only about the logic of the problem but also with the flow of control. Prolog is smart enough to backtrack and find all solutions when the search space is small enough, but when it is infinite (or even very large), the programmer still has a responsibility to guide the flow of control. It is possible to devise languages that do much more in terms of automatic flow of control. Prolog is a convenient and efficient middle ground between imperative languages and pure logic.

---

4 See the MU-Prolog and NU-Prolog languages (Naish 1986).
Approaches to Backtracking

Suppose you are asked to make a "small" change to an existing program. The problem is that some function, f, which was thought to be single-valued, is now known to return two or more valid answers in certain circumstances. In other words, f is nondeterministic. (Perhaps f is sqrt, and we now want to deal with negative numbers). What are your alternatives as a programmer? Five possibilities can be identified:

- **Guess.** Choose one possibility and discard the others. This requires a means of making the right guesses, or recovering from wrong guesses.

- **Know.** Sometimes you can provide additional information that is enough to decide what the right choice is. This means changing the calling function(s) to provide the additional information.

- **Return a list.** This means that the calling function(s) must be changed to expect a list of replies.

- **Return a pipe, as defined in section 9.3.** Again, the calling function(s) must be changed to expect a pipe.

- **Guess and save.** Choose one possibility and return it, but record enough information to allow computing the other possibilities later. This requires saving the current state of the computation as well as some information on the remaining possibilities.

The last alternative is the most desirable. It is efficient, because it doesn't require computing answers that are never used. It is unobtrusive, because it doesn't require changing the calling function (and the calling function's calling function) to expect a list or pipe of answers. Unfortunately, it does have one major difficulty: there has to be a way of packaging up the current state of the computation and saving it away so that it can be returned to when the first choice does not work. For our Prolog interpreter, the current state is succinctly represented as a list of goals. In other problems, it is not so easy to summarize the entire state.

We will see in section 22.4 that the Scheme dialect of Lisp provides a function, call-with-current-continuation, that does exactly what we want: it packages the current state of the computation into a function, which can be stored away and invoked later. Unfortunately, there is no corresponding function in Common Lisp.

Anonymous Variables

Before moving on, it is useful to introduce the notion of an anonymous variable. This is a variable that is distinct from all others in a clause or query, but which the
programmer does not want to bother to name. In real Prolog, the underscore is used for anonymous variables, but we will use a single question mark. The definition of member that follows uses anonymous variables for positions within terms that are not needed within a clause:

\[
(<- \text{(member } ?\text{item } (?\text{item . ?)}))
\]
\[
(<- \text{(member } ?\text{item } (? . ?\text{rest})) \text{(member } ?\text{item } ?\text{rest}))
\]

However, we also want to allow several anonymous variables in a clause but still be able to keep each anonymous variable distinct from all other variables. One way to do that is to replace each anonymous variable with a unique variable. The function replace-?-vars uses gensym to do just that. It is installed in the top-level macros <- and ?- so that all clauses and queries get the proper treatment.

\[
\text{(defmacro } \text{<- (&rest } \text{clause)}
\]
\[
\text{"Add a clause to the data base."}
\]
\[
\text{ '(add-clause ',(replace-?-vars clause)))
\]
\[
\text{(defmacro } ?- \text{ (&rest } \text{goals})
\]
\[
\text{"Make a query and print answers."}
\]
\[
\text{ '(top-level-prove ',(replace-?-vars goals)))
\]
\[
\text{(defun replace-?-vars (exp)}
\]
\[
\text{"Replace any ? within exp with a var of the form ?123."}
\]
\[
\text{ (cond ((eq exp '?) (gensym '?'))}
\]
\[
\text{ ((atom exp) exp)}
\]
\[
\text{ (t (reuse-cons (replace-?-vars (first exp))}
\]
\[
\text{ (replace-?-vars (rest exp))}
\]
\[
\text{ exp))\})
\]

A named variable that is used only once in a clause can also be considered an anonymous variable. This is addressed in a different way in section 12.3.

