

CHAPTER 15

Symbolic Mathematics with Canonical Forms

Anything simple always interests me.

—David Hockney

Chapter 8 started with high hopes: to take an existing pattern matcher, copy down some mathematical identities out of a reference book, and come up with a usable symbolic algebra system. The resulting system *was* usable for some purposes, and it showed that the technique of rule-based translation is a powerful one. However, the problems of section 8.5 show that not everything can be done easily and efficiently within the rule-based pattern matching framework.

There are important mathematical transformations that are difficult to express in the rule-based approach. For example, dividing two polynomials to obtain a quotient and remainder is a task that is easier to express as an algorithm—a program—than as a rule or set of rules.

In addition, there is a problem with efficiency. Pieces of the input expressions are simplified over and over again, and much time is spent interpreting rules that do not apply. Section 9.6 showed some techniques for speeding up the program by a factor of 100 on inputs of a dozen or so symbols, but for expressions with a hundred or so symbols, the speed-up is not enough. We can do better by designing a specialized representation from the ground up.

Serious algebraic manipulation programs generally enforce a notion of *canonical simplification*. That is, expressions are converted into a canonical internal format that may be far removed from the input form. They are then manipulated, and translated back to external form for output. Of course, the simplifier we have already does this kind of translation, to some degree. It translates $(3 + x + -3 + y)$ into $(+ x y)$ internally, and then outputs it as $(x + y)$. But a *canonical* representation must have the property that any two expressions that are equal have identical canonical forms. In our system the expression $(5 + y + x + -5)$ is translated to the internal form $(+ y x)$, which is not identical to $(+ x y)$, even though the two expressions are equal. Thus, our system is not canonical. Most of the problems of the previous section stem from the lack of a canonical form.

Adhering to canonical form imposes grave restrictions on the representation. For example, $x^2 - 1$ and $(x - 1)(x + 1)$ are equal, so they must be represented identically. One way to insure this is to multiply out all factors and collect similar terms. So $(x - 1)(x + 1)$ is $x^2 - x + x - 1$, which simplifies to $x^2 - 1$, in whatever the canonical internal form is. This approach works fine for $x^2 - 1$, but for an expression like $(x - 1)^{1000}$, multiplying out all factors would be quite time- (and space-) consuming. It is hard to find a canonical form that is ideal for all problems. The best we can do is choose one that works well for the problems we are most likely to encounter.

15.1 A Canonical Form for Polynomials

This section will concentrate on a canonical form for *polynomials*. Mathematically speaking, a polynomial is a function (of one or more variables) that can be computed using only addition and multiplication. We will speak of a polynomial's *main variable*, *coefficients*, and *degree*. In the polynomial:

$$5 \times x^3 + b \times x^2 + c \times x + 1$$

the main variable is x , the degree is 3 (the highest power of x), and the coefficients are 5, b , c and 1. We can define an input format for polynomials as follows:

1. Any Lisp number is a polynomial.
2. Any Lisp symbol is a polynomial.

3. If p and q are polynomials, so are $(p + q)$ and $(p * q)$.
4. If p is a polynomial and n is a positive integer, then $(p \wedge n)$ is a polynomial.

However, the input format cannot be used as the canonical form, because it would admit both $(x + y)$ and $(y + x)$, and both 4 and $(2 + 2)$.

Before considering a canonical form for polynomials, let us see why polynomials were chosen as the target domain. First, the volume of programming needed to support canonical forms for a larger class of expressions grows substantially. To make things easier, we have eliminated complications like log and trig functions. Polynomials are a good choice because they are closed under addition and multiplication: the sum or product of any two polynomials is a polynomial. If we had allowed division, the result would not be closed, because the quotient of two polynomials need not be a polynomial. As a bonus, polynomials are also closed under differentiation and integration, so we can include those operators as well.

Second, for sufficiently large classes of expressions it becomes not just difficult but impossible to define a canonical form. This may be surprising, and we don't have space here to explain exactly why it is so, but here is an argument: Consider what would happen if we added enough functionality to duplicate all of Lisp. Then "converting to canonical form" would be the same as "running a program." But it is an elementary result of computability theory that it is in general impossible to determine the result of running an arbitrary program (this is known as the halting problem). Thus, it is not surprising that it is impossible to canonicalize complex expressions.

Our task is to convert a polynomial as previously defined into some canonical form.¹ Much of the code and some of the commentary on this format and the routines to manipulate it was written by Richard Fateman, with some enhancements made by Peter Klier.

The first design decision is to assume that we will be dealing mostly with *dense* polynomials, rather than *sparse* ones. That is, we expect most of the polynomials to be like $ax^3 + bx^2 + cx + d$, not like $ax^{100} + bx^{50} + c$. For dense polynomials, we can save space by representing the main variable (x in these examples) and the individual coefficients (a, b, c , and d in these examples) explicitly, but representing the exponents only implicitly, by position. Vectors will be used instead of lists, to save space and to allow fast access to any element. Thus, the representation of $5x^3 + 10x^2 + 20x + 30$ will be the vector:

```
 #(x 30 20 10 5)
```

¹In fact, the algebraic properties of polynomial arithmetic and its generalizations fit so well with ideas in data abstraction that an extended example (in Scheme) on this topic is provided in *Structure and Interpretation of Computer Programs* by Abelson and Sussman (see section 2.4.3, pages 153–166). We'll pursue a slightly different approach here.

The main variable, x , is in the 0th element of the vector, and the coefficient of the i th power of x is in element $i + 1$ of the vector. A single variable is represented as a vector whose first coefficient is 1, and a number is represented as itself:

```
#(x 30 20 10 5) represents  $5x^3 + 10x^2 + 20x + 30$ 
#(x 0 1)         represents  $x$ 
5                represents 5
```

The fact that a number is represented as itself is a possible source of confusion. The number 5, for example, is a polynomial by our mathematical definition of polynomials. But it is represented as 5, not as a vector, so (typep 5 'polynomial) will be false. The word “polynomial” is used ambiguously to refer to both the mathematical concept and the Lisp type, but it should be clear from context which is meant.

A glossary for the canonical simplifier program is given in figure 15.1.

The functions defining the type polynomial follow. Because we are concerned with efficiency, we proclaim certain short functions to be compiled inline, use the specific function svref (simple-vector reference) rather than the more general aref, and provide declarations for the polynomials using the special form the. More details on efficiency issues are given in Chapter 9.

```
(proclaim '(inline main-var degree coef
           var= var> poly make-poly))

(deftype polynomial () 'simple-vector)

(defun main-var (p) (svref (the polynomial p) 0))
(defun coef (p i)  (svref (the polynomial p) (+ i 1)))
(defun degree (p)  (- (length (the polynomial p)) 2))
```

We had to make another design decision in defining coef, the function to extract a coefficient from a polynomial. As stated above, the i th coefficient of a polynomial is in element $i + 1$ of the vector. If we required the caller of coef to pass in $i + 1$ to get i , we might be able to save a few addition operations. The design decision was that this would be too confusing and error prone. Thus, coef expects to be passed i and does the addition itself.

For our format, we will insist that main variables be symbols, while coefficients can be numbers or other polynomials. A “production” version of the program might have to account for main variables like (sin x), as well as other complications like + and * with more than two arguments, and noninteger powers.

Now we can extract information from a polynomial, but we also need to build and modify polynomials. The function poly takes a variable and some coefficients and builds a vector representing the polynomial. make-poly takes a variable and a degree and produces a polynomial with all zero coefficients.

	Top-Level Functions
canon-simplifier	A read-canonicalize-print loop.
canon	Canonicalize argument and convert it back to infix.
	Data Types
polynomial	A vector of main variable and coefficients.
	Major Functions
prefix->canon	Convert a prefix expression to canonical polynomial.
canon->prefix	Convert a canonical polynomial to a prefix expression.
poly+poly	Add two polynomials.
poly*poly	Multiply two polynomials.
poly^n	Raise polynomial p to the nth power, $n \geq 0$.
deriv-poly	Return the derivative, dp/dx , of the polynomial p.
	Auxiliary Functions
poly	Construct a polynomial with given coefficients.
make-poly	Construct a polynomial of given degree.
coef	Pick out the ith coefficient of a polynomial.
main-var	The main variable of a polynomial.
degree	The degree of a polynomial; (degree x^2) = 2.
var=	Are two variables identical?
var>	Is one variable ordered before another?
poly+	Unary or binary polynomial addition.
poly-	Unary or binary polynomial subtraction.
k+poly	Add a constant k to a polynomial p.
k*poly	Multiply a polynomial p by a constant k.
poly+same	Add two polynomials with the same main variable.
poly*same	Multiply two polynomials with the same main variable.
normalize-poly	Alter a polynomial by dropping trailing zeros.
exponent->prefix	Used to convert to prefix.
args->prefix	Used to convert to prefix.
rat-numerator	Select the numerator of a rational.
rat-denominator	Select the denominator of a rational.
rat*rat	Multiply two rationals.
rat+rat	Add two rationals.
rat/rat	Divide two rationals.

Figure 15.1: Glossary for the Symbolic Manipulation Program

```
(defun poly (x &rest coefs)
  "Make a polynomial with main variable x
  and coefficients in increasing order."
  (apply #'vector x coefs))

(defun make-poly (x degree)
  "Make the polynomial 0 + 0*x + 0*x^2 + ... 0*x^degree"
  (let ((p (make-array (+ degree 2) :initial-element 0)))
    (setf (main-var p) x)
    p))
```

A polynomial can be altered by setting its main variable or any one of its coefficients using the following `defsetf` forms.

```
(defsetf main-var (p) (val)
  '(setf (svref (the polynomial ,p) 0) ,val))

(defsetf coef (p i) (val)
  '(setf (svref (the polynomial ,p) (+ ,i 1)) ,val))
```

The function `poly` constructs polynomials in a fashion similar to `list` or `vector`: with an explicit list of the contents. `make-poly`, on the other hand, is like `make-array`: it makes a polynomial of a specified size.

We provide `setf` methods for modifying the main variable and coefficients. Since this is the first use of `defsetf`, it deserves some explanation. A `defsetf` form takes a function (or macro) name, an argument list, and a second argument list that must consist of a single argument, the value to be assigned. The body of the form is an expression that stores the value in the proper place. So the `defsetf` for `main-var` says that `(setf (main-var p) val)` is equivalent to `(setf (svref (the polynomial p) 0) val)`. A `defsetf` is much like a `defmacro`, but there is a little less burden placed on the writer of `defsetf`. Instead of passing `p` and `val` directly to the `setf` method, Common Lisp binds local variables to these expressions, and passes those variables to the `setf` method. That way, the writer does not have to worry about evaluating the expressions in the wrong order or the wrong number of times. It is also possible to gain finer control over the whole process with `define-setf-method`, as explained on page 884.

The functions `poly+poly`, `poly*poly` and `poly^n` perform addition, multiplication, and exponentiation of polynomials, respectively. They are defined with several helping functions. `k*poly` multiplies a polynomial by a constant, `k`, which may be a number or another polynomial that is free of polynomial `p`'s main variable. `poly*same` is used to multiply two polynomials with the same main variable. For addition, the functions `k+poly` and `poly+same` serve analogous purposes. With that in mind, here's the function to convert from prefix to canonical form:

```
(defun prefix->canon (x)
  "Convert a prefix Lisp expression to canonical form.
  Exs: (+ (^ x 2) (* 3 x)) => #(x 0 3 1)
       (- (* (- x 1) (+ x 1)) (- (^ x 2) 1)) => 0"
  (cond ((numberp x) x)
        ((symbolp x) (poly x 0 1))
        ((and (exp-p x) (get (exp-op x) 'prefix->canon))
         (apply (get (exp-op x) 'prefix->canon)
                 (mapcar #'prefix->canon (exp-args x))))
        (t (error "Not a polynomial: ~a" x))))
```

It is data-driven, based on the `prefix->canon` property of each operator. In the following we install the appropriate functions. The existing functions `poly*poly` and `poly^n` can be used directly. But other operators need interface functions. The operators `+` and `-` need interface functions that handle both unary and binary.

```
(dolist (item '((+ poly+) (- poly-) (* poly*poly)
              (^ poly^n) (D deriv-poly)))
  (setf (get (first item) 'prefix->canon) (second item)))

(defun poly+ (&rest args)
  "Unary or binary polynomial addition."
  (ecase (length args)
    (1 (first args))
    (2 (poly+poly (first args) (second args)))))

(defun poly- (&rest args)
  "Unary or binary polynomial subtraction."
  (ecase (length args)
    (1 (poly*poly -1 (first args)))
    (2 (poly+poly (first args) (poly*poly -1 (second args)))))
```

The function `prefix->canon` accepts inputs that were not part of our definition of polynomials: unary positive and negation operators and binary subtraction and differentiation operators. These are permissible because they can all be reduced to the elementary `+` and `*` operations.

Remember that our problems with canonical form all began with the inability to decide which was simpler: $(+ x y)$ or $(+ y x)$. In this system, we define a canonical form by imposing an ordering on variables (we use alphabetic ordering as defined by `string>`). The rule is that a polynomial `p` can have coefficients that are polynomials in a variable later in the alphabet than `p`'s main variable, but no coefficients that are polynomials in variables earlier than `p`'s main variable. Here's how to compare variables:

```
(defun var= (x y) (eq x y))
(defun var> (x y) (string> x y))
```

The canonical form of the variable x will be $\#(x\ 0\ 1)$, which is $0 \times x^0 + 1 \times x^1$. The canonical form of $(+ x y)$ is $\#(x\ \#(y\ 0\ 1)\ 1)$. It couldn't be $\#(y\ \#(x\ 0\ 1)\ 1)$, because then the resulting polynomial would have a coefficient with a lesser main variable. The policy of ordering variables assures canonicity, by properly grouping like variables together and by imposing a particular ordering on expressions that would otherwise be commutative.

Here, then, is the code for adding two polynomials:

```
(defun poly+poly (p q)
  "Add two polynomials."
  (normalize-poly
   (cond
    ((numberp p) (k+poly p q))
    ((numberp q) (k+poly q p))
    ((var= (main-var p) (main-var q)) (poly+same p q))
    ((var> (main-var q) (main-var p)) (k+poly q p))
    (t (k+poly p q)))))

(defun k+poly (k p)
  "Add a constant k to a polynomial p."
  (cond ((eql k 0) p) ;; 0 + p = p
        ((and (numberp k) (numberp p))
         (+ k p)) ;; Add numbers
        (t (let ((r (copy-poly p))) ;; Add k to x^0 term of p
              (setf (coef r 0) (poly+poly (coef r 0) k))
              r))))

(defun poly+same (p q)
  "Add two polynomials with the same main variable."
  ;; First assure that q is the higher degree polynomial
  (if (> (degree p) (degree q))
      (poly+same q p)
      ;; Add each element of p into r (which is a copy of q).
      (let ((r (copy-poly q)))
        (loop for i from 0 to (degree p) do
              (setf (coef r i) (poly+poly (coef r i) (coef p i))))
        r)))

(defun copy-poly (p)
  "Make a copy a polynomial."
  (copy-seq p))
```


Both `poly+poly` and `poly*poly` make use of the function `normalize-poly` to “normalize” the result. The idea is that `(- (^ x 5) (^ x 5))` should return 0, not `#(x 0 0 0 0 0)`. Note that `normalize-poly` is a destructive operation: it calls `delete`, which can actually alter its argument. Normally this is a dangerous thing, but since `normalize-poly` is replacing something with its conceptual equal, no harm is done.

```
(defun normalize-poly (p)
  "Alter a polynomial by dropping trailing zeros."
  (if (numberp p)
      p
      (let ((p-degree (- (position 0 p :test (complement #'eql)
                          :from-end t)
                        1)))
        (cond ((<= p-degree 0) (normalize-poly (coef p 0)))
              ((< p-degree (degree p))
               (delete 0 p :start p-degree))
              (t p))))))
```

There are a few loose ends to clean up. First, the exponentiation function:

```
(defun poly^n (p n)
  "Raise polynomial p to the nth power, n>=0."
  (check-type n (integer 0 *))
  (cond ((= n 0) (assert (not (eql p 0))) 1)
        ((integerp p) (expt p n))
        (t (poly*poly p (poly^n p (- n 1))))))
```

15.2 Differentiating Polynomials

The differentiation routine is easy, mainly because there are only two operators (`+` and `*`) to deal with:

```
(defun deriv-poly (p x)
  "Return the derivative, dp/dx, of the polynomial p."
  ;; If p is a number or a polynomial with main-var > x,
  ;; then p is free of x, and the derivative is zero;
  ;; otherwise do real work.
  ;; But first, make sure X is a simple variable,
  ;; of the form #(X 0 1).
  (assert (and (typep x 'polynomial) (= (degree x) 1)
              (eql (coef x 0) 0) (eql (coef x 1) 1))))
```

```

(cond
  ((numberp p) 0)
  ((var> (main-var p) (main-var x)) 0)
  ((var= (main-var p) (main-var x))
   ;; d(a + bx + cx^2 + dx^3)/dx = b + 2cx + 3dx^2
   ;; So, shift the sequence p over by 1, then
   ;; put x back in, and multiply by the exponents
   (let ((r (subseq p 1)))
     (setf (main-var r) (main-var x))
     (loop for i from 1 to (degree r) do
       (setf (coef r i) (poly*poly (+ i 1) (coef r i))))
     (normalize-poly r)))
  (t ;; Otherwise some coefficient may contain x. Ex:
   ;; d(z + 3x + 3zx^2 + z^2x^3)/dz
   ;; = 1 + 0 + 3x^2 + 2zx^3
   ;; So copy p, and differentiate the coefficients.
   (let ((r (copy-poly p)))
     (loop for i from 0 to (degree p) do
       (setf (coef r i) (deriv-poly (coef r i) x)))
     (normalize-poly r))))))

```

? **Exercise 15.1 [h]** Integrating polynomials is not much harder than differentiating them. For example:

$$\int ax^2 + bx \, dx = \frac{ax^3}{3} + \frac{bx^2}{2} + c.$$

Write a function to integrate polynomials and install it in `prefix->canon`.

? **Exercise 15.2 [m]** Add support for *definite* integrals, such as $\int_a^b y \, dx$. You will need to make up a suitable notation and properly install it in both `infix->prefix` and `prefix->canon`. A full implementation of this feature would have to consider infinity as a bound, as well as the problem of integrating over singularities. You need not address these problems.

15.3 Converting between Infix and Prefix

All that remains is converting from canonical form back to prefix form, and from there back to infix form. This is a good point to extend the prefix form to allow expressions with more than two arguments. First we show an updated version of `prefix->infix` that handles multiple arguments:

```

(defun prefix->infix (exp)
  "Translate prefix to infix expressions.
  Handles operators with any number of args."
  (if (atom exp)
      exp
      (intersperse
       (exp-op exp)
       (mapcar #'prefix->infix (exp-args exp)))))

(defun intersperse (op args)
  "Place op between each element of args.
  Ex: (intersperse '+ '(a b c)) => '(a + b + c)"
  (if (length=1 args)
      (first args)
      (rest (loop for arg in args
                  collect op
                  collect arg))))

```

Now we need only convert from canonical form to prefix:

```

(defun canon->prefix (p)
  "Convert a canonical polynomial to a lisp expression."
  (if (numberp p)
      p
      (args->prefix
       '+ 0
       (loop for i from (degree p) downto 0
             collect (args->prefix
                      '* 1
                      (list (canon->prefix (coef p i))
                            (exponent->prefix
                             (main-var p) i)))))))

(defun exponent->prefix (base exponent)
  "Convert canonical base^exponent to prefix form."
  (case exponent
    (0 1)
    (1 base)
    (t '(^ ,base ,exponent))))

(defun args->prefix (op identity args)
  "Convert arg1 op arg2 op ... to prefix form."
  (let ((useful-args (remove identity args)))
    (cond ((null useful-args) identity)
          ((and (eq op '*') (member 0 args)) 0)
          ((length=1 args) (first useful-args))
          (t (cons op (mappend
                    #'(lambda (exp)

```

```

      (if (starts-with exp op)
          (exp-args exp)
          (list exp)))
    useful-args))))))

```

Finally, here's a top level to make use of all this:

```

(defun canon (infix-exp)
  "Canonicalize argument and convert it back to infix"
  (prefix->infix
   (canon->prefix
    (prefix->canon
     (infix->prefix infix-exp))))))

(defun canon-simplifier ()
  "Read an expression, canonicalize it, and print the result."
  (loop
   (print 'canon>)
   (print (canon (read)))))

```

and an example of it in use:

```

> (canon-simplifier)
CANON> (3 + x + 4 - x)
7
CANON> (x + y + y + x)
((2 * X) + (2 * Y))
CANON> (3 * x + 4 * x)
(7 * X)
CANON> (3 * x + y + x + 4 * x)
((8 * X) + Y)
CANON> (3 * x + y + z + x + 4 * x)
((8 * X) + (Y + Z))
CANON> ((x + 1) ^ 10)
((X ^ 10) + (10 * (X ^ 9)) + (45 * (X ^ 8)) + (120 * (X ^ 7))
 + (210 * (X ^ 6)) + (252 * (X ^ 5)) + (210 * (X ^ 4))
 + (120 * (X ^ 3)) + (45 * (X ^ 2)) + (10 * X) + 1)
CANON> ((x + 1) ^ 10 + (x - 1) ^ 10)
((2 * (X ^ 10)) + (90 * (X ^ 8)) + (420 * (X ^ 6))
 + (420 * (X ^ 4)) + (90 * (X ^ 2)) + 2)
CANON> ((x + 1) ^ 10 - (x - 1) ^ 10)
((20 * (X ^ 8)) + (240 * (X ^ 7)) + (504 * (X ^ 5))
 + (240 * (X ^ 3)) + (20 * X))
CANON> (3 * x ^ 3 + 4 * x * y * (x - 1) + x ^ 2 * (x + y))
((4 * (X ^ 3)) + ((5 * Y) * (X ^ 2)) + ((-4 * Y) * X))
CANON> (3 * x ^ 3 + 4 * x * w * (x - 1) + x ^ 2 * (x + w))
(((5 * (X ^ 2)) + (-4 * X)) * W) + (4 * (X ^ 3)))

```

```

CANON> (d (3 * x ^ 2 + 2 * x + 1) / d x)
((6 * X) + 2)
CANON> (d(z + 3 * x + 3 * z * x ^ 2 + z ^ 2 * x ^ 3) / d z)
(((2 * Z) * (X ^ 3)) + (3 * (X ^ 2)) + 1)
CANON> [Abort]

```

15.4 Benchmarking the Polynomial Simplifier

Unlike the rule-based program, this version gets all the answers right. Not only is the program correct (at least as far as these examples go), it is also fast. We can compare it to the canonical simplifier originally written for MACSYMA by William Martin (circa 1968), and modified by Richard Fateman. The modified version was used by Richard Gabriel in his suite of Common Lisp benchmarks (1985). The benchmark program is called `frpoly`, because it deals with polynomials and was originally written in the dialect Franz Lisp. The `frpoly` benchmark encodes polynomials as lists rather than vectors, and goes to great lengths to be efficient. Otherwise, it is similar to the algorithms used here (although the code itself is quite different, using `progs` and `gos` and other features that have fallen into disfavor in the intervening decades). The particular benchmark we will use here is raising $1 + x + y + z$ to the 15th power:

```

(defun r15-test ()
  (let ((r (prefix->canon '(+ 1 (+ x (+ y z))))))
    (time (poly^n r 15))
    nil))

```

This takes .97 seconds on our system. The equivalent test with the original `frpoly` code takes about the same time: .98 seconds. Thus, our program is as fast as production-quality code. In terms of storage space, vectors use about half as much storage as lists, because half of each cons cell is a pointer, while vectors are all useful data.²

How much faster is the polynomial-based code than the rule-based version? Unfortunately, we can't answer that question directly. We can `time (simp '(1 + x + y + z) ^ 15))`. This takes only a tenth of a second, but that is because it is doing no work at all—the answer is the same as the input! Alternately, we can take the expression computed by `(poly^n r 15)`, convert it to prefix, and pass that to `simplify`. `simplify` takes 27.8 seconds on this, so the rule-based version is

²Note: systems that use “cdr-coding” take about the same space for lists that are allocated all at once as for vectors. But cdr-coding is losing favor as RISC chips replace microcoded processors.

much slower. Section 9.6 describes ways to speed up the rule-based program, and a comparison of timing data appears on page 525.

There are always surprises when it comes down to measuring timing data. For example, the alert reader may have noticed that the version of `poly^n` defined above requires n multiplications. Usually, exponentiation is done by squaring a value when the exponent is even. Such an algorithm takes only $\log n$ multiplications instead of n . We can add a line to the definition of `poly^n` to get an $O(\log n)$ algorithm:

```
(defun poly^n (p n)
  "Raise polynomial p to the nth power, n>=0."
  (check-type n (integer 0 *))
  (cond ((= n 0) (assert (not (eql p 0))) 1)
        ((integerp p) (expt p n))
        ((evenp n) (poly^2 (poly^n p (/ n 2)))) ;***
        (t (poly*poly p (poly^n p (- n 1))))))

(defun poly^2 (p) (poly*poly p p))
```

The surprise is that this takes *longer* to raise `*r*` to the 15th power. Even though it does fewer `poly*poly` operations, it is doing them on more complex arguments, and there is more work altogether. If we use this version of `poly^n`, then `r15-test` takes 1.6 seconds instead of .98 seconds.

By the way, this is a perfect example of the conceptual power of recursive functions. We took an existing function, `poly^n`, added a single `cond` clause, and changed it from an $O(n)$ to $O(\log n)$ algorithm. (This turned out to be a bad idea, but that's beside the point. It would be a good idea for raising integers to powers.) The reasoning that allows the change is simple: First, p^n is certainly equal to $(p^{n/2})^2$ when n is even, so the change can't introduce any wrong answers. Second, the change continues the policy of decrementing n on every recursive call, so the function must eventually terminate (when $n = 0$). If it gives no wrong answers, and it terminates, then it must give the right answer.

In contrast, making the change for an iterative algorithm is more complex. The initial algorithm is simple:

```
(defun poly^n (p n)
  (let ((result 1))
    (loop repeat n do (setf result (poly*poly p result)))
    result))
```

But to change it, we have to change the `repeat` loop to a `while` loop, explicitly put in the decrement of n , and insert a test for the even case:

```
(defun poly^n (p n)
  (let ((result 1))
    (loop while (> n 0)
      do (if (evenp n)
            (setf p (poly^2 p)
                  n (/ n 2))
          (setf result (poly*poly p result)
                  n (- n 1))))
    result))
```

For this problem, it is clear that thinking recursively leads to a simpler function that is easier to modify.

It turns out that this is not the final word. Exponentiation of polynomials can be done even faster, with a little more mathematical sophistication. Richard Fateman's 1974 paper on Polynomial Multiplication analyzes the complexity of a variety of exponentiation algorithms. Instead of the usual asymptotic analysis (e.g. $O(n)$ or $O(n^2)$), he uses a fine-grained analysis that computes the constant factors (e.g. $1000 \times n$ or $2 \times n^2$). Such analysis is crucial for small values of n . It turns out that for a variety of polynomials, an exponentiation algorithm based on the binomial theorem is best. The binomial theorem states that

$$(a + b)^n = \sum_{i=0}^n \frac{n!}{i!(n-i)!} a^i b^{n-i}$$

for example,

$$(a + b)^3 = b^3 + 3ab^2 + 3a^2b + a^3$$

We can use this theorem to compute a power of a polynomial all at once, instead of computing it by repeated multiplication or squaring. Of course, a polynomial will in general be a sum of more than two components, so we have to decide how to split it into the a and b pieces. There are two obvious ways: either cut the polynomial in half, so that a and b will be of equal size, or split off one component at a time. Fateman shows that the latter method is more efficient in most cases. In other words, a polynomial $k_1x^n + k_2x^{n-1} + k_3x^{n-2} + \dots$ will be treated as the sum $a + b$ where $a = k_1x^n$ and b is the rest of the polynomial.

Following is the code for binomial exponentiation. It is somewhat messy, because the emphasis is on efficiency. This means reusing some data and using `p-add-into!` instead of the more general `poly+poly`.

```
(defun poly^n (p n)
  "Raise polynomial p to the nth power, n>=0."
  ;; Uses the binomial theorem
  (check-type n (integer 0 *))
  (cond
    ((= n 0) 1)
```



```

((integerp p) (expt p n))
(t ;; First: split the polynomial p = a + b, where
  ;; a = k*x^d and b is the rest of p
  (let ((a (make-poly (main-var p) (degree p)))
        (b (normalize-poly (subseq p 0 (- (length p) 1))))
        ;; Allocate arrays of powers of a and b:
        (a^n (make-array (+ n 1)))
        (b^n (make-array (+ n 1)))
        ;; Initialize the result:
        (result (make-poly (main-var p) (* (degree p) n))))
    (setf (coef a (degree p)) (coef p (degree p)))
    ;; Second: Compute powers of a^i and b^i for i up to n
    (setf (aref a^n 0) 1)
    (setf (aref b^n 0) 1)
    (loop for i from 1 to n do
      (setf (aref a^n i) (poly*poly a (aref a^n (- i 1))))
      (setf (aref b^n i) (poly*poly b (aref b^n (- i 1))))))
    ;; Third: add the products into the result,
    ;; so that result[i] = (n choose i) * a^i * b^(n-i)
    (let ((c 1) ;; c helps compute (n choose i) incrementally
          (loop for i from 0 to n do
            (p-add-into! result c
                          (poly*poly (aref a^n i)
                                       (aref b^n (- n i))))
            (setf c (/ (* c (- n i)) (+ i 1)))))
      (normalize-poly result))))))

(defun p-add-into! (result c p)
  "Destructively add c*p into result."
  (if (or (numberp p)
          (not (var= (main-var p) (main-var result))))
      (setf (coef result 0)
            (poly+poly (coef result 0) (poly*poly c p)))
      (loop for i from 0 to (degree p) do
        (setf (coef result i)
              (poly+poly (coef result i) (poly*poly c (coef p i))))))
  result)

```

Using this version of `poly^n`, `r15-test` takes only .23 seconds, four times faster than the previous version. The following table compares the times for `r15-test` with the three versions of `poly^n`, along with the times for applying simply to the `r15` polynomial, for various versions of `simplify`:

	program	secs	speed-up
rule-based versions			
1	original	27.8	-
2	memoization	7.7	4
3	memo+index	4.0	7
4	compilation only	2.5	11
5	memo+compilation	1.9	15
canonical versions			
6	squaring poly ⁿ	1.6	17
7	iterative poly ⁿ	.98	28
8	binomial poly ⁿ	.23	120

As we remarked earlier, the general techniques of memoization, indexing, and compilation provide for dramatic speed-ups. However, in the end, they do not lead to the fastest program. Instead, the fastest version was achieved by throwing out the original rule-based program, replacing it with a canonical-form-based program, and fine-tuning the algorithms within that program, using mathematical analysis.

Now that we have achieved a sufficiently fast system, the next two sections concentrate on making it more powerful.

15.5 A Canonical Form for Rational Expressions

A *rational* number is defined as a fraction: the quotient of two integers. A *rational expression* is hereby defined as the quotient of two polynomials. This section presents a canonical form for rational expressions.

First, a number or polynomial will continue to be represented as before. The quotient of two polynomials will be represented as a cons cells of numerator and denominator pairs. However, just as Lisp automatically reduces rational numbers to simplest form ($6/8$ is represented as $3/4$), we must reduce rational expressions. So, for example, $(x^2 - 1)/(x - 1)$ must be reduced to $x + 1$, not left as a quotient of two polynomials.

The following functions build and access rational expressions but do not reduce to simplest form, except in the case where the denominator is a number. Building up the rest of the functionality for full rational expressions is left to a series of exercises:

```
(defun make-rat (numerator denominator)
  "Build a rational: a quotient of two polynomials."
  (if (numberp denominator)
      (k*poly (/ 1 denominator) numerator)
      (cons numerator denominator)))
```

```

(defun rat-numerator (rat)
  "The numerator of a rational expression."
  (typecase rat
    (cons (car rat))
    (number (numerator rat))
    (t rat)))

(defun rat-denominator (rat)
  "The denominator of a rational expression."
  (typecase rat
    (cons (cdr rat))
    (number (denominator rat))
    (t 1)))

```

- ?** **Exercise 15.3 [s]** Modify `prefix->canon` to accept input of the form x / y and to return rational expressions instead of polynomials. Also allow for input of the form x^n .
- ?** **Exercise 15.4 [m]** Add arithmetic routines for multiplication, addition, and division of rational expressions. Call them `rat*rat`, `rat+rat`, and `rat/rat` respectively. They will call upon `poly*poly`, `poly+poly` and a new function, `poly/poly`, which is defined in the next exercise.
- ?** **Exercise 15.5 [h]** Define `poly-gcd`, which computes the greatest common divisor of two polynomials.
- ?** **Exercise 15.6 [h]** Using `poly-gcd`, define the function `poly/poly`, which will implement division for polynomials. Polynomials are closed under addition and multiplication, so `poly+poly` and `poly*poly` both returned polynomials. Polynomials are not closed under division, so `poly/poly` will return a rational expression.

15.6 Extending Rational Expressions

Now that we can divide polynomials, the final step is to reinstate the logarithmic, exponential, and trigonometric functions. The problem is that if we allow all these functions, we get into problems with canonical form again. For example, the following three expressions are all equivalent:

$$\begin{aligned} & \sin(x) \\ & \cos\left(x - \frac{\pi}{2}\right) \\ & \frac{e^{ix} - e^{-ix}}{2i} \end{aligned}$$




If we are interested in assuring we have a canonical form, the safest thing is to allow only e^x and $\log(x)$. All the other functions can be defined in terms of these two. With this extension, the set of expressions we can form is closed under differentiation, and it is possible to canonicalize expressions. The result is a mathematically sound construction known as a *differentiable field*. This is precisely the construct that is assumed by the Risch integration algorithm (Risch 1969, 1979).

The disadvantage of this minimal extension is that answers may be expressed in unfamiliar terms. The user asks for $d \sin(x^2)/dx$, expecting a simple answer in terms of \cos , and is surprised to see a complex answer involving e^{ix} . Because of this problem, most computer algebra systems have made more radical extensions, allowing \sin , \cos , and other functions. These systems are treading on thin mathematical ice. Algorithms that would be guaranteed to work over a simple differentiable field may fail when the domain is extended this way. In general, the result will not be a wrong answer but rather the failure to find an answer at all.

15.7 History and References

A brief history of symbolic algebra systems is given in chapter 8. Fateman (1979), Martin and Fateman (1971), and Davenport et al. (1988) give more details on the MACSYMA system, on which this chapter is loosely based. Fateman (1991) discusses the `frpoly` benchmark and introduces the vector implementation used in this chapter.

15.8 Exercises

-  **Exercise 15.7 [h]** Implement an extension of the rationals to include logarithmic, exponential, and trigonometric functions.
-  **Exercise 15.8 [m]** Modify `deriv` to handle the extended rational expressions.
-  **Exercise 15.9 [d]** Adapt the integration routine from section 8.6 (page 252) to the rational expression representation. Davenport et al. 1988 may be useful.

- ?** **Exercise 15.10 [s]** Give several reasons why constant polynomials, like 3, are represented as integers rather than as vectors.

15.9 Answers

Answer 15.4

```
(defun rat*rat (x y)
  "Multiply rationals: a/b * c/d = a*c/b*d"
  (poly/poly (poly*poly (rat-numerator x)
                        (rat-numerator y))
             (poly*poly (rat-denominator x)
                        (rat-denominator y))))

(defun rat+rat (x y)
  "Add rationals: a/b + c/d = (a*d + c*b)/b*d"
  (let ((a (rat-numerator x))
        (b (rat-denominator x))
        (c (rat-numerator y))
        (d (rat-denominator y)))
    (poly/poly (poly+poly (poly*poly a d) (poly*poly c b))
              (poly*poly b d))))

(defun rat/rat (x y)
  "Divide rationals: a/b / c/d = a*d/b*c"
  (rat*rat x (make-rat (rat-denominator y) (rat-numerator y))))
```

Answer 15.6

```
(defun poly/poly (p q)
  "Divide p by q: if d is the greatest common divisor of p and q
  then p/q = (p/d) / (q/d). Note if q=1, then p/q = p."
  (if (eql q 1)
      p
      (let ((d (poly-gcd p q)))
        (make-rat (poly/poly p d)
                  (poly/poly q d)))))
```

Answer 15.10 (1) An integer takes less time and space to process. (2) Representing numbers as a polynomial would cause an infinite regress, because the coefficients would be numbers. (3) Unless a policy was decided upon, the representation would not be canonical, since $\#(x\ 3)$ and $\#(y\ 3)$ both represent 3.

CHAPTER 16

Expert Systems

*An expert is one who knows more and more
about less and less.*

—Nicholas Murray Butler (1862–1947)

In the 1970s there was terrific interest in the area of *knowledge-based expert systems*. An expert system or knowledge-based system is one that solves problems by applying knowledge that has been garnered from one or more experts in a field. Since these experts will not in general be programmers, they will very probably express their expertise in terms that cannot immediately be translated into a program. It is the goal of expert-system research to come up with a representation that is flexible enough to handle expert knowledge, but still capable of being manipulated by a computer program to come up with solutions.

A plausible candidate for this representation is as logical facts and rules, as in Prolog. However, there are three areas where Prolog provides poor support for a general knowledge-based system:

- Reasoning with uncertainty. Prolog only deals with the black-and-white world of facts that are clearly true or false (and it doesn't even handle false very well). Often experts will express rules of thumb that are "likely" or "90% certain."
- Explanation. Prolog gives solutions to queries but no indication of how those solutions were derived. A system that can explain its solutions to the user in understandable terms will be trusted more.
- Flexible flow of control. Prolog works by backward-chaining from the goal. In some cases, we may need more varied control strategy. For example, in medical diagnosis, there is a prescribed order for acquiring certain information about the patient. A medical system must follow this order, even if it doesn't fit in with the backward-chaining strategy.

The early expert systems used a wide variety of techniques to attack these problems. Eventually, it became clear that certain techniques were being used frequently, and they were captured in *expert-system shells*: specialized programming environments that helped acquire knowledge from the expert and use it to solve problems and provide explanations. The idea was that these shells would provide a higher level of abstraction than just Lisp or Prolog and would make it easy to write new expert systems.

The MYCIN expert system was one of the earliest and remains one of the best known. It was written by Dr. Edward Shortliffe in 1974 as an experiment in medical diagnosis. MYCIN was designed to prescribe antibiotic therapy for bacterial blood infections, and when completed it was judged to perform this task as well as experts in the field. Its name comes from the common suffix in drugs it prescribes: erythromycin, clindamycin, and so on. The following is a slightly modified version of one of MYCIN's rules, along with an English paraphrase generated by the system:

```
(defrule 52
  if (site culture is blood)
      (gram organism is neg)
      (morphology organism is rod)
      (burn patient is serious)
  then .4
      (identity organism is pseudomonas))
```

Rule 52:

If

- 1) THE SITE OF THE CULTURE IS BLOOD
- 2) THE GRAM OF THE ORGANISM IS NEG
- 3) THE MORPHOLOGY OF THE ORGANISM IS ROD
- 4) THE BURN OF THE PATIENT IS SERIOUS

Then there is weakly suggestive evidence (0.4) that

- 1) THE IDENTITY OF THE ORGANISM IS PSEUDOMONAS

MYCIN led to the development of the EMYCIN expert-system shell. EMYCIN stands for "essential MYCIN," although it is often misrepresented as "empty MYCIN." Either way, the name refers to the shell for acquiring knowledge, reasoning with it, and explaining the results, without the specific medical knowledge.

EMYCIN is a backward-chaining rule interpreter that has much in common with Prolog. However, there are four important differences. First, and most importantly, EMYCIN deals with uncertainty. Instead of insisting that all predications be true or false, EMYCIN associates a *certainty factor* with each predication. Second, EMYCIN caches the results of its computations so that they need not be duplicated. Third, EMYCIN provides an easy way for the system to ask the user for information. Fourth, it provides explanations of its behavior. This can be summed up in the equation:

$$\text{EMYCIN} = \text{Prolog} + \text{uncertainty} + \text{caching} + \text{questions} + \text{explanations}$$

We will first cover the ways EMYCIN is different from Prolog. After that we will return to the main core of EMYCIN, the backward-chaining rule interpreter. Finally, we will show how to add some medical knowledge to EMYCIN to reconstruct MYCIN. A glossary of the program is in figure 16.1.

16.1 Dealing with Uncertainty

EMYCIN deals with uncertainty by replacing the two boolean values, true and false, with a range of values called *certainty factors*. These are numbers from -1 (false) to $+1$ (true), with 0 representing a complete unknown. In Lisp:

```
(defconstant true +1.0)
(defconstant false -1.0)
(defconstant unknown 0.0)
```

To define the logic of certainty factors, we need to define the logical operations, such as and, or, and not. The first operation to consider is the combination of two distinct pieces of evidence expressed as certainty factors. Suppose we are trying to

emycin mycin	Top-Level Functions for the Client Run the shell on a list of contexts representing a problem. Run the shell on the microbial infection domain.
defcontext defparm defrule	Top-Level Functions for the Expert Define a context. Define a parameter. Define a rule.
true false unknown cf-cut-off	Constants A certainty factor of +1. A certainty factor of -1. A certainty factor of 0. Below this certainty we cut off search.
context parm rule yes/no	Data Types A subdomain concerning a particular problem. A parameter. A backward-chaining rule with certainty factors. The type with members yes and no.
get-context-data find-out get-db use-rules use-rule new-instance report-findings	Major Functions within Emycin Collect data and draw conclusions. Determine values by knowing, asking, or using rules. Retrieve a fact from the data base. Apply all rules relevant to a parameter. Apply one rule. Create a new instance of a context. Print the results.
cf-or cf-and true-p false-p cf-p put-db clear-db get-vals get-cf update-cf ask-vals prompt-and-read-vals inst-name check-reply parse-reply parm-type get-parm put-rule get-rules clear-rules satisfy-premises eval-condition reject-premise conclude is check-conditions print-rule print-conditions print-condition cf->english print-why	Auxiliary Functions Combine certainty factors (CFs) with OR. Combine certainty factors (CFs) with AND. Is this CF true for purposes of search? Is this CF false for purposes of search? Is this a certainty factor? Place a fact in the data base. Clear all facts from the data base. Get value and CF for a parameter/instance. Get CF for a parameter/instance/value triplet. Change CF for a parameter/instance/value triplet. Ask the user for value/CF for a parameter/instance. Print a prompt and read a reply. The name of an instance. See if reply is valid list of CF/values. Convert reply into list of CF/values. Values of this parameter must be of this type. Find or make a parameter structure for this name. Add a new rule, indexed under each conclusion. Retrieve rules that help determine a parameter. Remove all rules. Calculate the combined CF for the premises. Determine the CF for a condition. Rule out a premise if it is clearly false. Add a parameter/instance/value/CF to the data base. An alias for equal. Make sure a rule is valid. Print a rule. Print a list of conditions. Print a single condition. Convert .7 to "suggestive evidence," etc. Say why a rule is being used.

Figure 16.1: Glossary for the EMYCIN Program

determine the chances of a patient having disease X. Assume we have a population of prior patients that have been given two lab tests. One test says that 60% of the patients have the disease and the other says that 40% have it. How should we combine these two pieces of evidence into one? Unfortunately, there is no way to answer that question correctly without knowing more about the *dependence* of the two sources on each other. Suppose the first test says that 60% of the patients (who all happen to be male) have the disease, and the second says that 40% (who all happen to be female) have it. Then we should conclude that 100% have it, because the two tests cover the entire population. On the other hand, if the first test is positive only for patients that are 70 years old or older, and the second is positive only for patients that are 80 or older, then the second is just a subset of the first. This adds no new information, so the correct answer is 60% in this case.

In section 16.9 we will consider ways to take this kind of reasoning into account. For now, we will present the combination method actually used in EMYCIN. It is defined by the formula:

combine (A, B) =

$$\begin{array}{l}
 A + B - AB; \quad A, B > 0 \\
 A + B + AB; \quad A, B < 0 \\
 \frac{A + B}{1 - \min(|A|, |B|)}; \text{ otherwise}
 \end{array}$$

According to this formula, combine(.60,.40) = .76, which is a compromise between the extremes of .60 and 1.00. It is the same as the probability p(A or B), assuming that A and B are independent.

However, it should be clear that certainty factors are not the same thing as probabilities. Certainty factors attempt to deal with disbelief as well as belief, but they do not deal with dependence and independence. The EMYCIN combination function has a number of desirable properties:

- It always computes a number between -1 and +1.
- Combining unknown (zero) with anything leaves it unchanged.
- Combining true with anything (except false) gives true.
- Combining true and false is an error.
- Combining two opposites gives unknown.
- Combining two positives (except true) gives a larger positive.
- Combining a positive and a negative gives something in between.

So far we have seen how to combine two separate pieces of evidence for the same hypothesis. In other words, if we have the two rules:

$$\begin{aligned} A &\Rightarrow C \\ B &\Rightarrow C \end{aligned}$$

and we know A with certainty factor (cf) .6 and B with cf .4, then we can conclude C with cf .76. But consider a rule with a conjunction in the premise:

$$A \text{ and } B \Rightarrow C$$

Combining A and B in this case is quite different from combining them when they are in separate rules. EMYCIN chooses to combine conjunctions by taking the minimum of each conjunct's certainty factor. If certainty factors were probabilities, this would be equivalent to assuming dependence between conjuncts in a rule. (If the conjuncts were independent, then the product of the probabilities would be the correct answer.) So EMYCIN is making the quite reasonable (but sometimes incorrect) assumption that conditions that are tied together in a single rule will be dependent on one another, while conditions in separate rules are independent.

The final complication is that rules themselves may be uncertain. That is, MYCIN accommodates rules that look like:

$$A \text{ and } B \Rightarrow .9 C$$

to say that A and B imply C with .9 certainty. EMYCIN simply multiplies the rule's cf by the combined cf of the premise. So if A has cf .6 and B has cf .4, then the premise as a whole has cf .4 (the minimum of A and B), which is multiplied by .9 to get .36. The .36 is then combined with any existing cf for C. If C is previously unknown, then combining .36 with 0 will give .36. If C had a prior cf of .76, then the new cf would be $.36 + .76 - (.36 \times .76) = .8464$.

Here are the EMYCIN certainty factor combination functions in Lisp:

```
(defun cf-or (a b)
  "Combine the certainty factors for the formula (A or B).
  This is used when two rules support the same conclusion."
  (cond ((and (> a 0) (> b 0))
        (+ a b (* -1 a b)))
        ((and (< a 0) (< b 0))
        (+ a b (* a b)))
        (t (/ (+ a b)
              (- 1 (min (abs a) (abs b)))))))

(defun cf-and (a b)
  "Combine the certainty factors for the formula (A and B)."
  (min a b))
```

Certainty factors can be seen as a generalization of truth values. EMYCIN is a

backward-chaining rule system that combines certainty factors according to the functions laid out above. But if we only used the certainty factors true and false, then EMYCIN would behave exactly like Prolog, returning only answers that are definitely true. It is only when we provide fractional certainty factors that the additional EMYCIN mechanism makes a difference.


Truth values actually serve two purposes in Prolog. They determine the final answer, yes, but they also determine when to cut off search: if any one of the premises of a rule is false, then there is no sense looking at the other premises. If in EMYCIN we only cut off the search when one of the premises was absolutely false, then we might have to search through a lot of rules, only to yield answers with very low certainty factors. Instead, EMYCIN arbitrarily cuts off the search and considers a premise false when it has a certainty factor below .2. The following functions support this arbitrary cutoff point:

```
(defconstant cf-cut-off 0.2
  "Below this certainty we cut off search.")

(defun true-p (cf)
  "Is this certainty factor considered true?"
  (and (cf-p cf) (> cf cf-cut-off)))

(defun false-p (cf)
  "Is this certainty factor considered false?"
  (and (cf-p cf) (< cf (- cf-cut-off 1.0))))

(defun cf-p (x)
  "Is X a valid numeric certainty factor?"
  (and (numberp x) (<= false x true)))
```

 **Exercise 16.1 [m]** Suppose you read the headline “Elvis Alive in Kalamazoo” in a tabloid newspaper to which you attribute a certainty factor of .01. If you combine certainties using EMYCIN’s combination rule, how many more copies of the newspaper would you need to see before you were .95 certain Elvis is alive?