### 11.4 The Zebra Puzzle

Here is an example of something Prolog is very good at: a logic puzzle. There are fifteen facts, or constraints, in the puzzle:

1. There are five houses in a line, each with an owner, a pet, a cigarette, a drink, and a color.

2. The Englishman lives in the red house.

3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.

5. The Ukrainian drinks tea.

6. The green house is immediately to the right of the ivory house.

7. The Winston smoker owns snails.

8. Kools are smoked in the yellow house.

9. Milk is drunk in the middle house.

10. The Norwegian lives in the first house on the left.

11. The man who smokes Chesterfields lives next to the man with the fox.

12. Kools are smoked in the house next to the house with the horse.

13. The Lucky Strike smoker drinks orange juice.


15. The Norwegian lives next to the blue house.

The questions to be answered are: who drinks water and who owns the zebra? To solve this puzzle, we first define the relations nextto (for “next to”) and right (for “immediately to the right of”). They are closely related to member, which is repeated here.

\[
\begin{align*}
&\text{(member ?x ?y ?rest))} \\
&\text{(- (member ?x ?rest))} \\
&\text{(- (nextto ?x ?y ?rest))} \\
&\text{(- (right ?x ?y ?rest))} \\
&\text{(- (right ?left ?right ?rest))} \\
&\text{(- (= ?x ?y))}
\end{align*}
\]

We also defined the identity relation, =. It has a single clause that says that any x is equal to itself. One might think that this implements eq or equal. Actually, since Prolog uses unification to see if the two arguments of a goal each unify with ?x, this means that = is unification.

Now we are ready to define the zebra puzzle with a single (long) clause. The variable ?x represents the list of five houses, and each house is represented by a term of the form (house nationality pet cigarette drink color). The variable ?w is the water drinker, and ?z is the zebra owner. Each of the 15 constraints in the puzzle is listed
in the body of zebra, although constraints 9 and 10 have been combined into the first one. Consider constraint 2, "The Englishman lives in the red house." This is interpreted as "there is a house whose nationality is Englishman and whose color is red, and which is a member of the list of houses": in other words, (member (house englishman ?? red) ?h). The other constraints are similarly straightforward.

(<= (zebra ?h ?w ?z)
  ;; Each house is of the form:
  ;; (house nationality pet cigarette drink house-color)
  (= ?h ((house norwegian ?? ??) ;1,10
  ?
   (house ? ? milk ??) ??)) ;9
  (member (house englishman ?? red) ?h) ;2
  (member (house spaniard dog ?? ??) ?h) ;3
  (member (house ?? coffee green) ?h) ;4
  (member (house ukrainian ?? tea) ?h) ;5
  (t fright (house ?? ?? ivory)
   ;6
   (house ?? ?? green) ?h))
  (member (house snails winston ??) ?h) ;7
  (member (house ?? kools ?? yellow) ?h) ;8
  (nextto (house ?? chesterfield ??)
   ;11
   (house fox ?? ??) ?h)
  (nextto (house ?? kools ??)
  ;12
   (house horse ?? ??) ?h)
  (member (house ?? luckystrike orange-juice ??) ?h) ;13
  (member (house japanese ?? parliaments ??) ?h) ;14
  (nextto (house norwegian ?? ??)
  ;15
   (house ?? ?? blue) ?h))
  ;; Now for the questions:
  (member (house ?w ?? water ??) ?h) ;01
  (member (house ?? zebra ?? ??) ?h)) ;02

Here's the query and solution to the puzzle:

> (?- (zebra ?houses ?water-drinker ?zebra-owner))

?HOUSES = ((HOUSE NORWEGIAN FOX KOOLS WATER YELLOW)
  (HOUSE UKRAINIAN HORSE CHESTERFIELD TEA BLUE)
  (HOUSE ENGLISHMAN SNAILS WINSTON MILK RED)
  (HOUSE SPANIARD DOG LUCKYSTRIKE ORANGE-JUICE IVORY)
  (HOUSE JAPANESE ZEBRA PARLIAMS COFFEE GREEN))

?WATER-DRINKER = NORWEGIAN
?ZEBRA-OWNER = JAPANESE.