16.2 Caching Derived Facts

The second thing that makes EMYCIN different from Prolog is that EMYCIN *caches* all the facts it derives in a data base. When Prolog is asked to prove the same goal twice, it performs the same computation twice, no matter how laborious. EMYCIN performs the computation the first time and just fetches it the second time.

We can implement a simple data base by providing three functions: `put-db` to add an association between a key and a value, `get-db` to retrieve a value, and `clear-db` to empty the data base and start over:

```
(let ((db (make-hash-table :test #'equal)))
  (defun get-db (key) (gethash key db))
  (defun put-db (key val) (setf (gethash key db) val))
  (defun clear-db () (clrhash db)))
```

This data base is general enough to hold any association between key and value. However, most of the information we will want to store is more specific. *EMYCIN* is designed to deal with objects (or *instances*) and attributes (or *parameters*) of those objects. For example, each patient has a name parameter. Presumably, the value of this parameter will be known exactly. On the other hand, each microscopic organism has an *identity* parameter that is normally not known at the start of the consultation. Applying the rules will lead to several possible values for this parameter, each with its own certainty factor. In general, then, the data base will have keys of the form (*parameter instance*) with values of the form ((*val*₁ *cf*₁) (*val*₂ *cf*₂)...). In the following code, `get-vals` returns the list of value/cf pairs for a given parameter and instance, `get-cf` returns the certainty factor for a parameter/instance/value triplet, and `update-cf` changes the certainty factor by combining the old one with a new one. Note that the first time `update-cf` is called on a given parameter/instance/value triplet, `get-cf` will return unknown (zero). Combining that with the given cf yields cf itself. Also note that the data base has to be an equal hash table, because the keys may include freshly consed lists.

```
(defun get-vals (parm inst)
  "Return a list of (val cf) pairs for this (parm inst)."  
  (get-db (list parm inst)))

(defun get-cf (parm inst val)
  "Look up the certainty factor or return unknown."  
  (or (second (assoc val (get-vals parm inst)))  
      unknown))

(defun update-cf (parm inst val cf)
  "Change the certainty factor for (parm inst is val),  
  by combining the given cf with the old."  
  (let ((new-cf (cf-or cf (get-cf parm inst val))))  
    (put-db (list parm inst)  
            (cons (list val new-cf)  
                  (remove val (get-db (list parm inst))  
                           :key #'first)))))
```

The data base holds all information related to an instance of a problem. For example,

in the medical domain, the data base would hold all information about the current patient. When we want to consider a new patient, the data base is cleared.

There are three other sources of information that cannot be stored in this data base, because they have to be maintained from one problem to the next. First, the *rule base* holds all the rules defined by the expert. Second, there is a structure to define each parameter; these are indexed under the name of each parameter. Third, we shall see that the flow of control is managed in part by a list of *contexts* to consider. These are structures that will be passed to the `myc i n` function.

16.3 Asking Questions

The third way that EMYCIN differs from Prolog is in providing an automatic means of asking the user questions when answers cannot be derived from the rules. This is not a fundamental difference; after all, it is not too hard to write Prolog rules that print a query and read a reply. EMYCIN lets the knowledge-base designer write a simple declaration instead of a rule, and will even assume a default declaration if none is provided. The system also makes sure that the same question is never asked twice.

The following function `ask-val s` prints a query that asks for the parameter of an instance, and reads from the user the value or a list of values with associated certainty factors. The function first looks at the data base to make sure the question has not been asked before. It then checks each value and certainty factor to see if each is of the correct type, and it also allows the user to ask certain questions. A `?` reply will show what type answer is expected. `rule` will show the current rule that the system is working on. `why` also shows the current rule, but it explains in more detail what the system knows and is trying to find out. Finally, `help` prints the following summary:

```
(defconstant help-string
  "~&Type one of the following:
  ?   - to see possible answers for this parameter
  rule - to show the current rule
  why  - to see why this question is asked
  help - to see this list
  xxx  - (for some specific xxx) if there is a definite answer
  (xxx .5 yyy .4) - If there are several answers with
                different certainty factors.")
```

Here is `ask-val s`. Note that the `why` and `rule` options assume that the current rule has been stored in the data base. The functions `print-why`, `parm-type`, and `check-reply` will be defined shortly.

```

(defun ask-vals (parm inst)
  "Ask the user for the value(s) of inst's parm parameter,
  unless this has already been asked. Keep asking until the
  user types UNKNOWN (return nil) or a valid reply (return t)."
  (unless (get-db '(asked ,parm ,inst))
    (put-db '(asked ,parm ,inst) t)
    (loop
      (let ((ans (prompt-and-read-vals parm inst)))
        (case ans
          (help (format t help-string))
          (why (print-why (get-db 'current-rule) parm))
          (rule (princ (get-db 'current-rule)))
          ((unk unknown) (RETURN nil))
          (? (format t "~&A ~a must be of type ~a"
                    parm (parm-type parm)) nil)
          (t (if (check-reply ans parm inst)
                 (RETURN t)
                 (format t "~&Illegal reply. ~
                          Type ? to see legal ones."))))))))))

```

The following is `prompt-and-read-vals`, the function that actually asks the query and reads the reply. It basically calls `format` to print a prompt and `read` to get the reply, but there are a few subtleties. First, it calls `finish-output`. Some Lisp implementations buffer output on a line-by-line basis. Since the prompt may not end in a newline, `finish-output` makes sure the output is printed before the reply is read.

So far, all the code that refers to a `parm` is really referring to the name of a parameter—a symbol. The actual parameters themselves will be implemented as structures. We use `get-parm` to look up the structure associated with a symbol, and the selector functions `parm-prompt` to pick out the prompt for each parameter and `parm-reader` to pick out the reader function. Normally this will be the function `read`, but `read-line` is appropriate for reading string-valued parameters.

The macro `defparm` (shown here) provides a way to define prompts and readers for parameters.

```

(defun prompt-and-read-vals (parm inst)
  "Print the prompt for this parameter (or make one up) and
  read the reply."
  (fresh-line)
  (format t (parm-prompt (get-parm parm)) (inst-name inst) parm)
  (princ " ")
  (finish-output)
  (funcall (parm-reader (get-parm parm))))

```

```
(defun inst-name (inst)
  "The name of this instance."
  ;; The stored name is either like ("Jan Doe" 1.0) or nil
  (or (first (first (get-vals 'name inst)))
      inst))
```

The function `check-reply` uses `parse-reply` to convert the user's reply into a canonical form, and then checks that each value is of the right type, and that each certainty factor is valid. If so, the data base is updated to reflect the new certainty factors.

```
(defun check-reply (reply parm inst)
  "If reply is valid for this parm, update the DB.
  Reply should be a val or (val1 cf1 val2 cf2 ...).
  Each val must be of the right type for this parm."
  (let ((answers (parse-reply reply)))
    (when (every #'(lambda (pair)
                    (and (typep (first pair) (parm-type parm))
                        (cf-p (second pair))))
          answers)
      ;; Add replies to the data base
      (dolist (pair answers)
        (update-cf parm inst (first pair) (second pair)))
      answers)))

(defun parse-reply (reply)
  "Convert the reply into a list of (value cf) pairs."
  (cond ((null reply) nil)
        ((atom reply) '(,reply ,true))
        (t (cons (list (first reply) (second reply))
                  (parse-reply (rest2 reply))))))
```

Parameters are implemented as structures with six slots: the name (a symbol), the context the parameter is for, the prompt used to ask for the parameter's value, a Boolean that tells if we should ask the user before or after using rules, a type restriction describing the legal values, and finally, the function used to read the value of the parameter.

Parameters are stored on the property list of their names under the `parm` property, so getting the `parm-type` of a name requires first getting the `parm` structure, and then selecting the type restriction field. By default, a parameter is given type `t`, meaning that any value is valid for that type. We also define the type `yes/no`, which comes in handy for Boolean parameters.

We want the default prompt to be "What is the PARM of the INST?" But most user-defined prompts will want to print the `inst`, and not the `parm`. To make it easy to write user-defined prompts, `prompt-and-read-vals` makes the instance be the first argument to the format string, with the `parm` second. Therefore, in the default

prompt we need to use the format directive "`~*`" to skip the instance argument, and "`~2:*`" to back up two arguments to get back to the instance. (These directives are common in error calls, where one list of arguments is passed to two format strings.)

`defparm` is a macro that calls `new-parm`, the constructor function defined in the `parm` structure, and stores the resulting structure under the `parm` property of the parameter's name.

```
(defstruct (parm (:constructor
                 new-parm (name &optional context type-restriction
                             prompt ask-first reader)))
  name (context nil) (prompt "~&What is the ~*~a of ~2:~*~a?")
  (ask-first nil) (type-restriction t) (reader 'read))

(defmacro defparm (parm &rest args)
  "Define a parameter."
  `(setf (get ',parm 'parm) (apply #'new-parm ',parm ',args)))

(defun parm-type (parm-name)
  "What type is expected for a value of this parameter?"
  (parm-type-restriction (get-parm parm-name)))

(defun get-parm (parm-name)
  "Look up the parameter structure with this name."
  ;; If there is none, make one
  (or (get parm-name 'parm)
      (setf (get parm-name 'parm) (new-parm parm-name))))

(deftype yes/no () '(member yes no))
```

16.4 Contexts Instead of Variables

Earlier we gave an equation relating EMYCIN to Prolog. That equation was not quite correct, because EMYCIN lacks one of Prolog's most important features: the logic variable. Instead, EMYCIN uses *contexts*. So the complete equation is:

$$\text{EMYCIN} = \text{Prolog} + \text{uncertainty} + \text{caching} + \text{questions} + \text{explanations} \\ + \text{contexts} - \text{variables}$$

A context is defined by the designers of MYCIN as a situation within which the program reasons. But it makes more sense to think of a context simply as a data type. So the list of contexts supplied to the program will determine what types of objects can be reasoned about. The program keeps track of the most recent instance of each type, and the rules can refer to those instances only, using the name of the

type. In our version of MYCIN, there are three types or contexts: patients, cultures, and organisms. Here is an example of a rule that references all three contexts:

```
(defrule 52
  if (site culture is blood)
      (gram organism is neg)
      (morphology organism is rod)
      (burn patient is serious)
  then .4
      (identity organism is pseudomonas))
```

Ignoring certainty factors for the moment, this MYCIN rule is equivalent to a Prolog rule of the form:

```
(<- (identity ?o ?pseudomonas)
    (and (culture ?c) (site ?c blood)
         (organism ?o) (gram ?o neg) (morphology ?o rod)
         (patient ?p) (burn ?p serious)))
```

The context mechanism provides sufficient flexibility to handle many of the cases that would otherwise be handled by variables. One important thing that cannot be done is to refer to more than one instance of the same context. Only the most recent instance can be referred to. Contexts are implemented as structures with the following definition:

```
(defstruct context
  "A context is a sub-domain, a type."
  name (number 0) initial-data goals)

(defmacro defcontext (name &optional initial-data goals)
  "Define a context."
  '(make-context :name ',name :initial-data ',initial-data
                 :goals ',goals))
```

The name field is something like patient or organism. Instances of contexts are numbered; the number field holds the number of the most recent instance. Each context also has two lists of parameters. The initial-data parameters are asked for when each instance is created. Initial data parameters are normally known by the user. For example, a doctor will normally know the patient's name, age, and sex, and as a matter of training expects to be asked these questions first, even if they don't factor into every case. The goal parameters, on the other hand, are usually unknown to the user. They are determined through the backward-chaining process.

The following function creates a new instance of a context, writes a message, and stores the instance in two places in the data base: under the key current-instance,

and also under the name of the context. The contexts form a tree. In our example, the patient context is the root of the tree, and the current patient is stored in the data base under the key `patient`. The next level of the tree is for cultures taken from the patient; the current culture is stored under the `culture` key. Finally, there is a level for organisms found in each culture. The current organism is stored under both the `organism` and `current-instance` keys. The context tree is shown in figure 16.2.

```
(defun new-instance (context)
  "Create a new instance of this context."
  (let ((instance (format nil "~a-~d"
                          (context-name context)
                          (incf (context-number context)))))
    (format t "~&----- ~a -----~&" instance)
    (put-db (context-name context) instance)
    (put-db 'current-instance instance)))
```

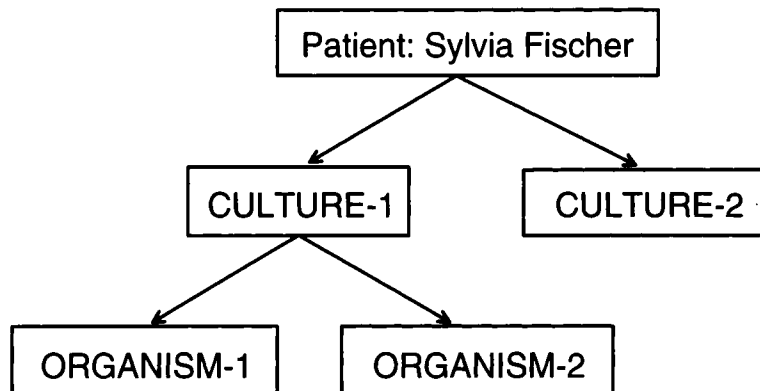


Figure 16.2: A Context Tree

16.5 Backward-Chaining Revisited

Now that we have seen how EMYCIN is different from Prolog, we are ready to tackle the way in which it is the same: the backward-chaining rule interpreter. Like Prolog, EMYCIN is given a goal and applies rules that are appropriate to the goal. Applying a rule means treating each premise of the rule as a goal and recursively applying rules that are appropriate to each premise.

There are still some remaining differences. In Prolog, a goal can be any expression, and appropriate rules are those whose heads unify with the goal. If any appropriate rule succeeds, then the goal is known to be true. In EMYCIN, a rule might give a goal a certainty of .99, but we still have to consider all the other rules that are appropriate to the goal, because they might bring the certainty down below the cutoff threshold. Thus, EMYCIN always gathers all evidence relating to a parameter/instance pair first, and only evaluates the goal after all the evidence is in. For example, if the goal was (temp patient > 98.6), EMYCIN would first evaluate all rules with conclusions about the current patient's temperature, and only then compare the temperature to 98.6.

Another way of looking at it is that Prolog has the luxury of searching depth-first, because the semantics of Prolog rules is such that if any rule says a goal is true, then it is true. EMYCIN must search breadth-first, because a goal with certainty of .99 might turn out to be false when more evidence is considered.

We are now ready to sketch out the design of the EMYCIN rule interpreter: To find-out a parameter of an instance: If the value is already stored in the data base, use the known value. Otherwise, the two choices are using the rules or asking the user. Do these in the order specified for this parameter, and if the first one succeeds, don't bother with the second. Note that ask-vals (defined above) will not ask the same question twice.

To use-rules, find all the rules that concern the given parameter and evaluate them with use-rule. After each rule has been tried, if any of them evaluate to true, then succeed.

To use-rule a rule, first check if any of the premises can be rejected outright. If we did not have this check, then the system could start asking the user questions that were obviously irrelevant. So we waste some of the program's time (checking each premise twice) to save the more valuable user time. (The function eval-condition takes an optional argument specifying if we should recursively ask questions in trying to accept or reject a condition.)

If no premise can be rejected, then evaluate each premise in turn with evaluate-condition, keeping track of the accumulated certainty factor with cf-and (which is currently just min), and cutting off evaluation when the certainty factor drops below threshold. If the premises evaluate true, then add the conclusions to the data base. The calling sequence looks like this. Note that the recursive call to find-out is what enables chaining to occur:

```

find-out      ; To find out a parameter for an instance:
  get-db      ; See if it is cached in the data base
  ask-vals    ; See if the user knows the answer
  use-rules   ; See if there is a rule for it:
    reject-premise ; See if the rule is outright false
    satisfy-premises ; Or see if each condition is true:
      eval-condition ; Evaluate each condition
      find-out      ; By finding the parameter's values

```

Before showing the interpreter, here is the structure definition for rules, along with the functions to maintain a data base of rules:

```
(defstruct (rule (:print-function print-rule))
  number premises conclusions cf)

(let ((rules (make-hash-table)))

  (defun put-rule (rule)
    "Put the rule in a table, indexed under each
    parm in the conclusion."
    (dolist (concl (rule-conclusions rule))
      (push rule (gethash (first concl) rules)))
    rule)

  (defun get-rules (parm)
    "A list of rules that help determine this parameter."
    (gethash parm rules))

  (defun clear-rules () (clrhash rules)))
```

Here, then, is the interpreter, find-out. It can find out the value(s) of a parameter three ways. First, it looks to see if the value is already stored in the data base. Next, it tries asking the user or using the rules. The order in which these two options are tried depends on the parm-ask-first property of the parameter. Either way, if an answer is determined, it is stored in the data base.

```
(defun find-out (parm &optional (inst (get-db 'current-instance)))
  "Find the value(s) of this parameter for this instance,
  unless the values are already known.
  Some parameters we ask first; others we use rules first."
  (or (get-db '(known ,parm ,inst))
      (put-db '(known ,parm ,inst)
              (if (parm-ask-first (get-parm parm))
                  (or (ask-vals parm inst) (use-rules parm))
                  (or (use-rules parm) (ask-vals parm inst))))))

(defun use-rules (parm)
  "Try every rule associated with this parameter.
  Return true if one of the rules returns true."
  (some #'true-p (mapcar #'use-rule (get-rules parm))))
```

```

(defun use-rule (rule)
  "Apply a rule to the current situation."
  ;; Keep track of the rule for the explanation system:
  (put-db 'current-rule rule)
  ;; If any premise is known false, give up.
  ;; If every premise can be proved true, then
  ;; draw conclusions (weighted with the certainty factor).
  (unless (some #'reject-premise (rule-premises rule))
    (let ((cf (satisfy-premises (rule-premises rule) true)))
      (when (true-p cf)
        (dolist (conclusion (rule-conclusions rule))
          (conclude conclusion (* cf (rule-cf rule)))
          cf))))))

(defun satisfy-premises (premises cf-so-far)
  "A list of premises is satisfied if they are all true.
  A combined cf is returned."
  ;; cf-so-far is an accumulator of certainty factors
  (cond ((null premises) cf-so-far)
        ((not (true-p cf-so-far)) false)
        (t (satisfy-premises
             (rest premises)
             (cf-and cf-so-far
                     (eval-condition (first premises)))))))

```

The function `eval-condition` evaluates a single condition, returning its certainty factor. If `find-out-p` is true, it first calls `find-out`, which may either query the user or apply appropriate rules. If `find-out-p` is false, it evaluates the condition using the current state of the data base. It does this by looking at each stored value for the parameter/instance pair and evaluating the operator on it. For example, if the condition is `(temp patient > 98.6)` and the values for `temp` for the current patient are `((98 .3) (99 .6) (100 .1))`, then `eval-condition` will test each of the values 98, 99, and 100 against 98.6 using the `>` operator. This test will succeed twice, so the resulting certainty factor is $.6 + .1 = .7$.

The function `reject-premise` is designed as a quick test to eliminate a rule. As such, it calls `eval-condition` with `find-out-p nil`, so it will reject a premise only if it is clearly false without seeking additional information.

If a rule's premises are true, then the conclusions are added to the data base by `conclude`. Note that `is` is the only operator allowed in conclusions. `is` is just an alias for `equal`.

```

(defun eval-condition (condition &optional (find-out-p t))
  "See if this condition is true, optionally using FIND-OUT
  to determine unknown parameters."
  (multiple-value-bind (parm inst op val)
    (parse-condition condition)

```

```

    (when find-out-p
      (find-out parm inst))
    ;; Add up all the (val cf) pairs that satisfy the test
    (loop for pair in (get-vals parm inst)
          when (funcall op (first pair) val)
            sum (second pair))))

(defun reject-premise (premise)
  "A premise is rejected if it is known false, without
  needing to call find-out recursively."
  (false-p (eval-condition premise nil)))

(defun conclude (conclusion cf)
  "Add a conclusion (with specified certainty factor) to DB."
  (multiple-value-bind (parm inst op val)
    (parse-condition conclusion)
    (update-cf parm inst val cf)))

(defun is (a b) (equal a b))

```

All conditions are of the form: (*parameter instance operator value*). For example: (morphology organism is rod). The function `parse-condition` turns a list of this form into four values. The trick is that it uses the data base to return the current instance of the context, rather than the context name itself:

```

(defun parse-condition (condition)
  "A condition is of the form (parm inst op val).
  So for (age patient is 21), we would return 4 values:
  (age patient-1 is 21), where patient-1 is the current patient."
  (values (first condition)
          (get-db (second condition))
          (third condition)
          (fourth condition)))

```

At this point a call like `(find-out 'identity 'organism-1)` would do the right thing only if we had somehow entered the proper information on the current patient, culture, and organism. The function `get-context-data` makes sure that each context is treated in order. First an instance is created, then `find-out` is used to determine both the initial data parameters and the goals. The findings for each goal are printed, and the program asks if there is another instance of this context. Finally, we also need a top-level function, `emycin`, which just clears the data base before calling `get-context-data`.

```

(defun emycin (contexts)
  "An Expert-System Shell. Accumulate data for instances of each
  context, and solve for goals. Then report the findings."
  (clear-db)
  (get-context-data contexts))

(defun get-context-data (contexts)
  "For each context, create an instance and try to find out
  required data. Then go on to other contexts, depth first,
  and finally ask if there are other instances of this context."
  (unless (null contexts)
    (let* ((context (first contexts))
           (inst (new-instance context)))
      (put-db 'current-rule 'initial)
      (mapc #'find-out (context-initial-data context))
      (put-db 'current-rule 'goal)
      (mapc #'find-out (context-goals context))
      (report-findings context inst)
      (get-context-data (rest contexts))
      (when (y-or-n-p "Is there another ~a?"
                    (context-name context))
          (get-context-data contexts))))))

```

16.6 Interacting with the Expert

At this point all the serious computational work is done: we have defined a backward-chaining rule mechanism that deals with uncertainty, caching, questions, and contexts. But there is still quite a bit of work to do in terms of input/output interaction. A programming language needs only to interface with programmers, so it is acceptable to make the programmer do all the work. But an expert-system shell is supposed to alleviate (if not abolish) the need for programmers. Expert-system shells really have two classes of users: the experts use the shell when they are developing the system, and the end users or clients use the resulting expert system when it is completed. Sometimes the expert can enter knowledge directly into the shell, but more often it is assumed the expert will have the help of a *knowledge engineer*—someone who is trained in the use of the shell and in eliciting knowledge, but who need not be either an expert in the domain or an expert programmer.

In our version of EMYCIN, we provide only the simplest tools for making the expert's job easier. The macros `defcontext` and `defparm`, defined above, are a little easier than calling `make-context` and `make-parm` explicitly, but not much. The macro `defrule` defines a rule and checks for some obvious errors:


```

(defmacro defrule (number &body body)
  "Define a rule with conditions, a certainty factor, and
  conclusions. Example: (defrule R001 if ... then .9 ...)"
  (assert (eq (first body) 'if))
  (let* ((then-part (member 'then body))
         (premises (ldiff (rest body) then-part))
         (conclusions (rest2 then-part))
         (cf (second then-part)))
    ;; Do some error checking:
    (check-conditions number premises 'premise)
    (check-conditions number conclusions 'conclusion)
    (when (not (cf-p cf))
      (warn "Rule ~a: Illegal certainty factor: ~a" number cf))
    ;; Now build the rule:
    `(put-rule
      (make-rule :number ',number :cf ,cf :premises ',premises
                 :conclusions ',conclusions))))

```

The function `check-conditions` makes sure that each rule has at least one premise and conclusion, that each condition is of the right form, and that the value of the condition is of the right type for the parameter. It also checks that conclusions use only the operator `is`:

```

(defun check-conditions (rule-num conditions kind)
  "Warn if any conditions are invalid."
  (when (null conditions)
    (warn "Rule ~a: Missing ~a" rule-num kind))
  (dolist (condition conditions)
    (when (not (consp condition))
      (warn "Rule ~a: Illegal ~a: ~a" rule-num kind condition))
    (multiple-value-bind (parm inst op val)
      (parse-condition condition)
      (declare (ignore inst))
      (when (and (eq kind 'conclusion) (not (eq op 'is)))
        (warn "Rule ~a: Illegal operator (~a) in conclusion: ~a"
              rule-num op condition))
      (when (not (typep val (parm-type parm)))
        (warn "Rule ~a: Illegal value (~a) in ~a: ~a"
              rule-num val kind condition))))))

```

The real EMYCIN had an interactive environment that prompted the expert for each context, parameter, and rule. Randall Davis (1977, 1979, Davis and Lenat 1982) describes the TEIRESIAS program, which helped experts enter and debug rules.

16.7 Interacting with the Client

Once the knowledge is in, we need some way to get it out. The client wants to run the system on his or her own problem and see two things: a solution to the problem, and an explanation of why the solution is reasonable. EMYCIN provides primitive facilities for both of these. The function `report-findings` prints information on all the goal parameters for a given instance:

```
(defun report-findings (context inst)
  "Print findings on each goal for this instance."
  (when (context-goals context)
    (format t "~&Findings for ~a:" (inst-name inst))
    (dolist (goal (context-goals context))
      (let ((values (get-vals goal inst)))
        ;; If there are any values for this goal,
        ;; print them sorted by certainty factor.
        (if values
            (format t "~& ~a:~{~{ ~a (~,3f) ~}~}" goal
                    (sort (copy-list values) #'> :key #'second))
            (format t "~& ~a: unknown" goal))))))
```

The only explanation facility our version of EMYCIN offers is a way to see the current rule. If the user types `rule` in response to a query, a pseudo-English translation of the current rule is printed. Here is a sample rule and its translation:

```
(defrule 52
  if (site culture is blood)
    (gram organism is neg)
    (morphology organism is rod)
    (burn patient is serious)
  then .4
  (identity organism is pseudomonas))
```

Rule 52:

If

- 1) THE SITE OF THE CULTURE IS BLOOD
- 2) THE GRAM OF THE ORGANISM IS NEG
- 3) THE MORPHOLOGY OF THE ORGANISM IS ROD
- 4) THE BURN OF THE PATIENT IS SERIOUS

Then there is weakly suggestive evidence (0.4) that

- 1) THE IDENTITY OF THE ORGANISM IS PSEUDOMONAS

The function `print-rule` generates this translation:

```

(defun print-rule (rule &optional (stream t) depth)
  (declare (ignore depth))
  (format stream "~&Rule ~a:~& If" (rule-number rule))
  (print-conditions (rule-premises rule) stream)
  (format stream "~& Then ~a (~a) that"
    (cf->english (rule-cf rule)) (rule-cf rule))
  (print-conditions (rule-conclusions rule) stream))

(defun print-conditions (conditions &optional
  (stream t) (num 1))
  "Print a list of numbered conditions."
  (dolist (condition conditions)
    (print-condition condition stream num)))

(defun print-condition (condition stream number)
  "Print a single condition in pseudo-English."
  (format stream "~& ~d){ ~a~}" number
    (let ((parm (first condition))
          (inst (second condition))
          (op (third condition))
          (val (fourth condition)))
      (case val
        (YES '(the ,inst ,op ,parm))
        (NO '(the ,inst ,op not ,parm))
        (T '(the ,parm of the ,inst ,op ,val))))))

(defun cf->english (cf)
  "Convert a certainty factor to an English phrase."
  (cond ((= cf 1.0) "there is certain evidence")
        (> cf .8) "there is strongly suggestive evidence")
        (> cf .5) "there is suggestive evidence")
        (> cf 0.0) "there is weakly suggestive evidence")
        (= cf 0.0) "there is NO evidence either way")
        (< cf 0.0) (concatenate 'string (cf->english (- cf))
                                " AGAINST the conclusion"))))

```

If the user types `why` in response to a query, a more detailed account of the same rule is printed. First, the premises that are already known are displayed, followed by the remainder of the rule. The parameter being asked for will always be the first premise in the remainder of the rule. The `current-rule` is stored in the data base by `use-rule` whenever a rule is applied, but it is also set by `get-context-data` to the atom `initial` or `goal` when the system is prompting for parameters. `print-why` checks for this case as well. Note the use of the `partition-if` function from page 256.

```

(defun print-why (rule parm)
  "Tell why this rule is being used. Print what is known,
  what we are trying to find out, and what we can conclude."
  (format t "~&[Why is the value of ~a being asked for?]" parm)
  (if (member rule '(initial goal))
      (format t "~&~a is one of the ~a parameters."
              parm rule)
      (multiple-value-bind (knowns unknowns)
          (partition-if #'(lambda (premise)
                          (true-p (eval-condition premise nil)))
                        (rule-premises rule))
        (when knowns
          (format t "~&It is known that:")
          (print-conditions knowns)
          (format t "~&Therefore,")
          (let ((new-rule (copy-rule rule)))
            (setf (rule-premises new-rule) unknowns)
            (print new-rule)))))))

```

That completes the definition of `emycin`. We are now ready to apply the shell to a specific domain, yielding the beginnings of an expert system.

16.8 MYCIN, A Medical Expert System

This section applies `emycin` to MYCIN's original domain: infectious blood disease. In our version of MYCIN, there are three contexts: first we consider a patient, then any cultures that have been grown from samples taken from the patient, and finally any infectious organisms in the cultures. The goal is to determine the identity of each organism. The real MYCIN was more complex, taking into account any drugs or operations the patient may previously have had. It also went on to decide the real question: what therapy to prescribe. However, much of this was done by special-purpose procedures to compute optimal dosages and the like, so it is not included here. The original MYCIN also made a distinction between current versus prior cultures, organisms, and drugs. All together, it had ten contexts to consider, while our version only has three:

```

(defun mycin ()
  "Determine what organism is infecting a patient."
  (emycin
   (list (defcontext patient (name sex age) ())
         (defcontext culture (site days-old) ())
         (defcontext organism () (identity))))))

```

These contexts declare that we will first ask each patient's name, sex, and age, and each culture's site and the number of days ago it was isolated. Organisms have no initial questions, but they do have a goal: to determine the identity of the organism.

The next step is to declare parameters for the contexts. Each parameter is given a type, and most are given prompts to improve the naturalness of the dialogue:

```

;;; Parameters for patient:
(defparm name patient t "Patient's name: " t read-line)
(defparm sex patient (member male female) "Sex:" t)
(defparm age patient number "Age:" t)
(defparm burn patient (member no mild serious)
  "Is ~a a burn patient? If so, mild or serious?" t)
(defparm compromised-host patient yes/no
  "Is ~a a compromised host?")

;;; Parameters for culture:
(defparm site culture (member blood)
  "From what site was the specimen for ~a taken?" t)
(defparm days-old culture number
  "How many days ago was this culture (~a) obtained?" t)

;;; Parameters for organism:
(defparm identity organism
  (member pseudomonas klebsiella enterobacteriaceae
  staphylococcus bacteroides streptococcus)
  "Enter the identity (genus) of ~a:" t)
(defparm gram organism (member acid-fast pos neg)
  "The gram stain of ~a:" t)
(defparm morphology organism (member rod coccus)
  "Is ~a a rod or coccus (etc.):")
(defparm aerobicity organism (member aerobic anaerobic))
(defparm growth-conformation organism
  (member chains pairs clumps))

```

Now we need some rules to help determine the identity of the organisms. The following rules are taken from Shortliffe 1976. The rule numbers refer to the pages on which they are listed. The real MYCIN had about 400 rules, dealing with a much wider variety of premises and conclusions.

```

(clear-rules)
(defrule 52
  if (site culture is blood)
    (gram organism is neg)
    (morphology organism is rod)
    (burn patient is serious)
  then .4
  (identity organism is pseudomonas))

```

```

(defrule 71
  if (gram organism is pos)
    (morphology organism is coccus)
    (growth-conformation organism is clumps)
  then .7
    (identity organism is staphylococcus))

(defrule 73
  if (site culture is blood)
    (gram organism is neg)
    (morphology organism is rod)
    (aerobicity organism is anaerobic)
  then .9
    (identity organism is bacteroides))

(defrule 75
  if (gram organism is neg)
    (morphology organism is rod)
    (compromised-host patient is yes)
  then .6
    (identity organism is pseudomonas))

(defrule 107
  if (gram organism is neg)
    (morphology organism is rod)
    (aerobicity organism is aerobic)
  then .8
    (identity organism is enterobacteriaceae))

(defrule 165
  if (gram organism is pos)
    (morphology organism is coccus)
    (growth-conformation organism is chains)
  then .7
    (identity organism is streptococcus))

```

Here is an example of the program in use:

```

> (mycin)
----- PATIENT-1 -----
Patient's name: Sylvia Fischer
Sex: female
Age: 27
----- CULTURE-1 -----
From what site was the specimen for CULTURE-1 taken? blood
How many days ago was this culture (CULTURE-1) obtained? 3
----- ORGANISM-1 -----
Enter the identity (genus) of ORGANISM-1: unknown
The gram stain of ORGANISM-1: ?

```

A GRAM must be of type (MEMBER ACID-FAST POS NEG)
 The gram stain of ORGANISM-1: neg

The user typed ? to see the list of valid responses. The dialog continues:

Is ORGANISM-1 a rod or coccus (etc.): rod
 What is the AEROBICITY of ORGANISM-1? why
 [Why is the value of AEROBICITY being asked for?]
 It is known that:
 1) THE GRAM OF THE ORGANISM IS NEG
 2) THE MORPHOLOGY OF THE ORGANISM IS ROD
 Therefore,
 Rule 107:
 If
 1) THE AEROBICITY OF THE ORGANISM IS AEROBIC
 Then there is suggestive evidence (0.8) that
 1) THE IDENTITY OF THE ORGANISM IS ENTEROBACTERIACEAE

The user wants to know why the system is asking about the organism's aerobicity. The reply shows the current rule, what is already known about the rule, and the fact that if the organism is aerobic, then we can conclude something about its identity. In this hypothetical case, the organism is in fact aerobic:

What is the AEROBICITY of ORGANISM-1? aerobic
 Is Sylvia Fischer a compromised host? yes
 Is Sylvia Fischer a burn patient? If so, mild or serious? why
 [Why is the value of BURN being asked for?]
 It is known that:
 1) THE SITE OF THE CULTURE IS BLOOD
 2) THE GRAM OF THE ORGANISM IS NEG
 3) THE MORPHOLOGY OF THE ORGANISM IS ROD
 Therefore,
 Rule 52:
 If
 1) THE BURN OF THE PATIENT IS SERIOUS
 Then there is weakly suggestive evidence (0.4) that
 1) THE IDENTITY OF THE ORGANISM IS PSEUDOMONAS
 Is Sylvia Fischer a burn patient? If so, mild or serious? serious
 Findings for ORGANISM-1:
 IDENTITY: ENTEROBACTERIACEAE (0.800) PSEUDOMONAS (0.760)

The system used rule 107 to conclude the identity might be enterobacteriaceae. The certainty is .8, the certainty for the rule itself, because all the conditions were known to be true with certainty. Rules 52 and 75 both support the hypothesis of pseudomonas. The certainty factors of the two rules, .6 and .4, are combined by the

formula $.6 + .4 - (.6 \times .4) = .76$. After printing the findings for the first organism, the system asks if another organism was obtained from this culture:

```
Is there another ORGANISM? (Y or N) Y
----- ORGANISM-2 -----
Enter the identity (genus) of ORGANISM-2: unknown
The gram stain of ORGANISM-2: (neg .8 pos .2)
Is ORGANISM-2 a rod or coccus (etc.): rod
What is the AEROBICITY of ORGANISM-2? anaerobic
```

For the second organism, the lab test was inconclusive, so the user entered a qualified answer indicating that it is probably gram-negative, but perhaps gram-positive. This organism was also a rod but was anaerobic. Note that the system does not repeat questions that it already knows the answers to. In considering rules 75 and 52 it already knows that the culture came from the blood, and that the patient is a compromised host and a serious burn patient. In the end, rule 73 contributes to the bacteroides conclusion, and rules 75 and 52 again combine to suggest pseudomonas, although with a lower certainty factor, because the neg finding had a lower certainty factor:

```
Findings for ORGANISM-2:
IDENTITY: BACTEROIDES (0.720) PSEUDOMONAS (0.646)
```

Finally, the program gives the user the opportunity to extend the context tree with new organisms, cultures, or patients:

```
Is there another ORGANISM? (Y or N) N
Is there another CULTURE? (Y or N) N
Is there another PATIENT? (Y or N) N
```

The set of rules listed above do not demonstrate two important features of the system: the ability to backward-chain, and the ability to use operators other than `is` in premises.

If we add the following three rules and repeat the case shown above, then evaluating rule 75 will back-chain to rule 1, 2, and finally 3 trying to determine if the patient is a compromised host. Note that the question asked will be "What is Sylvia Fischer's white blood cell count?" and not "Is the white blood cell count of Sylvia Fischer < 2.5?" The latter question would suffice for the premise at hand, but it would not be as useful for other rules that might refer to the WBC.

```
(defparm wbc patient number
 "What is ~a's white blood cell count?")
```



```
(defrule 1
  if (immunosuppressed patient is yes)
  then 1.0 (compromised-host patient is yes))

(defrule 2
  if (leukopenia patient is yes)
  then 1.0 (immunosuppressed patient is yes))

(defrule 3
  if (wbc patient < 2.5)
  then .9 (leukopenia patient is yes))
```

16.9 Alternatives to Certainty Factors

Certainty factors are a compromise. The good news is that a system based on rules with certainty factors requires the expert to come up with only a small set of numbers (one for each rule) and will allow fast computation of answers. The bad news is that the answer computed may lead to irrational decisions.

Certainty factors have been justified by their performance (MYCIN performed as well or better than expert doctors) and by intuitive appeal (they satisfy the criteria listed on page 534). However, they are subject to paradoxes where they compute bizarre results (as in Exercise 16.1, page 536). If the rules that make up the knowledge base are designed in a modular fashion, then problems usually do not arise, but it is certainly worrisome that the answers may be untrustworthy.

Before MYCIN, most reasoning with uncertainty was done using probability theory. The laws of probability—in particular, Bayes's law—provide a well-founded mathematical formalism that is not subject to the inconsistencies of certainty factors. Indeed, probability theory can be shown to be the only formalism that leads to rational behavior, in the sense that if you have to make a series of bets on some uncertain events, combining information with probability theory will give you the highest expected value for your bets. Despite this, probability theory was largely set aside in the mid-1970s. The argument made by Shortliffe and Buchanan (1975) was that probability theory required too many conditional probabilities, and that people were not good at estimating these. They argued that certainty factors were intuitively easier to deal with. Other researchers of the time shared this view. Shafer, with later refinements by Dempster, created a theory of belief functions that, like certainty factors, represented a combination of the belief for and against an event. Instead of representing an event by a single probability or certainty, Dempster-Shafer theory maintains two numbers, which are analagous to the lower and upper bound on the probability. Instead of a single number like .5, Dempster-Shafer theory would have an interval like [.4,.6] to represent a range of probabilities. A complete lack of knowledge would be represented by the range [0,1]. A great deal of effort in the late 1970s

and early 1980s was invested in these and other nonprobabilistic theories. Another example is Zadeh's fuzzy set theory, which is also based on intervals.

There is ample evidence that people have difficulty with problems involving probability. In a very entertaining and thought-provoking series of articles, Tversky and Kahneman (1974, 1983, 1986) show how people make irrational choices when faced with problems that are quite simple from a mathematical viewpoint. They liken these errors in choice to errors in visual perception caused by optical illusions. Even trained doctors and statisticians are subject to these errors.

As an example, consider the following scenario. Adrian and Dominique are to be married. Adrian goes for a routine blood test and is told that the results are positive for a rare genetic disorder, one that strikes only 1 in 10,000 people. The doctor says that the test is 99% accurate—it gives a false positive reading in only 1 in 100 cases. Adrian is despondent, being convinced that the probability of actually having the disease is 99%. Fortunately, Dominique happens to be a Bayesian, and quickly reassures Adrian that the chance is more like 1%. The reasoning is as follows: Take 10,001 people at random. Of these, only 1 is expected to have the disease. That person could certainly expect to test positive for the disease. But if the other 10,000 people all took the blood test, then 1% of them, or 100 people would also test positive. Thus, the chance of actually having the disease given that one tests positive is 1/101. Doctors are trained in this kind of analysis, but unfortunately many of them continue to reason more like Adrian than Dominique.

In the late 1980s, the tide started to turn back to subjective Bayesian probability theory. Cheeseman (1985) showed that, while Dempster-Shafer theory looks like it can, in fact it cannot help you make better decisions than probability theory. Heckerman (1986) re-examined MYCIN's certainty factors, showing how they could be interpreted as probabilities. Judea Pearl's 1988 book is an eloquent defense of probability theory. He shows that there are efficient algorithms for combining and propagating probabilities, as long as the network of interdependencies does not contain loops. It seems likely that uncertain reasoning in the 1990s will be based increasingly on Bayesian probability theory.


16.10 History and References

The MYCIN project is well documented in Buchanan and Shortliffe 1984. An earlier book, Shortliffe 1976, is interesting mainly for historical purposes. Good introductions to expert systems in general include Weiss and Kulikowski 1984, Waterman 1986, Luger and Stubblefield 1989, and Jackson 1990.


Dempster-Shafer evidence theory is presented enthusiastically in Gordon and Shortliffe 1984 and in a critical light in Pearl 1989/1978. Fuzzy set theory is presented in Zadeh 1979 and Dubois and Prade 1988.


Pearl (1988) captures most of the important points that lead to the renaissance of probability theory. Shafer and Pearl 1990 is a balanced collection of papers on all kinds of uncertain reasoning.


16.11 Exercises


 **Exercise 16.2[s]** Suppose the rule writer wanted to be able to use symbolic certainty factors instead of numbers. What would you need to change to support rules like this:


```
(defrule 100 if ... then true ...)  
(defrule 101 if ... then probably ...)
```


 **Exercise 16.3 [m]** Change `prompt-and-read-vals` so that it gives a better prompt for parameters of type yes/no.

 **Exercise 16.4 [m]** Currently, the rule writer can introduce a new parameter without defining it first. That is handy for rapid testing, but it means that the user of the system won't be able to see a nice English prompt, nor ask for the type of the parameter. In addition, if the rule writer simply misspells a parameter, it will be treated as a new one. Make a simple change to fix these problems.

 **Exercise 16.5 [d]** Write rules in a domain you are an expert in, or find and interview an expert in some domain, and write down rules coaxed from the expert. Evaluate your resulting system. Was it easier to develop your system with EMYCIN than it would have been without it?

 **Exercise 16.6 [s]** It is said that an early version of MYCIN asked if the patient was pregnant, even though the patient was male. Write a rule that would fix this problem.

 **Exercise 16.7 [m]** To a yes/no question, what is the difference between yes and (no -1)? What does this suggest?

 **Exercise 16.8 [m]** What happens if the user types *why* to the prompt about the patient's name? What happens if the expert wants to have more than one context with a name parameter? If there is a problem, fix it.

The remaining exercises discuss extensions that were in the original EMYCIN, but were not implemented in our version. Implementing all the extensions will result in a system that is very close to the full power of EMYCIN. These extensions are discussed in chapter 3 of Buchanan and Shortliffe 1984.

? **Exercise 16.9 [h]** Add a spelling corrector to `ask-val`s. If the user enters an invalid reply, and the parameter type is a member expression, check if the reply is “close” in spelling to one of the valid values, and if so, use that value. That way, the user can type just `entero` instead of `enterobacteriaceae`. You may experiment with the definition of “close,” but you should certainly allow for prefixes and at least one instance of a changed, missing, inserted, or transposed letter.

? **Exercise 16.10 [m]** Indent the output for each new branch in the context tree. In other words, have the prompts and findings printed like this:

```

----- PATIENT-1 -----
Patient's name: Sylvia Fischer
Sex: female
Age: 27
      ----- CULTURE-1 -----
      From what site was the specimen for CULTURE-1 taken? blood
      How many days ago was this culture (CULTURE-1) obtained? 3
            ----- ORGANISM-1 -----
            Enter the identity (genus) of ORGANISM-1: unknown
            The gram stain of ORGANISM-1: neg
            ...
            Findings for ORGANISM-1:
              IDENTITY: ENTEROBACTERIACEAE (0.800) PSEUDOMONAS (0.760)
              Is there another ORGANISM? (Y or N) N
            Is there another CULTURE? (Y or N) N
          Is there another PATIENT? (Y or N) N

```

? **Exercise 16.11 [h]** We said that our `emycin` looks at all possible rules for each parameter, because there is no telling how a later rule may affect the certainty factor. Actually, that is not quite true. If there is a rule that leads to a conclusion with certainty 1, then no other rules need be considered. This was called a *unity path*. Modify the program to look for unity paths first.

? **Exercise 16.12 [m]** Depending on whether a parameter is in `initial-data` or not, all the relevant rules are run either before or after asking the user for the value of the parameter. But there are some cases when not all initial data parameters

should be asked for. As an example, suppose that `identity` and `gram` were initial data parameters of `organism`. If the user gave a positive answer for `identity`, then it would be wasteful to ask for the `gram` parameter, since it could be determined directly from rules. After receiving complaints about this problem, a system of *antecedent rules* was developed. These rules were always run first, before asking questions. Implement antecedent rules.

- ?** **Exercise 16.13 [h]** It is useful to be able to write *default rules* that fill in a value after all other rules have failed to determine one. A default rule looks like this:

```
(defrule n if (parm inst unknown) then (parm inst is default))
```

It may also have other conjuncts in the premise. Beside details like writing the `unknown` operator, the difficult part is in making sure that these rules get run at the right time (after other rules have had a chance to fill in the parameter), and that infinite loops are avoided.

- ?** **Exercise 16.14 [h]** The context tree proved to be a limitation. Eventually, the need arose for a rule that said, "If any of the organisms in a culture has property X, then the culture has property Y." Implement a means of checking for some or every instance of a context.

- ?** **Exercise 16.15 [m]** As the rule base grew, it became increasingly hard to remember the justification for previous rules. Implement a mechanism that keeps track of the author and date of creation of each rule, and allows the author to add documentation explaining the rationale for the rule.

- ?** **Exercise 16.16 [m]** It is difficult to come up with the perfect prompt for each parameter. One solution is not to insist that one prompt fits all users, but rather to allow the expert to supply three different prompts: a normal prompt, a verbose prompt (or reprompt) for when the user replies with a `?`, and a terse prompt for the experienced user. Modify `defparm` to accommodate this concept, add a command for the user to ask for the terse prompts, and change `ask-vals` to use the proper prompt.

The remaining exercises cover three additional replies the user can make: `how`, `stop`, and `change`.

- ?** **Exercise 16.17 [d]** In addition to `why` replies, `EMYCIN` also allowed for `how` questions. The user can ask how the value of a particular parameter/instance pair was determined, and the system will reply with a list of rules and the evidence they supplied for

or against each value. Implement this mechanism. It will require storing additional information in the data base.

? **Exercise 16.18 [m]** There was also a stop command that immediately halted the session. Implement it.

? **Exercise 16.19 [d]** The original EMYCIN also had a change command to allow the user to change the answer to certain questions without starting all over. Each question was assigned a number, which was printed before the prompt. The command change, followed by a list of numbers, causes the system to look up the questions associated with each number and delete the answer to these questions. The system also throws away the entire context tree and all derived parameter values. At that point the entire consultation is restarted, using only the data obtained from the unchanged questions. Although it may seem wasteful to start over from the beginning, it will not be wasteful of the user's time, since correct answers will not be asked again.

Identify what needs to be altered to implement change and make the alterations.

? **Exercise 16.20 [h]** Change the definition of cf-and and cf-or to use fuzzy set theory instead of certainty factors. Do the same for Dempster-Shafer theory.