No.

This took 278 seconds, and profiling (see page 288) reveals that the function prove was called 12,825 times. A call to prove has been termed a logical inference, so our system
is performing $12825/278 = 46$ logical inferences per second, or LIPS. Good Prolog systems perform at 10,000 to 100,000 LIPS or more, so this is barely limping along.

Small changes to the problem can greatly affect the search time. For example, the relation next to holds when the first house is immediately right of the second, or when the second is immediately right of the first. It is arbitrary in which order these clauses are listed, and one might think it would make no difference in which order they were listed. In fact, if we reverse the order of these two clauses, the execution time is roughly cut in half.

11.5 The Synergy of Backtracking and Unification

Prolog's backward chaining with backtracking is a powerful technique for generating the possible solutions to a problem. It makes it easy to implement a generate-and-test strategy, where possible solutions are considered one at a time, and when a candidate solution is rejected, the next is suggested. But generate-and-test is only feasible when the space of possible solutions is small.

In the zebra puzzle, there are five attributes for each of the five houses. Thus there are $5^5$, or over 24 billion candidate solutions, far too many to test one at a time. It is the concept of unification (with the corresponding notion of a logic variable) that makes generate-and-test feasible on this puzzle. Instead of enumerating complete candidate solutions, unification allows us to specify partial candidates. We start out knowing that there are five houses, with the Norwegian living on the far left and the milk drinker in the middle. Rather than generating all complete candidates that satisfy these two constraints, we leave the remaining information vague, by unifying the remaining houses and attributes with anonymous logic variables. The next constraint (number 2) places the Englishman in the red house. Because of the way member is written, this first tries to place the Englishman in the leftmost house. This is rejected, because Englishman and Norwegian fail to unify, so the next possibility is considered, and the Englishman is placed in the second house. But no other features of the second house are specified—we didn’t have to make separate guesses for the Englishman’s house being green, yellow, and so forth. The search continues, filling in only as much as is necessary and backing up whenever a unification fails.

For this problem, unification serves the same purpose as the delay macro (page 281). It allows us to delay deciding the value of some attribute as long as possible, but to immediately reject a solution that tries to give two different values to the same attribute. That way, we save time if we end up backtracking before the computation is made, but we are still able to fill in the value later on.

It is possible to extend unification so that it is doing more work, and backtracking is doing less work. Consider the following computation:
(?- (length ?1 4)
  (member d ?1) (member a ?1) (member c ?1) (member b ?1)
  (= ?1 (a b c d)))

The first two lines generate permutations of the list (d a c b), and the third line tests for a permutation equal to (a b c d). Most of the work is done by backtracking. An alternative is to extend unification to deal with lists, as well as constants and variables. Predicates like length and member would be primitives that would have to know about the representation of lists. Then the first two lines of the above program would set ?1 to something like #s(list :length 4 :members (d a c d)). The third line would be a call to the extended unification procedure, which would further specify ?1 to be something like:

#s(list :length 4 :members (d a c d) :order (a b c d))

By making the unification procedure more complex, we eliminate the need for backtracking entirely.

Exercise 11.3 [s] Would a unification algorithm that delayed member tests be a good idea or a bad idea for the zebra puzzle?

11.6 Destructive Unification

As we saw in section 11.2, keeping track of a binding list of variables is a little tricky. It is also prone to inefficiency if the binding list grows large, because the list must be searched linearly, and because space must be allocated to hold the binding list. An alternative implementation is to change unify to a destructive operation. In this approach, there are no binding lists. Instead, each variable is represented as a structure that includes a field for its binding. When the variable is unified with another expression, the variable’s binding field is modified to point to the expression. Such variables will be called vars to distinguish them from the implementation of variables as symbols starting with a question mark. vars are defined with the following code:

(defconstant unbound "Unbound")
(defstruct var name (binding unbound))
(defun bound-p (var) (not (eq (var-binding var) unbound)))

The macro deref gets at the binding of a variable, returning its argument when it is an
unbound variable or a nonvariable expression. It includes a loop because a variable can be bound to another variable, which in turn is bound to the ultimate value.