16.12 Answers

Answer 16.1 Because EMYCIN assumes independence, each reading of the same headline would increase the certainty factor. The following computation shows that 298 more copies would be needed to reach .95 certainty. A more sophisticated reasoner would realize that multiple copies of a newspaper are completely dependent on one another, and would not change the certainty with each new copy.

```
> (loop for cf = .01 then (cf-or .01 cf)
    until (> cf .95)
    count t)
298
```

Answer 16.2 The defrule expands to (make-rule :number '101 :cf true ...); that is, the certainty factor is unquoted, so it is already legal to use true as a certainty factor! To support probably and other hedges, just define new constants.

Answer 16.4 Just make the default parameter type be `nil` (by changing `t` to `nil` in `parm-type`). Then any rule that uses an undefined parameter will automatically generate a warning.

Answer 16.6

```
(defrule 4
  if (sex patient is male)
  then -1 (pregnant patient is yes))
```

Answer 16.7 Logically, there should be no difference, but to EMYCIN there is a big difference. EMYCIN would not complain if you answered (yes 1 no 1). This suggests that the system should have some way of dealing with mutually exclusive answers. One way would be to accept only yes responses for Boolean parameters, but have the input routine translate no to (yes -1) and (no *cf*) to (yes 1-*cf*). Another possibility would be to have `update-cf` check to see if any certainty factor on a mutually exclusive value is 1, and if so, change the other values to -1.

Answer 16.18 Add the clause (stop (throw 'stop nil)) to the case statement in `ask-vals` and wrap a (catch 'stop ...) around the code in `emycin`.

CHAPTER 17

Line-Diagram Labeling by Constraint Satisfaction

It is wrong to think of Waltz's work only as a statement of the epistemology of line drawings of polyhedra. Instead I think it is an elegant case study of a paradigm we can expect to see again and again.

—Patrick Winston

The Psychology of Computer Vision (1975)

This book touches only the areas of AI that deal with abstract reasoning. There is another side of AI, the field of *robotics*, that deals with interfacing abstract reasoning with the real world through sensors and motors. A robot receives input from cameras, microphones, sonar, and touch-sensitive devices, and produces “output” by moving its appendages or generating sounds. The real world is a messier place than the abstract worlds we have been covering. A robot must deal with noisy data, faulty components, and other agents and events in the world that can affect changes in the environment.

Computer vision is the subfield of robotics that deals with interpreting visual information. Low-level vision takes its input directly from a camera and detects lines, regions and textures. We will not be concerned with this. High-level vision uses the findings of the low-level component to build a three-dimensional model of the objects depicted in the scene. This chapter covers one small aspect of high-level vision.

17.1 The Line-Labeling Problem

In this chapter we look at the line-diagram labeling problem: Given a list of lines and the vertexes at which they intersect, how can we determine what the lines represent? For example, given the nine lines in figure 17.1, how can we interpret the diagram as a cube?

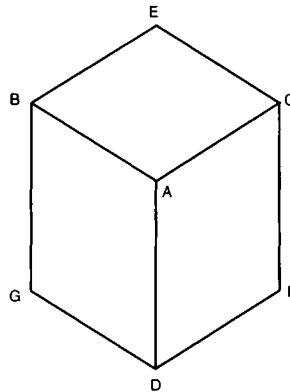


Figure 17.1: A Cube

Before we can arrive at an interpretation, we have to agree on what the candidates are. After all, figure 17.1 could be just a hexagon with three lines in the middle. For the purposes of this chapter, we will consider only diagrams that depict one or more *polyhedra*—three-dimensional solid figures whose surfaces are flat faces bounded by straight lines. In addition, we will only allow *trihedral* vertexes. That is, each vertex must be formed by the intersection of three faces, as in the corner of a cube, where the top, front, and side of the cube come together. A third restriction on diagrams is that no so-called *accidental* vertexes are allowed. For example, figure 17.1 might be a picture of three different cubes hanging in space, which just happen to line up so that the edge of one is aligned with the edge of another from our viewpoint. We will assume that this is not the case.

Given a diagram that fits these three restrictions, our goal is to identify each line, placing it in one of three classes:

1. A convex line separates two visible faces of a polyhedron such that a line from one face to the other would lie inside the polyhedron. It will be marked with a plus sign: +.
2. A concave line separates two faces of two polyhedra such that a line between the two spaces would pass through empty space. It will be marked with a minus sign: -.
3. A boundary line denotes the same physical situation as a convex line, but the diagram is oriented in such a way that only one of the two faces of the polyhedron is visible. Thus, the line marks the boundary between the polyhedron and the background. It will be marked with an arrow: \rightarrow . Traveling along the line from the tail to the point of the arrow, the polyhedron is on the right, and the background is on the left.

Figure 17.2 shows a labeling of the cube using these conventions. Vertex A is the near corner of the cube, and the three lines coming out of it are all convex lines. Lines GD and DF are concave lines, indicating the junction between the cube and the surface on which it is resting. The remaining lines are boundary lines, indicating that there is no physical connection between the cube and the background there, but that there are other sides of the cube that cannot be seen.

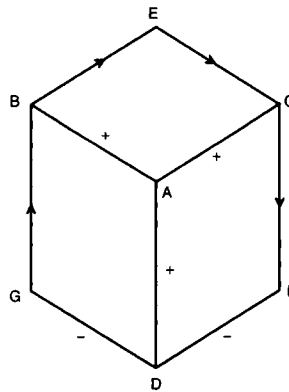


Figure 17.2: A Line-labeled Cube

The line-labeling technique developed in this chapter is based on a simple idea. First we enumerate all the possible vertexes, and all the possible labelings for each

vertex. It turns out there are only four different vertex types in the trihedral polygon world. We call them L, Y, W, and T vertexes, because of their shape. The Y and W vertexes are also known as forks and arrows, respectively. The vertexes are listed in figure 17.3. Each vertex imposes some constraints on the lines that compose it. For example, in a W vertex, the middle line can be labeled with a + or -, but not with an arrow.

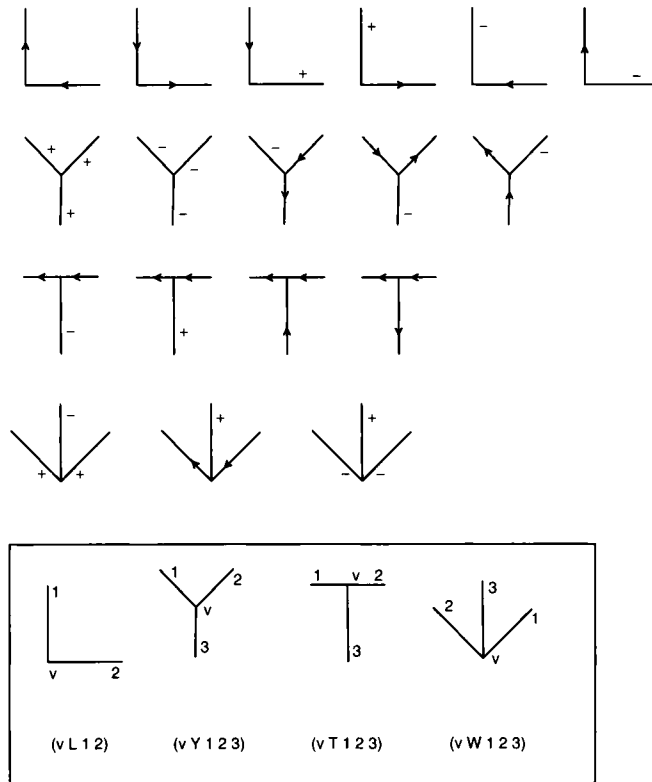


Figure 17.3: The Possible Vertexes and Labels

Each line connects two vertexes, so it must satisfy both constraints. This suggests a simple algorithm for labeling a diagram based on constraint propagation: First, label each vertex with all the possible labelings for the vertex type. An L vertex has six possibilities, Y has five, T has four, and W has three. Next, pick a vertex, V. Consider a neighboring vertex, N (that is, N and V are connected by a line). N will also have a set of possible labelings. If N and V agree on the possible labelings for the line between them, then we have gained nothing. But if the intersection of the two possibility sets is smaller than V's possibility set, then we have found a constraint on

the diagram. We adjust N and V 's possible labelings accordingly. Every time we add a constraint at a vertex, we repeat the whole process for all the neighboring vertexes, to give the constraint a chance to propagate as far as possible. When every vertex has been visited at least once and there are no more constraints to propagate, then we are done.

Figure 17.4 illustrates this process. On the left we start with a cube. All vertexes have all possible labelings, except that we know line GD is concave ($-$), indicating that the cube is resting on a surface. This constrains vertex D in such a way that line DA must be convex ($+$). In the middle picture the constraint on vertex D has propagated to vertex A , and in the right-hand picture it propagates to vertex B . Soon, the whole cube will be uniquely labeled.

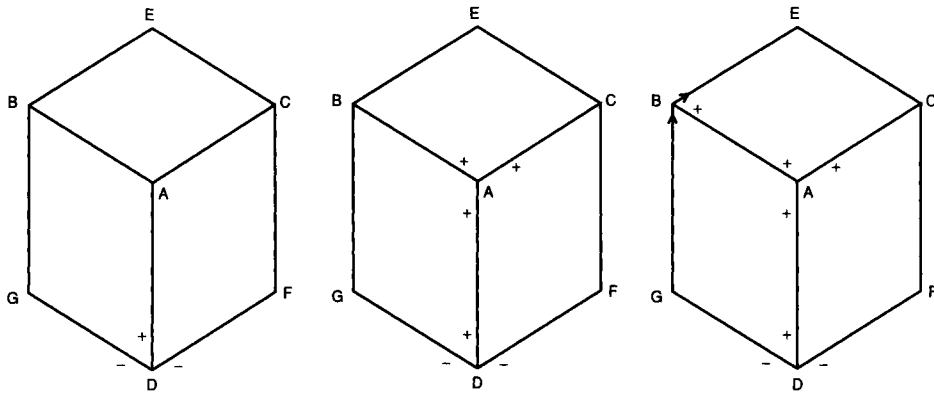


Figure 17.4: Propagating Constraints

Many diagrams will be labeled uniquely by this constraint propagation process. Some diagrams, however, are ambiguous. They will still have multiple labelings after constraint propagation has finished. In this case, we can search for a solution. Simply choose an ambiguous vertex, choose one of the possible labelings for that vertex, and repeat the constraint propagation/search process. Keep going until the diagram is either unambiguous or inconsistent.

That completes the sketch of the line-labeling algorithm. We are now ready to implement a labeling program. It's glossary is in figure 17.5.

The two main data structures are the `diagram` and the `vertex`. It would have been possible to implement a data type for `lines`, but it is not necessary: lines are defined implicitly by the two vertexes at their end points.

A `diagram` is completely specified by its list of vertexes, so the structure `diagram` needs only one slot. A `vertex`, on the other hand, is a more complex structure. Each vertex has an identifying name (usually a single letter), a vertex type (L, Y, W, or T), a

print-labelings	Top-Level Functions Label the diagram by propagating constraints and then searching.
diagram vertex	Data Types A diagram is a list of vertexes. A vertex has a name, type, and list of neighbors and labelings.
find-labelings propagate-constraints consistent-labelings search-solutions defdiagram diagram ground	Major Functions Do the same constraint propagation, but don't print anything. Reduce the number of labelings on vertex by considering neighbors. Return the set of labelings that are consistent with neighbors. Try all labelings for one ambiguous vertex, and propagate. (macro) Define a diagram. Retrieve a diagram stored by name. Attach the line between the two vertexes to the ground.
labels-for reverse-label ambiguous-vertex-p number-of-labelings find-vertex matrix-transpose possible-labelings print-vertex show-vertex show-diagram construct-diagram construct-vertex make-copy-diagram check-diagram	Auxiliary Functions Return all the labels for the line going to vertex. Reverse left and right on arrow labels. A vertex is ambiguous if it has more than one labeling. Number of labels on a vertex. Find the vertex with the given name. Turn a matrix on its side. The list of possible labelings for a given vertex type. Print a vertex in the short form. Print a vertex in a long form, on a new line. Print a diagram in a long form. Include a title. Build a new diagram from a set of vertex descriptions. Build a new vertex from a vertex description. Make a copy of a diagram, preserving connectivity. Check if the description appears consistent.

Figure 17.5: Glossary for the Line-Labeling Program

list of neighboring vertexes, and a list of possible labelings. A labeling is a list of line labels. For example, a Y vertex will initially have a list of five possible labelings. If it is discovered that the vertex is the interior of a concave corner, then it will have the single labeling (- - -). We give type information on the slots of vertex because it is a complicated data type. The syntax of `defstruct` is such that you cannot specify a `:type` without first specifying a default value. We chose `L` as the default value for the type slot at random, but note that it would have been an error to give `nil` as the default value, because `nil` is not of the right type.

```
(defstruct diagram "A diagram is a list of vertexes." vertexes)
(defstruct (vertex (:print-function print-vertex))
  (name      nil :type atom)
  (type      'L :type (member L Y W T))
  (neighbors nil :type list) ; of vertex
  (labelings nil :type list)) ; of lists of (member + - L R))))
```

An ambiguous vertex will have several labelings, while an unambiguous vertex has exactly one, and a vertex with no labelings indicates an impossible diagram. Initially we don't know which vertexes are what, so they all start with several possible labelings. Note that a labeling is a list, not a set: the order of the labels is significant and matches the order of the neighboring vertexes. The function `possible-labelings` gives a list of all possible labelings for each vertex type. We use R and L instead of arrows as labels, because the orientation of the arrows is significant. An R means that as you travel from the vertex to its neighbor, the polyhedron is on the right and the background object is on the left. Thus, an R is equivalent to an arrow pointing away from the vertex. The L is just the reverse.

```
(defun ambiguous-vertex-p (vertex)
  "A vertex is ambiguous if it has more than one labeling."
  (> (number-of-labelings vertex) 1))

(defun number-of-labelings (vertex)
  (length (vertex-labelings vertex)))

(defun impossible-vertex-p (vertex)
  "A vertex is impossible if it has no labeling."
  (null (vertex-labelings vertex)))

(defun impossible-diagram-p (diagram)
  "An impossible diagram is one with an impossible vertex."
  (some #'impossible-vertex-p (diagram-vertexes diagram)))

(defun possible-labelings (vertex-type)
  "The list of possible labelings for a given vertex type."
  ;; In these labelings, R means an arrow pointing away from
  ;; the vertex, L means an arrow pointing towards it.
  (case vertex-type
    ((L) '((R L) (L R) (+ R) (L +) (- L) (R -)))
    ((Y) '((+ + +) (- - -) (L R -) (- L R) (R - L)))
    ((T) '((R L +) (R L -) (R L L) (R L R)))
    ((W) '((L R +) (- - +) (+ + -))))
```

17.2 Combining Constraints and Searching

The main function `print-labelings` takes a diagram as input, reduces the number of labelings on each vertex by constraint propagation, and then searches for all consistent interpretations. Output is printed before and after each step.

```

(defun print-labelings (diagram)
  "Label the diagram by propagating constraints and then
  searching for solutions if necessary. Print results."
  (show-diagram diagram "~&The initial diagram is:")
  (every #'propagate-constraints (diagram-vertexes diagram))
  (show-diagram diagram
    "~2&After constraint propagation the diagram is:")
  (let* ((solutions (if (impossible-diagram-p diagram)
                        nil
                        (search-solutions diagram)))
         (n (length solutions)))
    (unless (= n 1)
      (format t "~2&There are ~r solution~:p:" n)
      (mapc #'show-diagram solutions)))
    (values))

```

The function `propagate-constraints` takes a vertex and considers the constraints imposed by neighboring vertexes to get a list of all the `consistent-labelings` for the vertex. If the number of consistent labelings is less than the number before we started, then the neighbors' constraints have had an effect on this vertex, so we propagate the new-found constraints on this vertex back to each neighbor. The function returns `nil` and thus immediately stops the propagation if there is an impossible vertex. Otherwise, propagation continues until there are no more changes to the labelings.

The whole propagation algorithm is started by a call to `every` in `print-labelings`, which propagates constraints from each vertex in the diagram. But it is not obvious that this is all that is required. After propagating from each vertex once, couldn't there be another vertex that needs relabeling? The only vertex that could possibly need relabeling would be one that had a neighbor changed since its last update. But any such vertex would have been visited by `propagate-constraint`, since we propagate to all neighbors. Thus, a single pass through the vertexes, compounded with recursive calls, will find and apply all possible constraints.

The next question worth asking is if the algorithm is guaranteed to terminate. Clearly, it is, because `propagate-constraints` can only produce recursive calls when it removes a labeling. But since there are a finite number of labelings initially (no more than six per vertex), there must be a finite number of calls to `propagate-constraints`.

```

(defun propagate-constraints (vertex)
  "Reduce the labelings on vertex by considering neighbors.
  If we can reduce, propagate the constraints to each neighbor."
  ;; Return nil only when the constraints lead to an impossibility
  (let ((old-num (number-of-labelings vertex)))
    (setf (vertex-labelings vertex) (consistent-labelings vertex))
    (unless (impossible-vertex-p vertex)
      (when (< (number-of-labelings vertex) old-num)
        (every #'propagate-constraints (vertex-neighbors vertex)))
      t)))

```

The function `consistent-labelings` is passed a vertex. It gets all the labels for this vertex from the neighboring vertexes, collecting them in `neighbor-labels`. It then checks all the labels on the current vertex, keeping only the ones that are consistent with all the neighbors' constraints. The auxiliary function `labels-for` finds the labels for a particular neighbor at a vertex, and `reverse-label` accounts for the fact that L and R labels are interpreted with respect to the vertex they point at.

```
(defun consistent-labelings (vertex)
  "Return the set of labelings that are consistent with neighbors."
  (let ((neighbor-labels
        (mapcar #'(lambda (neighbor) (labels-for neighbor vertex))
                (vertex-neighbors vertex))))
    ;; Eliminate labelings that don't have all lines consistent
    ;; with the corresponding line's label from the neighbor.
    ;; Account for the L-R mismatch with reverse-label.
    (find-all-if
     #'(lambda (labeling)
         (every #'member (mapcar #'reverse-label labeling)
                 neighbor-labels))
     (vertex-labelings vertex))))
```

Constraint propagation is often sufficient to yield a unique interpretation. But sometimes the diagram is still underconstrained, and we will have to search for solutions. The function `search-solutions` first checks to see if the diagram is ambiguous, by seeing if it has an ambiguous vertex, v . If the diagram is unambiguous, then it is a solution, and we return it (in a list, since `search-solutions` is designed to return a list of all solutions). Otherwise, for each of the possible labelings for the ambiguous vertex, we create a brand new copy of the diagram and set v 's labeling in the copy to one of the possible labelings. In effect, we are guessing that a labeling is a correct one. We call `propagate-constraints`; if it fails, then we have guessed wrong, so there are no solutions with this labeling. But if it succeeds, then we call `search-solutions` recursively to give us the list of solutions generated by this labeling.

```
(defun search-solutions (diagram)
  "Try all labelings for one ambiguous vertex, and propagate."
  ;; If there is no ambiguous vertex, return the diagram.
  ;; If there is one, make copies of the diagram trying each of
  ;; the possible labelings. Propagate constraints and append
  ;; all the solutions together.
  (let ((v (find-if #'ambiguous-vertex-p
                    (diagram-vertexes diagram))))
    (if (null v)
        (list diagram)
        (mapcan
         #'(lambda (v-labeling)
```



```

(let* ((diagram2 (make-copy-diagram diagram))
      (v2 (find-vertex (vertex-name v) diagram2)))
  (setf (vertex-labelings v2) (list v-labeling))
  (if (propagate-constraints v2)
      (search-solutions diagram2)
      nil)))
(vertex-labelings v))))

```

That's all there is to the algorithm; all that remains are some auxiliary functions. Here are three of them:

```

(defun labels-for (vertex from)
  "Return all the labels for the line going to vertex."
  (let ((pos (position from (vertex-neighbors vertex))))
    (mapcar #'(lambda (labeling) (nth pos labeling))
            (vertex-labelings vertex))))

(defun reverse-label (label)
  "Account for the fact that one vertex's right is another's left."
  (case label (L 'R) (R 'L) (otherwise label)))

(defun find-vertex (name diagram)
  "Find the vertex in the given diagram with the given name."
  (find name (diagram-vertexes diagram) :key #'vertex-name))

```

Here are the printing functions. `print-vertex` prints a vertex in short form. It obeys the `print` convention of returning the first argument. The functions `show-vertex` and `show-diagram` print more detailed forms. They obey the convention for `describe`-like functions of returning no values at all.

```

(defun print-vertex (vertex stream depth)
  "Print a vertex in the short form."
  (declare (ignore depth))
  (format stream "~a/~d" (vertex-name vertex)
          (number-of-labelings vertex))
  vertex)

(defun show-vertex (vertex &optional (stream t))
  "Print a vertex in a long form, on a new line."
  (format stream "& ~a ~d:" vertex (vertex-type vertex))
  (mapc #'(lambda (neighbor labels)
            (format stream " ~a~a=[~{~a~}]" (vertex-name vertex)
                    (vertex-name neighbor) labels))
        (vertex-neighbors vertex)
        (matrix-transpose (vertex-labelings vertex))))
  (values))

```

```
(defun show-diagram (diagram &optional (title "~2&Diagram:"))
  (stream t))
  "Print a diagram in a long form. Include a title."
  (format stream title)
  (mapc #'show-vertex (diagram-vertexes diagram))
  (let ((n (reduce #'* (mapcar #'number-of-labelings
                              (diagram-vertexes diagram)))))
    (when (> n 1)
      (format stream "~&For ~:d interpretation~:p." n))
    (values)))
```

Note that `matrix-transpose` is called by `show-vertex` to turn the matrix of labelings on its side. It works like this:

```
> (possible-labelings 'Y)
((+ + +)
 (- - -)
 (L R -)
 (- L R)
 (R - L))

> (matrix-transpose (possible-labelings 'Y))
((+ - L - R)
 (+ - R L -)
 (+ - - R L))
```

The implementation of `matrix-transpose` is surprisingly concise. It is an old Lisp trick, and well worth understanding:

```
(defun matrix-transpose (matrix)
  "Turn a matrix on its side."
  (if matrix (apply #'mapcar #'list matrix)))
```

The remaining code has to do with creating diagrams. We need some handy way of specifying diagrams. One way would be with a line-recognizing program operating on digitized input from a camera or bitmap display. Another possibility is an interactive drawing program using a mouse and bitmap display. But since there is not yet a Common Lisp standard for interacting with such devices, we will have to settle for a textual description. The macro `defdiagram` defines and names a diagram. The name is followed by a list of vertex descriptions. Each description is a list consisting of the name of a vertex, the vertex type (Y, A, L, or T), and the names of the neighboring vertexes. Here again is the `defdiagram` description for the cube shown in figure 17.6.

```
(defdiagram cube
  (a Y b c d)
  (b W g e a)
  (c W e f a)
  (d W f g a)
  (e L c b)
  (f L d c)
  (g L b d))
```

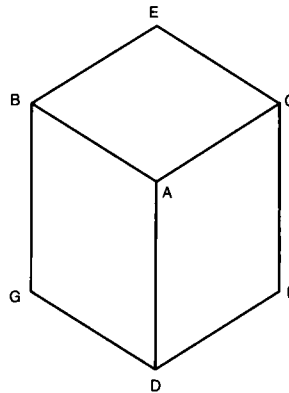


Figure 17.6: A Cube

The macro `defdiagram` calls `construct-diagram` to do the real work. It would be feasible to have `defdiagram` expand into a `defvar`, making the names be special variables. But then it would be the user's responsibility to make copies of such a variable before passing it to a destructive function. Instead, I use `put-diagram` and `diagram` to put and get diagrams in a table. `diagram` retrieves the named diagram and makes a copy of it. Thus, the user cannot corrupt the original diagrams stored in the table. Another possibility would be to have `defdiagram` expand into a function definition for `name` that returns a copy of the diagram. I chose to keep the diagram name space separate from the function name space, since names like `cube` make sense in both spaces.

```
(defmacro defdiagram (name &rest vertex-descriptors)
  "Define a diagram. A copy can be gotten by (diagram name)."
  `(put-diagram ',name (construct-diagram ',vertex-descriptors)))

(let ((diagrams (make-hash-table)))
```

```
(defun diagram (name)
  "Get a fresh copy of the diagram with this name."
  (make-copy-diagram (gethash name diagrams)))

(defun put-diagram (name diagram)
  "Store a diagram under a name."
  (setf (gethash name diagrams) diagram)
  name))
```

The function `construct-diagram` translates each vertex description, using `construct-vertex`, and then fills in the neighbors of each vertex.

```
(defun construct-diagram (vertex-descriptors)
  "Build a new diagram from a set of vertex descriptor."
  (let ((diagram (make-diagram)))
    ;; Put in the vertexes
    (setf (diagram-vertexes diagram)
          (mapcar #'construct-vertex vertex-descriptors))
    ;; Put in the neighbors for each vertex
    (dolist (v-d vertex-descriptors)
      (setf (vertex-neighbors (find-vertex (first v-d) diagram))
            (mapcar #'(lambda (neighbor)
                        (find-vertex neighbor diagram))
                    (v-d-neighbors v-d))))
    diagram))

(defun construct-vertex (vertex-descriptor)
  "Build the vertex corresponding to the descriptor."
  ;; Descriptors are like: (x L y z)
  (make-vertex
   :name (first vertex-descriptor)
   :type (second vertex-descriptor)
   :labelings (possible-labelings (second vertex-descriptor))))

(defun v-d-neighbors (vertex-descriptor)
  "The neighboring vertex names in a vertex descriptor."
  (rest (rest vertex-descriptor)))
```

The `defstruct` for `diagram` automatically creates the function `copy-diagram`, but it just copies each field, without copying the contents of each field. Thus we need `make-copy-diagram` to create a copy that shares no structure with the original.

```
(defun make-copy-diagram (diagram)
  "Make a copy of a diagram, preserving connectivity."
  (let* ((new (make-diagram
               :vertexes (mapcar #'copy-vertex
                                 (diagram-vertexes diagram))))
         ;; Put in the neighbors for each vertex
         (dolist (v (diagram-vertexes new))
           (setf (vertex-neighbors v)
                 (mapcar #'(lambda (neighbor)
                             (find-vertex (vertex-name neighbor) new))
                         (vertex-neighbors v))))
         new))
```

17.3 Labeling Diagrams

We are now ready to try labeling diagrams. First the cube:

```
> (print-labelings (diagram 'cube))
The initial diagram is:
A/5 Y: AB=[+-L-R] AC=[+-RL-] AD=[+--RL]
B/3 W: BG=[L-+] BE=[R-+] BA=[++-]
C/3 W: CE=[L-+] CF=[R-+] CA=[++-]
D/3 W: DF=[L-+] DG=[R-+] DA=[++-]
E/6 L: EC=[RL+L-R] EB=[LRR+L-]
F/6 L: FD=[RL+L-R] FC=[LRR+L-]
G/6 L: GB=[RL+L-R] GD=[LRR+L-]
For 29,160 interpretations.
```

```
After constraint propagation the diagram is:
A/1 Y: AB=[+] AC=[+] AD=[+]
B/2 W: BG=[L-] BE=[R-] BA=[++]
C/2 W: CE=[L-] CF=[R-] CA=[++]
D/2 W: DF=[L-] DG=[R-] DA=[++]
E/3 L: EC=[R-R] EB=[LL-]
F/3 L: FD=[R-R] FC=[LL-]
G/3 L: GB=[R-R] GD=[LL-]
For 216 interpretations.
```

There are four solutions:

Diagram:

A/1 Y: AB=[+] AC=[+] AD=[+]
 B/1 W: BG=[L] BE=[R] BA=[+]
 C/1 W: CE=[L] CF=[R] CA=[+]
 D/1 W: DF=[L] DG=[R] DA=[+]
 E/1 L: EC=[R] EB=[L]
 F/1 L: FD=[R] FC=[L]
 G/1 L: GB=[R] GD=[L]

Diagram:

A/1 Y: AB=[+] AC=[+] AD=[+]
 B/1 W: BG=[L] BE=[R] BA=[+]
 C/1 W: CE=[L] CF=[R] CA=[+]
 D/1 W: DF=[-] DG=[-] DA=[+]
 E/1 L: EC=[R] EB=[L]
 F/1 L: FD=[-] FC=[L]
 G/1 L: GB=[R] GD=[-]

Diagram:

A/1 Y: AB=[+] AC=[+] AD=[+]
 B/1 W: BG=[L] BE=[R] BA=[+]
 C/1 W: CE=[-] CF=[-] CA=[+]
 D/1 W: DF=[L] DG=[R] DA=[+]
 E/1 L: EC=[-] EB=[L]
 F/1 L: FD=[R] FC=[-]
 G/1 L: GB=[R] GD=[L]

Diagram:

A/1 Y: AB=[+] AC=[+] AD=[+]
 B/1 W: BG=[-] BE=[-] BA=[+]
 C/1 W: CE=[L] CF=[R] CA=[+]
 D/1 W: DF=[L] DG=[R] DA=[+]
 E/1 L: EC=[R] EB=[-]
 F/1 L: FD=[R] FC=[L]
 G/1 L: GB=[-] GD=[L]

The four interpretations correspond, respectively, to the cases where the cube is free floating, attached to the floor (GD and DF = -), attached to a wall on the right (EC and CF = -), or attached to a wall on the left (BG and BE = -). These are shown in figure 17.7. It would be nice if we could supply information about where the cube is attached, and see if we can get a unique interpretation. The function ground takes a diagram and modifies it by making one or more lines be grounded lines—lines that have a concave (-) label, corresponding to a junction with the ground.

We can see how this works on the cube:

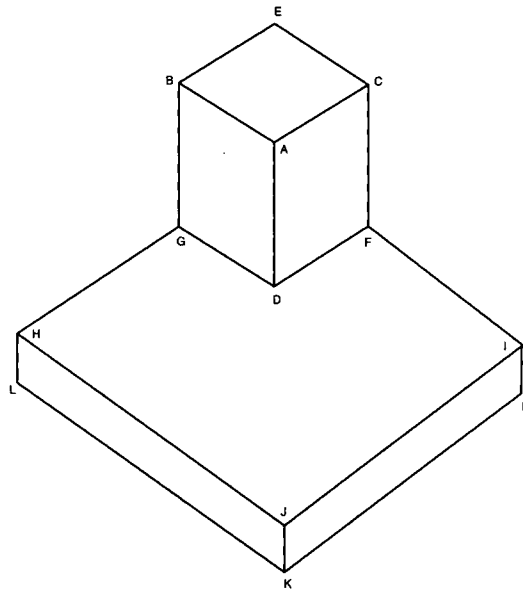


Figure 17.8: Cube on a Plate

```
> (print-labelings (ground (diagram 'cube) 'g 'd))
```

The initial diagram is:

```
A/5 Y: AB=[+-L-R] AC=[+-RL-] AD=[+--RL]
B/3 W: BG=[L-+] BE=[R-+] BA=[++-]
C/3 W: CE=[L-+] CF=[R-+] CA=[++-]
D/3 W: DF=[L-+] DG=[R-+] DA=[++-]
E/6 L: EC=[RL+L-R] EB=[LRR+L-]
F/6 L: FD=[RL+L-R] FC=[LRR+L-]
G/1 L: GB=[R] GD=[-]
```

For 4,860 interpretations.

After constraint propagation the diagram is:

```
A/1 Y: AB=[+] AC=[+] AD=[+]
B/1 W: BG=[L] BE=[R] BA=[+]
C/1 W: CE=[L] CF=[R] CA=[+]
D/1 W: DF=[-] DG=[-] DA=[+]
E/1 L: EC=[R] EB=[L]
F/1 L: FD=[-] FC=[L]
G/1 L: GB=[R] GD=[-]
```


Note that the user only had to specify one of the two ground lines, GD. The program found that DF is also grounded. Similarly, in programming ground-line, we only had to update one of the vertexes. The rest is done by constraint propagation.

The next example yields the same four interpretations, in the same order (free floating, attached at bottom, attached at right, and attached at left) when interpreted ungrounded. The grounded version yields the unique solution shown in the following output and in figure 17.9.

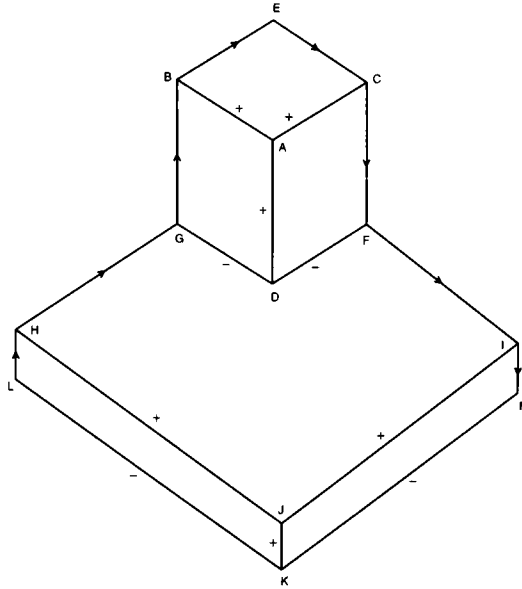


Figure 17.9: Labeled Cube on a Plate

```
(defdiagram cube-on-plate
  (a Y b c d)
  (b W g e a)
  (c W e f a)
  (d W f g a)
  (e L c b)
  (f Y d c i)
  (g Y b d h)
  (h W l g j)
  (i W f m j)
  (j Y h i k)
  (k W m l j)
  (l L h k)
```

```

(m L k i))
> (print-labelings (ground (diagram 'cube-on-plate) 'k 'm))
The initial diagram is:
A/5 Y: AB=[+-L-R] AC=[+-RL-] AD=[+--RL]
B/3 W: BG=[L-+] BE=[R-+] BA=[+-]
C/3 W: CE=[L-+] CF=[R-+] CA=[+-]
D/3 W: DF=[L-+] DG=[R-+] DA=[+-]
E/6 L: EC=[RL+L-R] EB=[LRR+L-]
F/5 Y: FD=[+-L-R] FC=[+-RL-] FI=[+--RL]
G/5 Y: GB=[+-L-R] GD=[+-RL-] GH=[+--RL]
H/3 W: HL=[L-+] HG=[R-+] HJ=[+-]
I/3 W: IF=[L-+] IM=[R-+] IJ=[+-]
J/5 Y: JH=[+-L-R] JI=[+-RL-] JK=[+--RL]
K/1 W: KM=[-] KL=[-] KJ=[+]
L/6 L: LH=[RL+L-R] LK=[LRR+L-]
M/6 L: MK=[RL+L-R] MI=[LRR+L-]
For 32,805,000 interpretations.

```

```

After constraint propagation the diagram is:
A/1 Y: AB=[+] AC=[+] AD=[+]
B/1 W: BG=[L] BE=[R] BA=[+]
C/1 W: CE=[L] CF=[R] CA=[+]
D/1 W: DF=[-] DG=[-] DA=[+]
E/1 L: EC=[R] EB=[L]
F/1 Y: FD=[-] FC=[L] FI=[R]
G/1 Y: GB=[R] GD=[-] GH=[L]
H/1 W: HL=[L] HG=[R] HJ=[+]
I/1 W: IF=[L] IM=[R] IJ=[+]
J/1 Y: JH=[+] JI=[+] JK=[+]
K/1 W: KM=[-] KL=[-] KJ=[+]
L/1 L: LH=[R] LK=[-]
M/1 L: MK=[-] MI=[L]

```

It is interesting to try the algorithm on an “impossible” diagram. It turns out the algorithm correctly finds no interpretation for this well-known illusion:

```

(defdiagram poiuyt
(a L b g)
(b L j a)
(c L d l)
(d L h c)
(e L f i)
(f L k e)
(g L a l)
(h L l d)
(i L e k)
(j L k b)

```

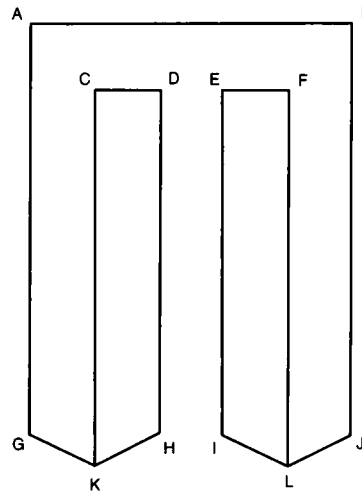


Figure 17.10: An Impossible Figure (A Poiuyt)

```
(k w j i f)
(l w h g c))

> (print-labelings (diagram 'poiuyt))
The initial diagram is:
A/6 L: AB=[RL+L-R] AG=[LRR+L-]
B/6 L: BJ=[RL+L-R] BA=[LRR+L-]
C/6 L: CD=[RL+L-R] CL=[LRR+L-]
D/6 L: DH=[RL+L-R] DC=[LRR+L-]
E/6 L: EF=[RL+L-R] EI=[LRR+L-]
F/6 L: FK=[RL+L-R] FE=[LRR+L-]
G/6 L: GA=[RL+L-R] GL=[LRR+L-]
H/6 L: HL=[RL+L-R] HD=[LRR+L-]
I/6 L: IE=[RL+L-R] IK=[LRR+L-]
J/6 L: JK=[RL+L-R] JB=[LRR+L-]
K/3 W: KJ=[L-+] KI=[R-+] KF=[++-]
L/3 W: LH=[L-+] LG=[R-+] LC=[++-]
For 544,195,584 interpretations.

After constraint propagation the diagram is:
A/5 L: AB=[RL+-R] AG=[LRR-L-]
B/5 L: BJ=[RLL-R] BA=[LR+L-]
C/2 L: CD=[LR] CL=[+-]
D/3 L: DH=[RL-] DC=[LRL]
E/3 L: EF=[RLR] EI=[LR-]
F/2 L: FK=[+-] FE=[RL]
```

G/4 L: GA=[RL-R] GL=[L+L-]
 H/4 L: HL=[R+-R] HD=[LRL-]
 I/4 L: IE=[RL-R] IK=[L+L-]
 J/4 L: JK=[R+-R] JB=[LRL-]
 K/3 W: KJ=[L-+] KI=[R-+] KF=[++-]
 L/3 W: LH=[L-+] LG=[R-+] LC=[++-]

For 2,073,600 interpretations.

There are zero solutions:

Now we try a more complex diagram:

```
(defdiagram tower
  (a Y b c d) (n L q o)
  (b W g e a) (o W y j n)
  (c W e f a) (p L r i)
  (d W f g a) (q W n s w)
  (e L c b) (r W s p x)
  (f Y d c i) (s L r q)
  (g Y b d h) (t W w x z)
  (h W l g j) (u W x y z)
  (i W f m p) (v W y w z)
  (j Y h o k) (w Y t v q)
  (k W m l j) (x Y r u t)
  (l L h k) (y Y v u o)
  (m L k i) (z Y t u v))

> (print-labelings (ground (diagram 'tower) 'i 'k))
The initial diagram is:
A/5 Y: AB=[+-L-R] AC=[+-RL-] AD=[+--RL]
B/3 W: BG=[L-+] BE=[R-+] BA=[++-]
C/3 W: CE=[L-+] CF=[R-+] CA=[++-]
D/3 W: DF=[L-+] DG=[R-+] DA=[++-]
E/6 L: EC=[RL+L-R] EB=[LRR+L-]
F/5 Y: FD=[+-L-R] FC=[+-RL-] FI=[+--RL]
G/5 Y: GB=[+-L-R] GD=[+-RL-] GH=[+--RL]
H/3 W: HL=[L-+] HG=[R-+] HJ=[++-]
I/3 W: IF=[L-+] IM=[R-+] IP=[++-]
J/5 Y: JH=[+-L-R] JO=[+-RL-] JK=[+--RL]
K/3 W: KM=[L-+] KL=[R-+] KJ=[++-]
L/1 L: LH=[R] LK=[-]
M/6 L: MK=[RL+L-R] MI=[LRR+L-]
N/6 L: NQ=[RL+L-R] NO=[LRR+L-]
O/3 W: OY=[L-+] OJ=[R-+] ON=[++-]
P/6 L: PR=[RL+L-R] PI=[LRR+L-]
Q/3 W: QN=[L-+] QS=[R-+] QW=[++-]
R/3 W: RS=[L-+] RP=[R-+] RX=[++-]
S/6 L: SR=[RL+L-R] SQ=[LRR+L-]
```

T/3 W: TW=[L-+] TX=[R-+] TZ=[++-]
 U/3 W: UX=[L-+] UY=[R-+] UZ=[++-]
 V/3 W: VY=[L-+] VW=[R-+] VZ=[++-]
 W/5 Y: WT=[+-L-R] WV=[+-RL-] WQ=[+-RL]
 X/5 Y: XR=[+-L-R] XU=[+-RL-] XT=[+-RL]
 Y/5 Y: YV=[+-L-R] YU=[+-RL-] YO=[+-RL]
 Z/5 Y: ZT=[+-L-R] ZU=[+-RL-] ZV=[+-RL]

For 1,614,252,037,500,000 interpretations.

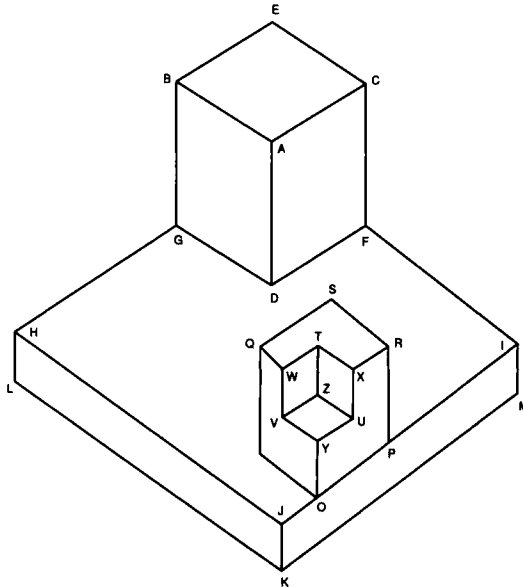


Figure 17.11: A Tower

After constraint propagation the diagram is:

A/1 Y: AB=[+] AC=[+] AD=[+]
 B/1 W: BG=[L] BE=[R] BA=[+]
 C/1 W: CE=[L] CF=[R] CA=[+]
 D/1 W: DF=[-] DG=[-] DA=[+]
 E/1 L: EC=[R] EB=[L]
 F/1 Y: FD=[-] FC=[L] FI=[R]
 G/1 Y: GB=[R] GD=[-] GH=[L]
 H/1 W: HL=[L] HG=[R] HJ=[+]
 I/1 W: IF=[L] IM=[R] IP=[+]
 J/1 Y: JH=[+] JO=[+] JK=[+]

```

K/1 W: KM=[-] KL=[-] KJ=[+]
L/1 L: LH=[R] LK=[-]
M/1 L: MK=[-] MI=[L]
N/1 L: NQ=[R] NO=[-]
O/1 W: OY=[+] OJ=[+] ON=[-]
P/1 L: PR=[L] PI=[+]
Q/1 W: QN=[L] QS=[R] QW=[+]
R/1 W: RS=[L] RP=[R] RX=[+]
S/1 L: SR=[R] SQ=[L]
T/1 W: TW=[+] TX=[+] TZ=[-]
U/1 W: UX=[+] UY=[+] UZ=[-]
V/1 W: VY=[+] VW=[+] VZ=[-]
W/1 Y: WT=[+] WV=[+] WQ=[+]
X/1 Y: XR=[+] XU=[+] XT=[+]
Y/1 Y: YV=[+] YU=[+] YO=[+]
Z/1 Y: ZT=[-] ZU=[-] ZV=[-]

```

We see that the algorithm was able to arrive at a single interpretation. Moreover, even though there were a large number of possibilities—over a quadrillion—the computation is quite fast. Most of the time is spent printing, so to get a good measurement, we define a function to find solutions without printing anything:

```

(defun find-labelings (diagram)
  "Return a list of all consistent labelings of the diagram."
  (every #'propagate-constraints (diagram-vertexes diagram))
  (search-solutions diagram))

```

When we time the application of `find-labelings` to the grounded tower and the `poiuyt`, we find the tower takes 0.11 seconds, and the `poiuyt` 21 seconds. This is over 180 times longer, even though the `poiuyt` has only half as many vertexes and only about half a million interpretations, compared to the tower's quadrillion. The `poiuyt` takes a long time to process because there are few local constraints, so violations are discovered only by considering several widely separated parts of the figure all at the same time. It is interesting that the same fact that makes the processing of the `poiuyt` take longer is also responsible for its interest as an illusion.

17.4 Checking Diagrams for Errors

This section considers one more example, and considers what to do when there are apparent errors in the input. The example is taken from Charniak and McDermott's *Introduction to Artificial Intelligence*, page 138, and shown in figure 17.12.

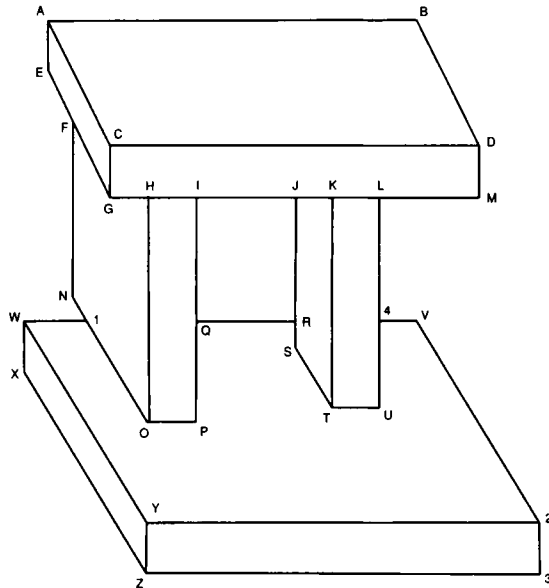


Figure 17.12: Diagram of an arch

```

(defdiagram arch
  (a W e b c) (p L o q)
  (b L d a) (q T p i r)
  (c Y a d g) (r T j s q)
  (d Y c b m) (s L r t)
  (e L a f) (t W v s k)
  (f T e g n) (u L t l)
  (g W h f c) (v L 2 4)
  (h T g i o) (w W x 1 y)
  (i T h j q) (x L w z)
  (j T i k r) (y Y w 2 z)
  (k T j l t) (z W 3 x y)
  (l T k m v) (1 T n o w)
  (m L l d) (2 W v 3 y)
  (n L f 1) (3 L z 2)
  (o W p 1 h) (4 T u 1 v))

```

Unfortunately, running this example results in no consistent interpretations after constraint propagation. This seems wrong. Worse, when we try to ground the diagram on the line XZ and call `print-labelings` on that, we get the following error:

```
>>>ERROR: The first argument to NTH was of the wrong type.
The function expected a fixnum >= zero.
While in the function LABELS-FOR =< CONSISTENT-LABELINGS

Debugger entered while in the following function:
```

```
LABELS-FOR (P.C. = 23)
  Arg 0 (VERTEX): U/6
  Arg 1 (FROM): 4/4
```

What has gone wrong? A good guess is that the diagram is somehow inconsistent—somewhere an error was made in transcribing the diagram. It could be that the diagram is in fact impossible, like the poiuyt. But that is unlikely, as it is easy for us to provide an intuitive interpretation. We need to debug the diagram, and it would also be a good idea to handle the error more gracefully.