Normally, it would be considered bad practice to implement `deref` as a macro, since it could be implemented as an inline function, provided the caller was willing to write `(setq x (deref x))` instead of `(deref x)`. However, `deref` will appear in code generated by some versions of the Prolog compiler that will be presented in the next section. Therefore, to make the generated code look neater, I have allowed myself the luxury of the `deref` macro.

```lisp
(defmacro deref (exp)
  "Follow pointers for bound variables."
  '(progn (loop while (and (var-p ,exp) (bound-p ,exp))
              do (setq ,exp (var-binding ,exp)))
       ,exp))
```

The function `unify!` below is the destructive version of `unify`. It is a predicate that returns true for success and false for failure, and has the side effect of altering variable bindings.

```lisp
(defun unify! (x y)
  "Destructively unify two expressions"
  (cond ((eql (deref x) (deref y)) t)
        ((var-p x) (set-binding! x y))
        ((var-p y) (set-binding! y x))
        ((and (consp x) (consp y))
         (and (unify! (first x) (first y))
              (unify! (rest x) (rest y))))
        (t nil)))

(defun set-binding! (var value)
  "Set var's binding to value. Always succeeds (returns t)."
  (setf (var-binding var) value)
  t)
```

To make vars easier to read, we can install a `:print-function`:

```lisp
(defun print-var (var stream depth)
  (if (or (and (numberp *print-level*)
               (= depth *print-level*))
          (var-p (deref var)))
      (format stream "?-a" (var-name var))
      (write var :stream stream)))
```

```lisp
(defun print-function (print-var)
  (defstruct (var (:print-function print-var))
    (name (binding unbound)))
  (defun print-var (var stream depth)
    (if (or (and (numberp *print-level*)
                 (= depth *print-level*))
            (var-p (deref var)))
        (format stream "?-a" (var-name var))
        (write var :stream stream)))
```

```lisp
(defun print-function (print-var)
  (defstruct (var (:print-function print-var))
    (name (binding unbound)))
  (defun print-var (var stream depth)
    (if (or (and (numberp *print-level*)
                 (= depth *print-level*))
            (var-p (deref var)))
        (format stream "?-a" (var-name var))
        (write var :stream stream)))
```

```lisp
(defun print-function (print-var)
  (defstruct (var (:print-function print-var))
    (name (binding unbound)))
  (defun print-var (var stream depth)
    (if (or (and (numberp *print-level*)
                 (= depth *print-level*))
            (var-p (deref var)))
        (format stream "?-a" (var-name var))
        (write var :stream stream)))
This is the first example of a carefully crafted print-function. There are three things to notice about it. First, it explicitly writes to the stream passed as the argument. It does not write to a default stream. Second, it checks the variable depth against *print-level*, and prints just the variable name when the depth is exceeded. Third, it uses write to print the bindings. This is because write pays attention to the current values of *print-escape*, *print-pretty*, and so on. Other printing functions such as print or print do not pay attention to these variables.

Now, for backtracking purposes, we want to make set-binding! keep track of the bindings that were made, so they can be undone later:

```lisp
(defun *trail* (make-array 200 :fill-pointer 0 :adjustable t))

(defun set-binding! (var value)
  "Set var's binding to value, after saving the variable in the trail. Always returns t."
  (unless (eq var value)
    (vector-push-extend var *trail*)
    (setf (var-binding var) value))
  t)

(defun undo-bindings! (old-trail)
  "Undo all bindings back to a given point in the trail."
  (loop until (= (fill-pointer *trail*) old-trail)
    do (setf (var-binding (vector-pop *trail*)) unbound)))
```

Now we need a way of making new variables, where each one is distinct. That could be done by gensym-ing a new name for each variable, but a quicker solution is just to increment a counter. The constructor function var is defined to generate a new variable with a name that is a new integer. This is not strictly necessary; we could have just used the automatically provided constructor make-var. However, I thought that the operation of providing new anonymous variable was different enough from providing a named variable that it deserved its own function. Besides, make-var may be less efficient, because it has to process the keyword arguments. The function var has no arguments; it just assigns the default values specified in the slots of the var structure.