One property of the diagram that is easy to check for is that every line should be mentioned twice. If there is a line between vertexes A and B, there should be two entries in the vertex descriptors of the following form:

```
(A ? ... B ...)
(B ? ... A ...)
```

Here the symbol “?” means we aren’t concerned about the type of the vertexes, only with the presence of the line in two places. The following code makes this check when a diagram is defined. It also checks that each vertex is one of the four legal types, and has the right number of neighbors.

```
(defmacro defdiagram (name &rest vertex-descriptors)
  "Define a diagram. A copy can be gotten by (diagram name)."
  '(put-diagram ',name (construct-diagram
    (check-diagram ',vertex-descriptors))))

(defun check-diagram (vertex-descriptors)
  "Check if the diagram description appears consistent."
  (let ((errors 0))
    (dolist (v-d vertex-descriptors)
      ;; v-d is like: (a Y b c d)
      (let ((A (first v-d))
            (v-type (second v-d)))
        ;; Check that the number of neighbors is right for
        ;; the vertex type (and that the vertex type is legal)
        (when (/= (length (v-d-neighbors v-d))
                  (case v-type ((W Y T) 3) ((L) 2) (t -1)))
          (warn "Illegal type/neighbor combo: ~a" v-d)
          (incf errors))
        ;; Check that each neighbor B is connected to
```



```

;; this vertex, A, exactly once
(dolist (B (v-d-neighbors v-d))
  (when (/= 1 (count-if
              #'(lambda (v-d2)
                  (and (eql (first v-d2) B)
                       (member A (v-d-neighbors v-d2))))
              vertex-descriptors))
    (warn "Inconsistent vertex: ~a~a" A B)
    (incf errors))))))
(when (> errors 0)
  (error "Inconsistent diagram. ~d total error~:p."
        errors)))
vertex-descriptors)

```

Now let's try the arch again:

```

(defdiagram arch
  (a W e b c) (p L o q)
  (b L d a) (q T p i r)
  (c Y a d g) (r T j s q)
  (d Y c b m) (s L r t)
  (e L a f) (t W v s k)
  (f T e g n) (u L t l)
  (g W h f c) (v L 2 4)
  (h T g i o) (w W x l y)
  (i T h j q) (x L w z)
  (j T i k r) (y Y w 2 z)
  (k T j l t) (z W 3 x y)
  (l T k m v) (1 T n o w)
  (m L l d) (2 W v 3 y)
  (n L f l) (3 L z 2)
  (o W p l h) (4 T u l v))
Warning: Inconsistent vertex: T-V
Warning: Inconsistent vertex: U-T
Warning: Inconsistent vertex: U-L
Warning: Inconsistent vertex: L-V
Warning: Inconsistent vertex: 4-U
Warning: Inconsistent vertex: 4-L

>>ERROR: Inconsistent diagram. 6 total errors.

```

The `defdiagram` was transcribed from a hand-labeled diagram, and it appears that the transcription has fallen prey to one of the oldest problems in mathematical notation: confusing a “u” with a “v.” The other problem was in seeing the line U-L as a single line, when in fact it is broken up into two segments, U-4 and 4-L. Repairing these bugs gives the diagram:

```
(defdiagram arch
  (a W e b c) (p L o q)
  (b L d a) (q T p i r)
  (c Y a d g) (r T j s q)
  (d Y c b m) (s L r t)
  (e L a f) (t W u s k) ;t-u not t-v
  (f T e g n) (u L t 4) ;u-4 not u-l
  (g W h f c) (v L 2 4)
  (h T g i o) (w W x 1 y)
  (i T h j q) (x L w z)
  (j T i k r) (y Y w 2 z)
  (k T j l t) (z W 3 x y)
  (l T k m 4) (1 T n o w) ;l-4 not l-v
  (m L l d) (2 W v 3 y)
  (n L f 1) (3 L z 2)
  (o W p 1 h) (4 T u 1 v))
```

This time there are no errors detected by check-diagram, but running print-labelings again still does not give a solution. To get more information about which constraints are applied, I modified propagate-constraints to print out some information:

```
(defun propagate-constraints (vertex)
  "Reduce the number of labelings on vertex by considering neighbors.
  If we can reduce, propagate the new constraint to each neighbor."
  ;; Return nil only when the constraints lead to an impossibility
  (let ((old-num (number-of-labelings vertex)))
    (setf (vertex-labelings vertex) (consistent-labelings vertex))
    (unless (impossible-vertex-p vertex)
      (when (< (number-of-labelings vertex) old-num)
        (format t "~&; ~a: ~14a ~a" vertex ;***
                (vertex-neighbors vertex) ;***
                (vertex-labelings vertex)) ;***
          (every #'propagate-constraints (vertex-neighbors vertex)))
        vertex)))
```

Running the problem again gives the following trace:

```
> (print-labelings (ground (diagram 'arch) 'x 'z))
The initial diagram is:
A/3 W: AE=[L-+] AB=[R-+] AC=[+-]
P/6 L: PO=[RL+L-R] PQ=[LRR+L-]
B/6 L: BD=[RL+L-R] BA=[LRR+L-]
Q/4 T: QP=[RRRR] QI=[LLLL] QR=[+-LR]
C/5 Y: CA=[+-L-R] CD=[+-RL-] CG=[+-RL]
R/4 T: RJ=[RRRR] RS=[LLLL] RQ=[+-LR]
D/5 Y: DC=[+-L-R] DB=[+-RL-] DM=[+-RL]
```

S/6 L: SR=[RL+L-R] ST=[LRR+L-]
 E/6 L: EA=[RL+L-R] EF=[LRR+L-]
 T/3 W: TU=[L-+] TS=[R-+] TK=[++-]
 F/4 T: FE=[RRRR] FG=[LLLL] FN=[+-LR]
 U/6 L: UT=[RL+L-R] U4=[LRR+L-]
 G/3 W: GH=[L-+] GF=[R-+] GC=[++-]
 V/6 L: V2=[RL+L-R] V4=[LRR+L-]
 H/4 T: HG=[RRRR] HI=[LLLL] HO=[+-LR]
 W/3 W: WX=[L-+] W1=[R-+] WY=[++-]
 I/4 T: IH=[RRRR] IJ=[LLLL] IQ=[+-LR]
 X/1 L: XW=[R] XZ=[-]
 J/4 T: JI=[RRRR] JK=[LLLL] JR=[+-LR]
 Y/5 Y: YW=[+-L-R] Y2=[+-RL-] YZ=[+-RL]
 K/4 T: KJ=[RRRR] KL=[LLLL] KT=[+-LR]
 Z/3 W: Z3=[L-+] ZX=[R-+] ZY=[++-]
 L/4 T: LK=[RRRR] LM=[LLLL] L4=[+-LR]
 1/4 T: 1N=[RRRR] 1O=[LLLL] 1W=[+-LR]
 M/6 L: ML=[RL+L-R] MD=[LRR+L-]
 2/3 W: 2V=[L-+] 23=[R-+] 2Y=[++-]
 N/6 L: NF=[RL+L-R] N1=[LRR+L-]
 3/6 L: 3Z=[RL+L-R] 32=[LRR+L-]
 O/3 W: OP=[L-+] O1=[R-+] OH=[++-]
 4/4 T: 4U=[RRRR] 4L=[LLLL] 4V=[+-LR]

For 2,888,816,545,234,944,000 interpretations.

; P/2: (O/3 Q/4) ((R L) (- L))
 ; O/1: (P/2 1/4 H/4) ((L R +))
 ; P/1: (O/1 Q/4) ((R L))
 ; 1/3: (N/6 O/1 W/3) ((R L +) (R L -) (R L L))
 ; N/2: (F/4 1/3) ((R L) (- L))
 ; F/2: (E/6 G/3 N/2) ((R L -) (R L L))
 ; E/2: (A/3 F/2) ((R L) (- L))
 ; A/2: (E/2 B/6 C/5) ((L R +) (- - +))
 ; B/3: (D/5 A/2) ((R L) (- L) (R -))
 ; D/3: (C/5 B/3 M/6) ((- - -) (- L R) (R - L))
 ; W/1: (X/1 1/3 Y/5) ((L R +))
 ; 1/1: (N/2 O/1 W/1) ((R L L))
 ; Y/1: (W/1 2/3 Z/3) ((+ + +))
 ; 2/2: (V/6 3/6 Y/1) ((L R +) (- - +))
 ; V/3: (2/2 4/4) ((R L) (- L) (R -))
 ; 4/2: (U/6 L/4 V/3) ((R L -) (R L R))
 ; U/2: (T/3 4/2) ((R L) (- L))
 ; T/2: (U/2 S/6 K/4) ((L R +) (- - +))
 ; S/2: (R/4 T/2) ((R L) (R -))
 ; K/1: (J/4 L/4 T/2) ((R L +))
 ; J/1: (I/4 K/1 R/4) ((R L L))
 ; I/1: (H/4 J/1 Q/4) ((R L R))
 ; L/1: (K/1 M/6 4/2) ((R L R))
 ; M/2: (L/1 D/3) ((R L) (R -))

```

; 3/3: (Z/3 2/2)      ((R L) (- L) (R -))
; Z/1: (3/3 X/1 Y/1) ((- - +))
; 3/1: (Z/1 2/2)      ((- L))
; 2/1: (V/3 3/1 Y/1) ((L R +))
; V/2: (2/1 4/2)      ((R L) (R -))

```

After constraint propagation the diagram is:

```

A/0 W:
P/1 L: PO=[R] PQ=[L]
B/0 L:
Q/4 T: QP=[RRRR] QI=[LLLL] QR=[+-LR]
C/0 Y:
R/4 T: RJ=[RRRR] RS=[LLLL] RQ=[+-LR]
D/0 Y:
S/2 L: SR=[RR] ST=[L-]
E/2 L: EA=[R-] EF=[LL]
T/2 W: TU=[L-] TS=[R-] TK=[++]
F/2 T: FE=[RR] FG=[LL] FN=[-L]
U/2 L: UT=[R-] U4=[LL]
G/0 W:
V/2 L: V2=[RR] V4=[L-]
H/0 T:
W/1 W: WX=[L] WI=[R] WY=[+]
I/1 T: IH=[R] IJ=[L] IQ=[R]
X/1 L: XW=[R] XZ=[-]
J/1 T: JI=[R] JK=[L] JR=[L]
Y/1 Y: YW=[+] Y2=[+] YZ=[+]
K/1 T: KJ=[R] KL=[L] KT=[+]
Z/1 W: Z3=[-] ZX=[-] ZY=[+]
L/1 T: LK=[R] LM=[L] L4=[R]
1/1 T: 1N=[R] 1O=[L] 1W=[L]
M/2 L: ML=[RR] MD=[L-]
2/1 W: 2V=[L] 23=[R] 2Y=[+]
N/2 L: NF=[R-] N1=[LL]
3/1 L: 3Z=[-] 32=[L]
O/1 W: OP=[L] O1=[R] OH=[+]
4/2 T: 4U=[RR] 4L=[LL] 4V=[-R]

```

From the diagram after constraint propagation we can see that the vertexes A,B,C,D,G, and H have no interpretations, so they are a good place to look first for an error. From the trace generated by propagate-constraints (the lines beginning with a semi-colon), we see that constraint propagation started at P and after seven propagations reached some of the suspect vertexes:

```

; A/2: (E/2 B/6 C/5) ((L R +) (- - +))
; B/3: (D/5 A/2)      ((R L) (- L) (R -))
; D/3: (C/5 B/3 M/6) ((- - -) (- L R) (R - L))

```

A and B look acceptable, but look at the entry for vertex D. It shows three interpretations, and it shows that the neighbors are C, B, and M. Note that line DC, the first entry in each of the interpretations, must be either -, - or R. But this is an error, because the “correct” interpretation has DC as a + line. Looking more closely, we notice that D is in fact a W-type vertex, not a Y vertex as written in the definition. We should have:

```

(defdiagram arch
  (a W e b c) (p L o q)
  (b L d a)   (q T p i r)
  (c Y a d g) (r T j s q)
  (d W b m c) (s L r t) ; d is a W, not Y
  (e L a f)   (t W u s k)
  (f T e g n) (u L t 4)
  (g W h f c) (v L 2 4)
  (h T g i o) (w W x 1 y)
  (i T h j q) (x L w z)
  (j T i k r) (y Y w 2 z)
  (k T j l t) (z W 3 x y)
  (l T k m 4) (1 T n o w)
  (m L l d)   (2 W v 3 y)
  (n L f 1)   (3 L z 2)
  (o W p 1 h) (4 T u 1 v))

```

By running the problem again and inspecting the trace output, we soon discover the real root of the problem: the most natural interpretation of the diagram is beyond the scope of the program! There are many interpretations that involve blocks floating in air, but if we ground lines OP, TU and XZ, we run into trouble. Remember, we said that we were considering trihedral vertexes only. But vertex 1 would be a quad-hedral vertex, formed by the intersection of four planes: the top and back of the base, and the bottom and left-hand side of the left pillar. The intuitively correct labeling for the diagram would have O1 be a concave (-) line and A1 be an occluding line, but our repertoire of labelings for T vertexes does not allow this. Hence, the diagram cannot be labeled consistently.

Let’s go back and consider the error that came up in the first version of the diagram. Even though the error no longer occurs on this diagram, we want to make sure that it won’t show up in another case. Here’s the error:

```

>>>ERROR: The first argument to NTH was of the wrong type.
The function expected a fixnum >= zero.
While in the function LABELS-FOR =< CONSISTENT-LABELINGS

Debugger entered while in the following function:

LABELS-FOR (P.C. = 23)
  Arg 0 (VERTEX): U/6
  Arg 1 (FROM): 4/4

```

Looking at the definition of `labels-for`, we see that it is looking for the `from` vertex, which in this case is 4, among the neighbors of U. It was not found, so `pos` became `nil`, and the function `nth` complained that it was not given an integer as an argument. So this error, if we had pursued it earlier, would have pointed out that 4 was not listed as a neighbor of U, when it should have been. Of course, we found that out by other means. In any case, there is no bug here to fix—as long as a diagram is guaranteed to be consistent, the `labels-for` bug will not appear again.

This section has made two points: First, write code that checks the input as thoroughly as possible. Second, even when input checking is done, it is still up to the user to understand the limitations of the program.

17.5 History and References






Guzman (1968) was one of the first to consider the problem of interpreting line diagrams. He classified vertexes, and defined some heuristics for combining information from adjacent vertexes. Huffman (1971) and Clowes (1971) independently came up with more formal and complete analyses, and David Waltz (1975) extended the analysis to handle shadows, and introduced the constraint propagation algorithm to cut down on the need for search. The algorithm is sometimes called “Waltz filtering” in his honor. With shadows and nontriangular angles, there are thousands of vertex labelings instead of 18, but there are also more constraints, so the constraint propagation actually does better than it does in our limited world. Waltz’s approach and the Huffman-Clowes labels are covered in most introductory AI books, including Rich and Knight 1990, Charniak and McDermott 1985, and Winston 1984. Waltz’s original paper appears in *The Psychology of Computer Vision* (Winston 1975), an influential volume collecting early work done at MIT. He also contributed a summary article on Waltz filtering (Waltz 1990).

Many introductory AI texts give vision short coverage, but Charniak and McDermott (1985) and Tanimoto (1990) provide good overviews of the field. Zucker (1990) provides an overview of low-level vision.

Ramsey and Barrett (1987) give an implementation of a line-recognition program. It would make a good project to connect their program to the one presented in this chapter, and thereby go all the way from pixels to 3-D descriptions.

17.6 Exercises

This chapter has solved the problem of line-labeling for polyhedra made of trihedral vertexes. The following exercises extend this solution.

-  **Exercise 17.1 [h]** Use the line-labeling to produce a face labeling. Write a function that takes a labeled diagram as input and produces a list of the faces (planes) that comprise the diagram.
-  **Exercise 17.2 [h]** Use the face labeling to produce a polyhedron labeling. Write a function that takes a list of faces and a diagram and produces a list of polyhedra (blocks) that comprise the diagram.
-  **Exercise 17.3 [d]** Extend the system to include quad-hedral vertexes and/or shadows. There is no conceptual difficulty in this, but it is a very demanding task to find all the possible vertex types and labelings for them. Consult Waltz 1975.
-  **Exercise 17.4 [d]** Implement a program to recognize lines from pixels.
-  **Exercise 17.5 [d]** If you have access to a workstation with a graphical interface, implement a program to allow a user to draw diagrams with a mouse. Have the program generate output in the form expected by `construct-diagram`.

CHAPTER 18

Search and the Game of Othello

*In the beginner's mind there are
endless possibilities;
in the expert's there are few.*

—Suzuki Roshi, Zen Master

Game playing has been the target of much early work in AI for three reasons. First, the rules of most games are formalized, and they can be implemented in a computer program rather easily. Second, in many games the interface requirements are trivial. The computer need only print out its moves and read in the opponent's moves. This is true for games like chess and checkers, but not for ping-pong and basketball, where vision and motor skills are crucial. Third, playing a good game of chess is considered by many an intellectual achievement. Newell, Shaw, and Simon say, "Chess is the intellectual game *par excellence*," and Donald Michie called chess the "*Drosophila melanogaster* of machine intelligence," meaning that chess is a relatively simple yet interesting domain that can lead to advances in AI, just as study of the fruit fly served to advance biology.

Today there is less emphasis on game playing in AI. It has been realized that techniques that work well in the limited domain of a board game do not necessarily lead to intelligent behavior in other domains. Also, as it turns out, the techniques that allow computers to play well are not the same as the techniques that good human players use. Humans are capable of recognizing abstract patterns learned from previous games, and formulating plans of attack and defense. While some computer programs try to emulate this approach, the more successful programs work by rapidly searching thousands of possible sequences of moves, making fairly superficial evaluations of the worth of each sequence.

While much previous work on game playing has concentrated on chess and checkers, this chapter demonstrates a program to play the game of Othello.¹ Othello is a variation on the nineteenth-century game Reversi. It is an easy game to program because the rules are simpler than chess. Othello is also a rewarding game to program, because a simple search technique can yield an excellent player. There are two reasons for this. First, the number of legal moves per turn is low, so the search is not too explosive. Second, a single Othello move can flip a dozen or more opponent pieces. This makes it difficult for human players to visualize the long-range consequences of a move. Search-based programs are not confused, and thus do well relative to humans.

The very name “Othello” derives from the fact that the game is so unpredictable, like the Moor of Venice. The name may also be an allusion to the line, “Your daughter and the Moor are now making the beast with two backs,”² since the game pieces do indeed have two backs, one white and one black. In any case, the association between the game and the play carries over to the name of several programs: Cassio, Iago, and Bill. The last two will be discussed in this chapter. They are equal to or better than even champion human players. We will be able to develop a simplified version that is not quite a champion but is much better than beginning players.

18.1 The Rules of the Game

Othello is played on a 8-by-8 board, which is initially set up with four pieces in the center, as shown in figure 18.1. The two players, black and white, alternate turns, with black playing first. On each turn, a player places a single piece of his own color on the board. No piece can be moved once it is placed, but subsequent moves may flip a piece from one color to another. Each piece must be placed so that it *brackets* one or more opponent pieces. That is, when black plays a piece there must be a line (horizontal, vertical, or diagonal) that goes through the piece just played, then through one or more white pieces, and then to another black piece. The intervening

¹Othello is a registered trademark of CBS Inc. Gameboard design © 1974 CBS Inc.

²*Othello*, [I. i. 117] William Shakespeare.

white pieces are flipped over to black. If there are bracketed white pieces in more than one direction, they are all flipped. Figure 18.2 (a) indicates the legal moves for black with small dots. Figure 18.2 (b) shows the position after black moves to square b4. Players alternate turns, except that a player who has no legal moves must pass. When neither player has any moves, the game is over, and the player with the most pieces on the board wins. This usually happens because there are no empty squares left, but it occasionally happens earlier in the game.

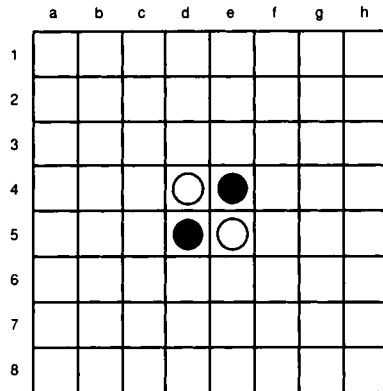


Figure 18.1: The Othello Board

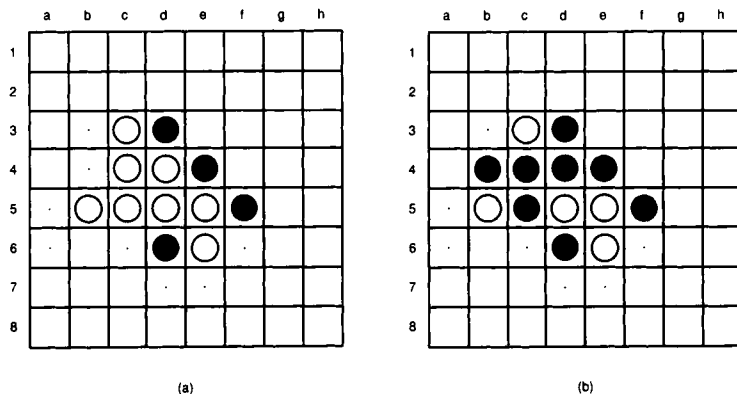


Figure 18.2: Legal Othello Moves

18.2 Representation Choices

In developing an Othello program, we will want to test out various strategies, playing those strategies against each other and against human players. We may also want our program to allow two humans to play a game. Therefore, our main function, `othello`, will be a monitoring function that takes as arguments two strategies. It uses these strategies to get each player's moves, and then applies these moves to a representation of the game board, perhaps printing out the board as it goes.

The first choice to make is how to represent the board and the pieces on it. The board is an 8-by-8 square, and each square can be filled by a black or white piece or can be empty. Thus, an obvious representation choice is to make the board an 8-by-8 array, where each element of the array is the symbol `black`, `white`, or `nil`.

Notice what is happening here: we are following the usual Lisp convention of implementing an *enumerated type* (the type of pieces that can fill a square) as a set of symbols. This is an appropriate representation because it supports the primary operation on elements of an enumerated type: test for equality using `eq`. It also supports input and output quite handily.

In many other languages (such as C or Pascal), enumerated types are implemented as integers. In Pascal one could declare:

```
type piece = (black, white, empty);
```

to define `piece` as a set of three elements that is treated as a subtype of the integers. The language does not allow for direct input and output of such types, but equality can be checked. An advantage of this approach is that an element can be packed into a small space. In the Othello domain, we anticipate that efficiency will be important, because one way to pick a good move is to look at a large number of possible sequences of moves, and choose a sequence that leads toward a favorable result. Thus, we are willing to look hard at alternative representations to find an efficient one. It takes only two bits to represent one of the three possible types, while it takes many more (perhaps 32) to represent a symbol. Thus, we may save space by representing pieces as small integers rather than symbols.

Next, we consider the board. The two-dimensional array seems like such an obvious choice that it is hard to imagine a better representation. We could consider an 8-element list of 8-element lists, but this would just waste space (for the cons cells) and time (in accessing the later elements of the lists). However, we will have to implement two other abstract data types that we have not yet considered: the square and the direction. We will need, for example, to represent the square that a player chooses to move into. This will be a pair of integers, such as 4,5. We could represent this as a two-element list, or more compactly as a cons cell, but this still means that we may have to generate garbage (create a cons cell) every time we want to refer to a new square. Similarly, we need to be able to scan in a given direction from a

square, looking for pieces to flip. Directions will be represented as a pair of integers, such as +1,-1. One clever possibility is to use complex numbers for both squares and directions, with the real component mapped to the horizontal axis and the imaginary component mapped to the vertical axis. Then moving in a given direction from a square is accomplished by simply adding the direction to the square. But in most implementations, creating new complex numbers will also generate garbage.

Another possibility is to represent squares (and directions) as two distinct integers, and have the routines that manipulate them accept two arguments instead of one. This would be efficient, but it is losing an important abstraction: that squares (and directions) are conceptually single objects.

A way out of this dilemma is to represent the board as a one-dimensional vector. Squares are represented as integers in the range 0 to 63. In most implementations, small integers (fixnums) are represented as immediate data that can be manipulated without generating garbage. Directions can also be implemented as integers, representing the numerical difference between adjacent squares along that direction. To get a feel for this, take a look at the board:

```

0  1  2  3  4  5  6  7
8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63

```

You can see that the direction +1 corresponds to movement to the right, +7 corresponds to diagonal movement downward and to the left, +8 is downward, and +9 is diagonally downward and to the right. The negations of these numbers (-1, -7, -8, -9) represent the opposite directions.

There is one complication with this scheme: we need to know when we hit the edge of the board. Starting at square 0, we can move in direction +1 seven times to arrive at the right edge of the board, but we aren't allowed to move in that direction yet again to arrive at square 8. It is possible to check for the edge of the board by considering quotients and remainders modulo 8, but it is somewhat complicated and expensive to do so.

A simpler solution is to represent the edge of the board explicitly, by using a 100-element vector instead of a 64-element vector. The outlying elements are filled with a marker indicating that they are outside the board proper. This representation wastes some space but makes edge detection much simpler. It also has the minor advantage that legal squares are represented by numbers in the range 11-88, which makes them easier to understand while debugging. Here's the new 100-element board:

```

0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99

```

The horizontal direction is now ± 1 , vertical is ± 10 , and the diagonals are ± 9 and ± 11 . We'll tentatively adopt this latest representation, but leave open the possibility of changing to another format. With this much decided, we are ready to begin. Figure 18.3 is the glossary for the complete program. A glossary for a second version of the program is on page 623.

What follows is the code for directions and pieces. We explicitly define the type `piece` to be a number from empty to outer (0 to 3), and define the function `name-of` to map from a piece number to a character: a dot for empty, @ for black, 0 for white, and a question mark (which should never be printed) for outer.

```

(defconstant all-directions '(-11 -10 -9 -1 1 9 10 11))

(defconstant empty 0 "An empty square")
(defconstant black 1 "A black piece")
(defconstant white 2 "A white piece")
(defconstant outer 3 "Marks squares outside the 8x8 board")

(deftype piece () '(integer ,empty ,outer))

(defun name-of (piece) (char ".@0?" piece))

(defun opponent (player) (if (eql player black) white black))

```

And here is the code for the board. Note that we introduce the function `bref`, for “board reference” rather than using the built-in function `aref`. This facilitates possible changes to the representation of boards. Also, even though there is no contiguous range of numbers that represents the legal squares, we can define the constant `all-squares` to be a list of the 64 legal squares, computed as those numbers from 11 to 88 whose value mod 10 is between 1 and 8.

```

(deftype board () '(simple-array piece (100)))

(defun bref (board square) (aref board square))
(defsetf bref (board square) (val)
  '(setf (aref ,board ,square) ,val))

```

othello	Top-Level Function Play a game of Othello. Return the score.
empty black white outer all-directions all-squares winning-value losing-value	Constants 0 represents an empty square. 1 represents a black piece. 2 represents a white piece. 3 represents a piece outside the 8 × 8 board. A list of integers representing the eight directions. A list of all legal squares. The best possible evaluation. The worst possible evaluation.
piece board	Data Types An integer from empty to outer. A vector of 100 pieces.
get-move make-move human random-strategy maximize-difference maximizer weighted-squares modified-weighted-squares minimax minimax-searcher alpha-beta alpha-beta-searcher	Major Functions Call the player's strategy function to get a move. Update board to reflect move by player. A strategy that prompts a human player. Make any legal move. A strategy that maximizes the difference in pieces. Return a strategy that maximizes some measure. Sum of the weights of player's squares minus opponent's. Like above, but treating corners better. Find the best move according to EVAL-FN, searching PLY levels. Return a strategy that uses minimax to search. Find the best move according to EVAL-FN, searching PLY levels. Return a strategy that uses alpha-beta to search.
bref copy-board initial-board print-board count-difference name-of opponent valid-p legal-p make-flips would-flip? find-bracketing-piece any-legal-move? next-to-play legal-moves final-value neighbors switch-strategies	Auxiliary Functions Reference to a position on the board. Make a new board. Return a board, empty except for four pieces in the middle. Print a board, along with some statistics. Count player's pieces minus opponent's pieces. A character used to print a piece. The opponent of black is white, and vice-versa. A syntactically valid square. A legal move on the board. Make any flips in the given direction. Would this move result in any flips in this direction? Return the square number of the bracketing piece. Does player have any legal moves in this position? Compute the player to move next, or NIL if nobody can move. Returns a list of legal moves for player. Is this a win, loss, or draw for player? Return a list of all squares adjacent to a square. Play one strategy for a while, then switch.
random-elt	Previously Defined Functions Choose a random element from a sequence. (pg. 36)

Figure 18.3: Glossary for the Othello Program

```

(defun copy-board (board)
  (copy-seq board))

(defconstant all-squares
  (loop for i from 11 to 88 when (<= 1 (mod i 10) 8) collect i))

(defun initial-board ()
  "Return a board, empty except for four pieces in the middle."
  ;; Boards are 100-element vectors, with elements 11-88 used,
  ;; and the others marked with the sentinel OUTER. Initially
  ;; the 4 center squares are taken, the others empty.
  (let ((board (make-array 100 :element-type 'piece
                          :initial-element outer)))
    (dolist (square all-squares)
      (setf (bref board square) empty))
    (setf (bref board 44) white (bref board 45) black
          (bref board 54) black (bref board 55) white)
    board))

(defun print-board (board)
  "Print a board, along with some statistics."
  (format t "~2& 1 2 3 4 5 6 7 8 [~c=~2a ~c=~2a (~@d)]"
          (name-of black) (count black board)
          (name-of white) (count white board)
          (count-difference black board))
  (loop for row from 1 to 8 do
    (format t "~& ~d " (* 10 row))
    (loop for col from 1 to 8
      for piece = (bref board (+ col (* 10 row)))
      do (format t "~c " (name-of piece))))
  (format t "~2&"))

(defun count-difference (player board)
  "Count player's pieces minus opponent's pieces."
  (- (count player board)
     (count (opponent player) board)))

```

Now let's take a look at the initial board, as it is printed by `print-board`, and by a raw `write` (I added the line breaks to make it easier to read):

```

> (write (initial-board) :array t)
#(3 3 3 3 3 3 3 3 3 3
  3 0 0 0 0 0 0 0 0 3
  3 0 0 0 0 0 0 0 0 3
  3 0 0 0 0 0 0 0 0 3
  3 0 0 0 2 1 0 0 0 3
  3 0 0 0 1 2 0 0 0 3
  3 0 0 0 0 0 0 0 0 3
  3 0 0 0 0 0 0 0 0 3
  3 0 0 0 0 0 0 0 0 3
  3 3 3 3 3 3 3 3 3 3)
#<ART-2B-100 -72570734>

> (print-board (initial-board))
      1 2 3 4 5 6 7 8 [O=2 O=2 (+0)]
10 . . . . .
20 . . . . .
30 . . . . .
40 . . . 0 O . . .
50 . . . O 0 . . .
60 . . . . .
70 . . . . .
80 . . . . .
NIL

```

Notice that `print-board` provides some additional information: the number of pieces that each player controls, and the difference between these two counts.

The next step is to handle moves properly: given a board and a square to move to, update the board to reflect the effects of the player moving to that square. This means flipping some of the opponent's pieces. One design decision is whether the procedure that makes moves, `make-move`, will be responsible for checking for error conditions. My choice is that `make-move` assumes it will be passed a legal move. That way, a strategy can use the function to explore sequences of moves that are known to be valid without slowing `make-move` down. Of course, separate procedures will have to insure that a move is legal. Here we introduce two terms: a *valid* move is one that is syntactically correct: an integer from 11 to 88 that is not off the board. A *legal* move is a valid move into an empty square that will flip at least one opponent. Here's the code:

```

(defun valid-p (move)
  "Valid moves are numbers in the range 11-88 that end in 1-8."
  (and (integerp move) (<= 11 move 88) (<= 1 (mod move 10) 8)))

(defun legal-p (move player board)
  "A Legal move must be into an empty square, and it must
  flip at least one opponent piece."
  (and (eql (bref board move) empty)
        (some #'(lambda (dir) (would-flip? move player board dir))
              all-directions)))

(defun make-move (move player board)
  "Update board to reflect move by player"
  ;; First make the move, then make any flips
  (setf (bref board move) player)
  (dolist (dir all-directions)
    (make-flips move player board dir))
  board)

```


Now all we need is to make-flips. To do that, we search in all directions for a *bracketing* piece: a piece belonging to the player who is making the move, which sandwiches a string of opponent pieces. If there are no opponent pieces in that direction, or if an empty or outer piece is hit before the player's piece, then no flips are made. Note that `would-flip?` is a semipredicate that returns false if no flips would be made in the given direction, and returns the square of the bracketing piece if there is one.

```
(defun make-flips (move player board dir)
  "Make any flips in the given direction."
  (let ((bracketer (would-flip? move player board dir)))
    (when bracketer
      (loop for c from (+ move dir) by dir until (eql c bracketer)
            do (setf (bref board c) player))))))

(defun would-flip? (move player board dir)
  "Would this move result in any flips in this direction?
  If so, return the square number of the bracketing piece."
  ;; A flip occurs if, starting at the adjacent square, c, there
  ;; is a string of at least one opponent pieces, bracketed by
  ;; one of player's pieces
  (let ((c (+ move dir)))
    (and (eql (bref board c) (opponent player))
         (find-bracketing-piece (+ c dir) player board dir))))

(defun find-bracketing-piece (square player board dir)
  "Return the square number of the bracketing piece."
  (cond ((eql (bref board square) player) square)
        ((eql (bref board square) (opponent player))
         (find-bracketing-piece (+ square dir) player board dir))
        (t nil)))
```

Finally we can write the function that actually monitors a game. But first we are faced with one more important choice: how will we represent a player? We have already distinguished between black and white's pieces, but we have not decided how to ask black or white for their moves. I choose to represent player's strategies as functions. Each function takes two arguments: the color to move (black or white) and the current board. The function should return a legal move number.

```
(defun othello (bl-strategy wh-strategy &optional (print t))
  "Play a game of Othello. Return the score, where a positive
  difference means black (the first player) wins."
  (let ((board (initial-board)))
    (loop for player = black
          then (next-to-play board player print)
          for strategy = (if (eql player black)
```

```

                bl-strategy
                wh-strategy)
    until (null player)
    do (get-move strategy player board print))
(when print
  (format t "~&The game is over. Final result:")
  (print-board board))
(count-difference black board)))

```

We need to be able to determine who plays next at any point. The rules say that players alternate turns, but if one player has no legal moves, the other can move again. When neither has a legal move, the game is over. This usually happens because there are no empty squares left, but it sometimes happens earlier in the game. The player with more pieces at the end of the game wins. If neither player has more, the game is a draw.

```

(defun next-to-play (board previous-player print)
  "Compute the player to move next, or NIL if nobody can move."
  (let ((opp (opponent previous-player)))
    (cond ((any-legal-move? opp board) opp)
          ((any-legal-move? previous-player board)
           (when print
             (format t "~&~c has no moves and must pass."
                     (name-of opp)))
            previous-player)
          (t nil))))

(defun any-legal-move? (player board)
  "Does player have any legal moves in this position?"
  (some #'(lambda (move) (legal-p move player board))
        all-squares))

```

Note that the argument `print` (of `othello`, `next-to-play`, and below, `get-move`) determines if information about the progress of the game will be printed. For an interactive game, `print` should be true, but it is also possible to play a “batch” game with `print` set to false.

In `get-move` below, the player’s strategy function is called to determine his move. Illegal moves are detected, and proper moves are reported when `print` is true. The strategy function is passed a number representing the player to move (black or white) and a copy of the board. If we passed the *real* game board, the function could cheat by changing the pieces on the board!

```
(defun get-move (strategy player board print)
  "Call the player's strategy function to get a move.
  Keep calling until a legal move is made."
  (when print (print-board board))
  (let ((move (funcall strategy player (copy-board board))))
    (cond
     ((and (valid-p move) (legal-p move player board))
      (when print
        (format t "~&~c moves to ~d." (name-of player) move))
      (make-move move player board))
     (t (warn "Illegal move: ~d" move)
        (get-move strategy player board print))))))
```

Here we define two simple strategies:

```
(defun human (player board)
  "A human player for the game of Othello"
  (declare (ignore board))
  (format t "~&~c to move: " (name-of player))
  (read))

(defun random-strategy (player board)
  "Make any legal move."
  (random-elt (legal-moves player board)))

(defun legal-moves (player board)
  "Returns a list of legal moves for player"
  (loop for move in all-squares
        when (legal-p move player board) collect move))
```

We are now in a position to play the game. The expression `(othello #'human #'human)` will let two people play against each other. Alternately, `(othello #'random-strategy #'human)` will allow us to match our wits against a particularly poor strategy. The rest of this chapter shows how to develop a better strategy.

18.3 Evaluating Positions

The random-move strategy is, of course, a poor one. We would like to make a good move rather than a random move, but so far we don't know what makes a good move. The only positions we are able to evaluate for sure are final positions: when the game is over, we know that the player with the most pieces wins. This suggests a strategy: choose the move that maximizes count-difference, the piece differential.

The function `maximize-difference` does just that. It calls `maximizer`, a higher-order function that chooses the best move according to an arbitrary evaluation function.

```
(defun maximize-difference (player board)
  "A strategy that maximizes the difference in pieces."
  (funcall (maximizer #'count-difference) player board))

(defun maximizer (eval-fn)
  "Return a strategy that will consider every legal move,
  apply EVAL-FN to each resulting board, and choose
  the move for which EVAL-FN returns the best score.
  FN takes two arguments: the player-to-move and board"
  #'(lambda (player board)
      (let* ((moves (legal-moves player board))
            (scores (mapcar #'(lambda (move)
                                (funcall
                                 eval-fn
                                 player
                                 (make-move move player
                                           (copy-board board))))
                            moves))
            (best (apply #'max scores)))
        (elt moves (position best scores)))))
```

? **Exercise 18.1** Play some games with `maximize-difference` against `random-strategy` and human. How good is `maximize-difference`?

Those who complete the exercise will quickly see that the `maximize-difference` player does better than random, and may even beat human players in their first game or two. But most humans are able to improve, learning to take advantage of the overly greedy play of `maximize-difference`. Humans learn that the edge squares, for example, are valuable because the player dominating the edges can surround the opponent, while it is difficult to recapture an edge. This is especially true of corner squares, which can never be recaptured.

Using this knowledge, a clever player can temporarily sacrifice pieces to obtain edge and corner squares in the short run, and win back pieces in the long run. We can approximate some of this reasoning with the `weighted-squares` evaluation function. Like `count-difference`, it adds up all the player's pieces and subtracts the opponents, but each piece is weighted according to the square it occupies. Edge squares are weighted highly, corner squares higher still, and squares adjacent to the corners and edges have negative weights, because occupying these squares often gives the opponent a means of capturing the desirable square. Figure 18.4 shows the standard nomenclature for edge squares: X, A, B, and C. In general, X and C

squares are to be avoided, because taking them gives the opponent a chance to take the corner. The weighted-squares evaluation function reflects this.

	a	b	c	d	e	f	g	h
1		C	A	B	B	A	C	
2	C	X					X	C
3	A							A
4	B							B
5	B							B
6	A							A
7	C	X					X	C
8		C	A	B	B	A	C	

Figure 18.4: Names for Edge Squares

```
(defparameter *weights*
  #(0 0 0 0 0 0 0 0 0 0
    0 120 -20 20 5 5 20 -20 120 0
    0 -20 -40 -5 -5 -5 -5 -40 -20 0
    0 20 -5 15 3 3 15 -5 20 0
    0 5 -5 3 3 3 3 -5 5 0
    0 5 -5 3 3 3 3 -5 5 0
    0 20 -5 15 3 3 15 -5 20 0
    0 -20 -40 -5 -5 -5 -5 -40 -20 0
    0 120 -20 20 5 5 20 -20 120 0
    0 0 0 0 0 0 0 0 0 0))

(defun weighted-squares (player board)
  "Sum of the weights of player's squares minus opponent's."
  (let ((opp (opponent player)))
    (loop for i in all-squares
          when (eql (bref board i) player)
            sum (aref *weights* i)
          when (eql (bref board i) opp)
            sum (- (aref *weights* i)))))
```

? **Exercise 18.2** Compare strategies by evaluating the two forms below. What happens? Is this a good test to determine which strategy is better?

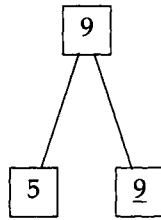
```
(othello (maximizer #'weighted-squares)
         (maximizer #'count-difference) nil)

(othello (maximizer #'count-difference)
         (maximizer #'weighted-squares) nil)
```

18.4 Searching Ahead: Minimax

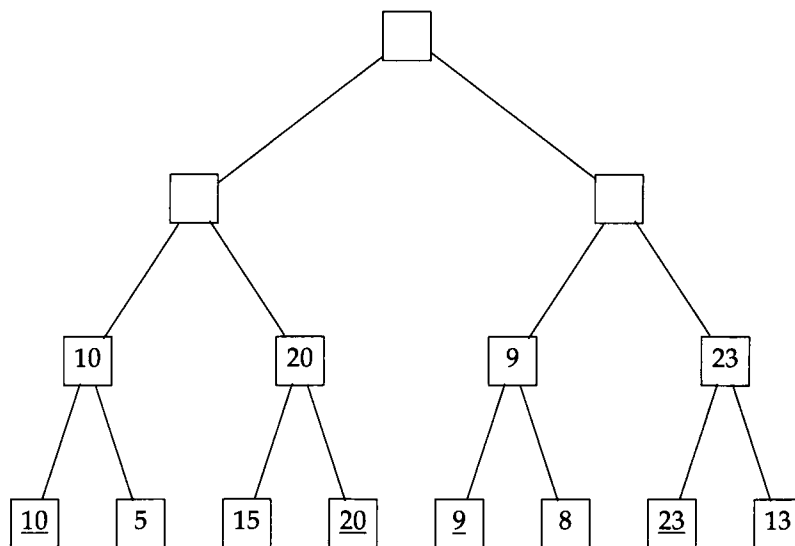
Even the weighted-squares strategy is no match for an experienced player. There are two ways we could improve the strategy. First, we could modify the evaluation function to take more information into account. But even without changing the evaluation function, we can improve the strategy by searching ahead. Instead of choosing the move that leads immediately to the highest score, we can also consider the opponent's possible replies, our replies to those replies, and so on. By searching through several levels of moves, we can steer away from potential disaster and find good moves that were not immediately apparent.

Another way to look at the `maximizer` function is as a search function that searches only one level, or *ply*, deep:



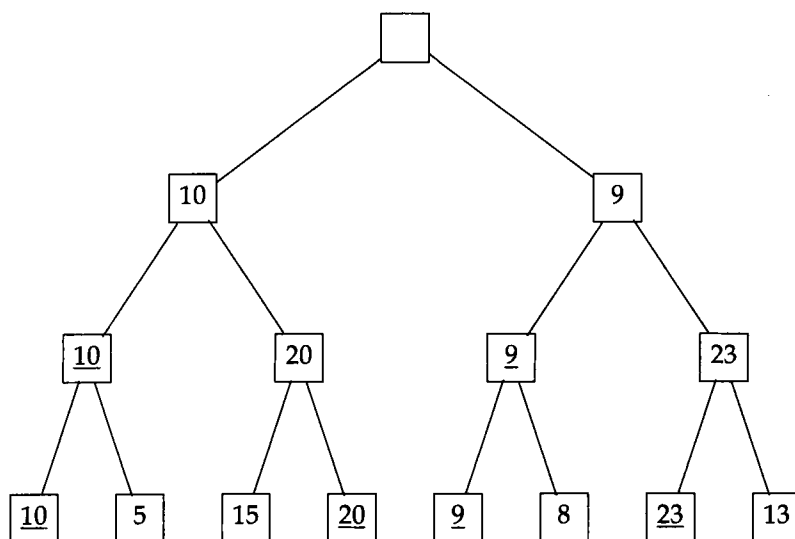
The top of the tree is the current board position, and the squares below that indicate possible moves. The `maximizer` function evaluates each of these and picks the best move, which is underlined in the diagram.

Now let's see how a 3-ply search might go. The first step is to apply `maximizer` to the positions just above the bottom of the tree. Suppose we get the following values:

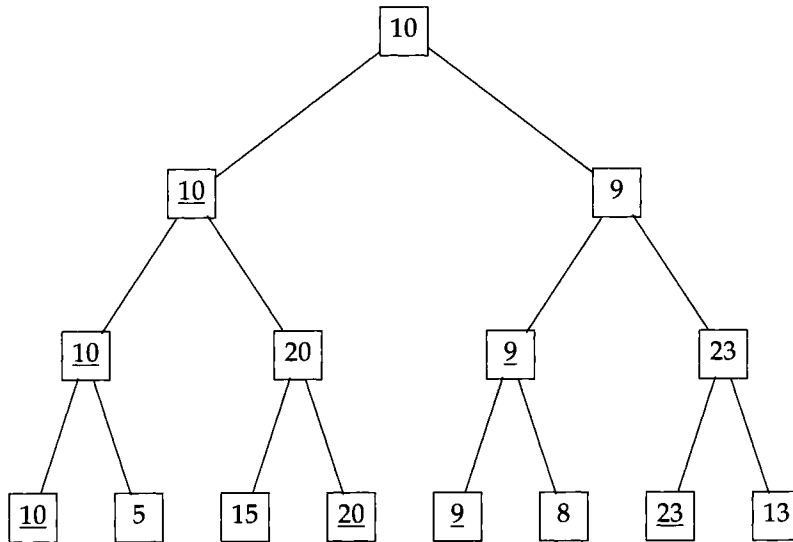


Each position is shown as having two possible legal moves, which is unrealistic but makes the diagram fit on the page. In a real game, five to ten legal moves per position is typical. The values at the leaves of the tree were computed by applying the evaluation function, while the values one level up were computed by maximizer. The result is that we know what our best move is for any of the four positions just above the bottom of the tree.

Going up a level, it is the opponent's turn to move. We can assume the opponent will choose the move that results in the minimal value to us, which would be the maximal value to the opponent. Thus, the opponent's choices would be the 10- and 9-valued positions, avoiding the 20- and 23-valued positions.



Now it is our turn to move again, so we apply maximizer once again to get the final value of the top-level position:



If the opponent plays as expected, we will always follow the left branch of the tree and end up at the position with value 10. If the opponent plays otherwise, we will end up at a position with a better value.

This kind of search is traditionally called a *minimax* search, because of the alternate application of the maximizer and a hypothetical minimizer function. Notice that only the leaf positions in the tree are looked at by the evaluation function. The value of all other positions is determined by minimizing and maximizing.

We are almost ready to code the minimax algorithm, but first we have to make a few design decisions. First, we could write two functions, `minimax` and `maximin`, which correspond to the two players' analyses. However, it is easier to write a single function that maximizes the value of a position for a particular player. In other words, by adding the player as a parameter, we avoid having to write two otherwise identical functions.

Second, we have to decide if we are going to write a general minimax searcher or an Othello-specific searcher. I decided on the latter for efficiency reasons, and because there are some Othello-specific complications that need to be accounted for. First, it is possible that a player will not have any legal moves. In that case, we want to continue the search with the opponent to move. If the opponent has no moves either, then the game is over, and the value of the position can be determined with finality by counting the pieces.

Third, we need to decide the interaction between the normal evaluation function and this final evaluation that occurs when the game is over. We could insist that

each evaluation function determine when the game is over and do the proper computation. But that overburdens the evaluation functions and may lead to wasteful checking for the end of game. Instead, I implemented a separate `final-value` evaluation function, which returns 0 for a draw, a large positive number for a win, and a large negative number for a loss. Because fixnum arithmetic is most efficient, the constants `most-positive-fixnum` and `most-negative-fixnum` are used. The evaluation functions must be careful to return numbers that are within this range. All the evaluation functions in this chapter will be within range if fixnums are 20 bits or more.