```lisp
(defvar *var-counter* 0)

(defun make-var ()
  (defstruct var (:constructor ? ()
                (:print-function print-var)
                (name (incf *var-counter*))
                (binding unbound)))
```

A reasonable next step would be to use destructive unification to make a more efficient interpreter. This is left as an exercise, however, and instead we put the interpreter aside, and in the next chapter develop a compiler.
11.7 Prolog in Prolog

As stated at the start of this chapter, Prolog has many of the same features that make Lisp attractive for program development. Just as it is easy to write a Lisp interpreter in Lisp, it is easy to write a Prolog interpreter in Prolog. The following Prolog metainterpreter has three main relations. The relation clause is used to store clauses that make up the rules and facts that are to be interpreted. The relation prove is used to prove a goal. It calls prove-all, which attempts to prove a list of goals. prove-all succeeds in two ways: (1) if the list is empty, or (2) if there is some clause whose head matches the first goal, and if we can prove the body of that clause, followed by the remaining goals:

\[
\begin{align*}
&\langle- \text{ (prove } \text{ ?goal) (prove-all } \text{ (?goal)))} \\
&\langle- \text{ (prove-all } \text{ nil}) \\
&\langle- \text{ (prove-all } \text{ (?goal . ?goals))} \\
&\quad \text{ (clause } \langle- \text{ ?goal . ?body}) \\
&\quad \text{ (concat ?body ?goals ?new-goals) } \\
&\quad \text{ (prove-all } \text{ ?new-goals))}
\end{align*}
\]

Now we add two clauses to the database to define the member relation:

\[
\begin{align*}
&\langle- \text{ (clause } \langle- \text{ (mem } \text{ ?x (} ?x \text{ . } ?y))}) \\
&\langle- \text{ (clause } \langle- \text{ (mem } \text{ ?x (} ?x \text{ . } ?z)) (\text{mem } ?x ?z))))
\end{align*}
\]

Finally, we can prove a goal using our interpreter:

\[
\begin{align*}
&?- \text{ (prove (mem } ?x \text{ (}1 \text{ 2 3))}) \\
&?X = 1; \\
&?X = 2; \\
&?X = 3; \\
&\text{No.}
\end{align*}
\]

11.8 Prolog Compared to Lisp

Many of the features that make Prolog a successful language for AI (and for program development in general) are the same as Lisp's features. Let's reconsider the list of features that make Lisp different from conventional languages (see page 25) and see what Prolog has to offer:
• **Built-in Support for Lists (and other data types).** New data types can be created easily using lists or structures (structures are preferred). Support for reading, printing, and accessing components is provided automatically. Numbers, symbols, and characters are also supported. However, because logic variables cannot be altered, certain data structures and operations are not provided. For example, there is no way to update an element of a vector in Prolog.

• **Automatic Storage Management.** The programmer can allocate new objects without worrying about reclaiming them. Reclaiming is usually faster in Prolog than in Lisp, because most data can be stack-allocated instead of heap-allocated.

• **Dynamic Typing.** Declarations are not required. Indeed, there is no standard way to make type declarations, although some implementations allow for them. Some Prolog systems provide only fixnums, so that eliminates the need for a large class of declarations.

• **First-Class Functions.** Prolog has no equivalent of lambda, but the built-in predicate call allows a term—a piece of data—to be called as a goal. Although backtracking choice points are not first-class objects, they can be used in a way very similar to continuations in Lisp.

• **Uniform Syntax.** Like Lisp, Prolog has a uniform syntax for both programs and data. This makes it easy to write interpreters and compilers in Prolog. While Lisp's prefix-operator list notation is more uniform, Prolog allows infix and postfix operators, which may be more natural for some applications.

• **Interactive Environment.** Expressions can be immediately evaluated. High-quality Prolog systems offer both a compiler and interpreter, along with a host of debugging tools.

• **Extensibility.** Prolog syntax is extensible. Because programs and data share the same format, it is possible to write the equivalent of macros in Prolog and to define embedded languages. However, it can be harder to ensure that the resulting code will be compiled efficiently. The details of Prolog compilation are implementation-dependent.

To put things in perspective, consider that Lisp is at once one of the highest-level languages available and a universal assembly language. It is a high-level language because it can easily capture data, functional, and control abstractions. It is a good assembly language because it is possible to write Lisp in a style that directly reflects the operations available on modern computers.

Prolog is generally not as efficient as an assembly language, but it can be more concise as a specification language, at least for some problems. The user writes specifications: lists of axioms that describe the relationships that can hold in the problem domain. If these specifications are in the right form, Prolog's automatic
backtracking can find a solution, even though the programmer does not provide an explicit algorithm. For other problems, the search space will be too large or infinite, or Prolog’s simple depth-first search with backup will be too inflexible. In this case, Prolog must be used as a programming language rather than a specification language. The programmer must be aware of Prolog’s search strategy, using it to implement an appropriate algorithm for the problem at hand.

Prolog, like Lisp, has suffered unfairly from some common myths. It has been thought to be an inefficient language because early implementations were interpreted, and because it has been used to write interpreters. But modern compiled Prolog can be quite efficient (see Warren et al. 1977 and Van Roy 1990). There is a temptation to see Prolog as a solution in itself rather than as a programming language. Those who take that view object that Prolog’s depth-first search strategy and basis in predicate calculus is too inflexible. This objection is countered by Prolog programmers who use the facilities provided by the language to build more powerful search strategies and representations, just as one would do in Lisp or any other language.

11.9 History and References

Cordell Green (1968) was the first to articulate the view that mathematical results on theorem proving could be used to make deductions and thereby answer queries. However, the major technique in use at the time, resolution theorem proving (see Robinson 1965), did not adequately constrain search, and thus was not practical. The idea of goal-directed computing was developed in Carl Hewitt’s work (1971) on the PLANNER language for robot problem solving. He suggested that the user provide explicit hints on how to control deduction.

At about the same time and independently, Alain Colmerauer was developing a system to perform natural language analysis. His approach was to weaken the logical language so that computationally complex statements (such as logical disjunctions) could not be made. Colmerauer and his group implemented the first Prolog interpreter using Algol-W in the summer of 1972 (see Roussel 1975). It was Roussel’s wife, Jacqueline, who came up with the name Prolog as an abbreviation for “programmation en logique.” The first large Prolog program was their natural language system, also completed that year (Colmerauer et al. 1973). For those who read English better than French, Colmerauer (1985) presents an overview of Prolog. Robert Kowalski is generally considered the co-inventor of Prolog. His 1974 article outlines his approach, and his 1988 article is a historical review on the early logic programming work.

There are now dozens of text books on Prolog. In my mind, six of these stand out. Clocksin and Mellish’s Programming in Prolog (1987) was the first and remains one of the best. Sterling and Shapiro’s The Art of Prolog (1986) has more substantial examples but is not as complete as a reference. An excellent overview from a slightly