In a tournament, it is not only important who wins and loses, but also by how much. If we were trying to maximize the margin of victory, then `final-value` would be changed to include a small factor for the final difference.

```
(defconstant winning-value most-positive-fixnum)
(defconstant losing-value most-negative-fixnum)

(defun final-value (player board)
  "Is this a win, loss, or draw for player?"
  (case (signum (count-difference player board))
    (-1 losing-value)
    ( 0 0)
    (+1 winning-value)))
```

Fourth, and finally, we need to decide on the parameters for the minimax function. Like the other evaluation functions, it needs the `player` to move and the current board as parameters. It also needs an indication of how many ply to search, and the static evaluation function to apply to the leaf positions. Thus, `minimax` will be a function of four arguments. What will it return? It needs to return the best move, but it also needs to return the value of that move, according to the static evaluation function. We use multiple values for this.

```
(defun minimax (player board ply eval-fn)
  "Find the best move, for PLAYER, according to EVAL-FN,
  searching PLY levels deep and backing up values."
  (if (= ply 0)
      (funcall eval-fn player board)
      (let ((moves (legal-moves player board)))
        (if (null moves)
            (if (any-legal-move? (opponent player) board)
                (- (minimax (opponent player) board
                           (- ply 1) eval-fn))
                (final-value player board))
            (let ((best-move nil)
                  (best-val nil))
              (dolist (move moves)
```

```

(let* ((board2 (make-move move player
                        (copy-board board)))
      (val (- (minimax
              (opponent player) board2
              (- ply 1) eval-fn))))
      (when (or (null best-val)
                (> val best-val))
        (setf best-val val)
        (setf best-move move)))
      (values best-val best-move))))

```

The `minimax` function cannot be used as a strategy function as is, because it takes too many arguments and returns too many values. The functional `minimax-searcher` returns an appropriate strategy. Remember that a strategy is a function of two arguments: the player and the board. `get-move` is responsible for passing the right arguments to the function, so the strategy need not worry about where the arguments come from.

```

(defun minimax-searcher (ply eval-fn)
  "A strategy that searches PLY levels and then uses EVAL-FN."
  #'(lambda (player board)
      (multiple-value-bind (value move)
        (minimax player board ply eval-fn)
        (declare (ignore value))
        move)))

```

We can test the `minimax` strategy, and see that searching ahead 3 ply is indeed better than looking at only 1 ply. I show only the final result, which demonstrates that it is indeed an advantage to be able to look ahead:

```

> (othello (minimax-searcher 3 #'count-difference)
      (maximizer #'count-difference))

```

...

The game is over. Final result:

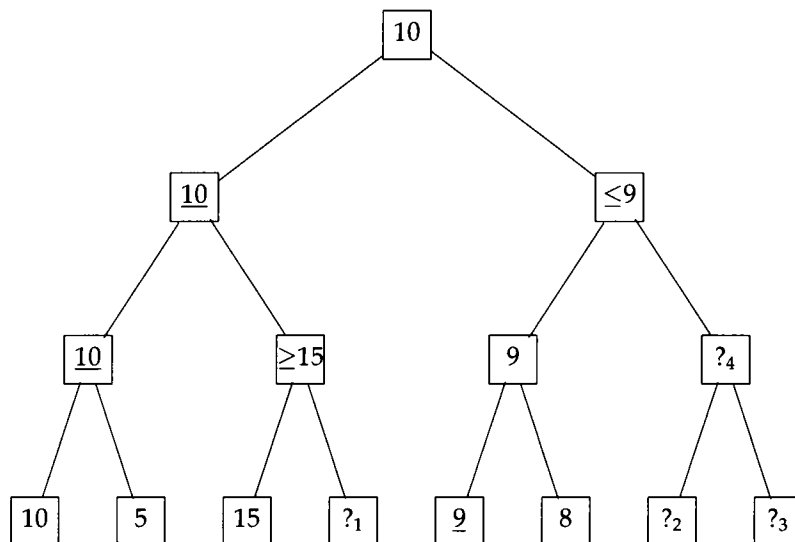
```

      1 2 3 4 5 6 7 8  [O=53 O=0 (+53)]
10  @ @ @ @ @ @ @ @
20  @ @ @ @ @ @ @ @
30  @ @ @ @ @ @ @ @
40  @ @ @ @ @ @ @ @
50  @ @ @ @ @ @ @ @
60  . . @ @ @ @ @ @
70  . . . @ @ @ @ @
80  . . . . @ @ . .

```

18.5 Smarter Searching: Alpha-Beta Search

The problem with a full minimax search is that it considers too many positions. It looks at every line of play, including many improbable ones. Fortunately, there is a way to find the optimal line of play without looking at every possible position. Let's go back to our familiar search tree:



Here we have marked certain positions with question marks. The idea is that the whole search tree evaluates to 10 regardless of the value of the positions labeled $?_i$. Consider the position labeled $?_1$. It does not matter what this position evaluates to, because the opponent will always choose to play toward the 10-position, to avoid the possibility of the 15. Thus, we can cut off the search at this point and not consider the $?_1$ -position. This kind of cutoff has historically been called a *beta* cutoff.

Now consider the position labeled $?_4$. It does not matter what this position evaluates to, because we will always prefer to choose the 10 position at the left branch, rather than giving the opponent a chance to play to the 9-position. This is an *alpha* cutoff. Notice that it cuts off a whole subtree of positions below it (labeled $?_2$ and $?_3$).

In general, we keep track of two parameters that bound the true value of the current position. The lower bound is a value we know we can achieve by choosing a certain line of play. The idea is that we need not even consider moves that will lead to a value lower than this. The lower bound has traditionally been called *alpha*, but we will name it *achievable*. The upper bound represents a value the opponent can achieve by choosing a certain line of play. It has been called *beta*, but we will call it *cutoff*. Again, the idea is that we need not consider moves with a higher value than this (because then the opponent would avoid the move that is so good for us). The

alpha-beta algorithm is just minimax, but with some needless evaluations pruned by these two parameters.

In deeper trees with higher branching factors, many more evaluations can be pruned. In general, a tree of depth d and branching factor b requires b^d evaluations for full minimax, and as few as $b^{d/2}$ evaluations with alpha-beta minimax.

To implement alpha-beta search, we add two more parameters to the function `minimax` and rename it `alpha-beta`. `achievable` is the best score the player can achieve; it is what we want to maximize. The `cutoff` is a value that, when exceeded, will make the opponent choose another branch of the tree, thus making the rest of the current level of the tree irrelevant. The test `until (>= achievable cutoff)` in the penultimate line of `minimax` does the cutoff; all the other changes just involve passing the parameters around properly.

```
(defun alpha-beta (player board achievable cutoff ply eval-fn)
  "Find the best move, for PLAYER, according to EVAL-FN,
  searching PLY levels deep and backing up values,
  using cutoffs whenever possible."
  (if (= ply 0)
      (funcall eval-fn player board)
      (let ((moves (legal-moves player board)))
        (if (null moves)
            (if (any-legal-move? (opponent player) board)
                (- (alpha-beta (opponent player) board
                              (- cutoff) (- achievable)
                              (- ply 1) eval-fn))
                (final-value player board))
            (let ((best-move (first moves)))
              (loop for move in moves do
                (let* ((board2 (make-move move player
                                          (copy-board board)))
                      (val (- (alpha-beta
                              (opponent player) board2
                              (- cutoff) (- achievable)
                              (- ply 1) eval-fn))))
                  (when (> val achievable)
                    (setf achievable val)
                    (setf best-move move)))
                until (>= achievable cutoff))
              (values achievable best-move))))))

(defun alpha-beta-searcher (depth eval-fn)
  "A strategy that searches to DEPTH and then uses EVAL-FN."
  #'(lambda (player board)
      (multiple-value-bind (value move)
        (alpha-beta player board losing-value winning-value
                    depth eval-fn)
```

```
(declare (ignore value))
move)))
```

It must be stressed that alpha-beta computes the exact same result as the full-search version of minimax. The only advantage of the cutoffs is making the search go faster by considering fewer positions.

18.6 An Analysis of Some Games

Now is a good time to stop and analyze where we have gone. We've demonstrated a program that can play a *legal* game of Othello, and some strategies that may or may not play a *good* game. First, we'll look at some individual games to see the mistakes made by some strategies, and then we'll generate some statistics for series of games.

Is the weighted-squares measure a good one? We can compare it to a strategy of maximizing the number of pieces. Such a strategy would of course be perfect if it could look ahead to the end of the game, but the speed of our computers limits us to searching only a few ply, even with cutoffs. Consider the following game, where black is maximizing the difference in the number of pieces, and white is maximizing the weighted sum of squares. Both search to a depth of 4 ply:

```
> (othello (alpha-beta-searcher 4 #'count-difference)
      (alpha-beta-searcher 4 #'weighted-squares))
```

Black is able to increase the piece difference dramatically as the game progresses. After 17 moves, white is down to only one piece:

```
      1 2 3 4 5 6 7 8  [W=20 O=1 (+19)]
10 0 @ . . . . .
20 . @ . . . @ @ .
30 @ @ @ @ @ @ . .
40 . @ . @ @ . . .
50 @ @ @ @ @ @ . .
60 . @ . . . . .
70 . . . . . . .
80 . . . . . . .
```

Although behind by 19 points, white is actually in a good position, because the piece in the corner is safe and threatens many of black's pieces. White is able to maintain good position while being numerically far behind black, as shown in these positions later in the game:

```

      1 2 3 4 5 6 7 8  [O=32 O=15 (+17)]
10 0 0 0 0 O O 0 0
20 O O 0 O O O O O
30 O O 0 0 O 0 O O
40 0 0 O O O O O O
50 O 0 O O O O . .
60 O O 0 O O 0 . .
70 O . . O O . . .
80 . . . . . . . .

```

```

      1 2 3 4 5 6 7 8  [O=34 O=19 (+15)]
10 0 0 0 0 O O 0 0
20 O O 0 O O O O O
30 O O 0 0 O 0 O O
40 0 O 0 O O O O O
50 0 O 0 O O O O .
60 0 O 0 O O O . .
70 0 O O O O . . .
80 0 O 0 . . . . .

```

After some give-and-take, white gains the advantage for good by capturing eight pieces on a move to square 85 on the third-to-last move of the game:

```

      1 2 3 4 5 6 7 8  [O=31 O=30 (+1)]
10 0 0 0 0 O O 0 0
20 O O 0 0 O O O 0
30 O O 0 0 0 O O 0
40 0 O 0 0 0 O O 0
50 0 O 0 O 0 O O 0
60 0 O 0 O O O O 0
70 0 O O O O O 0 0
80 0 O O O . . . 0

```

O moves to 85.

```

      1 2 3 4 5 6 7 8  [O=23 O=39 (-16)]
10 0 0 0 0 O O 0 0
20 O O 0 0 O O O 0
30 O O 0 0 0 O O 0
40 0 O 0 0 0 O O 0
50 0 O 0 O 0 O O 0
60 0 O 0 O 0 O 0 0
70 0 O O 0 0 0 0 0
80 0 0 0 0 0 . . 0

```

O moves to 86.

```

      1 2 3 4 5 6 7 8  [O=26 0=37 (-11)]
10 0 0 0 0 O O 0 0
20 O O 0 0 O O O 0
30 O O 0 0 0 O O 0
40 0 O 0 0 0 O O 0
50 0 O 0 O 0 O O 0
60 0 O 0 O 0 O 0 0
70 0 O O 0 O O 0 0
80 0 0 0 0 0 0 O . 0

```

0 moves to 87.

The game is over. Final result:

```

      1 2 3 4 5 6 7 8  [O=24 0=40 (-16)]
10 0 0 0 0 O O 0 0
20 O O 0 0 O O O 0
30 O O 0 0 0 O O 0
40 0 O 0 0 0 O O 0
50 0 O 0 O 0 O O 0
60 0 O 0 O 0 O 0 0
70 0 O O 0 O 0 0 0
80 0 0 0 0 0 0 0 0
-16

```

White ends up winning by 16 pieces. Black's strategy was too greedy: black was willing to give up position (all four corners and all but four of the edge squares) for temporary gains in material.

Increasing the depth of search does not compensate for a faulty evaluation function. In the following game, black's search depth is increased to 6 ply, while white's is kept at 4. The same things happen, although black's doom takes a bit longer to unfold.

```

> (othello (alpha-beta-searcher 6 #'count-difference)
      (alpha-beta-searcher 4 #'weighted-squares))

```

Black slowly builds up an advantage:

```

      1 2 3 4 5 6 7 8  [O=21 0=8 (+13)]
10 . . O O O O .
20 . O . O 0 O .
30 0 O O 0 O 0 .
40 . O . O 0 O .
50 . O O O O .
60 . O . O 0 .
70 . . . . .
80 . . . . .

```

But at this point white has clear access to the upper left corner, and through that corner threatens to take the whole top edge. Still, black maintains a material edge as the game goes on:

```

      1 2 3 4 5 6 7 8   [O=34 0=11 (+23)]
10 0 . @ @ @ @ @ .
20 . 0 0 @ @ @ . .
30 0 @ 0 0 @ @ @ @
40 @ @ @ @ 0 @ @ .
50 @ @ @ @ @ 0 @ .
60 @ @ @ @ @ @ 0 0
70 @ . . @ . . @ 0
80 . . . . . . . .

```

But eventually white's weighted-squares strategy takes the lead:

```

      1 2 3 4 5 6 7 8   [O=23 0=27 (-4)]
10 0 0 0 0 0 0 0 0
20 @ @ 0 @ @ @ . .
30 0 @ 0 0 @ @ @ @
40 0 @ 0 @ 0 @ @ .
50 0 @ 0 @ @ 0 @ .
60 0 0 0 @ @ @ 0 0
70 0 . 0 @ . . @ 0
80 0 . . . . . . .

```

and is able to hold on to win:

```

      1 2 3 4 5 6 7 8   [O=24 0=40 (-16)]
10 0 0 0 0 0 0 0 0
20 @ @ 0 @ 0 0 @ @
30 0 @ 0 0 @ @ @ @
40 0 @ 0 0 @ @ @ 0
50 0 0 @ @ 0 @ 0 0
60 0 0 0 @ 0 @ @ 0
70 0 0 0 0 @ @ 0 0
80 0 0 0 0 0 @ @ 0

```

-16

This shows that brute-force searching is not a panacea. While it is helpful to be able to search deeper, greater gains can be made by making the evaluation function more accurate. There are many problems with the weighted-squares evaluation function. Consider again this position from the first game above:


```

      1 2 3 4 5 6 7 8  [ @=-20 0=1 (+19)]
10 0 @ . . . . .
20 . @ . . . @ @ .
30 @ @ @ @ @ @ . .
40 . @ . @ @ . . .
50 @ @ @ @ @ @ . .
60 . @ . . . . .
70 . . . . . . .
80 . . . . . . .

```

Here white, playing the weighted-squares strategy, chose to play 66. This is probably a mistake, as 13 would extend white's dominance of the top edge, and allow white to play again (since black would have no legal moves). Unfortunately, white rejects this move, primarily because square 12 is weighted as -20. Thus, there is a disincentive to taking this square. But 12 is weighted -20 because it is a bad idea to take such a square when the corner is empty—the opponent will then have a chance to capture the corner, regaining the 12 square as well. Thus, we want squares like 12 to have a negative score when the corner is empty, but not when it is already occupied. The modified-weighted-squares evaluation function does just that.

```

(defun modified-weighted-squares (player board)
  "Like WEIGHTED-SQUARES, but don't take off for moving
  near an occupied corner."
  (let ((w (weighted-squares player board)))
    (dolist (corner '(11 18 81 88))
      (when (not (eql (bref board corner) empty))
        (dolist (c (neighbors corner))
          (when (not (eql (bref board c) empty))
            (incf w (* (- 5 (aref *weights* c)
                          (if (eql (bref board c) player)
                              +1 -1)))))))
        w))

  (let ((neighbor-table (make-array 100 :initial-element nil)))
    ;; Initialize the neighbor table
    (dolist (square all-squares)
      (dolist (dir all-directions)
        (if (valid-p (+ square dir))
            (push (+ square dir)
                  (aref neighbor-table square))))))

    (defun neighbors (square)
      "Return a list of all squares adjacent to a square."
      (aref neighbor-table square)))

```

18.7 The Tournament Version of Othello

While the `othello` function serves as a perfectly good moderator for casual play, there are two points that need to be fixed for tournament-level play. First, tournament games are played under a strict time limit: a player who takes over 30 minutes total to make all the moves forfeits the game. Second, the standard notation for Othello games uses square names in the range a1 to h8, rather than in the 11 to 88 range that we have used so far. a1 is the upper left corner, a8 is the lower left corner, and h8 is the lower right corner. We can write routines to translate between this notation and the one we were using by creating a table of square names.

```
(let ((square-names
      (cross-product #'symbol
                    '(? a b c d e f g h ?)
                    '(? 1 2 3 4 5 6 7 8 ?))))

  (defun h8->88 (str)
    "Convert from alphanumeric to numeric square notation."
    (or (position (string str) square-names :test #'string-equal)
        str))

  (defun 88->h8 (num)
    "Convert from numeric to alphanumeric square notation."
    (if (valid-p num)
        (elt square-names num)
        num)))

(defun cross-product (fn xlist ylist)
  "Return a list of all (fn x y) values."
  (mappend #'(lambda (y)
              (mapcar #'(lambda (x) (funcall fn x y))
                  xlist))
           ylist))
```

Note that these routines return their input unchanged when it is not one of the expected values. This is to allow commands other than moving to a particular square. For example, we will add a feature that recognizes `resign` as a move.

The human player needs to be changed slightly to read moves in this format. While we're at it, we'll also print the list of possible moves:

```
(defun human (player board)
  "A human player for the game of Othello"
  (format t "~&~c to move ~a: " (name-of player)
          (mapcar #'88->h8 (legal-moves player board))))
(h8->88 (read)))
```

othello-series	Top-Level Functions
random-othello-series	Play a series of N games.
round-robin	Play a series of games, starting from a random position.
	Play a tournament among strategies.
	Special Variables
clock	A copy of the game clock (tournament version only).
board	A copy of the game board (tournament version only).
move-number	Number of moves made (tournament version only).
ply-boards	A vector of boards; used as a resource to avoid consing.
	Data Structures
node	Holds a board and its evaluation.
	Main Functions
alpha-beta2	Sorts moves by static evaluation.
alpha-beta-searcher2	Strategy using alpha-beta2.
alpha-beta3	Uses the killer heuristic.
alpha-beta-searcher3	Strategy using alpha-beta3.
Iago-eval	Evaluation function based on Rosenbloom's program.
Iago	Strategy using Iago-eval.
	Auxiliary Functions
h8->88	Convert from alphanumeric to numeric square notation.
88->h8	Convert from numeric to alphanumeric square notation.
time-string	Convert internal time units to a mm:ss string.
switch-strategies	Play one strategy for a while, then another.
mobility	A strategy that counts the number of legal moves.
legal-nodes	A list of legal moves sorted by their evaluation.
negate-node	Set the value of a node to its negative.
put-first	Put the killer move first, if it is legal.
	Previously Defined Functions
cross-product	Apply fn to all pairs of arguments. (pg. 47)
symbol	Build a symbol by concatenating components.

Figure 18.5: Glossary for the Tournament Version of Othello

The `othello` function needn't worry about notation, but it does need to monitor the time. We make up a new data structure, the `clock`, which is an array of integers saying how much time (in internal units) each player has left. For example, `(aref clock black)` is the amount of time `black` has left to make all his moves. In Pascal, we would declare the `clock` array as `array[black..white]`, but in Common Lisp all arrays are zero-based, so we need an array of three elements to allow the subscript `black`, which is 2.

The `clock` is passed to `get-move` and `print-board` but is otherwise unused. I could have complicated the main game loop by adding tests for forfeits because of expired time and, as we shall see later, resignation by either player. However, I felt that would add a great deal of complexity for rarely used options. Instead, I wrap the whole game loop, along with the computation of the final score, in a `catch` special form. Then, if

get-move encounters a forfeit or resignation, it can throw an appropriate final score: 64 or -64, depending on which player forfeits.

```
(defvar *move-number* 1 "The number of the move to be played")

(defun othello (bl-strategy wh-strategy
              &optional (print t) (minutes 30))
  "Play a game of othello. Return the score, where a positive
  difference means black, the first player, wins."
  (let ((board (initial-board))
        (clock (make-array (+ 1 (max black white))
                           :initial-element
                           (* minutes 60
                              internal-time-units-per-second))))
    (catch 'game-over
      (loop for *move-number* from 1
            for player = black then (next-to-play board player print)
            for strategy = (if (eql player black)
                               bl-strategy
                               wh-strategy)
            until (null player)
            do (get-move strategy player board print clock))
      (when print
        (format t "~&The game is over. Final result:")
        (print-board board clock))
      (count-difference black board))))
```

Strategies now have to comply with the time-limit rule, so they may want to look at the time remaining. Rather than passing the clock in as an argument to the strategy, I decided to store the clock in the special variable `*clock*`. The new version of `othello` also keeps track of the `*move-number*`. This also could have been passed to the strategy functions as a parameter. But adding these extra arguments would require changes to all the strategies we have developed so far. By storing the information in special variables, strategies that want to can look at the clock or the move number, but other strategies don't have to know about them.

We still have the security problem—we don't want a strategy to be able to set the opponent's remaining time to zero and thereby win the game. Thus, we use `*clock*` only as a copy of the "real" game clock. The function `replace` copies the real clock into `*clock*`, and also copies the real board into `*board*`.

```
(defvar *clock* (make-array 3) "A copy of the game clock")
(defvar *board* (initial-board) "A copy of the game board")
```

```

(defun get-move (strategy player board print clock)
  "Call the player's strategy function to get a move.
  Keep calling until a legal move is made."
  ;; Note we don't pass the strategy function the REAL board.
  ;; If we did, it could cheat by changing the pieces on the board.
  (when print (print-board board clock))
  (replace *clock* clock)
  (let* ((t0 (get-internal-real-time))
         (move (funcall strategy player (replace *board* board)))
         (t1 (get-internal-real-time)))
    (decf (elt clock player) (- t1 t0))
    (cond
     ((< (elt clock player) 0)
      (format t "~&~c has no time left and forfeits."
              (name-of player))
      (THROW 'game-over (if (eq player black) -64 64)))
     (eq move 'resign)
      (THROW 'game-over (if (eq player black) -64 64)))
     (and (valid-p move) (legal-p move player board))
      (when print
        (format t "~&~c moves to ~a."
                (name-of player) (88->h8 move)))
        (make-move move player board))
     (t (warn "Illegal move: ~a" (88->h8 move))
        (get-move strategy player board print clock))))))

```

Finally, the function `print-board` needs to print the time remaining for each player; this requires an auxiliary function to get the number of minutes and seconds from an internal-format time interval. Note that we make the arguments optional, so that in debugging one can say just `(print-board)` to see the current situation. Also note the esoteric format option: `"~2, '0d"` prints a decimal number using at least two places, padding on the left with zeros.

```

(defun print-board (&optional (board *board*) clock)
  "Print a board, along with some statistics."
  ;; First print the header and the current score
  (format t "~2&   a b c d e f g h   [~c=~2a ~c=~2a (~@)]"
          (name-of black) (count black board)
          (name-of white) (count white board)
          (count-difference black board))
  ;; Print the board itself
  (loop for row from 1 to 8 do
        (format t "~& ~d " row)
        (loop for col from 1 to 8
              for piece = (bref board (+ col (* 10 row)))
              do (format t "~c " (name-of piece))))))

```

```

;; Finally print the time remaining for each player
(when clock
  (format t " [~c=~a ~c=~a]~2&"
    (name-of black) (time-string (elt clock black))
    (name-of white) (time-string (elt clock white))))

(defun time-string (time)
  "Return a string representing this internal time in min:secs."
  (multiple-value-bind (min sec)
    (floor (round time internal-time-units-per-second) 60))
  (format nil "~2d:~2,'0d" min sec)))

```

18.8 Playing a Series of Games

A single game is not enough to establish that one strategy is better than another. The following function allows two strategies to compete in a series of games:

```

(defun othello-series (strategy1 strategy2 n-pairs)
  "Play a series of 2*n-pairs games, swapping sides."
  (let ((scores (loop repeat n-pairs
    collect (othello strategy1 strategy2 nil)
    collect (- (othello strategy2 strategy1 nil))))))
    ;; Return the number of wins, (1/2 for a tie),
    ;; the total of the point differences, and the
    ;; scores themselves, all from strategy1's point of view.
    (values (+ (count-if #'plusp scores)
      (/ (count-if #'zerop scores) 2))
      (apply #'+ scores)
      scores)))

```

Let's see what happens when we use it to pit the two weighted-squares functions against each other in a series of ten games:

```

> (othello-series
  (alpha-beta-searcher 2 #'modified-weighted-squares)
  (alpha-beta-searcher 2 #'weighted-squares) 5)
0
60
(-28 40 -28 40 -28 40 -28 40 -28 40)

```

Something is suspicious here—the same scores are being repeated. A little thought reveals why: neither strategy has a random component, so the exact same game was played five times with one strategy going first, and another game was played

five times when the other strategy goes first! A more accurate appraisal of the two strategies' relative worth would be gained by starting each game from some random position and playing from there.

Think for a minute how you would design to run a series of games starting from a random position. One possibility would be to change the function `othello` to accept an optional argument indicating the initial state of the board. Then `othello-series` could be changed to somehow generate a random board and pass it to `othello`. While this approach is feasible, it means changing two existing working functions, as well as writing another function, `generate-random-board`. But we couldn't generate just any random board: it would have to be a legal board, so it would have to call `othello` and somehow get it to stop before the game was over.

An alternative is to leave both `othello` and `othello-series` alone and build another function on top of it, one that works by passing in two new strategies: strategies that make a random move for the first few moves and then revert to the normal specified behavior. This is a better solution because it uses existing functions rather than modifying them, and because it requires no new functions besides `switch-strategies`, which could prove useful for other purposes, and `random-othello-series`, which does nothing more than call `othello-series` with the proper arguments.

```
(defun random-othello-series (strategy1 strategy2
                             n-pairs &optional (n-random 10))
  "Play a series of 2*n games, starting from a random position."
  (othello-series
   (switch-strategies #'random-strategy n-random strategy1)
   (switch-strategies #'random-strategy n-random strategy2)
   n-pairs))

(defun switch-strategies (strategy1 m strategy2)
  "Make a new strategy that plays strategy1 for m moves,
  then plays according to strategy2."
  #'(lambda (player board)
      (funcall (if (<= *move-number* m) strategy1 strategy2)
               player board)))
```

There is a problem with this kind of series: it may be that one of the strategies just happens to get better random positions. A fairer test would be to play two games from each random position, one with the each strategy playing first. One way to do that is to alter `othello-series` so that it saves the random state before playing the first game of a pair, and then restores the saved random state before playing the second game. That way the same random position will be duplicated.

```
(defun othello-series (strategy1 strategy2 n-pairs)
  "Play a series of 2*n-pairs games, swapping sides."
  (let ((scores
        (loop repeat n-pairs
              for random-state = (make-random-state)
              collect (othello strategy1 strategy2 nil)
              do (setf *random-state* random-state)
              collect (- (othello strategy2 strategy1 nil))))))
    ;; Return the number of wins (1/2 for a tie),
    ;; the total of the point differences, and the
    ;; scores themselves, all from strategy1's point of view.
    (values (+ (count-if #'plusp scores)
              (/ (count-if #'zerop scores) 2))
            (apply #'+ scores)
            scores)))
```

Now we are in a position to do a more meaningful test. In the following, the weighted-squares strategy wins 4 out of 10 games against the modified strategy, losing by a total of 76 pieces, with the actual scores indicated.

```
> (random-othello-series
   (alpha-beta-searcher 2 #'weighted-squares)
   (alpha-beta-searcher 2 #'modified-weighted-squares)
   5)
4
-76
(-8 -40 22 -30 10 -10 12 -18 4 -18)
```

The random-othello-series function is useful for comparing two strategies. When there are more than two strategies to be compared at the same time, the following function can be useful:

```
(defun round-robin (strategies n-pairs &optional
                  (n-random 10) (names strategies))
  "Play a tournament among the strategies.
  N-PAIRS = games each strategy plays as each color against
  each opponent. So with N strategies, a total of
  N*(N-1)*N-PAIRS games are played."
  (let* ((N (length strategies))
         (totals (make-array N :initial-element 0))
         (scores (make-array (list N N)
                              :initial-element 0)))
    ;; Play the games
    (dotimes (i N)
      (loop for j from (+ i 1) to (- N 1) do
        (let* ((wins (random-othello-series
```



```

        (elt strategies i)
        (elt strategies j)
        n-pairs n-random))
      (losses (- (* 2 n-pairs) wins)))
      (incf (aref scores i j) wins)
      (incf (aref scores j i) losses)
      (incf (aref totals i) wins)
      (incf (aref totals j) losses))))
;; Print the results
(dotimes (i N)
  (format t "~&~a~20T ~4f: " (elt names i) (elt totals i))
  (dotimes (j N)
    (format t "~4f " (if (= i j) '---
                        (aref scores i j))))))

```

Here is a comparison of five strategies that search only 1 ply:

```

(defun mobility (player board)
  "The number of moves a player has."
  (length (legal-moves player board)))

> (round-robin
   (list (maximizer #'count-difference)
         (maximizer #'mobility)
         (maximizer #'weighted-squares)
         (maximizer #'modified-weighted-squares)
         #'random-strategy)
   5 10
   '(count-difference mobility weighted modified-weighted random))

COUNT-DIFFERENCE   12.5:  ---  3.0  2.5  0.0  7.0
MOBILITY             20.5:  7.0  ---  1.5  5.0  7.0
WEIGHTED             28.0:  7.5  8.5  ---  3.0  9.0
MODIFIED-WEIGHTED   31.5: 10.0  5.0  7.0  ---  9.5
RANDOM                7.5:  3.0  3.0  1.0  0.5  ---

```

The parameter `n-pairs` is 5, meaning that each strategy plays five games as black and five as white against each of the other four strategies, for a total of 40 games for each strategy and 100 games overall. The first line of output says that the count-difference strategy won 12.5 of its 40 games, including 3 against the mobility strategy, 2.5 against the weighted strategy, none against the modified weighted, and 7 against the random strategy. The fact that the random strategy manages to win 7.5 out of 40 games indicates that the other strategies are not amazingly strong. Now we see what happens when the search depth is increased to 4 ply (this will take a while to run):

```

> (round-robin
  (list (alpha-beta-searcher 4 #'count-difference)
        (alpha-beta-searcher 4 #'weighted-squares)
        (alpha-beta-searcher 4 #'modified-weighted-squares)
        #'random-strategy)
  5 10
  '(count-difference weighted modified-weighted random))

COUNT-DIFFERENCE    12.0:  ---  2.0  0.0 10.0
WEIGHTED              23.5:  8.0  ---  5.5 10.0
MODIFIED-WEIGHTED    24.5: 10.0  4.5  --- 10.0
RANDOM                 0.0:  0.0  0.0  0.0  ---

```

Here the random strategy does not win any games—an indication that the other strategies are doing something right. Notice that the modified weighted-squares has only a slight advantage over the weighted-squares, and in fact it lost their head-to-head series, four games to five, with one draw. So it is not clear which strategy is better.

The output does not break down wins by black or white, nor does it report the numerical scores. I felt that that would clutter up the output too much, but you're welcome to add this information. It turns out that white wins 23 (and draws 1) of the 40 games played between 4-ply searching strategies. Usually, Othello is a fairly balanced game, because black has the advantage of moving first but white usually gets to play last. It is clear that these strategies do not play well in the opening game, but for the last four ply they play perfectly. This may explain white's slight edge, or it may be a statistical aberration.

18.9 More Efficient Searching

The alpha-beta cutoffs work when we have established a good move and another move proves to be not as good. Thus, we will be able to make cutoffs earlier if we ensure that good moves are considered first. Our current algorithm loops through the list of `legal-moves`, but `legal-moves` makes no attempt to order the moves in any way. We will call this the *random-ordering* strategy (even though the ordering is not random at all—square 11 is always considered first, then 12, etc.).

One way to try to generate good moves first is to search highly weighted squares first. Since `legal-moves` considers squares in the order defined by `all-squares`, all we have to do is redefine the list `all-squares`³:

³Remember, when a constant is redefined, it may be necessary to recompile any functions that use the constant.

```
(defconstant all-squares
  (sort (loop for i from 11 to 88
            when (<= 1 (mod i 10) 8) collect i)
        #'> :key #'(lambda (sq) (elt *weights* sq))))
```

Now the corner squares will automatically be considered first, followed by the other highly weighted squares. We call this the *static-ordering* strategy, because the ordering is not random, but it does not change depending on the situation.

A more informed way to try to generate good moves first is to sort the moves according to the evaluation function. This means making more evaluations. Previously, only the boards at the leaves of the search tree were evaluated. Now we need to evaluate every board. In order to avoid evaluating a board more than once, we make up a structure called a node, which holds a board, the square that was taken to result in that board, and the evaluation value of that board. The search is the same except that nodes are passed around instead of boards, and the nodes are sorted by their value.

```
(defstruct (node) square board value)

(defun alpha-beta-searcher2 (depth eval-fn)
  "Return a strategy that does A-B search with sorted moves."
  #'(lambda (player board)
      (multiple-value-bind (value node)
        (alpha-beta2
         player (make-node :board board
                          :value (funcall eval-fn player board))
         losing-value winning-value depth eval-fn)
        (declare (ignore value))
        (node-square node))))

(defun alpha-beta2 (player node achievable cutoff ply eval-fn)
  "A-B search, sorting moves by eval-fn"
  ;; Returns two values: achievable-value and move-to-make
  (if (= ply 0)
      (values (node-value node) node)
      (let* ((board (node-board node))
             (nodes (legal-nodes player board eval-fn)))
        (if (null nodes)
            (if (any-legal-move? (opponent player) board)
                (values (- (alpha-beta2 (opponent player)
                                       (negate-value node)
                                       (- cutoff) (- achievable)
                                       (- ply 1) eval-fn))
                        nil)
                (values (final-value player board) nil))
            (let ((best-node (first nodes)))
              (loop for move in nodes
```

```

    for val = (- (alpha-beta2
                 (opponent player)
                 (negate-value move)
                 (- cutoff) (- achievable)
                 (- ply 1) eval-fn))
    do (when (> val achievable)
        (setf achievable val)
        (setf best-node move))
    until (>= achievable cutoff))
(values achievable best-node))))))

(defun negate-value (node)
  "Set the value of a node to its negative."
  (setf (node-value node) (- (node-value node)))
  node)

(defun legal-nodes (player board eval-fn)
  "Return a list of legal moves, each one packed into a node."
  (let ((moves (legal-moves player board)))
    (sort (map-into
           moves
           #'(lambda (move)
               (let ((new-board (make-move move player
                                           (copy-board board))))
                 (make-node
                  :square move :board new-board
                  :value (funcall eval-fn player new-board))))
           moves)
          #'> :key #'node-value)))

```

(Note the use of the function `map-into`. This is part of ANSI Common Lisp, but if it is not a part of your implementation, a definition is provided on page 857.)

The following table compares the performance of the random-ordering strategy, the sorted-ordering strategy and the static-ordering strategy in the course of a single game. All strategies search 6 ply deep. The table measures the number of boards investigated, the number of those boards that were evaluated (in all cases the evaluation function was modified-weighted-squares) and the time in seconds to compute a move.

random order			sorted order			static order		
boards	evals	secs	boards	evals	secs	boards	evals	secs
13912	10269	69	5556	5557	22	2365	1599	19
9015	6751	56	6571	6572	25	3081	2188	18
9820	7191	46	11556	11557	45	5797	3990	31
4195	3213	20	5302	5303	17	2708	2019	15
10890	7336	60	10709	10710	38	3743	2401	23
13325	9679	63	6431	6432	24	4222	2802	24
13163	9968	58	9014	9015	32	6657	4922	31
16642	12588	70	9742	9743	33	10421	7488	51
18016	13366	80	11002	11003	37	9508	7136	41
23295	17908	104	15290	15291	48	26435	20282	111
34120	25895	143	22994	22995	75	20775	16280	78
56117	43230	224	46883	46884	150	48415	36229	203
53573	41266	209	62252	62253	191	37803	28902	148
43943	33184	175	31039	31040	97	33180	24753	133
51124	39806	193	45709	45710	135	19297	15064	69
24743	18777	105	20003	20004	65	15627	11737	66
1.0	1.0	1.0	.81	1.07	.62	.63	.63	.63

The last two lines of the table give the averages and the averages normalized to the random-ordering strategy's performance. The sorted-ordering strategy takes only 62% of the time of the random-ordering strategy, and the static-ordering takes 63%. These times are not to be trusted too much, because a large-scale garbage collection was taking place during the latter part of the game, and it may have thrown off the times. The board and evaluation count may be better indicators, and they both show the static-ordering strategy doing the best.

We have to be careful how we evaluate these results. Earlier I said that alpha-beta search makes more cutoffs when it is presented first with better moves. The actual truth is that it makes more cutoffs when presented first with moves that *the evaluation function thinks* are better. In this case the evaluation function and the static-ordering strategy are in strong agreement on what are the best moves, so it is not surprising that static ordering does so well. As we develop evaluation functions that vary from the weighted-squares approach, we will have to run experiments again to see if the static-ordering is still the best.

18.10 It Pays to Precycle

The progressive city of Berkeley, California, has a strong recycling program to reclaim glass, paper, and aluminum that would otherwise be discarded as garbage. In 1989,

Berkeley instituted a novel program of *precycling*: consumers are encouraged to avoid buying products that come in environmentally wasteful packages.

Your Lisp system also has a recycling program: the Lisp garbage collector automatically recycles any unused storage. However, there is a cost to this program, and you the consumer can get better performance by precycling your data. Don't buy wasteful data structures when simpler ones can be used or reused. You, the Lisp programmer, may not be able to save the rain forests or the ozone layer, but you can save valuable processor time.

We saw before that the search routines look at tens of thousands of boards per move. Currently, each board position is created anew by copy-board and discarded soon thereafter. We could avoid generating all this garbage by reusing the same board at each ply. We'd still need to keep the board from the previous ply for use when the search backs up. Thus, a vector of boards is needed. In the following we assume that we will never search deeper than 40 ply. This is a safe assumption, as even the fastest Othello programs can only search about 15 ply before running out of time.

```
(defvar *ply-boards*
  (apply #'vector (loop repeat 40 collect (initial-board))))
```

Now that we have sharply limited the number of boards needed, we may want to reevaluate the implementation of boards. Instead of having the board as a vector of pieces (to save space), we may want to implement boards as vectors of bytes or full words. In some implementations, accessing elements of such vectors is faster. (In other implementations, there is no difference.)

An implementation using the vector of boards will be done in the next section. Note that there is another alternative: use only one board, and update it by making and retracting moves. This is a good alternative in a game like chess, where a move only alters two squares. In Othello, many squares can be altered by a move, so copying the whole board over and making the move is not so bad.

It should be mentioned that it is worth looking into the problem of copying a position from one board to another. The function `replace` copies one sequence (or part of it) into another, but it is a generic function that may be slow. In particular, if each element of a board is only 2 bits, then it may be much faster to use displaced arrays to copy 32 bits at a time. The advisability of this approach depends on the implementation, and so it is not explored further here.

18.11 Killer Moves

In section 18.9, we considered the possibility of searching moves in a different order, in an attempt to search the better moves first, thereby getting more alpha-beta pruning. In this section, we consider the *killer heuristic*, which states that a move that

has proven to be a good one in one line of play is also likely to be a good one in another line of play. To use chess as perhaps a more familiar example, suppose I consider one move, and it leads to the opponent replying by capturing my queen. This is a killer move, one that I would like to avoid. Therefore, when I consider other possible moves, I want to immediately consider the possibility of the opponent making that queen-capturing move.

The function `alpha-beta3` adds the parameter `killer`, which is the best move found so far at the current level. After we determine the `legal-moves`, we use `put-first` to put the killer move first, if it is in fact a legal move. When it comes time to search the next level, we keep track of the best move in `killer2`. This requires keeping track of the value of the best move in `killer2-val`. Everything else is unchanged, except that we get a new board by recycling the `*ply-boards*` vector rather than by allocating fresh ones.

```
(defun alpha-beta3 (player board achievable cutoff ply eval-fn
                   killer)
  "A-B search, putting killer move first."
  (if (= ply 0)
      (funcall eval-fn player board)
      (let ((moves (put-first killer (legal-moves player board))))
        (if (null moves)
            (if (any-legal-move? (opponent player) board)
                (- (alpha-beta3 (opponent player) board
                                (- cutoff) (- achievable)
                                (- ply 1) eval-fn nil))
                (final-value player board))
            (let ((best-move (first moves))
                  (new-board (aref *ply-boards* ply))
                  (killer2 nil)
                  (killer2-val winning-value))
              (loop for move in moves
                    do (multiple-value-bind (val reply)
                        (alpha-beta3
                         (opponent player)
                         (make-move move player
                                     (replace new-board board))
                         (- cutoff) (- achievable)
                         (- ply 1) eval-fn killer2)
                      (setf val (- val))
                      (when (> val achievable)
                        (setf achievable val)
                        (setf best-move move))
                      (when (and reply (< val killer2-val))
                        (setf killer2 reply)
                        (setf killer2-val val))))
                until (>= achievable cutoff))
```

```

        (values achievable best-move))))))
(defun alpha-beta-searcher3 (depth eval-fn)
  "Return a strategy that does A-B search with killer moves."
  #'(lambda (player board)
      (multiple-value-bind (value move)
        (alpha-beta3 player board losing-value winning-value
          depth eval-fn nil)
        (declare (ignore value))
        move)))
(defun put-first (killer moves)
  "Move the killer move to the front of moves,
  if the killer move is in fact a legal move."
  (if (member killer moves)
      (cons killer (delete killer moves))
      moves))

```

Another experiment on a single game reveals that adding the killer heuristic to static-ordering search (again at 6-ply) cuts the number of boards and evaluations, and the total time, all by about 20%. To summarize, alpha-beta search at 6 ply with random ordering takes 105 seconds per move (in our experiment), adding static-ordering cuts it to 66 seconds, and adding killer moves to that cuts it again to 52 seconds. This doesn't include the savings that alpha-beta cutoffs give over full minimax search. At 6 ply with a branching factor of 7, full minimax would take about nine times longer than static ordering with killers. The savings increase with increased depth. At 7 ply and a branching factor of 10, a small experiment shows that static-ordering with killers looks at only 28,000 boards in about 150 seconds. Full minimax would evaluate 10 million boards and take 350 times longer. The times for full minimax are estimates based on the number of boards per second, not on an actual experiment.

The algorithm in this section just keeps track of one killer move. It is of course possible to keep track of more than one. The Othello program Bill (Lee and Mahajan 1990b) merges the idea of killer moves with legal move generation: it keeps a list of possible moves at each level, sorted by their value. The legal move generator then goes down this list in sorted order.

It should be stressed once again that all this work on alpha-beta cutoffs, ordering, and killer moves has not made any change at all in the moves that are selected. We still end up choosing the same move that would be made by a full minimax search to the given depth, we are just doing it faster, without looking at possibilities that we can prove are not as good.

18.12 Championship Programs: Iago and Bill

As mentioned in the introduction, the unpredictability of Othello makes it a difficult game for humans to master, and thus programs that search deeply can do comparatively well. In fact, in 1981 the reigning champion, Jonathan Cerf, proclaimed "In my opinion the top programs . . . are now equal (if not superior) to the best human players." In discussing Rosenbloom's Iago program (1982), Cerf went on to say "I understand Paul Rosenbloom is interested in arranging a match against me. Unfortunately my schedule is very full, and I'm going to see that it remains that way for the foreseeable future."

In 1989, another program, Bill (Lee and Mahajan 1990) beat the highest rated American Othello player, Brian Rose, by a score of 56-8. Bill's evaluation function is fast enough to search 6-8 ply under tournament conditions, yet it is so accurate that it beats its creator, Kai-Fu Lee, searching only 1 ply. (However, Lee is only a novice Othello player; his real interest is in speech recognition; see Waibel and Lee 1991.) There are other programs that also play at a high level, but they have not been written up in the AI literature as Iago and Bill have.

In this section we present an evaluation function based on Iago's, although it also contains elements of Bill, and of an evaluation function written by Eric Wefald in 1989. The evaluation function makes use of two main features: *mobility* and *edge stability*.

Mobility

Both Iago and Bill make heavy use of the concept of *mobility*. Mobility is a measure of the ability to make moves; basically, the more moves one can make, the better. This is not quite true, because there is no advantage in being able to make bad moves, but it is a useful heuristic. We define *current mobility* as the number of legal moves available to a player, and *potential mobility* as the number of blank squares that are adjacent to opponent's pieces. These include the legal moves. A better measure of mobility would try to count only good moves. The following function computes both current and potential mobility for a player:

```
(defun mobility (player board)
  "Current mobility is the number of legal moves.
  Potential mobility is the number of blank squares
  adjacent to an opponent that are not legal moves.
  Returns current and potential mobility for player."
  (let ((opp (opponent player))
        (current 0) ; player's current mobility
        (potential 0)) ; player's potential mobility
    (dolist (square all-squares)
      (when (eql (bref board square) empty)
        (cond ((legal-p square player board)
```

```

(incf current))
((some #'(lambda (sq) (eq1 (bref board sq) opp))
  (neighbors square))
 (incf potential))))
(values current (+ current potential))))

```

Edge Stability

Success at Othello often hinges around edge play, and both Iago and Bill evaluate the edges carefully. Edge analysis is made easier by the fact that the edges are fairly independent of the interior of the board: once a piece is placed on the edge, no interior moves can flip it. This independence allows a simplifying assumption: to evaluate a position's edge strength, evaluate each of the four edges independently, without consideration of the interior of the board. The evaluation can be made more accurate by considering the X-squares to be part of the edge.

Even evaluating a single edge is a time-consuming task, so Bill and Iago compile away the evaluation by building a table of all possible edge positions. An "edge" according to Bill is ten squares: the eight actual edge squares and the two X-squares. Since each square can be black, white, or empty, there are 3^{10} or 59,049 possible edge positions—a large but manageable number.

The value of each edge position is determined by a process of successive approximation. Just as in a minimax search, we will need a static edge evaluation function to determine the value of a edge position without search. This static edge evaluation function is applied to every possible edge position, and the results are stored in a 59,049 element vector. The static evaluation is just a weighted sum of the occupied squares, with different weights given depending on if the piece is stable or unstable.

Each edge position's evaluation can be improved by a process of search. Iago uses a single ply search: given a position, consider all moves that could be made (including no move at all). Some moves will be clearly legal, because they flip pieces on the edge, but other moves will only be legal if there are pieces in the interior of the board to flip. Since we are only considering the edge, we don't know for sure if these moves are legal. They will be assigned probabilities of legality. The updated evaluation of a position is determined by the values and probabilities of each move. This is done by sorting the moves by value and then summing the product of the value times the probability that the move can be made. This process of iterative approximation is repeated five times for each position. At that point, Rosenbloom reports, the values have nearly converged.

In effect, this extends the depth of the normal alpha-beta search by including an edge-only search in the evaluation function. Since each edge position with n pieces is evaluated as a function of the positions with $n + 1$ pieces, the search is complete—it is an implicit 10-ply search.

Calculating edge stability is a bit more complicated than the other features. The first step is to define a variable, `*edge-table*`, which will hold the evaluation of each edge position, and a constant, `edge-and-x-lists`, which is a list of the squares on each of the four edges. Each edge has ten squares because the X-squares are included.

```
(defvar *edge-table* (make-array (expt 3 10))
  "Array of values to player-to-move for edge positions.")

(defconstant edge-and-x-lists
  '((22 11 12 13 14 15 16 17 18 27)
    (72 81 82 83 84 85 86 87 88 77)
    (22 11 21 31 41 51 61 71 81 72)
    (27 18 28 38 48 58 68 78 88 77))
  "The four edges (with their X-squares).")
```

Now for each edge we can compute an index into the edge table by building a 10-digit base-3 number, where each digit is 1 if the corresponding edge square is occupied by the player, 2 if by the opponent, and 0 if empty. The function `edge-index` computes this, and `edge-stability` sums the values of the four edge indexes.

```
(defun edge-index (player board squares)
  "The index counts 1 for player; 2 for opponent,
  on each square---summed as a base 3 number."
  (let ((index 0))
    (dolist (sq squares)
      (setq index (+ (* index 3)
                    (cond ((eql (bref board sq) empty) 0)
                          ((eql (bref board sq) player) 1)
                          (t 2)))))
    index))

(defun edge-stability (player board)
  "Total edge evaluation for player to move on board."
  (loop for edge-list in edge-and-x-lists
        sum (aref *edge-table*
                  (edge-index player board edge-list))))
```

The function `edge-stability` is all we will need in Iago's evaluation function, but we still need to generate the edge table. Since this needs to be done only once, we don't have to worry about efficiency. In particular, rather than invent a new data structure to represent edges, we will continue to use complete boards, even though they will be mostly empty. The computations for the edge table will be made on the top edge, from the point of view of black, with black to play. But the same table can be used for white, or for one of the other edges, because of the way the edge index is computed.

Each position in the table is first initialized to a static value computed by a kind of weighted-squares metric, but with different weights depending on if a piece is in

danger of being captured. After that, each position is updated by considering the possible moves that can be made from the position, and the values of each of these moves.

```
(defconstant top-edge (first edge-and-x-lists))

(defun init-edge-table ()
  "Initialize *edge-table*, starting from the empty board."
  ;; Initialize the static values
  (loop for n-pieces from 0 to 10 do
    (map-edge-n-pieces
      #'(lambda (board index)
          (setf (aref *edge-table* index)
                (static-edge-stability black board)))
        black (initial-board) n-pieces top-edge 0))
  ;; Now iterate five times trying to improve:
  (dotimes (i 5)
    ;; Do the indexes with most pieces first
    (loop for n-pieces from 9 downto 1 do
      (map-edge-n-pieces
        #'(lambda (board index)
            (setf (aref *edge-table* index)
                  (possible-edge-moves-value
                    black board index)))
          black (initial-board) n-pieces top-edge 0))))
```

The function `map-edge-n-pieces` iterates through all edge positions with a total of `n` pieces (of either color), applying a function to each such position. It also keeps a running count of the edge index as it goes. The function should accept two arguments: the board and the index. Note that a single board can be used for all the positions because squares are reset after they are used. The function has three cases: if the number of squares remaining is less than `n`, then it will be impossible to place `n` pieces on those squares, so we give up. If there are no more squares then `n` must also be zero, so this is a valid position, and the function `fn` is called. Otherwise we first try leaving the current square blank, then try filling it with player's piece, and then with the opponent's piece, in each case calling `map-edge-n-pieces` recursively.

```
(defun map-edge-n-pieces (fn player board n squares index)
  "Call fn on all edges with n pieces."
  ;; Index counts 1 for player; 2 for opponent
  (cond
    ((< (length squares) n) nil)
    ((null squares) (funcall fn board index))
    (t (let ((index3 (* 3 index))
              (sq (first squares)))
         (map-edge-n-pieces fn player board n (rest squares) index3))
```

```

(when (and (> n 0) (eql (bref board sq) empty))
  (setf (bref board sq) player)
  (map-edge-n-pieces fn player board (- n 1) (rest squares)
                    (+ 1 index3))
  (setf (bref board sq) (opponent player))
  (map-edge-n-pieces fn player board (- n 1) (rest squares)
                    (+ 2 index3))
  (setf (bref board sq) empty))))))

```

The function `possible-edge-moves-value` searches through all possible moves to determine an edge value that is more accurate than a static evaluation. It loops through every empty square on the edge, calling `possible-edge-move` to return a (*probability value*) pair. Since it is also possible for a player not to make any move at all on an edge, the pair (1.0 *current-value*) is also included.

```

(defun possible-edge-moves-value (player board index)
  "Consider all possible edge moves.
  Combine their values into a single number."
  (combine-edge-moves
   (cons
    (list 1.0 (aref *edge-table* index)) ;; no move
    (loop for sq in top-edge
          when (eql (bref board sq) empty)
            collect (possible-edge-move player board sq)))
   player))

```

The value of each position is determined by making the move on the board, then looking up in the table the value of the resulting position for the opponent, and negating it (since we are interested in the value to us, not to our opponent).

```

(defun possible-edge-move (player board sq)
  "Return a (prob val) pair for a possible edge move."
  (let ((new-board (replace (aref *ply-boards* player) board)))
    (make-move sq player new-board)
    (list (edge-move-probability player board sq)
          (- (aref *edge-table*
                  (edge-index (opponent player)
                              new-board top-edge))))))

```

The possible moves are combined with `combine-edge-moves`, which sorts the moves best-first. (Since `init-edge-table` started from black's perspective, black tries to maximize and white tries to minimize scores.) We then go down the moves, increasing the total value by the value of each move times the probability of the move, and decreasing the remaining probability by the probability of the move. Since there will

always be a least one move (pass) with probability 1.0, this is guaranteed to converge. In the end we round off the total value, so that we can do the run-time calculations with fixnums.

```
(defun combine-edge-moves (possibilities player)
  "Combine the best moves."
  (let ((prob 1.0)
        (val 0.0)
        (fn (if (eql player black) #'> #'<)))
    (loop for pair in (sort possibilities fn :key #'second)
          while (>= prob 0.0)
          do (incf val (* prob (first pair) (second pair)))
              (decf prob (* prob (first pair))))
    (round val)))
```

We still need to compute the probability that each possible edge move is legal. These probabilities should reflect things such as the fact that it is easy to capture a corner if the opponent is in the adjacent X-square, and very difficult otherwise. First we define some functions to recognize corner and X-squares and relate them to their neighbors:

```
(let ((corner/xsqs '((11 . 22) (18 . 27) (81. 72) (88 . 77))))
  (defun corner-p (sq) (assoc sq corner/xsqs))
  (defun x-square-p (sq) (rassoc sq corner/xsqs))
  (defun x-square-for (corner) (cdr (assoc corner corner/xsqs)))
  (defun corner-for (xsq) (car (rassoc xsq corner/xsqs))))
```

Now we consider the probabilities. There are four cases. First, since we don't know anything about the interior of the board, we assume each player has a 50% chance of being able to play in an X-square. Second, if we can show that a move is legal (because it flips opponent pieces on the edge) then it has 100% probability. Third, for the corner squares, we assign a 90% chance if the opponent occupies the X-square, 10% if it is empty, and only .1% if we occupy it. Otherwise, the probability is determined by the two neighboring squares: if a square is next to one or more opponents it is more likely we can move there; if it is next to our pieces it is less likely. If it is legal for the opponent to move into the square, then the chances are cut in half (although we may still be able to move there, since we move first).

```
(defun edge-move-probability (player board square)
  "What's the probability that player can move to this square?"
  (cond
    ((x-square-p square) .5) ;; X-squares
    ((legal-p square player board) 1.0) ;; immediate capture
    ((corner-p square) ;; move to corner depends on X-square
```

```

(let ((x-sq (x-square-for square)))
  (cond
    ((eql (bref board x-sq) empty) .1)
    ((eql (bref board x-sq) player) 0.001)
    (t .9))))
(t (/ (aref
      '#2A((.1 .4 .7)
           (.05 .3 *)
           (.01 * *))
      (count-edge-neighbors player board square)
      (count-edge-neighbors (opponent player) board square))
      (if (legal-p square (opponent player) board) 2 1))))

(defun count-edge-neighbors (player board square)
  "Count the neighbors of this square occupied by player."
  (count-if #'(lambda (inc)
                (eql (bref board (+ square inc)) player))
            '(+1 -1)))

```

Now we return to the problem of determining the static value of an edge position. This is computed by a weighted-squares metric, but the weights depend on the *stability* of each piece. A piece is called stable if it cannot be captured, unstable if it is in immediate danger of being captured, and semistable otherwise. A table of weights follows for each edge square and stability. Note that corner squares are always stable, and X-squares we will call semistable if the adjacent corner is taken, and unstable otherwise.

```

(defparameter *static-edge-table*
  '#2A(;stab semi un
      ( * 0 -2000) ; X
      ( 700 * *) ; corner
      (1200 200 -25) ; C
      (1000 200 75) ; A
      (1000 200 50) ; B
      (1000 200 50) ; B
      (1000 200 75) ; A
      (1200 200 -25) ; C
      ( 700 * *) ; corner
      ( * 0 -2000) ; X
  ))

```

The static evaluation then just sums each piece's value according to this table:

```
(defun static-edge-stability (player board)
  "Compute this edge's static stability"
  (loop for sq in top-edge
        for i from 0
        sum (cond
              ((eq1 (bref board sq) empty) 0)
              ((eq1 (bref board sq) player)
               (aref *static-edge-table* i
                    (piece-stability board sq)))
              (t (- (aref *static-edge-table* i
                          (piece-stability board sq)))))))
```

The computation of stability is fairly complex. It centers around finding the two "pieces," p1 and p2, which lay on either side of the piece in question and which are not of the same color as the piece. These "pieces" may be empty, or they may be off the board. A piece is unstable if one of the two is empty and the other is the opponent; it is semistable if there are opponents on both sides and at least one empty square to play on, or if it is surrounded by empty pieces. Finally, if either p1 or p2 is nil then the piece is stable, since it must be connected by a solid wall of pieces to the corner.

```
(let ((stable 0) (semi-stable 1) (unstable 2))
```

```
(defun piece-stability (board sq)
  (cond
    ((corner-p sq) stable)
    ((x-square-p sq)
     (if (eq1 (bref board (corner-for sq)) empty)
         unstable semi-stable))
    (t (let* ((player (bref board sq))
              (opp (opponent player))
              (p1 (find player board :test-not #'eq1
                       :start sq :end 19))
              (p2 (find player board :test-not #'eq1
                       :start 11 :end sq
                       :from-end t)))
          (cond
            ;; unstable pieces can be captured immediately
            ;; by playing in the empty square
            ((or (and (eq1 p1 empty) (eq1 p2 opp))
                 (and (eq1 p2 empty) (eq1 p1 opp))))
             unstable)
            ;; semi-stable pieces might be captured
            ((and (eq1 p1 opp) (eq1 p2 opp))
```



```

        (find empty board :start 11 :end 19))
      semi-stable)
    ((and (eql p1 empty) (eql p2 empty))
      semi-stable)
    ;; Stable pieces can never be captured
    (t stable))))))

```

The edge table can now be built by a call to `init-edge-table`. After the table is built once, it is a good idea to save it so that we won't need to repeat the initialization. We could write simple routines to dump the table into a file and read it back in, but it is faster and easier to use existing tools that already do this job quite well: `compile-file` and `load`. All we have to do is create and compile a file containing the single line:

```
(setf *edge-table* '#.*edge-table*)
```

The `#.` read macro evaluates the following expression at read time. Thus, the compiler will see and compile the current edge table. It will be able to store this more compactly and load it back in more quickly than if we printed the contents of the vector in decimal (or any other base).

Combining the Factors

Now we have a measure of the three factors: current mobility, potential mobility, and edge stability. All that remains is to find a good way to combine them into a single evaluation metric. The combination function used by Rosenbloom (1982) is a linear combination of the three factors, but each factor's coefficient is dependent on the move number. Rosenbloom's features are normalized to the range $[-1000, 1000]$; we normalize to the range $[-1, 1]$ by doing a division after multiplying by the coefficient. That allows us to use `fixnums` for the coefficients. Since our three factors are not calculated in quite the same way as Rosenbloom's, it is not surprising that his coefficients are not the best for our program. The edge coefficient was doubled and the potential coefficient cut by a factor of five.

```

(defun Iago-eval (player board)
  "Combine edge-stability, current mobility and
  potential mobility to arrive at an evaluation."
  ;; The three factors are multiplied by coefficients
  ;; that vary by move number:
  (let ((c-edg (+ 312000 (* 6240 *move-number*)))
        (c-cur (if (< *move-number* 25)
                    (+ 50000 (* 2000 *move-number*))
                    (+ 75000 (* 1000 *move-number*))))
        (c-pot 20000))

```

```

(multiple-value-bind (p-cur p-pot)
  (mobility player board)
  (multiple-value-bind (o-cur o-pot)
    (mobility (opponent player) board)
    ;; Combine the three factors into one sum:
    (+ (round (* c-edg (edge-stability player board)) 32000)
       (round (* c-cur (- p-cur o-cur)) (+ p-cur o-cur 2))
       (round (* c-pot (- p-pot o-pot)) (+ p-pot o-pot 2))))))

```

Finally, we are ready to code the Iago function. Given a search depth, Iago returns a strategy that will do alpha-beta search to that depth using the Iago-eval evaluation function. This version of Iago was able to defeat the modified weighted-squares strategy in 8 of 10 games at 3 ply, and 9 of 10 at 4 ply. On an Explorer II, 4-ply search takes about 20 seconds per move. At 5 ply, many moves take over a minute, so the program runs the risk of forfeiting. At 3 ply, the program takes only a few seconds per move, but it still was able to defeat the author in five straight games, by scores of 50-14, 64-0, 51-13, 49-15 and 36-28. Despite these successes, it is likely that the evaluation function could be improved greatly with a little tuning of the parameters.

```

(defun Iago (depth)
  "Use an approximation of Iago's evaluation function."
  (alpha-beta-searcher3 depth #'iago-eval))

```

18.13 Other Techniques

There are many other variations that can be tried to speed up the search and improve play. Unfortunately, choosing among the techniques is a bit of a black art. You will have to experiment to find the combination that is best for each domain and each evaluation function. Most of the following techniques were incorporated, or at least considered and rejected, in Bill.

Iterative Deepening

We have seen that the average branching factor for Othello is about 10. This means that searching to depth $n + 1$ takes roughly 10 times longer than search to depth n . Thus, we should be willing to go to a lot of overhead before we search one level deeper, to assure two things: that search will be done efficiently, and that we won't forfeit due to running out of time. A by-now familiar technique, iterative deepening (see chapters 6 and 14), serves both these goals.

Iterative deepening is used as follows. The strategy determines how much of the remaining time to allocate to each move. A simple strategy could allocate a constant amount of time for each move, and a more sophisticated strategy could allocate more time for moves at crucial points in the game. Once the time allocation is determined for a move, the strategy starts an iterative deepening alpha-beta search. There are two complications: First, the search at n ply keeps track of the best moves, so that the search at $n + 1$ ply will have better ordering information. In many cases it will be faster to do both the n and $n + 1$ ply searches with the ordering information than to do only the $n + 1$ ply search without it. Second, we can monitor how much time has been taken searching each ply, and cut off the search when searching one more ply would exceed the allocated time limit. Thus, iterative-deepening search degrades gracefully as time limits are imposed. It will give a reasonable answer even with a short time allotment, and it will rarely exceed the allotted time.

Forward Pruning

One way to cut the number of positions searched is to replace the legal move generator with a *plausible* move generator: in other words, only consider good moves, and never even look at moves that seem clearly bad. This technique is called *forward pruning*. It has fallen on disfavor because of the difficulty in determining which moves are plausible. For most games, the factors that would go into a plausible move generator would be duplicated in the static evaluation function anyway, so forward pruning would require more effort without much gain. Worse, forward pruning could rule out a brilliant sacrifice—a move that looks bad initially but eventually leads to a gain.

For some games, forward pruning is a necessity. The game of Go, for example, is played on a 19 by 19 board, so the first player has 361 legal moves, and a 6-ply search would involve over 2 quadrillion positions. However, many good Go programs can be viewed as not doing forward pruning but doing abstraction. There might be 30 empty squares in one portion of the board, and the program would treat a move to any of these squares equivalently.

Bill uses forward pruning in a limited way to rule out certain moves adjacent to the corners. It does this not to save time but because the evaluation function might lead to such a move being selected, even though it is in fact a poor move. In other words, forward pruning is used to correct a bug in the evaluation function cheaply.

Nonspeculative Forward Pruning

This technique makes use of the observation that there are limits in the amount the evaluation function can change from one position to the next. For example, if we are using the count difference as the evaluation function, then the most a move can change the evaluation is +37 (one for placing a piece in the corner, and six captures in each of the three directions). The smallest change is 0 (if the player is forced to

pass). Thus, if there are 2 ply left in the search, and the backed-up value of position A has been established as 38 points better than the static value of position B , then it is useless to expand position B . This assumes that we are evaluating every position, perhaps to do sorted ordering or iterative deepening. It also assumes that no position in the search tree is a final position, because then the evaluation could change by more than 37 points. In conclusion, it seems that nonspeculative forward pruning is not very useful for Othello, although it may play a role in other games.

Aspiration Search

Alpha-beta search is initiated with the achievable and cutoff boundaries set to losing-value and winning-value, respectively. In other words, the search assumes nothing: the final position may be anything from a loss to a win. But suppose we are in a situation somewhere in the mid-game where we are winning by a small margin (say the static evaluation for the current position is 50). In most cases, a single move will not change the evaluation by very much. Therefore, if we invoked the alpha-beta search with a window defined by boundaries of, say, 0 and 100, two things can happen: if the actual backed-up evaluation for this position is in fact in the range 0 to 100, then the search will find it, and it will be found quickly, because the reduced window will cause more pruning. If the actual value is not in the range, then the value returned will reflect that, and we can search again using a larger window. This is called aspiration search, because we aspire to find a value within a given window. If the window is chosen well, then often we will succeed and will have saved some search time.

Pearl (1984) suggests an alternative called zero-window search. At each level, the first possible move, which we'll call m , is searched using a reasonably wide window to determine its exact value, which we'll call v . Then the remaining possible moves are searched using v as both the lower and upper bounds of the window. Thus, the result of the search will tell if each subsequent move is better or worse than m , but won't tell how much better or worse. There are three outcomes for zero-window search. If no move turns out to be better than m , then stick with m . If a single move is better, then use it. If several moves are better than m , then they have to be searched again using a wider window to determine which is best.

There is always a trade-off between time spent searching and information gained. Zero-window search makes an attractive trade-off: we gain some search time by losing information about the value of the best move. We are still guaranteed of finding the best move, we just don't know its exact value.

Bill's zero-window search takes only 63% of the time taken by full alpha-beta search. It is effective because Bill's move-ordering techniques ensure that the first move is often best. With random move ordering, zero-window search would not be effective.

Think-Ahead

A program that makes its move and then waits for the opponent's reply is wasting half the time available to it. A better use of time is to compute, or *think-ahead* while the opponent is moving. Think-ahead is one factor that helps Bill defeat Iago. While many programs have done think-ahead by choosing the most likely move by the opponent and then starting an iterative-deepening search assuming that move, Bill's algorithm is somewhat more complex. It can consider more than one move by the opponent, depending on how much time is available.

Hashing and Opening Book Moves

We have been treating the search space as a tree, but in general it is a directed acyclic graph (dag): there may be more than one way to reach a particular position, but there won't be any loops, because every move adds a new piece. This raises the question we explored briefly in section 6.4: should we treat the search space as a tree or a graph? By treating it as a graph we eliminate duplicate evaluations, but we have the overhead of storing all the previous positions, and of checking to see if a new position has been seen before. The decision must be based on the proportion of duplicate positions that are actually encountered in play. One compromise solution is to store in a hash table a partial encoding of each position, encoded as, say, a single fixnum (one word) instead of the seven or so words needed to represent a full board. Along with the encoding of each position, store the move to try first. Then, for each new position, look in the hash table, and if there is a hit, try the corresponding move first. The move may not even be legal, if there is an accidental hash collision, but there is a good chance that the move will be the right one, and the overhead is low.

One place where it is clearly worthwhile to store information about previous positions is in the opening game. Since there are fewer choices in the opening, it is a good idea to compile an opening "book" of moves and to play by it as long as possible, until the opponent makes a move that departs from the book. Book moves can be gleaned from the literature, although not very much has been written about Othello (as compared to openings in chess). However, there is a danger in following expert advice: the positions that an expert thinks are advantageous may not be the same as the positions from which our program can play well. It may be better to compile the book by playing the program against itself and determining which positions work out best.

The End Game

It is also a good idea to try to save up time in the midgame and then make an all-out effort to search the complete game tree to completion as soon as feasible. Bill can search to completion from about 14 ply out. Once the search is done, of course, the

most promising lines of play should be saved so that it won't be necessary to solve the game tree again.

Metareasoning

If it weren't for the clock, Othello would be a trivial game: just search the complete game tree all the way to the end, and then choose the best move. The clock imposes a complication: we have to make all our moves before we run out of time. The algorithms we have seen so far manage the clock by allocating a certain amount of time to each move, such that the total time is guaranteed (or at least very likely) to be less than the allotted time. This is a very crude policy. A finer-grained way of managing time is to consider computation itself as a possible move. That is, at every tick of the clock, we need to decide if it is better to stop and play the best move we have computed so far or to continue and try to compute a better move. It will be better to compute more only in the case where we eventually choose a better move; it will be better to stop and play only in the case where we would otherwise forfeit due to time constraints, or be forced to make poor choices later in the game. An algorithm that includes computation as a possible move is called a metareasoning system, because it reasons about how much to reason.

Russell and Wefald (1989) present an approach based on this view. In addition to an evaluation function, they assume a variance function, which gives an estimate of how much a given position's true value is likely to vary from its static value. At each step, their algorithm compares the value and variance of the best move computed so far and the second best move. If the best move is clearly better than the second best (taking variance into account), then there is no point computing any more. Also, if the top two moves have similar values but both have very low variance, then computing will not help much; we can just choose one of the two at random.

For example, if the board is in a symmetric position, then there may be two symmetric moves that will have identical value. By searching each move's subtree more carefully, we soon arrive at a low variance for both moves, and then we can choose either one, without searching further. Of course, we could also add special-case code to check for symmetry, but the metareasoning approach will work for nonsymmetric cases as well as symmetric ones. If there is a situation where two moves both lead to a clear win, it won't waste time choosing between them.

The only situation where it makes sense to continue computing is when there are two moves with high variance, so that it is uncertain if the true value of one exceeds the other. The metareasoning algorithm is predicated on devoting time to just this case.

Learning

From the earliest days of computer game playing, it was realized that a championship program would need to learn to improve itself. Samuel (1959) describes a program that plays checkers and learns to improve its evaluation function. The evaluation function is a linear combination of features, such as the number of pieces for each player, the number of kings, the number of possible forks, and so on. Learning is done by a hill-climbing search procedure: change one of the coefficients for one of the features at random, and then see if the changed evaluation function is better than the original one.

Without some guidance, this hill-climbing search would be very slow. First, the space is very large—Samuel used 38 different features, and although he restricted the coefficients to be a power of two between 0 and 20, that still leaves 21^{38} possible evaluation functions. Second, the obvious way of determining the relative worth of two evaluation functions—playing a series of games between them and seeing which wins more often—is quite time-consuming.

Fortunately, there is a faster way of evaluating an evaluation function. We can apply the evaluation function to a position and compare this static value with the backed-up value determined by an alpha-beta search. If the evaluation function is accurate, the static value should correlate well with the backed-up value. If it does not correlate well, the evaluation function should be changed in such a way that it does. This approach still requires the trial-and-error of hill-climbing, but it will converge much faster if we can gain information from every position, rather than just from every game.

In the past few years there has been increased interest in learning by a process of guided search. *Neural nets* are one example of this. They have been discussed elsewhere. Another example is *genetic learning* algorithms. These algorithms start with several candidate solutions. In our case, each candidate would consist of a set of coefficients for an evaluation function. On each generation, the genetic algorithm sees how well each candidate does. The worst candidates are eliminated, and the best ones “mate” and “reproduce”—two candidates are combined in some way to yield a new one. If the new offspring has inherited both its parents’ good points, then it will prosper; if it has inherited both its parents’ bad points, then it will quickly die out. Either way, the idea is that natural selection will eventually yield a high-quality solution. To increase the chances of this, it is a good idea to allow for mutations: random changes in the genetic makeup of one of the candidates.

18.14 History and References

Lee and Mahajan (1986, 1990) present the current top Othello program, Bill. Their description outlines all the techniques used but does not go into enough detail to allow

the reader to reconstruct the program. Bill is based in large part on Rosenbloom's Iago program. Rosenbloom's article (1982) is more thorough. The presentation in this chapter is based largely on this article, although it also contains some ideas from Bill and from other sources.




The journal *Othello Quarterly* is the definitive source for reports on both human and computer Othello games and strategies.

The most popular game for computer implementation is chess. Shannon (1950a,b) speculated that a computer might play chess. In a way, this was one of the boldest steps in the history of AI. Today, writing a chess program is a challenging but feasible project for an undergraduate. But in 1950, even suggesting that such a program might be possible was a revolutionary step that changed the way people viewed these arithmetic calculating devices. Shannon introduced the ideas of a game tree search, minimaxing, and evaluation functions—ideas that remain intact to this day. Marsland (1990) provides a good short introduction to computer chess, and David Levy has two books on the subject (1976, 1988). It was Levy, an international chess master, who in 1968 accepted a bet from John McCarthy, Donald Michie, and others that a computer chess program would not beat him in the next ten years. Levy won the bet. Levy's *Heuristic Programming* (1990) and *Computer Games* (1988) cover a variety of computer game playing programs. The studies by DeGroot (1965, 1966) give a fascinating insight into the psychology of chess masters.

Knuth and Moore (1975) analyze the alpha-beta algorithm, and Pearl's book *Heuristics* (1984) covers all kinds of heuristic search, games included.

Samuel (1959) is the classic work on learning evaluation function parameters. It is based on the game of checkers. Lee and Mahajan (1990) present an alternative learning mechanism, using Bayesian classification to learn an evaluation function that optimally distinguishes winning positions from losing positions. Genetic algorithms are discussed by L. Davis (1987, 1991) and Goldberg (1989).

18.15 Exercises

-  **Exercise 18.3 [s]** How many different Othello positions are there? Would it be feasible to store the complete game tree and thus have a perfect player?
-  **Exercise 18.4 [m]** At the beginning of this chapter, we implemented pieces as an enumerated type. There is no built-in facility in Common Lisp for doing this, so we had to introduce a series of `defconstant` forms. Define a macro for defining enumerated types. What else should be provided besides the constants?
-  **Exercise 18.5 [h]** Add `fixnum` and `speed` declarations to the Iago evaluation func-

tion and the alpha-beta code. How much does this speed up Iago? What other efficiency measures can you take?

- ?** **Exercise 18.6 [h]** Implement an iterative deepening search that allocates time for each move and checks between each iteration if the time is exceeded.
- ?** **Exercise 18.7 [h]** Implement zero-window search, as described in section 18.13.
- ?** **Exercise 18.8 [d]** Read the references on Bill (Lee and Mahajan 1990, and 1986 if you can get it), and reimplement Bill's evaluation function as best you can, using the table-based approach. It will also be helpful to read Rosenbloom 1982.
- ?** **Exercise 18.9 [d]** Improve the evaluation function by tuning the parameters, using one of the techniques described in section 18.13.
- ?** **Exercise 18.10 [h]** Write move-generation and evaluation functions for another game, such as chess or checkers.

18.16 Answers

Answer 18.2 The weighted-squares strategy wins the first game by 20 pieces, but when count-difference plays first, it captures all the pieces on its fifth move. These two games alone are not enough to determine the best strategy; the function `othello-series` on page 626 shows a better comparison.

Answer 18.3 $3^{64} = 3,433,683,820,292,512,484,657,849,089,281$. No.

Answer 18.4 Besides the constants, we provide a `deftype` for the type itself, and conversion routines between integers and symbols:

```
(defmacro define-enumerated-type (type &rest elements)
  "Represent an enumerated type with integers 0-n."
  `(progn
    (deftype ,type () '(integer 0 ,(- (length elements) 1)))
    (defun ,(symbol type '->symbol) (,type)
      (elt ',elements ,type))
    (defun ,(symbol 'symbol-> type) (symbol)
      (position symbol ',elements))
    ,@(loop for element in elements
            for i from 0
            collect '(defconstant ,element ,i))))
```

Here's how the macro would be used to define the piece data type, and the code produced:

```
> (macroexpand
  '(define-enumerated-type piece
    empty black white outer))

(PROGN
 (DEFTYPE PIECE () '(INTEGER 0 3))
 (DEFUN PIECE->SYMBOL (PIECE)
  (ELT '(EMPTY BLACK WHITE OUTER) PIECE))
 (DEFUN SYMBOL->PIECE (SYMBOL)
  (POSITION SYMBOL '(EMPTY BLACK WHITE OUTER)))
 (DEFCONSTANT EMPTY 0)
 (DEFCONSTANT BLACK 1)
 (DEFCONSTANT WHITE 2)
 (DEFCONSTANT OUTER 3))
```

A more general facility would, like `defstruct`, provide for several options. For example, it might allow for a documentation string for the type and each constant, and for a `:conc-name`, so the constants could have names like `piece-empty` instead of `empty`. This would avoid conflicts with other types that wanted to use the same names. The user might also want the ability to start the values at some number other than zero, or to assign specific values to some of the symbols.

CHAPTER 19

Introduction to Natural Language

Language is everywhere. It permeates our thoughts, mediates our relations with others, and even creeps into our dreams. The overwhelming bulk of human knowledge is stored and transmitted in language. Language is so ubiquitous that we take it for granted, but without it, society as we know it would be impossible.

—Ronand Langacker
Language and its Structure (1967)

Anatural language is a language spoken by people, such as English, German, or Tagalog. This is in opposition to artificial languages like Lisp, FORTRAN, or Morse code. Natural language processing is an important part of AI because language is intimately connected to thought. One measure of this is the number of important books that mention language and thought in the title: in AI, Schank and Colby's *Computer Models of Thought and Language*; in linguistics, Whorf's *Language, Thought, and Reality* (and Chomsky's *Language and Mind*;) in philosophy, Fodor's *The Language of Thought*; and in psychology, Vygotsky's *Thought and Language* and John Anderson's *Language, Memory, and Thought*. Indeed, language is

the trait many think of as being the most characteristic of humans. Much controversy has been generated over the question of whether animals, especially primates and dolphins, can use and “understand” language. Similar controversy surrounds the same question asked of computers.

The study of language has been traditionally separated into two broad classes: syntax, or grammar, and semantics, or meaning. Historically, syntax has achieved the most attention, largely because on the surface it is more amenable to formal and semiformal methods. Although there is evidence that the boundary between the two is at best fuzzy, we still maintain the distinction for the purposes of these notes. We will cover the “easier” part, syntax, first, and then move on to semantics.

A good artificial language, like Lisp or C, is unambiguous. There is only one interpretation for a valid Lisp expression. Of course, the interpretation may depend on the state of the current state of the Lisp world, such as the value of global variables. But these dependencies can be explicitly enumerated, and once they are spelled out, then there can only be one meaning for the expression.¹

Natural language does not work like this. Natural expressions are inherently ambiguous, depending on any number of factors that can never be quite spelled out completely. It is perfectly reasonable for two people to disagree on what some other person meant by a natural language expression. (Lawyers and judges make their living largely by interpreting natural language expressions—laws—that are meant to be unambiguous but are not.)

This chapter is a brief introduction to natural language processing. The next chapter gives a more thorough treatment from the point of view of logic grammars, and the chapter after that puts it all together into a full-fledged system.

19.1 Parsing with a Phrase-Structure Grammar

To parse a sentence means to recover the constituent structure of the sentence—to discover what sequence of generation rules could have been applied to come up with the sentence. In general, there may be several possible derivations, in which case we say the sentence is grammatically ambiguous. In certain circles, the term “parse” means to arrive at an understanding of a sentence’s meaning, not just its grammatical form. We will attack that more difficult question later.

¹Some erroneous expressions are underspecified and may return different results in different implementations, but we will ignore that problem.

We start with the grammar defined on page 39 for the generate program:

```
(defvar *grammar* "The grammar used by GENERATE.")

(defparameter *grammar1*
  '((Sentence -> (NP VP))
    (NP -> (Art Noun))
    (VP -> (Verb NP))
    (Art -> the a)
    (Noun -> man ball woman table)
    (Verb -> hit took saw liked)))
```

Our parser takes as input a list of words and returns a structure containing the parse tree and the unparsed words, if any. That way, we can parse the remaining words under the next category to get compound rules. For example, in parsing “the man saw the table,” we would first parse “the man,” returning a structure representing the noun phrase, with the remaining words “saw the table.” This remainder would then be parsed as a verb phrase, returning no remainder, and the two phrases could then be joined to form a parse that is a complete sentence with no remainder.

Before proceeding, I want to make a change in the representation of grammar rules. Currently, rules have a left-hand side and a list of alternative right-hand sides. But each of these alternatives is really a separate rule, so it would be more modular to write them separately. For the generate program it was fine to have them all together, because that made processing choices easier, but now I want a more flexible representation. Later on we will want to add more information to each rule, like the semantics of the assembled left-hand side, and constraints between constituents on the right-hand side, so the rules would become quite large indeed if we didn’t split up the alternatives. I also take this opportunity to clear up the confusion between words and category symbols. The convention is that a right-hand side can be either an atom, in which case it is a word, or a list of symbols, which are then all interpreted as categories. To emphasize this, I include “noun” and “verb” as nouns in the grammar `*grammar3*`, which is otherwise equivalent to the previous `*grammar1*`.

```
(defparameter *grammar3*
  '((Sentence -> (NP VP))
    (NP -> (Art Noun))
    (VP -> (Verb NP))
    (Art -> the) (Art -> a)
    (Noun -> man) (Noun -> ball) (Noun -> woman) (Noun -> table)
    (Noun -> noun) (Noun -> verb)
    (Verb -> hit) (Verb -> took) (Verb -> saw) (Verb -> liked)))

(setf *grammar* *grammar3*)
```

I also define the data types `rule`, `parse`, and `tree`, and some functions for getting

at the rules. Rules are defined as structures of type list with three slots: the left-hand side, the arrow (which should always be represented as the literal `->`) and the right-hand side. Compare this to the treatment on page 40.

```
(defstruct (rule (:type list)) lhs -> rhs)

(defstruct (parse) "A parse tree and a remainder." tree rem)

;; Trees are of the form: (lhs . rhs)
(defun new-tree (cat rhs) (cons cat rhs))
(defun tree-lhs (tree) (first tree))
(defun tree-rhs (tree) (rest tree))

(defun parse-lhs (parse) (tree-lhs (parse-tree parse)))

(defun lexical-rules (word)
  "Return a list of rules with word on the right-hand side."
  (find-all word *grammar* :key #'rule-rhs :test #'equal))

(defun rules-starting-with (cat)
  "Return a list of rules where cat starts the rhs."
  (find-all cat *grammar*
    :key #'(lambda (rule) (first-or-nil (rule-rhs rule)))))

(defun first-or-nil (x)
  "The first element of x if it is a list; else nil."
  (if (consp x) (first x) nil))
```

Now we're ready to define the parser. The main function `parser` takes a list of words to parse. It calls `parse`, which returns a list of all parses that parse some subsequence of the words, starting at the beginning. `parser` keeps only the parses with no remainder—that is, the parses that span all the words.

```
(defun parser (words)
  "Return all complete parses of a list of words."
  (mapcar #'parse-tree (complete-parses (parse words))))

(defun complete-parses (parses)
  "Those parses that are complete (have no remainder)."
  (find-all-if #'null parses :key #'parse-rem))
```

The function `parse` looks at the first word and considers each category it could be. It makes a parse of the first word under each category, and calls `extend-parse` to try to continue to a complete parse. `parse` uses `mapcar` to append together all the resulting parses. As an example, suppose we are trying to parse "the man took the ball." `parse` would find the single lexical rule for "the" and call `extend-parse` with a parse with tree (Art the) and remainder "man took the ball," with no more categories needed.

extend-parse has two cases. If the partial parse needs no more categories to be complete, then it returns the parse itself, along with any parses that can be formed by extending parses starting with the partial parse. In our example, there is one rule starting with Art, namely (NP -> (Art Noun)), so the function would try to extend the parse tree (NP (Art the)) with remainder "man took the ball," with the category Noun needed. That call to extend-parse represents the second case. We first parse "man took the ball," and for every parse that is of category Noun (there will be only one), we combine with the partial parse. In this case we get (NP (Art the) (Noun man)). This gets extended as a sentence with a VP needed, and eventually we get a parse of the complete list of words.

```
(defun parse (words)
  "Bottom-up parse, returning all parses of any prefix of words."
  (unless (null words)
    (mapcan #'(lambda (rule)
              (extend-parse (rule-lhs rule) (list (first words))
                           (rest words) nil))
            (lexical-rules (first words)))))

(defun extend-parse (lhs rhs rem needed)
  "Look for the categories needed to complete the parse."
  (if (null needed)
      ;; If nothing needed, return parse and upward extensions
      (let ((parse (make-parse :tree (new-tree lhs rhs) :rem rem)))
        (cons parse
              (mapcan
               #'(lambda (rule)
                   (extend-parse (rule-lhs rule)
                                 (list (parse-tree parse))
                                 rem (rest (rule-rhs rule))))
               (rules-starting-with lhs))))
      ;; otherwise try to extend rightward
      (mapcan
       #'(lambda (p)
           (if (eq (parse-lhs p) (first needed))
               (extend-parse lhs (append1 rhs (parse-tree p))
                             (parse-rem p) (rest needed))))
       (parse rem))))
```

This makes use of the auxiliary function append1:

```
(defun append1 (items item)
  "Add item to end of list of items."
  (append items (list item)))
```

Some examples of the parser in action are shown here:

```
> (parser '(the table))
((NP (ART THE) (NOUN TABLE)))

> (parser '(the ball hit the table))
((SENTENCE (NP (ART THE) (NOUN BALL))
            (VP (VERB HIT)
                  (NP (ART THE) (NOUN TABLE))))))

> (parser '(the noun took the verb))
((SENTENCE (NP (ART THE) (NOUN NOUN))
            (VP (VERB TOOK)
                  (NP (ART THE) (NOUN VERB))))))
```

19.2 Extending the Grammar and Recognizing Ambiguity

Overall, the parser seems to work fine, but the range of sentences we can parse is quite limited with the current grammar. The following grammar includes a wider variety of linguistic phenomena: adjectives, prepositional phrases, pronouns, and proper names. It also uses the usual linguistic conventions for category names, summarized in the table below:

	Category	Examples
S	Sentence	<i>John likes Mary</i>
NP	Noun Phrase	<i>John; a blue table</i>
VP	Verb Phrase	<i>likes Mary; hit the ball</i>
PP	Prepositional Phrase	<i>to Mary; with the man</i>
A	Adjective	<i>little; blue</i>
A+	A list of one or more adjectives	<i>little blue</i>
D	Determiner	<i>the; a</i>
N	Noun	<i>ball; table</i>
Name	Proper Name	<i>John; Mary</i>
P	Preposition	<i>to; with</i>
Pro	Pronoun	<i>you; me</i>
V	Verb	<i>liked; hit</i>

Here is the grammar:

```
(defparameter *grammar4*
  '((S -> (NP VP))
    (NP -> (D N))
    (NP -> (D A+ N))
    (NP -> (NP PP))
    (NP -> (Pro))
    (NP -> (Name))
    (VP -> (V NP))
    (VP -> (V))
    (VP -> (VP PP))
    (PP -> (P NP))
    (A+ -> (A))
    (A+ -> (A A+))
    (Pro -> I) (Pro -> you) (Pro -> he) (Pro -> she)
    (Pro -> it) (Pro -> me) (Pro -> him) (Pro -> her)
    (Name -> John) (Name -> Mary)
    (A -> big) (A -> little) (A -> old) (A -> young)
    (A -> blue) (A -> green) (A -> orange) (A -> perspicuous)
    (D -> the) (D -> a) (D -> an)
    (N -> man) (N -> ball) (N -> woman) (N -> table) (N -> orange)
    (N -> saw) (N -> saws) (N -> noun) (N -> verb)
    (P -> with) (P -> for) (P -> at) (P -> on) (P -> by) (P -> of) (P -> in)
    (V -> hit) (V -> took) (V -> saw) (V -> liked) (V -> saws)))

(setf *grammar* *grammar4*)
```

Now we can parse more interesting sentences, and we can see a phenomenon that was not present in the previous examples: ambiguous sentences. The sentence “The man hit the table with the ball” has two parses, one where the ball is the thing that hits the table, and the other where the ball is on or near the table. parser finds both of these parses (although of course it assigns no meaning to either parse):

```
> (parser '(The man hit the table with the ball))
((S (NP (D THE) (N MAN))
  (VP (VP (V HIT) (NP (D THE) (N TABLE)))
    (PP (P WITH) (NP (DTHE) (N BALL))))))
(S (NP (D THE) (N MAN))
  (VP (V HIT)
    (NP (NP (D THE) (N TABLE))
      (PP (P WITH) (NP (DTHE) (N BALL)))))))
```

Sentences are not the only category that can be ambiguous, and not all ambiguities have to be between parses in the same category. Here we see a phrase that is ambiguous between a sentence and a noun phrase:

```
> (parser '(the orange saw))
((S (NP (D THE) (N ORANGE)) (VP (V SAW)))
 (NP (D THE) (A+ (A ORANGE)) (N SAW)))
```

19.3 More Efficient Parsing

With more complex grammars and longer sentences, the parser starts to slow down. The main problem is that it keeps repeating work. For example, in parsing “The man hit the table with the ball,” it has to reparse “with the ball” for both of the resulting parses, even though in both cases it receives the same analysis, a PP. We have seen this problem before and have already produced an answer: memoization (see section 9.6). To see how much memoization will help, we need a benchmark:

```
> (setf s (generate 's))
( (THE PERSPICUOUS BIG GREEN BALL BY A BLUE WOMAN WITH A BIG MAN
  HIT A TABLE BY THE SAW BY THE GREEN ORANGE) )
> (time (length (parser s)))
Evaluation of (LENGTH (PARSER S)) took 33.11 Seconds of elapsed time.
10
```

The sentence S has 10 parses, since there are two ways to parse the subject NP and five ways to parse the VP. It took 33 seconds to discover these 10 parses with the parse function as it was written.

We can improve this dramatically by memoizing parse (along with the table-lookup functions). Besides memoizing, the only change is to clear the memoization table within parser.

```
(memoize 'lexical-rules)
(memoize 'rules-starting-with)
(memoize 'parse :test #'eq)

(defun parser (words)
  "Return all complete parses of a list of words."
  (clear-memoize 'parse) ;***
  (mapcar #'parse-tree (complete-parses (parse words))))
```

In normal human language use, memoization would not work very well, since the interpretation of a phrase depends on the context in which the phrase was uttered. But with context-free grammars we have a guarantee that the context cannot affect the interpretation. The call (parse words) must return all possible parses for the words. We are free to choose between the possibilities based on contextual information, but

context can never supply a new interpretation that is not in the context-free list of parses.

The function `use` is introduced to tell the table-lookup functions that they are out of date whenever the grammar changes:

```
(defun use (grammar)
  "Switch to a new grammar."
  (clear-memoize 'rules-starting-with)
  (clear-memoize 'lexical-rules)
  (length (setf *grammar* grammar)))
```

Now we run the benchmark again with the memoized version of `parse`:

```
> (time (length (parser s)))
Evaluation of (LENGTH (PARSER S 'S)) took .13 Seconds of elapsed time.
10
```

By memoizing `parse` we reduce the parse time from 33 to .13 seconds, a 250-fold speed-up. We can get a more systematic comparison by looking at a range of examples. For example, consider sentences of the form “The man hit the table [with the ball]*” for zero or more repetitions of the PP “with the ball.” In the following table we record N , the number of repetitions of the PP, along with the number of resulting parses², and for both memoized and unmemoized versions of `parse`, the number of seconds to produce the parse, the number of parses per second (PPS), and the number of recursive calls to `parse`. The performance of the memoized version is quite acceptable; for $N=5$, a 20-word sentence is parsed into 132 possibilities in .68 seconds, as opposed to the 20 seconds it takes in the unmemoized version.

²The number of parses of sentences of this kind is the same as the number of bracketings of an arithmetic expression, or the number of binary trees with a given number of leaves. The resulting sequence (1,2,5,14,42, . . .) is known as the Catalan Numbers. This kind of ambiguity is discussed by Church and Patil (1982) in their article *Coping with Syntactic Ambiguity, or How to Put the Block in the Box on the Table*.

N	Parses	Memoized			Unmemoized		
		Secs	PPS	Calls	Secs	PPS	Calls
0	1	0.02	60	4	0.02	60	17
1	2	0.02	120	11	0.07	30	96
2	5	0.05	100	21	0.23	21	381
3	14	0.10	140	34	0.85	16	1388
4	42	0.23	180	50	3.17	13	4999
5	132	0.68	193	69	20.77	6	18174
6	429	1.92	224	91	—		
7	1430	5.80	247	116	—		
8	4862	20.47	238	144	—		

? **Exercise 19.1 [h]** It seems that we could be more efficient still by memoizing with a table consisting of a vector whose length is the number of words in the input (plus one). Implement this approach and see if it entails less overhead than the more general hash table approach.

19.4 The Unknown-Word Problem

As it stands, the parser cannot deal with unknown words. Any sentence containing a word that is not in the grammar will be rejected, even if the program can parse all the rest of the words perfectly. One way of treating unknown words is to allow them to be any of the “open-class” categories—nouns, verbs, adjectives, and names, in our grammar. An unknown word will not be considered as one of the “closed-class” categories—prepositions, determiners, or pronouns. This can be programmed very simply by having `lexical-rules` return a list of these open-class rules for every word that is not already known.

```
(defparameter *open-categories* '(N V A Name)
  "Categories to consider for unknown words")

(defun lexical-rules (word)
  "Return a list of rules with word on the right-hand side."
  (or (find-all word *grammar* :key #'rule-rhs :test #'equal)
      (mapcar #'(lambda (cat) '(,cat -> ,word)) *open-categories*)))
```

With memoization of `lexical-rules`, this means that the lexicon is expanded every time an unknown word is encountered. Let's try this out:

```
> (parser '(John liked Mary))
((S (NP (NAME JOHN))
     (VP (V LIKED) (NP (NAME MARY)))))
```

```

> (parser '(Dana liked Dale))
((S (NP (NAME DANA))
    (VP (V LIKED) (NP (NAME DALE)))))

> (parser '(the rab zaggled the woogly quax))
((S (NP (D THE) (N RAB))
    (VP (V ZAGGLED) (NP (D THE) (A+ (A WOOGLY)) (N QUAX)))))

```

We see the parser works as well with words it knows (John and Mary) as with new words (Dana and Dale), which it can recognize as names because of their position in the sentence. In the last sentence in the example, it recognizes each unknown word unambiguously. Things are not always so straightforward, unfortunately, as the following examples show:

```

> (parser '(the slithy toves gymbled))
((S (NP (D THE) (N SLITHY)) (VP (V TOVES) (NP (NAME GYMBLED)))))
(S (NP (D THE) (A+ (A SLITHY)) (N TOVES)) (VP (V GYMBLED)))
(NP (D THE) (A+ (A SLITHY) (A+ (A TOVES))) (N GYMBLED)))

> (parser '(the slithy toves gymbled on the wabe))
((S (NP (D THE) (N SLITHY))
    (VP (VP (V TOVES) (NP (NAME GYMBLED)))
        (PP (P ON) (NP (D THE) (N WABE)))))
(S (NP (D THE) (N SLITHY))
    (VP (V TOVES) (NP (NP (NAME GYMBLED))
        (PP (P ON) (NP (D THE) (N WABE)))))
(S (NP (D THE) (A+ (A SLITHY)) (N TOVES))
    (VP (VP (V GYMBLED)) (PP (P ON) (NP (D THE) (N WABE)))))
(NP (NP (D THE) (A+ (A SLITHY) (A+ (A TOVES))) (N GYMBLED))
    (PP (P ON) (NP (D THE) (N WABE)))))

```

If the program knew morphology—that a *y* at the end of a word often signals an adjective, an *s* a plural noun, and an *ed* a past-tense verb—then it could do much better.

19.5 Parsing into a Semantic Representation

Syntactic parse trees of a sentence may be interesting, but by themselves they're not very useful. We use sentences to communicate ideas, not to display grammatical structures. To explore the idea of the semantics, or meaning, of a phrase, we need a domain to talk about. Imagine the scenario of a compact disc player capable of playing back selected songs based on their track number. Imagine further that this machine has buttons on the front panel indicating numbers, as well as words such as "play," "to," "and," and "without." If you then punch in the sequence of buttons "play

1 to 5 without 3," you could reasonably expect the machine to respond by playing tracks 1, 2, 4, and 5. After a few such successful interactions, you might say that the machine "understands" a limited language. The important point is that the utility of this machine would not be enhanced much if it happened to display a parse tree of the input. On the other hand, you would be justifiably annoyed if it responded to "play 1 to 5 without 3" by playing 3 or skipping 4.