more mathematical perspective is Pereira and Shieber’s *Prolog and Natural-Language Analysis* (1987). The book is worthwhile for its coverage of Prolog alone, and it also provides a good introduction to the use of logic programming for language understanding (see part V for more on this subject). O’Keefe’s *The Craft of Prolog* (1990) shows a number of advanced techniques. O’Keefe is certainly one of the most influential voices in the Prolog community. He has definite views on what makes for good and bad coding style and is not shy about sharing his opinions. The reader is warned that this book evolved from a set of notes on the Clocksin and Mellish book, and the lack of organization shows in places. However, it contains advanced material that can be found nowhere else. Another collection of notes that has been organized into a book is Coelho and Cotta’s *Prolog by Example*. Published in 1988, this is an update of their 1980 book, *How to Solve it in Prolog*. The earlier book was an underground classic in the field, serving to educate a generation of Prolog programmers. Both versions include a wealth of examples, unfortunately with little documentation and many typos. Finally, Ivan Bratko’s *Prolog Programming for Artificial Intelligence* (1990) covers some introductory AI material from the Prolog perspective.

Maier and Warren’s *Computing with Logic* (1988) is the best reference for those interested in implementing Prolog. It starts with a simple interpreter for a variable-free version of Prolog, and then moves up to the full language, adding improvements to the interpreter along the way. (Note that the second author, David S. Warren of Stonybrook, is different from David H. D. Warren, formerly at Edinburgh and now at Bristol. Both are experts on Prolog.)


There have been many attempts to extend Prolog to be closer to the ideal of Logic Programming. The language MU-Prolog and NU-Prolog (Naish 1986) and Prolog III (Colmerauer 1990) are particularly interesting. The latter includes a systematic treatment of the $\neq$ relation and an interpretation of infinite trees.

### 11.10 Exercises

> **Exercise 11.4 [m]** It is somewhat confusing to see “no” printed after one or more valid answers have appeared. Modify the program to print “no” only when there are no answers at all, and “no more” in other cases.

> **Exercise 11.5 [h]** At least six books (Abelson and Sussman 1985, Charniak and McDermott 1985, Charniak et al. 1986, Hennessy 1989, Wilensky 1986, and Winston and Horn 1988) present unification algorithms with a common error. They all have problems unifying $(?x ?y a)$ with $(?y ?x ?x)$. Some of these texts assume that unify
will be called in a context where no variables are shared between the two arguments. However, they are still suspect to the bug, as the following example points out:

\[
> \text{unify } \left( (f \ ?x \ ?y \ a) \ (\forall \ ?x \ ?x) \right) \ (f \ ?z \ ?z)
\]
\[
\left( (\forall \ ?y \ . \ a) \ (\forall \ ?x \ . \ ?y) \ (\forall \ ?z \ ?x \ ?A) \right)
\]

Despite this subtle bug, I highly recommend each of the books to the reader. It is interesting to compare different implementations of the same algorithm. It turns out there are more similarities than differences. This indicates two things: (1) there is a generally agreed-upon style for writing these functions, and (2) good programmers sometimes take advantage of opportunities to look at other’s code.

The question is: Can you give an informal proof of the correctness of the algorithm presented in this chapter? Start by making a clear statement of the specification. Apply that to the other algorithms, and show where they go wrong. Then see if you can prove that the unify function in this chapter is correct. Failing a complete proof, can you at least prove that the algorithm will always terminate? See Norvig 1991 for more on this problem.

exercise 11.6 [h] Since logic variables are so basic to Prolog, we would like them to be efficient. In most implementations, structures are not the best choice for small objects. Note that variables only have two slots: the name and the binding. The binding is crucial, but the name is only needed for printing and is arbitrary for most variables. This suggests an alternative implementation. Each variable will be a cons cell of the variable’s binding and an arbitrary marker to indicate the type. This marker would be checked by variable-p. Variable names can be stored in a hash table that is cleared before each query. Implement this representation for variables and compare it to the structure representation.

Exercise 11.7 [m] Consider the following alternative implementation for anonymous variables: Leave the macros <- and ?- alone, so that anonymous variables are allowed in assertions and queries. Instead, change unify so that it lets anything match against an anonymous variable:

```lisp
(defun unify (x y &optional (bindings no-bindings))
  "See if x and y match with given bindings."
  (cond ((eq bindings fail) fail)
        ((eq1 x y) bindings)
        ((or (eq x '?) (eq 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)))

Is this alternative correct? If so, give an informal proof. If not, give a counterexample.

Exercise 11.8 [h] Write a version of the Prolog interpreter that uses destructive unification instead of binding lists.

Exercise 11.9 [m] Write Prolog rules to express the terms father, mother, son, daughter, and grand-versions of each of them. Also define parent, child, wife, husband, brother, sister, uncle, and aunt. You will need to decide which relations are primitive (stored in the Prolog database) and which are derived by rules.

For example, here's a definition of grandfather that says that G is the grandfather of C if G is the father of some P, who is the parent of C:

(\- (grandfather ?g ?c)
 (father ?g ?p)
 (parent ?p ?c))

Exercise 11.10 [m] The following problem is presented in Wirth 1976:

I married a widow (let's call her W) who has a grown-up daughter (call her D). My father (F), who visited us often, fell in love with my step-daughter and married her. Hence my father became my son-in-law and my step-daughter became my mother. Some months later, my wife gave birth to a son (S1), who became the brother-in-law of my father, as well as my uncle. The wife of my father, that is, my step-daughter, also had a son (S2).

Represent this situation using the predicates defined in the previous exercise, verify its conclusions, and prove that the narrator of this tale is his own grandfather.

Exercise 11.11 [d] Recall the example:

?- (length (a b c d) ?n))
\n?n = (1+ (1+ (1+ (1+ 0))) ;

It is possible to produce 4 instead of (1+ (1+ (1+ (1+ 0))) by extending the notion of unification. Ait-Kaci et al. 1987 might give you some ideas how to do this.
Exercise 11.12 [h] The function rename-variables was necessary to avoid confusion between the variables in the first argument to unify and those in the second argument. An alternative is to change the unify so that it takes two binding lists, one for each argument, and keeps them separate. Implement this alternative.

11.11 Answers

Answer 11.9 We will choose as primitives the unary predicates male and female and the binary predicates child and married. The former takes the child first; the latter takes the husband first. Given these primitives, we can make the following definitions:

```prolog
(- (father ?f ?c) (male ?f) (parent ?f ?c))
(- (mother ?m ?c) (female ?m) (parent ?m ?c))
(- (son ?s ?p) (male ?s) (parent ?p ?s))
(- (daughter ?s ?p) (male ?s) (parent ?p ?s))
(- (grandfather ?g ?c) (father ?g ?p) (parent ?p ?c))
(- (grandmother ?g ?c) (mother ?g ?p) (parent ?p ?c))
(- (wife ?w ?h) (married ?h ?w))
(- (husband ?h ?w) (married ?h ?w))
(- (brother ?b ?x) (male ?b) (sibling ?b ?x))
(- (sister ?s ?x) (female ?s) (sibling ?s ?x))
```

Note that there is no way in Prolog to express a true definition. We would like to say that "P is the parent of C if and only if C is the child of P," but Prolog makes us express the biconditional in one direction only.
Answer 11.10

Because we haven't considered step-relations in the prior definitions, we have to extend the notion of parent to include step-parents. The definitions have to be written very carefully to avoid infinite loops. The strategy is to structure the defined terms into a strict hierarchy: the four primitives are at the bottom, then parent is defined in terms of the primitives, then the other terms are defined in terms of parent and the primitives.

We also provide a definition for son-in-law:

\[
\begin{align*}
\end{align*}
\]

Now we add the information from the story. Note that we only use the four primitives male, female, married, and child:

\[
\begin{align*}
&\text{(male I)} \quad \text{(male F)} \quad \text{(married I W)} \quad \text{(married F D)} \\
&\text{(female W)} \quad \text{(female D)} \quad \text{(child D W)} \quad \text{(child I F)} \\
&\text{(child SI I)} \quad \text{(child S2 F)}
\end{align*}
\]

Now we are ready to make the queries:

\[
\begin{align*}
&\text{(?- (son-in-law F I))} \\
&\text{Yes.} \\
&\text{(?- (mother D I))} \\
&\text{Yes.} \\
&\text{(?- (uncle SI I))} \\
&\text{Yes.} \\
&\text{(?- (grandfather I I))} \\
&\text{Yes.}
\end{align*}
\]