Now let's stretch the imagination one more time by assuming that this CD player comes equipped with a full Common Lisp compiler, and that we are now in charge of writing the parser for its input language. Let's first consider the relevant data structures. We need to add a component for the semantics to both the rule and tree structures. Once we've done that, it is clear that trees are nothing more than instances of rules, so their definitions should reflect that. Thus, I use an `:include defstruct` to define trees, and I specify no `copier` function, because `copy-tree` is already a Common Lisp function, and I don't want to redefine it. To maintain consistency with the old `new-tree` function (and to avoid having to put in all those keywords) I define the constructor `new-tree`. This option to `defstruct` makes `(new-tree a b c)` equivalent to `(make-tree :lhs a :sem b :rhs c)`.

```
(defstruct (rule (:type list))
  lhs -> rhs sem)

(defstruct (tree (:type list) (:include rule) (:copier nil)
               (:constructor new-tree (lhs sem rhs))))
```

We will adopt the convention that the semantics of a word can be any Lisp object. For example, the semantics of the word "1" could be the object 1, and the semantics of "without" could be the function `set-difference`. The semantics of a tree is formed by taking the semantics of the rule that generated the tree and applying it (as a function) to the semantics of the constituents of the tree. Thus, the grammar writer must insure that the semantic component of rules are functions that expect the right number of arguments. For example, given the rule

```
(NP -> (NP CONJ NP) infix-funcall)
```

then the semantics of the phrase "1 to 5 without 3" could be determined by first determining the semantics of "1 to 5" to be (1 2 3 4 5), of "without" to be `set-difference`, and of "3" to be (3). After these sub-constituents are determined, the rule is applied by calling the function `infix-funcall` with the three arguments (1 2 3 4 5), `set-difference`, and (3). Assuming that `infix-funcall` is defined to apply its second argument to the other two arguments, the result will be (1 2 4 5).

This may make more sense if we look at a complete grammar for the CD player problem:

```

(use
  '((NP -> (NP CONJ NP) infix-funcall)
    (NP -> (N) list)
    (NP -> (N P N) infix-funcall)
    (N -> (DIGIT) identity)
    (P -> to integers)
    (CONJ -> and union)
    (CONJ -> without set-difference)
    (N -> 1 1) (N -> 2 2) (N -> 3 3) (N -> 4 4) (N -> 5 5)
    (N -> 6 6) (N -> 7 7) (N -> 8 8) (N -> 9 9) (N -> 0 0)))

(defun integers (start end)
  "A list of all the integers in the range [start...end] inclusive."
  (if (> start end) nil
      (cons start (integers (+ start 1) end))))

(defun infix-funcall (arg1 function arg2)
  "Apply the function to the two arguments"
  (funcall function arg1 arg2))

```

Consider the first three grammar rules, which are the only nonlexical rules. The first says that when two NPs are joined by a conjunction, we assume the translation of the conjunction will be a function, and the translation of the phrase as a whole is derived by calling that function with the translations of the two NPs as arguments. The second rule says that a single noun (whose translation should be a number) translates into the singleton list consisting of that number. The third rule is similar to the first, but concerns joining Ns rather than NPs. The overall intent is that the translation of an NP will always be a list of integers, representing the songs to play.

As for the lexical rules, the conjunction “and” translates to the `union` function, “without” translates to the function that subtracts one set from another, and “to” translates to the function that generates a list of integers between two end points. The numbers “0” to “9” translate to themselves. Note that both lexical rules like “CONJ -> and” and nonlexical rules like “NP -> (N P N)” can have functions as their semantic translations; in the first case, the function will just be returned as the semantic translation, whereas in the second case the function will be applied to the list of constituents.

Only minor changes are needed to parse to support this kind of semantic processing. As we see in the following, we add a `sem` argument to `extend-parse` and arrange to pass the semantic components around properly. When we have gathered all the right-hand-side components, we actually do the function application. All changes are marked with *******. We adopt the convention that the semantic value `nil` indicates failure, and we discard all such parses.

```

(defun parse (words)
  "Bottom-up parse, returning all parses of any prefix of words.
  This version has semantics."
  (unless (null words)
    (mapcan #'(lambda (rule)
              (extend-parse (rule-lhs rule) (rule-sem rule) ;***
                            (list (first words) (rest words) nil))
              (lexical-rules (first words))))))

(defun extend-parse (lhs sem rhs rem needed) ;***
  "Look for the categories needed to complete the parse.
  This version has semantics."
  (if (null needed)
      ;; If nothing is needed, return this parse and upward extensions,
      ;; unless the semantics fails
      (let ((parse (make-parse :tree (new-tree lhs sem rhs) :rem rem)))
        (unless (null (apply-semantics (parse-tree parse))) ;***
          (cons parse
                (mapcan
                 #'(lambda (rule)
                     (extend-parse (rule-lhs rule) (rule-sem rule) ;***
                                   (list (parse-tree parse) rem
                                         (rest (rule-rhs rule))))
                 (rules-starting-with lhs))))))
      ;; otherwise try to extend rightward
      (mapcan
       #'(lambda (p)
           (if (eq (parse-lhs p) (first needed))
               (extend-parse lhs sem (append1 rhs (parse-tree p)) ;***
                             (parse-rem p) (rest needed))))
       (parse rem))))

```

We need to add some new functions to support this:

```

(defun apply-semantics (tree)
  "For terminal nodes, just fetch the semantics.
  Otherwise, apply the sem function to its constituents."
  (if (terminal-tree-p tree)
      (tree-sem tree)
      (setf (tree-sem tree)
            (apply (tree-sem tree)
                   (mapcar #'tree-sem (tree-rhs tree))))))

(defun terminal-tree-p (tree)
  "Does this tree have a single word on the rhs?"
  (and (length=1 (tree-rhs tree))
       (atom (first (tree-rhs tree)))))

```



```
(defun meanings (words)
  "Return all possible meanings of a phrase. Throw away the syntactic part."
  (remove-duplicates (mapcar #'tree-sem (parser words)):test #'equal))
```

Here are some examples of the meanings that the parser can extract:

```
> (meanings '(1 to 5 without 3))
((1 2 4 5))

> (meanings '(1 to 4 and 7 to 9))
((1 2 3 4 7 8 9))

> (meanings '(1 to 6 without 3 and 4))
((1 2 4 5 6)
 (1 2 5 6))
```

The example “(1 to 6 without 3 and 4)” is ambiguous. The first reading corresponds to “((1 to 6) without 3) and 4,” while the second corresponds to “(1 to 6) without (3 and 4).” The syntactic ambiguity leads to a semantic ambiguity—the two meanings have different lists of numbers in them. However, it seems that the second reading is somehow better, in that it doesn’t make a lot of sense to talk of adding 4 to a set that already includes it, which is what the first translation does.

We can upgrade the lexicon to account for this. The following lexicon insists that “and” conjoins disjoint sets and that “without” removes only elements that were already in the first argument. If these conditions do not hold, then the translation will return nil, and the parse will fail. Note that this also means that an empty list, such as “3 to 2,” will also fail.

The previous grammar only allowed for the numbers 0 to 9. We can allow larger numbers by stringing together digits. So now we have two rules for numbers: a number is either a single digit, in which case the value is the digit itself (the `identity` function), or it is a number followed by another digit, in which case the value is 10 times the number plus the digit. We could alternately have specified a number to be a digit followed by a number, or even a number followed by a number, but either of those formulations would require a more complex semantic interpretation.

```
(use
 '( (NP -> (NP CONJ NP) infix-funcall)
   (NP -> (N) list)
   (NP -> (N P N) infix-funcall)
   (N -> (DIGIT) identity)
   (N -> (N DIGIT) 10*N+D)
   (P -> to integers)
   (CONJ -> and union*)
   (CONJ -> without set-diff)
   (DIGIT -> 1 1) (DIGIT -> 2 2) (DIGIT -> 3 3)
```

```
(DIGIT -> 4 4) (DIGIT -> 5 5) (DIGIT -> 6 6)
(DIGIT -> 7 7) (DIGIT -> 8 8) (DIGIT -> 9 9)
(DIGIT -> 0 0)))

(defun union* (x y) (if (null (intersection x y)) (append x y)))
(defun set-diff (x y) (if (subsetp y x) (set-difference x y)))
(defun 10*N+D (N D) (+ (* 10 N) D))
```

With this new grammar, we can get single interpretations out of most reasonable inputs:

```
> (meanings '(1 to 6 without 3 and 4))
((1 2 5 6))

> (meanings '(1 and 3 to 7 and 9 without 5 and 6))
((1 3 4 7 9))

> (meanings '(1 and 3 to 7 and 9 without 5 and 2))
((1 3 4 6 7 9 2))

> (meanings '(1 9 8 to 2 0 1))
((198 199 200 201))

> (meanings '(1 2 3))
(123 (123))
```

The example “1 2 3” shows an ambiguity between the number 123 and the list (123), but all the others are unambiguous.

19.6 Parsing with Preferences

One reason we have unambiguous interpretations is that we have a very limited domain of interpretation: we are dealing with sets of numbers, not lists. This is perhaps typical of the requests faced by a CD player, but it does not account for all desired input. For example, if you had a favorite song, you couldn't hear it three times with the request “1 and 1 and 1” under this grammar. We need some compromise between the permissive grammar, which generated all possible parses, and the restrictive grammar, which eliminates too many parses. To get the “best” interpretation out of an arbitrary input, we will not only need a new grammar, we will also need to modify the program to compare the relative worth of candidate interpretations. In other words, we will assign each interpretation a numeric score, and then pick the interpretation with the highest score.

We start by once again modifying the rule and tree data types to include a score component. As with the `sem` component, this will be used to hold first a function to compute a score and then eventually the score itself.

```
(defstruct (rule (:type list)
               (:constructor
                rule (lhs -> rhs &optional sem score)))
  lhs -> rhs sem score)

(defstruct (tree (:type list) (:include rule) (:copier nil)
               (:constructor new-tree (lhs sem score rhs))))
```

Note that we have added the constructor function `rule`. The intent is that the `sem` and `score` component of grammar rules should be optional. The user does not have to supply them, but the function use will make sure that the function `rule` is called to fill in the missing `sem` and `score` values with `nil`.

```
(defun use (grammar)
  "Switch to a new grammar."
  (clear-memoize 'rules-starting-with)
  (clear-memoize 'lexical-rules)
  (length (setf *grammar*
                (mapcar #'(lambda (r) (apply #'rule r))
                        grammar))))
```

Now we modify the parser to keep track of the score. The changes are again minor, and mirror the changes needed to add semantics. There are two places where we put the score into trees as we create them, and one place where we apply the scoring function to its arguments.

```
(defun parse (words)
  "Bottom-up parse, returning all parses of any prefix of words.
  This version has semantics and preference scores."
  (unless (null words)
    (mapcar #'(lambda (rule)
                (extend-parse
                 (rule-lhs rule) (rule-sem rule)
                 (rule-score rule) (list (first words)) ;***
                 (rest words) nil))
            (lexical-rules (first words)))))

(defun extend-parse (lhs sem score rhs rem needed) ;***
  "Look for the categories needed to complete the parse.
  This version has semantics and preference scores."
  (if (null needed)
      ;; If nothing is needed, return this parse and upward extensions.
      ;; unless the semantics fails
      (let ((parse (make-parse :tree (new-tree lhs sem score rhs) ;***
                              :rem rem)))
        (unless (null (apply-semantics (parse-tree parse))))
```

```

      (apply-scorer (parse-tree parse)) ;***
      (cons parse
        (mapcan
          #'(lambda (rule)
            (extend-parse
              (rule-lhs rule) (rule-sem rule)
              (rule-score rule) (list (parse-tree parse)) ;***
              rem (rest (rule-rhs rule))))
            (rules-starting-with lhs))))
      ;; otherwise try to extend rightward
      (mapcan
        #'(lambda (p)
          (if (eq (parse-lhs p) (first needed))
            (extend-parse lhs sem score
              (append1 rhs (parse-tree p)) ;***
              (parse-rem p) (rest needed))))
          (parse rem))))

```

Again we need some new functions to support this. Most important is `apply-scorer`, which computes the score for a tree. If the tree is a terminal (a word), then the function just looks up the score associated with that word. In this grammar all words have a score of 0, but in a grammar with ambiguous words it would be a good idea to give lower scores for infrequently used senses of ambiguous words. If the tree is a nonterminal, then the score is computed in two steps. First, all the scores of the constituents of the tree are added up. Then, this is added to a measure for the tree as a whole. The rule associated with each tree will have either a number attached to it, which is added to the sum, or a function. In the latter case, the function is applied to the tree, and the result is added to obtain the final score. As a final special case, if the function returns `nil`, then we assume it meant to return zero. This will simplify the definition of some of the scoring functions.

```

(defun apply-scorer (tree)
  "Compute the score for this tree."
  (let ((score (or (tree-score tree) 0)))
    (setf (tree-score tree)
          (if (terminal-tree-p tree)
              score
              ;; Add up the constituent's scores,
              ;; along with the tree's score
              (+ (sum (tree-rhs tree) #'tree-score-or-0)
                 (if (numberp score)
                     score
                     (or (apply score (tree-rhs tree)) 0)))))))

```

Here is an accessor function to pick out the score from a tree:

```
(defun tree-score-or-0 (tree)
  (if (numberp (tree-score tree))
      (tree-score tree)
      0))
```

Here is the updated grammar. First, I couldn't resist the chance to add more features to the grammar. I added the postnominal adjectives "shuffled," which randomly permutes the list of songs, and "reversed," which reverses the order of play. I also added the operator "repeat," as in "1 to 3 repeat 5," which repeats a list a certain number of times. I also added brackets to allow input that says explicitly how it should be parsed.

```
(use
 '( (NP -> (NP CONJ NP) infix-funcall infix-scorer)
   (NP -> (N P N) infix-funcall infix-scorer)
   (NP -> (N) list)
   (NP -> ([ NP ]) arg2)
   (NP -> (NP ADJ) rev-funcall rev-scorer)
   (NP -> (NP OP N) infix-funcall)
   (N -> (D) identity)
   (N -> (N D) 10*N+D)
   (P -> to integers prefer<)
   ([ -> [ [ ]
   (] -> ] ])
   (OP -> repeat repeat)
   (CONJ -> and append prefer-disjoint)
   (CONJ -> without set-difference prefer-subset)
   (ADJ -> reversed reverse inv-span)
   (ADJ -> shuffled permute prefer-not-singleton)
   (D -> 1 1) (D -> 2 2) (D -> 3 3) (D -> 4 4) (D -> 5 5)
   (D -> 6 6) (D -> 7 7) (D -> 8 8) (D -> 9 9) (D -> 0 0)))
```

The following scoring functions take trees as inputs and compute bonuses or penalties for those trees. The scoring function `prefer<`, used for the word "to," gives a one-point penalty for reversed ranges: "5 to 1" gets a score of -1, while "1 to 5" gets a score of 0. The scorer for "and," `prefer-disjoint`, gives a one-point penalty for intersecting lists: "1 to 3 and 7 to 9" gets a score of 0, while "1 to 4 and 2 to 5" gets -1. The "x without y" scorer, `prefer-subset`, gives a three-point penalty when the y list has elements that aren't in the x list. It also awards points in inverse proportion to the length (in words) of the x phrase. The idea is that we should prefer to bind "without" tightly to some small expression on the left. If the final scores come out as positive or as nonintegers, then this scoring component is responsible, since all the other components are negative integers. The "x shuffled" scorer, `prefer-not-singleton`, is similar, except that there the penalty is for shuffling a list of less than two songs.

```

(defun prefer< (x y)
  (if (>= (sem x) (sem y)) -1))

(defun prefer-disjoint (x y)
  (if (intersection (sem x) (sem y)) -1))

(defun prefer-subset (x y)
  (+ (inv-span x) (if (subsetp (sem y) (sem x)) 0 -3)))

(defun prefer-not-singleton (x)
  (+ (inv-span x) (if (< (length (sem x)) 2) -4 0)))

```

The `infix-scorer` and `rev-scorer` functions don't add anything new, they just assure that the previously mentioned scoring functions will get applied in the right place.

```

(defun infix-scorer (arg1 scorer arg2)
  (funcall (tree-score scorer) arg1 arg2))

(defun rev-scorer (arg scorer) (funcall (tree-score scorer) arg))

```

Here are the functions mentioned in the grammar, along with some useful utilities:

```

(defun arg2 (a1 a2 &rest a-n) (declare (ignore a1 a-n)) a2)

(defun rev-funcall (arg function) (funcall function arg))

(defun repeat (list n)
  "Append list n times."
  (if (= n 0)
      nil
      (append list (repeat list (- n 1)))))

(defun span-length (tree)
  "How many words are in tree?"
  (if (terminal-tree-p tree) 1
      (sum (tree-rhs tree) #'span-length)))

(defun inv-span (tree) (/ 1 (span-length tree)))

(defun sem (tree) (tree-sem tree))

(defun integers (start end)
  "A list of all the integers in the range [start...end] inclusive.
  This version allows start > end."
  (cond ((< start end) (cons start (integers (+ start 1) end)))
        ((> start end) (cons start (integers (- start 1) end)))
        (t (list start))))

(defun sum (numbers &optional fn)
  "Sum the numbers, or sum (mapcar fn numbers)."
  (if fn
      (loop for x in numbers sum (funcall fn x))
      (loop for x in numbers sum x)))

```

```
(defun permute (bag)
  "Return a random permutation of the given input list."
  (if (null bag)
      nil
      (let ((e (random-elt bag)))
        (cons e (permute (remove e bag :count 1 :test #'eq)))))))
```

We will need a way to show off the preference rankings:

```
(defun all-parses (words)
  (format t "~%Score Semantics~25T~a" words)
  (format t "%===== ~25T=====~%" )
  (loop for tree in (sort (parser words) #'> :key #'tree-score)
        do (format t "~5,lf ~9a~25T~a~%" (tree-score tree) (tree-sem tree)
                  (bracketing tree)))
  (values))

(defun bracketing (tree)
  "Extract the terminals, bracketed with parens."
  (cond ((atom tree) tree)
        ((length=1 (tree-rhs tree))
         (bracketing (first (tree-rhs tree))))
        (t (mapcar #'bracketing (tree-rhs tree)))))
```

Now we can try some examples:

```
> (all-parses '(1 to 6 without 3 and 4))
Score Semantics (1 TO 6 WITHOUT 3 AND 4)
=====
0.3 (1 2 5 6) ((1 TO 6) WITHOUT (3 AND 4))
-0.7 (1 2 4 5 6 4) (((1 TO 6) WITHOUT 3) AND 4)

> (all-parses '(1 and 3 to 7 and 9 without 5 and 6))
Score Semantics (1 AND 3 TO 7 AND 9 WITHOUT 5 AND 6)
=====
0.2 (1 3 4 7 9) (1 AND (((3 TO 7) AND 9) WITHOUT (5 AND 6)))
0.1 (1 3 4 7 9) (((1 AND (3 TO 7)) AND 9) WITHOUT (5 AND 6))
0.1 (1 3 4 7 9) ((1 AND ((3 TO 7) AND 9)) WITHOUT (5 AND 6))
-0.8 (1 3 4 6 7 9 6) ((1 AND (((3 TO 7) AND 9) WITHOUT 5)) AND 6)
-0.8 (1 3 4 6 7 9 6) (1 AND (((3 TO 7) AND 9) WITHOUT 5) AND 6))
-0.9 (1 3 4 6 7 9 6) (((((1 AND (3 TO 7)) AND 9) WITHOUT 5) AND 6)
-0.9 (1 3 4 6 7 9 6) (((1 AND ((3 TO 7) AND 9)) WITHOUT 5) AND 6)
-2.0 (1 3 4 5 6 7 9) ((1 AND (3 TO 7)) AND (9 WITHOUT (5 AND 6)))
-2.0 (1 3 4 5 6 7 9) (1 AND ((3 TO 7) AND (9 WITHOUT (5 AND 6))))
-3.0 (1 3 4 5 6 7 9 6) (((1 AND (3 TO 7)) AND (9 WITHOUT 5)) AND 6)
-3.0 (1 3 4 5 6 7 9 6) ((1 AND (3 TO 7)) AND ((9 WITHOUT 5) AND 6))
-3.0 (1 3 4 5 6 7 9 6) ((1 AND ((3 TO 7) AND (9 WITHOUT 5))) AND 6)
```

```

-3.0 (1 3 4 5 6 7 9 6) (1 AND (((3 TO 7) AND (9 WITHOUT 5)) AND 6))
-3.0 (1 3 4 5 6 7 9 6) (1 AND ((3 TO 7) AND ((9 WITHOUT 5) AND 6)))

> (all-parses '(1 and 3 to 7 and 9 without 5 and 2))
Score Semantics (1 AND 3 TO 7 AND 9 WITHOUT 5 AND 2)
-----
0.2 (1 3 4 6 7 9 2) ((1 AND (((3 TO 7) AND 9) WITHOUT 5)) AND 2)
0.2 (1 3 4 6 7 9 2) (1 AND (((3 TO 7) AND 9) WITHOUT 5) AND 2))
0.1 (1 3 4 6 7 9 2) (((1 AND (3 TO 7)) AND 9) WITHOUT 5) AND 2)
0.1 (1 3 4 6 7 9 2) (((1 AND ((3 TO 7) AND 9)) WITHOUT 5) AND 2)
-2.0 (1 3 4 5 6 7 9 2) (((1 AND (3 TO 7)) AND (9 WITHOUT 5)) AND 2)
-2.0 (1 3 4 5 6 7 9 2) ((1 AND (3 TO 7)) AND ((9 WITHOUT 5) AND 2))
-2.0 (1 3 4 5 6 7 9) ((1 AND (3 TO 7)) AND (9 WITHOUT (5 AND 2)))
-2.0 (1 3 4 5 6 7 9 2) ((1 AND ((3 TO 7) AND (9 WITHOUT 5))) AND 2)
-2.0 (1 3 4 5 6 7 9 2) (1 AND ((3 TO 7) AND ((9 WITHOUT 5) AND 2)))
-2.0 (1 3 4 5 6 7 9 2) (1 AND ((3 TO 7) AND ((9 WITHOUT 5) AND 2)))
-2.0 (1 3 4 5 6 7 9) (1 AND ((3 TO 7) AND (9 WITHOUT (5 AND 2))))
-2.8 (1 3 4 6 7 9) (1 AND (((3 TO 7) AND 9) WITHOUT (5 AND 2)))
-2.9 (1 3 4 6 7 9) (((1 AND (3 TO 7)) AND 9) WITHOUT (5 AND 2))
-2.9 (1 3 4 6 7 9) ((1 AND ((3 TO 7) AND 9)) WITHOUT (5 AND 2))

```

In each case, the preference rules are able to assign higher scores to more reasonable interpretations. It turns out that, in each case, all the interpretations with positive scores represent the same set of numbers, while interpretations with negative scores seem worse. Seeing all the scores in gory detail may be of academic interest, but what we really want is something to pick out the best interpretation. The following code is appropriate for many situations. It picks the top scorer, if there is a unique one, or queries the user if several interpretations tie for the best score, and it complains if there are no valid parses at all. The query-user function may be useful in many applications, but note that `meaning` uses it only as a default; a program that had some automatic way of deciding could supply another tie-breaker function to `meaning`.

```

(defun meaning (words &optional (tie-breaker #'query-user))
  "Choose the single top-ranking meaning for the words."
  (let* ((trees (sort (parser words) #'> :key #'tree-score))
        (best-score (if trees (tree-score (first trees)) 0))
        (best-trees (delete best-score trees
                           :key #'tree-score :test-not #'eql))
        (best-sems (delete-duplicates (mapcar #'tree-sem best-trees)
                                      :test #'equal)))
    (case (length best-sems)
      (0 (format t "~&Sorry, I didn't understand that.") nil)
      (1 (first best-sems))
      (t (funcall tie-breaker best-sems))))))

```



```
(defun query-user (choices &optional
                  (header-str "~&Please pick one:")
                  (footer-str "~&Your choice? "))
  "Ask user to make a choice."
  (format *query-io* header-str)
  (loop for choice in choices for i from 1 do
        (format *query-io* "~&~3d: ~a" i choice))
  (format *query-io* footer-str)
  (nth (- (read) 1) choices))
```

Here we see some final examples:

```
> (meaning '(1 to 5 without 3 and 4))
(1 2 5)

> (meaning '(1 to 5 without 3 and 6))
(1 2 4 5 6)

> (meaning '(1 to 5 without 3 and 6 shuffled))
(6 4 1 2 5)

> (meaning '([ 1 to 5 without [ 3 and 6 ] ] reversed))
(5 4 2 1)

> (meaning '(1 to 5 to 9))
Sorry, I didn't understand that.
NIL

> (meaning '(1 to 5 without 3 and 7 repeat 2))
Please pick one:
  1: (1 2 4 5 7 1 2 4 5 7)
  2: (1 2 4 5 7 7)
Your choice? 1
(1 2 4 5 7 1 2 4 5 7)

> (all-parses '(1 to 5 without 3 and 7 repeat 2))
Score  Semantics          (1 TO 5 WITHOUT 3 AND 7 REPEAT 2)
-----
  0.3  (1 2 4 5 7 1 2 4 5 7) (((1 TO 5) WITHOUT 3) AND 7) REPEAT 2)
  0.3  (1 2 4 5 7 7)         (((1 TO 5) WITHOUT 3) AND (7 REPEAT 2))
 -2.7 (1 2 4 5 1 2 4 5)     (((1 TO 5) WITHOUT (3 AND 7)) REPEAT 2)
 -2.7 (1 2 4 5)            ((1 TO 5) WITHOUT ((3 AND 7) REPEAT 2))
 -2.7 (1 2 4 5)            ((1 TO 5) WITHOUT (3 AND (7 REPEAT 2)))
```

This last example points out a potential problem: I wasn't sure what was a good scoring function for "repeat," so I left it blank, it defaulted to 0, and we end up with two parses with the same score. This example suggests that "repeat" should probably involve `inv-span` like the other modifiers, but perhaps other factors should be involved as well. There can be a complicated interplay between phrases, and it

is not always clear where to assign the score. For example, it doesn't make much sense to repeat a "without" phrase; that is, the bracketing (x without (y repeat n)) is probably a bad one. But the scorer for "without" nearly handles that already. It assigns a penalty if its right argument is not a subset of its left. Unfortunately, repeated elements are not counted in sets, so for example, the list (1 2 3 1 2 3) is a subset of (1 2 3 4). However, we could change the scorer for "without" to test for sub-bag-p (not a built-in Common Lisp function) instead, and then "repeat" would not have to be concerned with that case.

19.7 The Problem with Context-Free Phrase-Structure Rules

The fragment of English grammar we specified in section 19.2 admits a variety of ungrammatical phrases. For example, it is equally happy with both "I liked her" and "me liked she." Only the first of these should be accepted; the second should be ruled out. Similarly, our grammar does not state that verbs have to agree with their subjects in person and number. And, since the grammar has no notion of meaning, it will accept sentences that are semantically anomalous (or at least unusual), such as "the table liked the man."

There are also some technical problems with context-free grammars. For example, it can be shown that no context-free grammar can be written to account for the language consisting of just the strings ABC, AABBC, AAABBBCCC, and so forth, where each string has an equal number of As, Bs, and Cs. Yet sentences roughly of that form show up (admittedly rarely) in natural languages. An example is "Robin and Sandy loved and hated Pat and Kim, respectively." While there is still disagreement over whether it is possible to generate natural languages with a context-free grammar, clearly it is much easier to use a more powerful grammatical formalism. For example, consider solving the subject-predicate agreement problem. It is possible to do this with a context-free language including categories like singular-NP, plural-NP, singular-VP, and plural-VP, but it is far easier to augment the grammatical formalism to allow passing features between constituents.

It should be noted that context-free phrase-structure rules turned out to be very useful for describing programming languages. Starting with Algol 60, the formalism has been used under the name *Backus-Naur Form* (BNF) by computer scientists. In this book we are more interested in natural languages, so in the next chapter we will see a more powerful formalism known as *unification grammar* that can handle the problem of agreement, as well as other difficulties. Furthermore, *unification grammars* allow a natural way of attaching semantics to a parse.

19.8 History and References

There is a class of parsing algorithms known as *chart parsers* that explicitly cache partial parses and reuse them in constructing larger parses. Earley's algorithm (1970) is the first example, and Martin Kay (1980) gives a good overview of the field and introduces a data structure, the *chart*, for storing substrings of a parse. Winograd (1983) gives a complex (five-page) specification of a chart parser. None of these authors have noticed that one can achieve the same results by augmenting a simple (one-page) parser with memoization. In fact, it is possible to write a top-down parser that is even more succinct. (See exercise 19.3 below.)

For a general overview of natural language processing, my preferences (in order) are Allen 1987, Winograd 1983 or Gazdar and Mellish 1989.

19.9 Exercises

? **Exercise 19.2 [m-h]** Experiment with the grammar and the parser. Find sentences it cannot parse correctly, and try to add new syntactic rules to account for them.

? **Exercise 19.3 [m-h]** The parser works in a bottom-up fashion. Write a top-down parser, and compare it to the bottom-up version. Can both parsers work with the same grammar? If not, what constraints on the grammar does each parsing strategy impose?

? **Exercise 19.4 [h]** Imagine an interface to a dual cassette deck. Whereas the CD player had one assumed verb, "play," this unit has three explicit verb forms: "record," "play," and "erase." There should also be modifiers "from" and "to," where the object of a "to" is either 1 or 2, indicating which cassette to use, and the object of a "from" is either 1 or 2, or one of the symbols PHONO, CD, or AUX. It's up to you to design the grammar, but you should allow input something like the following, where I have chosen to generate actual Lisp code as the meaning:

```
> (meaning '(play 1 to 5 from CD shuffled and
            record 1 to 5 from CD and 1 and 3 and 7 from 1))
(PROGN (PLAY '(1 5 2 3 4) :FROM 'CD)
       (RECORD '(1 2 3 4 5) :FROM 'CD)
       (RECORD '(1 3 7) :FROM '1))
```

This assumes that the functions `play` and `record` take keyword arguments (with defaults) for `:from` and `:to`. You could also extend the grammar to accommodate an automatic timer, with phrases like "at 3:00."


```

(defun extend-parse (lhs rhs rem needed num-words table)
  "Look for the categories needed to complete the parse."
  (if (null needed)
      ;; If nothing is needed, return this parse and upward extensions
      (let ((parse (make-parse :tree (new-tree lhs rhs) :rem rem)))
        (cons parse
              (mapcan
               #'(lambda (rule)
                   (extend-parse (rule-lhs rule)
                                (list (parse-tree parse)
                                      rem (rest (rule-rhs rule))
                                      num-words table))
                   (rules-starting-with lhs))))))
      ;; otherwise try to extend rightward
      (mapcan
       #'(lambda (p)
           (if (eq (parse-lhs p) (first needed))
               (extend-parse lhs (append1 rhs (parse-tree p))
                             (parse-rem p) (rest needed)
                             (length (parse-rem p)) table)))
           (parse rem num-words table))))))

```

It turns out that, for the Lisp system used in the timings above, this version is no faster than normal memoization.

Answer 19.3 Actually, the top-down parser is a little easier (shorter) than the bottom-up version. The problem is that the most straightforward way of implementing a top-down parser does not handle so-called *left recursive* rules—rules of the form $(X \rightarrow (X \dots))$. This includes rules we've used, like $(NP \rightarrow (NP \text{ and } NP))$. The problem is that the parser will postulate an NP, and then postulate that it is of the form $(NP \text{ and } NP)$, and that the first NP of that expression is of the form $(NP \text{ and } NP)$, and so on. An infinite structure of NPs is explored before even the first word is considered.

Bottom-up parsers are stymied by rules with null right-hand sides: $(X \rightarrow ())$. Note that I was careful to exclude such rules in my grammars earlier.

```

(defun parser (words &optional (cat 'S))
  "Parse a list of words; return only parses with no remainder."
  (mapcar #'parse-tree (complete-parses (parse words cat))))

(defun parse (tokens start-symbol)
  "Parse a list of tokens, return parse trees and remainders."
  (if (eq (first tokens) start-symbol)
      (list (make-parse :tree (first tokens) :rem (rest tokens))
            (mapcan #'(lambda (rule)
                        (extend-parse (lhs rule) nil tokens (rhs rule)))
                    (rules-for start-symbol))))))

```

```
(defun extend-parse (lhs rhs rem needed)
  "Parse the remaining needed symbols."
  (if (null needed)
      (list (make-parse :tree (cons lhs rhs) :rem rem))
      (mapcan
        #'(lambda (p)
            (extend-parse lhs (append rhs (list (parse-tree p)))
                          (parse-rem p) (rest needed)))
          (parse rem (first needed))))))

(defun rules-for (cat)
  "Return all the rules with category on lhs"
  (find-all cat *grammar* :key #'rule-lhs))
```

Answer 19.5 If it were omitted, then `:test` would default to `#'eq`, and it would be possible to remove the “wrong” element from the list. Consider the list `(1.0 1.0)` in an implementation where floating-point numbers are `eq` but not `equal`. If `random-elt` chooses the first `1.0` first, then everything is satisfactory—the result list is the same as the input list. However, if `random-elt` chooses the second `1.0`, then the second `1.0` will be the first element of the answer, but `remove` will remove the wrong `1.0`! It will remove the first `1.0`, and the final answer will be a list with two pointers to the second `1.0` and none to the first. In other words, we could have:

```
> (member (first x) (permute x) :test #'eq)
NIL
```

Answer 19.6

```
(defun permute (bag)
  "Return a random permutation of the bag."
  ;; It is done by converting the bag to a vector, but the
  ;; result is always the same type as the input bag.
  (let ((bag-copy (replace (make-array (length bag)) bag))
        (bag-type (if (listp bag) 'list (type-of bag))))
    (coerce (permute-vector! bag-copy) bag-type)))

(defun permute-vector! (vector)
  "Destructively permute (shuffle) the vector."
  (loop for i from (length vector) downto 2 do
    (rotatef (aref vector (- i 1))
             (aref vector (random i))))
  vector)
```

The answer uses `rotatef`, a relative of `setf` that swaps 2 or more values. That is, `(rotatef a b)` is like:

```
(let ((temp a))
  (setf a b)
  (setf b temp)
  nil)
```

Rarely, `rotatef` is used with more than two arguments. (`rotatef a b c`) is like:

```
(let ((temp a))
  (setf a b)
  (setf b c)
  (setf c temp)
  nil)
```

CHAPTER 20

Unification Grammars

Prolog was invented because Alain Colmerauer wanted a formalism to describe the grammar of French. His intuition was that the combination of Horn clauses and unification resulted in a language that was just powerful enough to express the kinds of constraints that show up in natural languages, while not as powerful as, for example, full predicate calculus. This lack of power is important, because it enables efficient implementation of Prolog, and hence of the language-analysis programs built on top of it.

Of course, Prolog has evolved and is now used for many applications besides natural language, but Colmerauer's underlying intuition remains a good one. This chapter shows how to view a grammar as a set of logic programming clauses. The clauses define what is a legal sentence and what isn't, without any explicit reference to the process of parsing or generation. The amazing thing is that the clauses can be defined in a way that leads to a very efficient parser. Furthermore, the same grammar can be used for both parsing and generation (at least in some cases).

20.1 Parsing as Deduction

Here's how we could express the grammar rule "A sentence can be composed of a noun phrase followed by a verb phrase" in Prolog:

```
(← (S ?s)
   (NP ?np)
   (VP ?vp)
   (concat ?np ?vp ?s))
```

The variables represent strings of words. As usual, they will be implemented as lists of symbols. The rule says that a given string of words *?s* is a sentence if there is a string that is noun phrase and one that is a verb phrase, and if they can be concatenated to form *?s*. Logically, this is fine, and it would work as a program to generate random sentences. However, it is a very inefficient program for parsing sentences. It will consider all possible noun phrases and verb phrases, without regard to the input words. Only when it gets to the `concat` goal (defined on page 411) will it test to see if the two constituents can be concatenated together to make up the input string. Thus, a better order of evaluation for parsing is:

```
(← (S ?s)
   (concat ?np ?vp ?s)
   (NP ?np)
   (VP ?vp))
```

The first version had NP and VP guessing strings to be verified by `concat`. In most grammars, there will be a very large or infinite number of NPs and VPs. This second version has `concat` guessing strings to be verified by NP and VP. If there are *n* words in the sentence, then `concat` can only make *n* + 1 guesses, quite an improvement. However, it would be better still if we could in effect have `concat` and NP work together to make a more constrained guess, which would then be verified by VP.

We have seen this type of problem before. In Lisp, the answer is to return multiple values. NP would be a function that takes a string as input and returns two values: an indication of success or failure, and a remainder string of words that have not yet been parsed. When the first value indicates success, then VP would be called with the remaining string as input. In Prolog, return values are just extra arguments. So each predicate will have two parameters: an input string and a remainder string. Following the usual Prolog convention, the output parameter comes after the input. In this approach, no calls to `concat` are necessary, no wild guesses are made, and Prolog's backtracking takes care of the necessary guessing:

```
(<- (S ?s0 ?s2)
      (NP ?s0 ?s1)
      (VP ?s1 ?s2))
```

This rule can be read as “The string from s_0 to s_2 is a sentence if there is an s_1 such that the string from s_0 to s_1 is a noun phrase and the string from s_1 to s_2 is a verb phrase.”

A sample query would be `(?- (S (The boy ate the apple) ()))`. With suitable definitions of NP and VP, this would succeed, with the following bindings holding within S:

```
?s0 = (The boy ate the apple)
?s1 =      (ate the apple)
?s2 =      ()
```

Another way of reading the goal `(NP ?s0 ?s1)`, for example, is as “IS the list `?s0` minus the list `?s1` a noun phrase?” In this case, `?s0` minus `?s1` is the list `(The boy)`. The combination of two arguments, an input list and an output list, is often called a *difference list*, to emphasize this interpretation. More generally, the combination of an input parameter and output parameter is called an *accumulator*. Accumulators, particularly difference lists, are an important technique throughout logic programming and are also used in functional programming, as we saw on page 63.

In our rule for S, the concatenation of difference lists was implicit. If we prefer, we could define a version of `concat` for difference lists and call it explicitly:

```
(<- (S ?s-in ?s-rem)
      (NP ?np-in ?np-rem)
      (VP ?vp-in ?vp-rem)
      (concat ?np-in ?np-rem ?vp-in ?vp-rem ?s-in ?s-rem))

(<- (concat ?a ?b ?b ?c ?a ?c))
```

Because this version of `concat` has a different arity than the old version, they can safely coexist. It states the difference list equation $(a - b) + (b - c) = (a - c)$.

In the last chapter we stated that context-free phrase-structure grammar is inconvenient for expressing things like agreement between the subject and predicate of a sentence. With the Horn-clause-based grammar formalism we are developing here, we can add an argument to the predicates NP and VP to represent agreement. In English, the agreement rule does not have a big impact. For all verbs except *be*, the difference only shows up in the third-person singular of the present tense:

	Singular	Plural
first person	I sleep	we sleep
second person	you sleep	you sleep
third person	he/she sleeps	they sleep

Thus, the agreement argument will take on one of the two values 3sg or ~3sg to indicate third-person-singular or not-third-person-singular. We could write:

```
(<- (S ?s0 ?s2)
      (NP ?agr ?s0 ?s1)
      (VP ?agr ?s1 ?s2))

(<- (NP 3sg (he . ?s) ?s))
(<- (NP ~3sg (they . ?s) ?s))

(<- (VP 3sg (sleeps . ?s) ?s))
(<- (VP ~3sg (sleep . ?s) ?s))
```

This grammar parses just the right sentences:

```
> (?- (S (He sleeps) ()))
Yes.

> (?- (S (He sleep) ()))
No.
```

Let's extend the grammar to allow common nouns as well as pronouns:

```
(<- (NP ?agr ?s0 ?s2)
      (Det ?agr ?s0 ?s1)
      (N ?agr ?s1 ?s2))

(<- (Det ?any (the . ?s) ?s))
(<- (N 3sg (boy . ?s) ?s))
(<- (N 3sg (girl . ?s) ?s))
```

The same grammar rules can be used to generate sentences as well as parse. Here are all possible sentences in this trivial grammar:

```
> (?- (S ?words ()))
?WORDS = (HE SLEEPS);
?WORDS = (THEY SLEEP);
?WORDS = (THE BOY SLEEPS);
?WORDS = (THE GIRL SLEEPS);
No.
```

So far all we have is a recognizer: a predicate that can separate sentences from

nonsentences. But we can add another argument to each predicate to build up the semantics. The result is not just a recognizer but a true parser:

```
(← (S (?pred ?subj) ?s0 ?s2)
    (NP ?agr ?subj ?s0 ?s1)
    (VP ?agr ?pred ?s1 ?s2))

(← (NP 3sg (the male) (he . ?s) ?s))
(← (NP ~3sg (some objects) (they . ?s) ?s))

(← (NP ?agr (?det ?n) ?s0 ?s2)
    (Det ?agr ?det ?s0 ?s1)
    (N ?agr ?n ?s1 ?s2))

(← (VP 3sg sleep (sleeps . ?s) ?s))
(← (VP ~3sg sleep (sleep . ?s) ?s))

(← (Det ?any the (the . ?s) ?s))
(← (N 3sg (young male human) (boy . ?s) ?s))
(← (N 3sg (young female human) (girl . ?s) ?s))
```

The semantic translations of individual words is a bit capricious. In fact, it is not too important at this point if the translation of *boy* is (young male human) or just *boy*. There are two properties of a semantic representation that are important. First, it should be unambiguous. The representation of *orange* the fruit should be different from *orange* the color (although the representation of the fruit might well refer to the color, or vice versa). Second, it should express generalities, or allow them to be expressed elsewhere. So either *sleep* and *sleeps* should have the same or similar representation, or there should be an inference rule relating them. Similarly, if the representation of *boy* does not say so explicitly, there should be some other rule saying that a boy is a male and a human.

Once the semantics of individual words is decided, the semantics of higher-level categories (sentences and noun phrases) is easy. In this grammar, the semantics of a sentence is the application of the predicate (the verb phrase) to the subject (the noun phrase). The semantics of a compound noun phrase is the application of the determiner to the noun.

This grammar returns the semantic interpretation but does not build a syntactic tree. The syntactic structure is implicit in the sequence of goals: S calls NP and VP, and NP can call Det and N. If we want to make this explicit, we can provide yet another argument to each nonterminal:

```
(← (S (?pred ?subj) (s ?np ?vp) ?s0 ?s2)
    (NP ?agr ?subj ?np ?s0 ?s1)
    (VP ?agr ?pred ?vp ?s1 ?s2))

(← (NP 3sg (the male) (np he) (he . ?s) ?s))
(← (NP ~3sg (some objects) (np they) (they . ?s) ?s))
```

```

(<- (NP ?agr (?det ?n) (np ?det-syn ?n-syn)?s0 ?s2)
  (Det ?agr ?det ?det-syn ?s0 ?s1)
  (N ?agr ?n ?n-syn ?s1 ?s2))

(<- (VP 3sg sleep (vp sleeps)(sleeps . ?s) ?s))
(<- (VP ~3sg sleep (vp sleep)(sleep . ?s) ?s))

(<- (Det ?any the (det the)(the . ?s) ?s))
(<- (N 3sg (young male human) (n boy)(boy . ?s) ?s))
(<- (N 3sg (young female human) (n girl)(girl . ?s) ?s))

```

This grammar can still be used to parse or generate sentences, or even to enumerate all syntax/semantics/sentence triplets:

```

;; Parsing:
> (?- (S ?sem ?syn (He sleeps) ()))
?SEM = (SLEEP (THE MALE))
?SYN = (S (NP HE) (VP SLEEPS)).

;; Generating:
> (?- (S (sleep (the male)) ? ?words ()))
?WORDS = (HE SLEEPS)

;; Enumerating:
> (?- (S ?sem ?syn ?words ()))
?SEM = (SLEEP (THE MALE))
?SYN = (S (NP HE) (VP SLEEPS))
?WORDS = (HE SLEEPS);

?SEM = (SLEEP (SOME OBJECTS))
?SYN = (S (NP THEY) (VP SLEEP))
?WORDS = (THEY SLEEP);

?SEM = (SLEEP (THE (YOUNG MALE HUMAN)))
?SYN = (S (NP (DET THE) (N BOY)) (VP SLEEPS))
?WORDS = (THE BOY SLEEPS);

?SEM = (SLEEP (THE (YOUNG FEMALE HUMAN)))
?SYN = (S (NP (DET THE) (N GIRL)) (VP SLEEPS))
?WORDS = (THE GIRL SLEEPS);

No.

```

20.2 Definite Clause Grammars

We now have a powerful and efficient tool for parsing sentences. However, it is getting to be a very messy tool—there are too many arguments to each goal, and it

is hard to tell which arguments represent syntax, which represent semantics, which represent in/out strings, and which represent other features, like agreement. So, we will take the usual step when our bare programming language becomes messy: define a new language.

Edinburgh Prolog recognizes assertions called *definite clause grammar* (DCG) rules. The term *definite clause* is just another name for a Prolog clause, so DCGs are also called “logic grammars.” They could have been called “Horn clause grammars” or “Prolog grammars” as well.

DCG rules are clauses whose main functor is an arrow, usually written `-->`. They compile into regular Prolog clauses with extra arguments. In normal DCG rules, only the string arguments are automatically added. But we will see later how this can be extended to add other arguments automatically as well.

We will implement DCG rules with the macro `rule` and an infix arrow. Thus, we want the expression:

```
(rule (S) --> (NP) (VP))
```

to expand into the clause:

```
(<- (S ?s0 ?s2)
    (NP ?s0 ?s1)
    (VP ?s1 ?s2))
```

While we’re at it, we may as well give `rule` the ability to deal with different types of rules, each one represented by a different type of arrow. Here’s the `rule` macro:

```
(defmacro rule (head &optional (arrow ':-) &body body)
  "Expand one of several types of logic rules into pure Prolog."
  ;; This is data-driven, dispatching on the arrow
  (funcall (get arrow 'rule-function) head body))
```

As an example of a rule function, the arrow `:-` will be used to represent normal Prolog clauses. That is, the form `(rule head :- body)` will be equivalent to `(<- head body)`.

```
(setf (get ':- 'rule-function)
      #'(lambda (head body) (<- ,head .,body)))
```

Before writing the rule function for DCG rules, there are two further features of the DCG formalism to consider. First, some goals in the body of a rule may be normal Prolog goals, and thus do not require the extra pair of arguments. In Edinburgh Prolog, such goals are surrounded in braces. One would write:

```
s(Sem) --> np(Subj), vp(Pred),
           {combine(Subj,Pred,Sem)}.
```

where the idea is that `combine` is not a grammatical constituent, but rather a Prolog predicate that could do some calculations on `Subj` and `Pred` to arrive at the proper semantics, `Sem`. We will mark such a test predicate not by brackets but by a list headed by the keyword `:test`, as in:

```
(rule (S ?sem) --> (NP ?subj) (VP ?pred)
      (:test (combine ?subj ?pred ?sem)))
```

Second, we need some way of introducing individual words on the right-hand side, as opposed to categories of words. In Prolog, brackets are used to represent a word or list of words on the right-hand side:

```
verb --> [sleeps].
```

We will use a list headed by the keyword `:word`:

```
(rule (NP (the male) 3sg) --> (:word he))
(rule (VP sleeps 3sg) --> (:word sleeps))
```

The following predicates test for these two special cases. Note that the cut is also allowed as a normal goal.

```
(defun dcg-normal-goal-p (x) (or (starts-with x :test) (eq x '!)))

(defun dcg-word-list-p (x) (starts-with x ':word))
```

At last we are in a position to present the rule function for DCG rules. The function `make-dcg` inserts variables to keep track of the strings that are being parsed.

```
(setf (get '-->' rule-function) 'make-dcg)

(defun make-dcg (head body)
  (let ((n (count-if (complement #'dcg-normal-goal-p) body)))
    '(<- (,@head ?s0 ,(symbol '?s n))
         ..(make-dcg-body body 0))))
```

```

(defun make-dcg-body (body n)
  "Make the body of a Definite Clause Grammar (DCG) clause.
  Add ?string-in and -out variables to each constituent.
  Goals like (:test goal) are ordinary Prolog goals,
  and goals like (:word hello) are literal words to be parsed."
  (if (null body)
      nil
      (let ((goal (first body)))
        (cond
         ((eq goal '!) (cons '! (make-dcg-body (rest body) n)))
         ((dcg-normal-goal-p goal)
          (append (rest goal)
                  (make-dcg-body (rest body) n)))
         ((dcg-word-list-p goal)
          (cons
           '(= ,(symbol '?s n)
              (,@(rest goal) .,(symbol '?s (+ n 1))))
           (make-dcg-body (rest body) (+ n 1))))
         (t (cons
              (append goal
                      (list (symbol '?s n)
                            (symbol '?s (+ n 1))))
              (make-dcg-body (rest body) (+ n 1))))))))))

```

? **Exercise 20.1 [m]** `make-dcg` violates one of the cardinal rules of macros. What does it do wrong? How would you fix it?

20.3 A Simple Grammar in DCG Format

Here is the trivial grammar from page 688 in DCG format.

```

(rule (S (?pred ?subj)) -->
      (NP ?agr ?subj)
      (VP ?agr ?pred))

(rule (NP ?agr (?det ?n)) -->
      (Det ?agr ?det)
      (N ?agr ?n))

```



```

(rule (NP 3sg (the male))      --> (:word he))
(rule (NP ~3sg (some objects)) --> (:word they))
(rule (VP 3sg sleep)          --> (:word sleeps))
(rule (VP ~3sg sleep)         --> (:word sleep))
(rule (Det ?any the)          --> (:word the))
(rule (N 3sg (young male human)) --> (:word boy))
(rule (N 3sg (young female human)) --> (:word girl))

```

This grammar is quite limited, generating only four sentences. The first way we will extend it is to allow verbs with objects: in addition to “The boy sleeps,” we will allow “The boy meets the girl.” To avoid generating ungrammatical sentences like “* The boy meets,”¹ we will separate the category of verb into two *subcategories*: transitive verbs, which take an object, and intransitive verbs, which don’t.

Transitive verbs complicate the semantic interpretation of sentences. We would like the interpretation of “Terry kisses Jean” to be (kiss Terry Jean). The interpretation of the noun phrase “Terry” is just Terry, but then what should the interpretation of the verb phrase “kisses Jean” be? To fit our predicate application model, it must be something equivalent to $(\lambda(x) (\text{kiss } x \text{ Jean}))$. When applied to the subject, we want to get the simplification:

$$((\lambda(x) (\text{kiss } x \text{ Jean})) \text{Terry}) \Rightarrow (\text{kiss Terry Jean})$$

Such simplification is not done automatically by Prolog, but we can write a predicate to do it. We will call it `funca11`, because it is similar to the Lisp function of that name, although it only handles replacement of the argument, not full evaluation of the body. (Technically, this is the lambda-calculus operation known as *beta-reduction*.) The predicate `funca11` is normally used with two input arguments, a function and its argument, and one output argument, the resulting reduction:

```
(<- (funca11 (lambda (?x) ?body) ?x ?body))
```

With this we could write our rule for sentences as:

```

(rule (S ?sem) -->
  (NP ?agr ?subj)
  (VP ?agr ?pred)
  (:test (funca11 ?pred ?subj ?sem)))

```

An alternative is to, in effect, compile away the call to `funca11`. Instead of having the semantic representation of VP be a single lambda expression, we can represent it as

¹The asterisk at the start of a sentence is the standard linguistic notation for an utterance that is ungrammatical or otherwise ill-formed.

two arguments: an input argument, ?subj, which acts as a parameter to the output argument, ?pred, which takes the place of the body of the lambda expression. By explicitly manipulating the parameter and body, we can eliminate the call to `funca11`. The trick is to make the parameter and the subject one and the same:

```
(rule (S ?pred) -->
  (NP ?agr ?subj)
  (VP ?agr ?subj ?pred))
```

One way of reading this rule is "To parse a sentence, parse a noun phrase followed by a verb phrase. If they have different agreement features then fail, but otherwise insert the interpretation of the noun phrase, ?subj, into the proper spot in the interpretation of the verb phrase, ?pred, and return ?pred as the final interpretation of the sentence."

The next step is to write rules for verb phrases and verbs. Transitive verbs are listed under the predicate `Verb/tr`, and intransitive verbs are listed as `Verb/intr`. The semantics of tenses (past and present) has been ignored.

```
(rule (VP ?agr ?subj ?pred) -->
  (Verb/tr ?agr ?subj ?pred ?obj)
  (NP ?any-agr ?obj))

(rule (VP ?agr ?subj ?pred) -->
  (Verb/intr ?agr ?subj ?pred))

(rule (Verb/tr ~3sg ?x (kiss ?x ?y) ?y) --> (:word kiss))
(rule (Verb/tr 3sg ?x (kiss ?x ?y) ?y) --> (:word kisses))
(rule (Verb/tr ?any ?x (kiss ?x ?y) ?y) --> (:word kissed))

(rule (Verb/intr ~3sg ?x (sleep ?x)) --> (:word sleep))
(rule (Verb/intr 3sg ?x (sleep ?x)) --> (:word sleeps))
(rule (Verb/intr ?any ?x (sleep ?x)) --> (:word slept))
```

Here are the rules for noun phrases and nouns:

```
(rule (NP ?agr ?sem) -->
  (Name ?agr ?sem))

(rule (NP ?agr (?det-sem ?noun-sem)) -->
  (Det ?agr ?det-sem)
  (Noun ?agr ?noun-sem))

(rule (Name 3sg Terry) --> (:word Terry))
(rule (Name 3sg Jean) --> (:word Jean))
```

```

(rule (Noun 3sg (young male human))      --> (:word boy))
(rule (Noun 3sg (young female human))     --> (:word girl))
(rule (Noun ~3sg (group (young male human))) --> (:word boys))
(rule (Noun ~3sg (group (young female human))) --> (:word girls))

(rule (Det ?any the) --> (:word the))
(rule (Det 3sg a) --> (:word a))

```

This grammar and lexicon generates more sentences, although it is still rather limited. Here are some examples:

```

> (?- (S ?sem (The boys kiss a girl) ()))
?SEM = (KISS (THE (GROUP (YOUNG MALE HUMAN)))
        (A (YOUNG FEMALE HUMAN))).

> (?- (S ?sem (The girls kissed the girls) ()))
?SEM = (KISS (THE (GROUP (YOUNG FEMALE HUMAN)))
        (THE (GROUP (YOUNG FEMALE HUMAN)))).

> (?- (S ?sem (Terry kissed the girl) ()))
?SEM = (KISS TERRY (THE (YOUNG FEMALE HUMAN))).

> (?- (S ?sem (The girls kisses the boys) ()))
No.

> (?- (S ?sem (Terry kissed a girls) ()))
No.

> (?- (S ?sem (Terry sleeps Jean) ()))
No.

```

The first three examples are parsed correctly, while the final three are correctly rejected. The inquisitive reader may wonder just what is going on in the interpretation of a sentence like “The girls kissed the girls.” Do the subject and object represent the same group of girls, or different groups? Does everyone kiss everyone, or are there fewer kissings going on? Until we define our representation more carefully, there is no way to tell. Indeed, it seems that there is a potential problem in the representation, in that the predicate `kiss` sometimes has individuals as its arguments, and sometimes groups. More careful representations of “The girls kissed the girls” include the following candidates, using predicate calculus:

$$\begin{aligned}
&\forall x \forall y \ x \in \text{girls} \wedge y \in \text{girls} \Rightarrow \text{kiss}(x,y) \\
&\forall x \forall y \ x \in \text{girls} \wedge y \in \text{girls} \wedge x \neq y \Rightarrow \text{kiss}(x,y) \\
&\forall x \exists y, z \ x \in \text{girls} \wedge y \in \text{girls} \wedge z \in \text{girls} \Rightarrow \text{kiss}(x,y) \wedge \text{kiss}(z,x) \\
&\forall x \exists y \ x \in \text{girls} \wedge y \in \text{girls} \Rightarrow \text{kiss}(x,y) \vee \text{kiss}(y,x)
\end{aligned}$$

The first of these says that every girl kisses every other girl. The second says the same thing, except that a girl need not kiss herself. The third says that every girl kisses

and is kissed by at least one other girl, but not necessarily all of them, and the fourth says that everybody is in on at least one kissing. None of these interpretations says anything about who “the girls” are.

Clearly, the predicate calculus representations are less ambiguous than the representation produced by the current system. On the other hand, it would be wrong to choose one of the representations arbitrarily, since in different contexts, “The girls kissed the girls” can mean different things. Maintaining ambiguity in a concise form is useful, as long as there is some way eventually to recover the proper meaning.

20.4 A DCG Grammar with Quantifiers

The problem in the representation we have been using becomes more acute when we consider other determiners, such as “every.” Consider the sentence “Every picture paints a story.” The preceding DCG, if given the right vocabulary, would produce the interpretation:

```
(paints (every picture) (a story))
```

This can be considered ambiguous between the following two meanings, in predicate calculus form:

$$\forall x \text{ picture}(x) \Rightarrow \exists y \text{ story}(y) \wedge \text{paint}(x,y)$$

$$\exists y \text{ story}(y) \wedge \forall x \text{ picture}(x) \Rightarrow \text{paint}(x,y)$$

The first says that for each picture, there is a story that it paints. The second says that there is a certain special story that every picture paints. The second is an unusual interpretation for this sentence, but for “Every U.S. citizen has a president,” the second interpretation is perhaps the preferred one. In the next section, we will see how to produce representations that can be transformed into either interpretation. For now, it is a useful exercise to see how we could produce just the first representation above, the interpretation that is usually correct. First, we need to transcribe it into Lisp:

```
(all ?x (-> (picture ?x) (exists ?y (and (story ?y) (paint ?x ?y)))))
```

The first question is how the `all` and `exists` forms get in there. They must come from the determiners, “every” and “a.” Also, it seems that `all` is followed by an implication arrow, `->`, while `exists` is followed by a conjunction, `and`. So the determiners will have translations looking like this:

```
(rule (Det ?any ?x ?p ?q (the ?x (and ?p ?q))) --> (:word the))
(rule (Det 3sg ?x ?p ?q (exists ?x (and ?p ?q))) --> (:word a))
(rule (Det 3sg ?x ?p ?q (all ?x (-> ?p ?q))) --> (:word every))
```

Once we have accepted these translations of the determiners, everything else follows. The formulas representing the determiners have two holes in them, ?p and ?q. The first will be filled by a predicate representing the noun, and the latter will be filled by the predicate that is being applied to the noun phrase as a whole. Notice that a curious thing is happening. Previously, translation to logical form was guided by the sentence's verb. Linguistically, the verb expresses the main predicate, so it makes sense that the verb's logical translation should be the main part of the sentence's translation. In linguistic terms, we say that the verb is the *head* of the sentence.

With the new translations for determiners, we are in effect turning the whole process upside down. Now the subject's determiner carries the weight of the whole sentence. The determiner's interpretation is a function of two arguments; it is applied to the noun first, yielding a function of one argument, which is in turn applied to the verb phrase's interpretation. This primacy of the determiner goes against intuition, but it leads directly to the right interpretation.

The variables ?p and ?q can be considered holes to be filled in the final interpretation, but the variable ?x fills a quite different role. At the end of the parse, ?x will not be filled by anything; it will still be a variable. But it will be referred to by the expressions filling ?p and ?q. We say that ?x is a *metavariable*, because it is a variable in the representation, not a variable in the Prolog implementation. It just happens that Prolog variables can be used to implement these metavariables.

Here are the interpretations for each word in our target sentence and for each intermediate constituent:

```
Every          = (all ?x (-> ?p1 ?q1))
picture        = (picture ?x)
paints         = (paint ?x ?y)
a              = (exists ?y (and ?p2 ?q2))
story          = (story ?y)

Every picture = (all ?x (-> (picture ?x) ?q1))
a story       = (exists ?y (and (story ?y) ?q2))
paints a story = (exists ?y (and (story ?y) (paint ?x ?y)))
```

The semantics of a noun has to fill the ?p hole of a determiner, possibly using the metavariable ?x. The three arguments to the Noun predicate are the agreement, the metavariable ?x, and the assertion that the noun phrase makes about ?x:

```
(rule (Noun 3sg ?x (picture ?x)) --> (:word picture))
(rule (Noun 3sg ?x (story ?x)) --> (:word story))
(rule (Noun 3sg ?x (and (young ?x) (male ?x) (human ?x))) -->
(:word boy))
```

The NP predicate is changed to take four arguments. First is the agreement, then the metavariable ?x. Third is a predicate that will be supplied externally, by the verb phrase. The final argument returns the interpretation of the NP as a whole. As we have stated, this comes from the determiner:

```
(rule (NP ?agr ?x ?pred ?pred) -->
(Name ?agr ?name))
;(rule (NP ?agr ?x ?pred ?np) -->
; (Det ?agr ?x ?noun ?pred ?np)
; (Noun ?agr ?x ?noun))
```

The rule for an NP with determiner is commented out because it is convenient to introduce an extended rule to replace it at this point. The new rule accounts for certain relative clauses, such as “the boy that paints a picture”:

```
(rule (NP ?agr ?x ?pred ?np) -->
(Det ?agr ?x ?noun&rel ?pred ?np)
(Noun ?agr ?x ?noun)
(rel-clause ?agr ?x ?noun ?noun&rel))
(rule (rel-clause ?agr ?x ?np ?np) --> )
(rule (rel-clause ?agr ?x ?np (and ?np ?rel)) -->
(:word that)
(VP ?agr ?x ?rel))
```

The new rule does not account for relative clauses where the object is missing, such as “the picture that the boy paints.” Nevertheless, the addition of relative clauses means we can now generate an infinite language, since we can always introduce a relative clause, which introduces a new noun phrase, which in turn can introduce yet another relative clause.

The rules for relative clauses are not complicated, but they can be difficult to understand. Of the four arguments to `rel-clause`, the first two hold the agreement features of the head noun and the metavariable representing the head noun. The last two arguments are used together as an accumulator for predications about the metavariable: the third argument holds the predications made so far, and the fourth will hold the predications including the relative clause. So, the first rule for `rel-clause` says that if there is no relative clause, then what goes in to the accumulator is the same as what goes out. The second rule says that what goes out is the conjunction of what comes in and what is predicated in the relative clause itself.

Verbs apply to either one or two metavariables, just as they did before. So we can use the definitions of `Verb/tr` and `Verb/intr` unchanged. For variety, I've added a few more verbs:

```
(rule (Verb/tr ~3sg ?x ?y (paint ?x ?y)) --> (:word paint))
(rule (Verb/tr 3sg ?x ?y (paint ?x ?y)) --> (:word paints))
(rule (Verb/tr ?any ?x ?y (paint ?x ?y)) --> (:word painted))

(rule (Verb/intr ~3sg ?x (sleep ?x)) --> (:word sleep))
(rule (Verb/intr 3sg ?x (sleep ?x)) --> (:word sleeps))
(rule (Verb/intr ?any ?x (sleep ?x)) --> (:word slept))

(rule (Verb/intr 3sg ?x (sells ?x)) --> (:word sells))
(rule (Verb/intr 3sg ?x (stinks ?x)) --> (:word stinks))
```

Verb phrases and sentences are almost as before. The only difference is in the call to `NP`, which now has extra arguments:

```
(rule (VP ?agr ?x ?vp) -->
  (Verb/tr ?agr ?x ?obj ?verb)
  (NP ?any-agr ?obj ?verb ?vp))

(rule (VP ?agr ?x ?vp) -->
  (Verb/intr ?agr ?x ?vp))

(rule (S ?np) -->
  (NP ?agr ?x ?vp ?np)
  (VP ?agr ?x ?vp))
```

With this grammar, we get the following correspondence between sentences and logical forms:

```
Every picture paints a story.
(ALL ?3 (-> (PICTURE ?3)
  (EXISTS ?14 (AND (STORY ?14) (PAINT ?3 ?14))))))

Every boy that paints a picture sleeps.
(ALL ?3 (-> (AND (AND (YOUNG ?3) (MALE ?3) (HUMAN ?3))
  (EXISTS ?19 (AND (PICTURE ?19)
    (PAINT ?3 ?19))))))
(SLEEP ?3)))

Every boy that sleeps paints a picture.
(ALL ?3 (-> (AND (AND (YOUNG ?3) (MALE ?3) (HUMAN ?3))
  (SLEEP ?3))
  (EXISTS ?22 (AND (PICTURE ?22) (PAINT ?3 ?22))))))
```

```

Every boy that paints a picture that sells
paints a picture that stinks.
(ALL ?3 (-> (AND (AND (YOUNG ?3) (MALE ?3) (HUMAN ?3))
                (EXISTS ?19 (AND (AND (PICTURE ?19) (SELLS ?19))
                                (PAINT ?3 ?19))))))
(EXISTS ?39 (AND (AND (PICTURE ?39) (STINKS ?39))
                (PAINT ?3 ?39))))

```

20.5 Preserving Quantifier Scope Ambiguity

Consider the simple sentence “Every man loves a woman.” This sentence is ambiguous between the following two interpretations:

$$\forall m \exists w \text{ man}(m) \wedge \text{woman}(w) \wedge \text{loves}(m,w)$$

$$\exists w \forall m \text{ man}(m) \wedge \text{woman}(w) \wedge \text{loves}(m,w)$$

The first interpretation is that every man loves some woman—his wife, perhaps. The second interpretation is that there is a certain woman whom every man loves—Natassja Kinski, perhaps. The meaning of the sentence is ambiguous, but the structure is not; there is only one syntactic parse.

In the last section, we presented a parser that would construct one of the two interpretations. In this section, we show how to construct a single interpretation that preserves the ambiguity, but can be disambiguated by a postsyntactic process. The basic idea is to construct an intermediate logical form that leaves the scope of quantifiers unspecified. This intermediate form can then be rearranged to recover the final interpretation.

To recap, here is the interpretation we would get for “Every man loves a woman,” given the grammar in the previous section:

```
(all ?m (-> (man ?m) (exists ?w) (and (woman ?w) (loves ?m ?w))))
```

We will change the grammar to produce instead the intermediate form:

```
(and (all ?m (man ?m))
      (exists ?w (woman ?w))
      (loves ?m ?w))
```

The difference is that logical components are produced in smaller chunks, with unscoped quantifiers. The typical grammar rule will build up an interpretation by conjoining constituents with *and*, rather than by fitting pieces into holes in other

pieces. Here is the complete grammar and a just-large-enough lexicon in the new format:

```
(rule (S (and ?np ?vp)) -->
  (NP ?agr ?x ?np)
  (VP ?agr ?x ?vp))

(rule (VP ?agr ?x (and ?verb ?obj)) -->
  (Verb/tr ?agr ?x ?o ?verb)
  (NP ?any-agr ?o ?obj))

(rule (VP ?agr ?x ?verb) -->
  (Verb/intr ?agr ?x ?verb))

(rule (NP ?agr ?name t) -->
  (Name ?agr ?name))

(rule (NP ?agr ?x ?det) -->
  (Det ?agr ?x (and ?noun ?rel) ?det)
  (Noun ?agr ?x ?noun)
  (rel-clause ?agr ?x ?rel))

(rule (rel-clause ?agr ?x t) --> )
(rule (rel-clause ?agr ?x ?rel) -->
  (:word that)
  (VP ?agr ?x ?rel))

(rule (Name 3sg Terry) --> (:word Terry))
(rule (Name 3sg Jean) --> (:word Jean))
(rule (Det 3sg ?x ?restr (all ?x ?restr)) --> (:word every))
(rule (Noun 3sg ?x (man ?x)) --> (:word man))
(rule (Verb/tr 3sg ?x ?y (love ?x ?y)) --> (:word loves))
(rule (Verb/intr 3sg ?x (lives ?x)) --> (:word lives))
(rule (Det 3sg ?x ?res (exists ?x ?res)) --> (:word a))
(rule (Noun 3sg ?x (woman ?x)) --> (:word woman))
```

This gives us the following parse for “Every man loves a woman”:

```
(and (all ?4 (and (man ?4) t))
  (and (love ?4 ?12) (exists ?12 (and (woman ?12) t))))
```

If we simplified this, eliminating the ts and joining ands, we would get the desired representation:

```
(and (all ?m (man ?m))
  (exists ?w (woman ?w))
  (loves ?m ?w))
```

From there, we could use what we know about syntax, in addition to what we know

about men, woman, and loving, to determine the most likely final interpretation. This will be covered in the next chapter.

20.6 Long-Distance Dependencies

So far, every syntactic phenomena we have considered has been expressible in a rule that imposes constraints only at a single level. For example, we had to impose the constraint that a subject agree with its verb, but this constraint involved two immediate constituents of a sentence, the noun phrase and verb phrase. We didn't need to express a constraint between, say, the subject and a modifier of the verb's object. However, there are linguistic phenomena that require just these kinds of constraints.

Our rule for relative clauses was a very simple one: a relative clause consists of the word "that" followed by a sentence that is missing its subject, as in "every man that loves a woman." Not all relative clauses follow this pattern. It is also possible to form a relative clause by omitting the object of the embedded sentence: "every man that a woman loves \square ." In this sentence, the symbol \square indicates a gap, which is understood as being filled by the head of the complete noun phrase, the man. This has been called a *filler-gap dependency*. It is also known as a *long-distance dependency*, because the gap can occur arbitrarily far from the filler. For example, all of the following are valid noun phrases:

The person that Lee likes \square
 The person that Kim thinks Lee likes \square
 The person that Jan says Kim thinks Lee likes \square

In each case, the gap is filled by the head noun, the person. But any number of relative clauses can intervene between the head noun and the gap.

The same kind of filler-gap dependency takes place in questions that begin with "who," "what," "where," and other interrogative pronouns. For example, we can ask a question about the subject of a sentence, as in "Who likes Lee?", or about the object, as in "Who does Kim like \square ?"

Here is a grammar that covers relative clauses with gapped subjects or objects. The rules for *S*, *VP*, and *NP* are augmented with a pair of arguments representing an accumulator for gaps. Like a difference list, the first argument minus the second represents the presence or absence of a gap. For example, in the first two rules for noun phrases, the two arguments are the same, ?g0 and ?g0. This means that the rule as a whole has no gap, since there can be no difference between the two arguments. In the third rule for *NP*, the first argument is of the form (gap . . .), and the second is nogap. This means that the right-hand side of the rule, an empty constituent, can be parsed as a gap. (Note that if we had been using true difference lists, the two

arguments would be ((gap ...) ?g0) and ?g0. But since we are only dealing with one gap per rule, we don't need true difference lists.)

The rule for S says that a noun phrase with gap ?g0 minus ?g1 followed by a verb phrase with gap ?g1 minus ?g2 comprise a sentence with gap ?g0 minus ?g2. The rule for relative clauses finds a sentence with a gap anywhere; either in the subject position or embedded somewhere in the verb phrase. Here's the complete grammar:

```
(rule (S ?g0 ?g2 (and ?np ?vp)) -->
  (NP ?g0 ?g1 ?agr ?x ?np)
  (VP ?g1 ?g2 ?agr ?x ?vp))

(rule (VP ?g0 ?g1 ?agr ?x (and ?obj ?verb)) -->
  (Verb/tr ?agr ?x ?o ?verb)
  (NP ?g0 ?g1 ?any-agr ?o ?obj))

(rule (VP ?g0 ?g0 ?agr ?x ?verb) -->
  (Verb/intr ?agr ?x ?verb))

(rule (NP ?g0 ?g0 ?agr ?name t) -->
  (Name ?agr ?name))

(rule (NP ?g0 ?g0 ?agr ?x ?det) -->
  (Det ?agr ?x (and ?noun ?rel) ?det)
  (Noun ?agr ?x ?noun)
  (rel-clause ?agr ?x ?rel))

(rule (NP (gap NP ?agr ?x) nogap ?agr ?x t) --> )

(rule (rel-clause ?agr ?x t) --> )

(rule (rel-clause ?agr ?x ?rel) -->
  (:word that)
  (S (gap NP ?agr ?x) nogap ?rel))
```

Here are some sentence/parse pairs covered by this grammar:

```
Every man that  $\sqcup$  loves a woman likes a person.
(AND (ALL ?28 (AND (MAN ?28)
  (AND T (AND (LOVE ?28 ?30)
    (EXISTS ?30 (AND (WOMAN ?30)
      T))))))
  (AND (EXISTS ?39 (AND (PERSON ?39) T)) (LIKE ?28 ?39)))
```

```
Every man that a woman loves  $\sqcup$  likes a person.
(AND (ALL ?37 (AND (MAN ?37)
  (AND (EXISTS ?20 (AND (WOMAN ?20) T))
    (AND T (LOVE ?20 ?37))))))
  (AND (EXISTS ?39 (AND (PERSON ?39) T)) (LIKE ?37 ?39)))
```

```

Every man that loves a bird that  $\sqcup$  flies likes a person.
(AND (ALL ?28 (AND (MAN ?28)
      (AND T (AND (EXISTS ?54
        (AND (BIRD ?54)
          (AND T (FLY ?54))))))
      (LOVE ?28 ?54))))))
(AND (EXISTS ?60 (AND (PERSON ?60) T)) (LIKE ?28 ?60)))

```

Actually, there are limitations on the situations in which gaps can appear. In particular, it is rare to have a gap in the subject of a sentence, except in the case of a relative clause. In the next chapter, we will see how to impose additional constraints on gaps.

20.7 Augmenting DCG Rules

In the previous section, we saw how to build up a semantic representation of a sentence by conjoining the semantics of the components. One problem with this approach is that the semantic interpretation is often something of the form (and (and t a) b), when we would prefer (and a b). There are two ways to correct this problem: either we add a step that takes the final semantic interpretation and simplifies it, or we complicate each individual rule, making it generate the simplified form. The second choice would be slightly more efficient, but would be very ugly and error prone. We should be doing all we can to make the rules simpler, not more complicated; that is the whole point of the DCG formalism. This suggests a third approach: change the rule interpreter so that it automatically generates the semantic interpretation as a conjunction of the constituents, unless the rule explicitly says otherwise. This section shows how to augment the DCG rules to handle common cases like this automatically.

Consider again a rule from section 20.4:

```

(rule (S (and ?np ?vp)) -->
  (NP ?agr ?x ?np)
  (VP ?agr ?x ?vp))

```

If we were to alter this rule to produce a simplified semantic interpretation, it would look like the following, where the predicate `and*` simplifies a list of conjunctions into a single conjunction:

```
(rule (S ?sem) -->
      (np ?agr ?x ?np)
      (vp ?agr ?x ?vp)
      (:test (and* (?np ?vp) ?sem)))
```

Many rules will have this form, so we adopt a simple convention: if the last argument of the constituent on the left-hand side of a rule is the keyword `:sem`, then we will build the semantics by replacing `:sem` with a conjunction formed by combining all the last arguments of the constituents on the right-hand side of the rule. A `==>` arrow will be used for rules that follow this convention, so the following rule is equivalent to the one above:

```
(rule (S :sem) ==>
      (NP ?agr ?x ?np)
      (VP ?agr ?x ?vp))
```

It is sometimes useful to introduce additional semantics that does not come from one of the constituents. This can be indicated with an element of the right-hand side that is a list starting with `:sem`. For example, the following rule adds to the semantics the fact that `?x` is the topic of the sentence:

```
(rule (S :sem) ==>
      (NP ?agr ?x ?np)
      (VP ?agr ?x ?vp)
      (:sem (topic ?x)))
```

Before implementing the rule function for the `==>` arrow, it is worth considering if there are other ways we could make things easier for the rule writer. One possibility is to provide a notation for describing examples. Examples make it easier to understand what a rule is designed for. For the `S` rule, we could add examples like this:

```
(rule (S :sem) ==>
      (:ex "John likes Mary" "He sleeps")
      (NP ?agr ?x ?np)
      (VP ?agr ?x ?vp))
```

These examples not only serve as documentation for the rule but also can be stored under `S` and subsequently run when we want to test if `S` is in fact implemented properly.

Another area where the rule writer could use help is in handling left-recursive rules. Consider the rule that says that a sentence can consist of two sentences joined by a conjunction:

```
(rule (S (?conj ?s1 ?s2)) ==>
  (:ex "John likes Mary and Mary likes John")
  (S ?s1)
  (Conj ?conj)
  (S ?s2))
```

While this rule is correct as a declarative statement, it will run into difficulty when run by the standard top-down depth-first DCG interpretation process. The top-level goal of parsing an *S* will lead immediately to the subgoal of parsing an *S*, and the result will be an infinite loop.

Fortunately, we know how to avoid this kind of infinite loop: split the offending predicate, *S*, into two predicates: one that supports the recursion, and one that is at a lower level. We will call the lower-level predicate *S_*. Thus, the following rule says that a sentence can consist of two sentences, where the first one is not conjoined and the second is possibly conjoined:

```
(rule (S (?conj ?s1 ?s2)) ==>
  (S_ ?s1)
  (Conj ?conj)
  (S ?s2))
```

We also need a rule that says that a possibly conjoined sentence can consist of a nonconjoined sentence:

```
(rule (S ?sem) ==> (S_ ?sem))
```

To make this work, we need to replace any mention of *S* in the left-hand side of a rule with *S_*. References to *S* in the right-hand side of rules remain unchanged.

```
(rule (S_ ?sem) ==> ...)
```

To make this all automatic, we will provide a macro, *conj-rule*, that declares a category to be one that can be conjoined. Such a declaration will automatically generate the recursive and nonrecursive rules for the category, and will insure that future references to the category on the left-hand side of a rule will be replaced with the corresponding lower-level predicate.

One problem with this approach is that it imposes a right-branching parse on multiple conjoined phrases. That is, we will get parses like "spaghetti and (meatballs and salad)" not "(spaghetti and meatballs) and salad." Clearly, that is the wrong interpretation for this sentence. Still, it can be argued that it is best to produce a single canonical parse, and then let the semantic interpretation functions worry about rearranging the parse in the right order. We will not attempt to resolve this

debate but will provide the automatic conjunction mechanism as a tool that can be convenient but has no cost for the user who prefers a different solution.

We are now ready to implement the extended DCG rule formalism that handles `:sem`, `:ex`, and automatic conjunctions. The function `make-augmented-dcg`, stored under the arrow `==>`, will be used to implement the formalism:

```
(setf (get '==> 'rule-function) 'make-augmented-dcg)

(defun make-augmented-dcg (head body)
  "Build an augmented DCG rule that handles :sem, :ex,
  and automatic conjunctions."
  (if (eq (last1 head) :sem)
      ;; Handle :sem
      (let* ((?sem (gensym "?SEM")))
        (make-augmented-dcg
         '(,@(butlast head) ,?sem)
         '(,@(remove :sem body :key #'first-or-nil)
           (:test ,(collect-sems body ?sem))))))
      ;; Separate out examples from body
      (multiple-value-bind (exs new-body)
        (partition-if #'(lambda (x) (starts-with x :ex)) body)
        ;; Handle conjunctions
        (let ((rule '(rule ,(handle-conj head) --> ,@new-body)))
          (if (null exs)
              rule
              '(progn (:ex ,head .,(mappend #'rest exs))
                    ,rule))))))
```

First we show the code that collects together the semantics of each constituent and conjoins them when `:sem` is specified. The function `collect-sems` picks out the semantics and handles the trivial cases where there are zero or one constituents on the right-hand side. If there are more than one, it inserts a call to the predicate `and*`.

```
(defun collect-sems (body ?sem)
  "Get the semantics out of each constituent in body,
  and combine them together into ?sem."
  (let ((sems (loop for goal in body
                    unless (or (dcg-normal-goal-p goal)
                                (dcg-word-list-p goal)
                                (starts-with goal :ex)
                                (atom goal))
                    collect (last1 goal))))
    (case (length sems)
      (0 '(= ,?sem t))
      (1 '(= ,?sem ,(first sems)))
      (t '(and* ,sems ,?sem))))))
```

We could have implemented `and*` with Prolog clauses, but it is slightly more efficient to do it directly in Lisp. A call to `conjuncts` collects all the conjuncts, and we then add an `and` if necessary:

```
(defun and*/2 (in out cont)
  "IN is a list of conjuncts that are conjoined into OUT."
  ;; E.g.: (and* (t (and a b) t (and c d) t) ?x) ==>
  ;;       ?x = (and a b c d)
  (if (unify! out (maybe-add 'and (conjuncts (cons 'and in)) t))
      (funcall cont)))

(defun conjuncts (exp)
  "Get all the conjuncts from an expression."
  (deref exp)
  (cond ((eq exp t) nil)
        ((atom exp) (list exp))
        ((eq (deref (first exp)) 'nil) nil)
        ((eq (first exp) 'and)
         (mappend #'conjuncts (rest exp)))
        (t (list exp))))
```

The next step is handling example phrases. The code in `make-augmented-dcg` turns examples into expressions of the form:

```
(:ex (S ?sem) "John likes Mary" "He sleeps")
```

To make this work, `:ex` will have to be a macro:

```
(defmacro :ex ((category . args) &body examples)
  "Add some example phrases, indexed under the category."
  `(add-examples ',category ',args ',examples))
```

`:ex` calls `add-examples` to do all the work. Each example is stored in a hash table indexed under the the category. Each example is transformed into a two-element list: the example phrase string itself and a call to the proper predicate with all arguments supplied. The function `add-examples` does this transformation and indexing, and `run-examples` retrieves the examples stored under a category, prints each phrase, and calls each goal. The auxiliary functions `get-examples` and `clear-examples` are provided to manipulate the example table, and `remove-punction`, `punctuation-p` and `string->list` are used to map from a string to a list of words.

```
(defvar *examples* (make-hash-table :test #'eq))
(defun get-examples (category) (gethash category *examples*))
(defun clear-examples () (clrhash *examples*))
```



```

(defun add-examples (category args examples)
  "Add these example strings to this category,
  and when it comes time to run them, use the args."
  (dolist (example examples)
    (when (stringp example)
      (let ((ex '(,example
                  (,category ,@args
                  ,(string->list
                     (remove-punctuation example)) ())))
        (unless (member ex (get-examples category)
                        :test #'equal)
          (setf (gethash category *examples*)
                (nconc (get-examples category) (list ex)))))))

(defun run-examples (&optional category)
  "Run all the example phrases stored under a category.
  With no category, run ALL the examples."
  (prolog-compile-symbols)
  (if (null category)
      (maphash #'(lambda (cat val)
                  (declare (ignore val))
                  (format t "~2&Examples of ~a:~%" cat)
                  (run-examples cat)
                  *examples*))
      (dolist (example (get-examples category))
        (format t "~2&EXAMPLE: ~{~a~&~9T~a~}" example)
        (top-level-prove (cdr example)))))

(defun remove-punctuation (string)
  "Replace punctuation with spaces in string."
  (substitute-if #\space #'punctuation-p string))

(defun string->list (string)
  "Convert a string to a list of words."
  (read-from-string (concatenate 'string "(" string ")")))

(defun punctuation-p (char) (find char "*_.,:;!#-()\\"))

```

The final part of our augmented DCG formalism is handling conjunctive constituents automatically. We already arranged to translate category symbols on the left-hand side of rules into the corresponding conjunctive category, as specified by the function `handle-conj`. We also want to generate automatically (or as easily as possible) rules of the following form:

```

(rule (S (?conj ?s1 ?s2)) ==>
      (S_ ?s1)
      (Conj ?conj)
      (S ?s2))

```

```
(rule (S ?sem) ==> (S_ ?sem))
```

But before we generate these rules, let's make sure they are exactly what we want. Consider parsing a nonconjoined sentence with these two rules in place. The first rule would parse the entire sentence as a *S*_—, and would then fail to see a *Conj*, and thus fail. The second rule would then duplicate the entire parsing process, thus doubling the amount of time taken. If we changed the order of the two rules we would be able to parse nonconjoined sentences quickly, but would have to backtrack on conjoined sentences.

The following shows a better approach. A single rule for *S* parses a sentence with *S*_—, and then calls *Conj_S*, which can be read as "either a conjunction followed by a sentence, or nothing." If the first sentence is followed by nothing, then we just use the semantics of the first sentence; if there is a conjunction, we have to form a combined semantics. I have added ... to show where arguments to the predicate other than the semantic argument fit in.

```
(rule (S ... ?s-combined) ==>
  (S_ ... ?sem1)
  (Conj_S ?sem1 ?s-combined))

(rule (Conj_S ?sem1 (?conj ?sem1 ?sem2)) ==>
  (Conj ?conj)
  (S ... ?sem2))

(rule (Conj_S ?sem1 ?sem1) ==>)
```

Now all we need is a way for the user to specify that these three rules are desired. Since the exact method of building up the combined semantics and perhaps even the call to *Conj* may vary depending on the specifics of the grammar being defined, the rules cannot be generated entirely automatically. We will settle for a macro, *conj-rule*, that looks very much like the second of the three rules above but expands into all three, plus code to relate *S*_— to *S*. So the user will type:

```
(conj-rule (Conj_S ?sem1 (?conj ?sem1 ?sem2)) ==>
  (Conj ?conj)
  (S ?a ?b ?c ?sem2))
```

Here is the macro definition:

```
(defmacro conj-rule ((conj-cat sem1 combined-sem) ==>
  conj (cat . args))
  "Define this category as an automatic conjunction."
  '(progn
    (setf (get ',cat 'conj-cat) ',(symbol cat '_))
```

```
(rule (,cat ,@(butlast args) ?combined-sem) ==>
      (,(symbol cat '_) ,@(butlast args) ,sem1)
      (,conj-cat ,sem1 ?combined-sem))
(rule (,conj-cat ,sem1 ,combined-sem) ==>
      ,conj
      (,cat ,@args))
(rule (,conj-cat ?sem1 ?sem1) ==>)))
```

and here we define `handle-conj` to substitute `S_` for `S` in the left-hand side of rules:

```
(defun handle-conj (head)
  "Replace (Cat ...) with (Cat_ ...) if Cat is declared
  as a conjunctive category."
  (if (and (listp head) (conj-category (predicate head)))
      (cons (conj-category (predicate head)) (args head))
      head))

(defun conj-category (predicate)
  "If this is a conjunctive predicate, return the Cat_ symbol."
  (get predicate 'conj-category))
```

20.8 History and References

As we have mentioned, Alain Colmerauer invented Prolog to use in his grammar of French (1973). His *metamorphosis grammar* formalism was more expressive but much less efficient than the standard DCG formalism.

The grammar in section 20.4 is essentially the same as the one presented in Fernando Pereira and David H. D. Warren's 1980 paper, which introduced the Definite Clause Grammar formalism as it is known today. The two developed a much more substantial grammar and used it in a very influential question-answering system called Chat-80 (Warren and Pereira, 1982). Pereira later teamed with Stuart Shieber on an excellent book covering logic grammars in more depth: *Prolog and Natural-Language Analysis* (1987). The book has many strong points, but unfortunately it does not present a grammar anywhere near as complete as the Chat-80 grammar.

The idea of a compositional semantics based on mathematical logic owes much to the work of the late linguist Richard Montague. The introduction by Dowty, Wall, and Peters (1981) and the collection by Rich Thomason (1974) cover Montague's approach.

The grammar in section 20.5 is based loosely on Michael McCord's modular logic grammar, as presented in Walker et al. 1990.

It should be noted that logic grammars are by no means the only approach to natural language processing. Woods (1970) presents an approach based on the

augmented transition network, or ATN. A transition network is like a context-free grammar. The *augmentation* is a way of manipulating features and semantic values. This is just like the extra arguments in DCGs, except that the basic operations are setting and testing variables rather than unification. So the choice between ATNs and DCGs is largely a matter of what programming approach you are most comfortable with: procedural for ATNs and declarative for DCGs. My feeling is that unification is a more suitable primitive than assignment, so I chose to present DCGs, even though this required bringing in Prolog's backtracking and unification mechanisms.

In either approach, the same linguistic problems must be addressed—agreement, long-distance dependencies, topicalization, quantifier-scope ambiguity, and so on. Comparing Woods's (1970) ATN grammar to Pereira and Warren's (1980) DCG grammar, the careful reader will see that the solutions have much in common. The analysis is more important than the notation, as it should be.

20.9 Exercises

- ? **Exercise 20.2 [m]** Modify the grammar (from section 20.4, 20.5, or 20.6) to allow for adjectives before a noun.

- ? **Exercise 20.3 [m]** Modify the grammar to allow for prepositional phrase modifiers on verb and noun phrases.

- ? **Exercise 20.4 [m]** Modify the grammar to allow for ditransitive verbs—verbs that take two objects, as in “give the dog a bone.”

- ? **Exercise 20.5** Suppose we wanted to adopt the Prolog convention of writing DCG tests and words in brackets and braces, respectively. Write a function that will alter the readtable to work this way.

- ? **Exercise 20.6 [m]** Define a rule function for a new type of DCG rule that automatically builds up a syntactic parse of the input. For example, the two rules:

```
(rule (s) => (np) (vp))
(rule (np) => (:word he))
```

should be equivalent to:

```
(rule (s (s ?1 ?2)) --> (np ?1) (vp ?2))
(rule (np (np he)) --> (:word he))
```

? **Exercise 20.7 [m]** There are advantages and disadvantages to the approach that Prolog takes in dividing predicates into clauses. The advantage is that it is easy to add a new clause. The disadvantage is that it is hard to alter an existing clause. If you edit a clause and then evaluate it, the new clause will be added to the end of the clause list, when what you really wanted was for the new clause to take the place of the old one. To achieve that effect, you have to call `clear-predicate`, and then reload all the clauses, not just the one that has been changed.

Write a macro named `-rule` that is just like `rule`, except that it attaches names to clauses. When a named rule is reloaded, it replaces the old clause rather than adding a new one.

? **Exercise 20.8 [h]** Extend the DCG rule function to allow `or` goals in the right-hand side. To make this more useful, also allow `and` goals. For example:

```
(rule (A) --> (B) (or (C) (and (D) (E))) (F))
```

should compile into the equivalent of:

```
(<- (A ?S0 ?S4)
    (B ?S0 ?S1)
    (OR (AND (C ?S1 ?S2) (= ?S2 ?S3))
        (AND (D ?S1 ?S2) (E ?S2 ?S3)))
    (F ?S3 ?S4))
```

20.10 Answers

Answer 20.1 It uses local variables (`?s0`, `?s1` ...) that are not guaranteed to be unique. This is a problem if the grammar writer wants to use these symbols anywhere in his or her rules. The fix is to gensym symbols that are guaranteed to be unique.

Answer 20.5

```

(defun setup-braces (&optional (on? t) (readtable *readtable*))
  "Make [a b] read as (:word a b) and {a b} as (:test a b c)
  if ON? is true; otherwise revert {} to normal."
  (if (not on?)
      (map nil #'(lambda (c)
                   (set-macro-character c (get-macro-character #\a)
                                         t readtable))
           "{[]}")
      (progn
        (set-macro-character
         #\[ (get-macro-character #\)) nil readtable)
        (set-macro-character
         #\} (get-macro-character #\)) nil readtable)
        (set-macro-character
         #\[ #'(lambda (s ignore)
                 (cons :word (read-delimited-list #\] s t)))
         nil readtable)
        (set-macro-character
         #\{ #'(lambda (s ignore)
                 (cons :test (read-delimited-list #\} s t)))
         nil readtable))))

```

CHAPTER 21

A Grammar of English

Prefer geniality to grammar.

—Henry Watson Fowler
The King's English (1906)

The previous two chapters outline techniques for writing grammars and parsers based on those grammars. It is quite straightforward to apply these techniques to applications like the CD player problem where input is limited to simple sentences like “Play 1 to 8 without 3.” But it is a major undertaking to write a grammar for unrestricted English input. This chapter develops a grammar that covers all the major syntactic constructions of English. It handles sentences of much greater complexity, such as “Kim would not have been persuaded by Lee to look after the dog.” The grammar is not comprehensive enough to handle sentences chosen at random from a book, but when augmented by suitable vocabulary it is adequate for a wide variety of applications.

This chapter is organized as a tour through the English language. We first cover noun phrases, then verb phrases, clauses, and sentences. For each category we introduce examples, analyze them linguistically, and finally show definite clause grammar rules that correspond to the analysis.

As the last chapter should have made clear, analysis more often results in complication than in simplification. For example, starting with a simple rule like ($S \rightarrow NP VP$), we soon find that we have to add arguments to handle agreement, semantics, and gapping information. Figure 21.1 lists the grammatical categories and their arguments. Note that the semantic argument, *sem*, is always last, and the gap accumulators, *gap1* and *gap2*, are next-to-last whenever they occur. All single-letter arguments denote metavariables; for example, each noun phrase (category NP) will have a semantic interpretation, *sem*, that is a conjunction of relations involving the variable *x*. Similarly, the *h* in *modifiers* is a variable that refers to the head—the thing that is being modified. The other arguments and categories will be explained in turn, but it is handy to have this figure to refer back to.

Category	Arguments
Preterminals	
name	agr name
verb	verb inflection slots v sem
rel-pro	case type
pronoun	agr case wh x sem
art	agr quant
adj	x sem
cardinal	number agr
ordinal	number
prep	prep sem
noun	agr slots x sem
aux	inflection needs-inflection v sem
adverb	x sem
Nonterminals	
S	s sem
aux-inv-S	subject s sem
clause	inflection x int-subj v gap1 gap2 sem
subject	agr x subj-slot int-subj gap1 gap2 sem
VP	inflection x subject-slot v gap1 gap2 vp
NP	agr case wh x gap1 gap2 np
NP2	agr case x gap1 gap2 sem
PP	prep role wh np x gap1 gap2 sem
XP	slot constituent wh x gap1 gap2 sem
Det	agr wh x restriction sem
rel-clause	agr x sem
modifiers	pre/post cat info slots h gap1 gap2 sem
complement	cat info slot h gap1 gap2 sem
adjunct	pre/post cat info h gap1 gap2 sem
advp	wh x gap1 gap2 sem

Figure 21.1: Grammatical Categories and their Arguments

21.1 Noun Phrases

The simplest noun phrases are names and pronouns, such as “Kim” and “them.” The rules for these cases are simple: we build up a semantic expression from a name or pronoun, and since there can be no gap, the two gap accumulator arguments are the same (?g1). Person and number agreement is propagated in the variable ?agr, and we also keep track of the *case* of the noun phrase. English has three cases that are reflected in certain pronouns. In the first person singular, “I” is the *nominative* or *subjective* case, “me” is the *accusative* or *objective* case, and “my” is the *genitive* case. To distinguish them from the genitive, we refer to the nominative and the objective cases as the *common* cases. Accordingly, the three cases will be marked by the expressions (common nom), (common obj), and gen, respectively. Many languages of the world have suffixes that mark nouns as being one case or another, but English does not. Thus, we use the expression (common ?) to mark nouns.

We also distinguish between noun phrases that can be used in questions, like “who,” and those that cannot. The ?wh variable has the value +wh for noun phrases like “who” or “which one” and -wh for nonquestion phrases. Here, then, are the rules for names and pronouns. The predicates name and pronoun are used to look up words in the lexicon.

```
(rule (NP ?agr (common ?) -wh ?x ?g1 ?g1 (the ?x (name ?name ?x))) ==>
  (name ?agr ?name))

(rule (NP ?agr ?case ?wh ?x ?g1 ?g1 ?sem) ==>
  (pronoun ?agr ?case ?wh ?x ?sem))
```

Plural nouns can stand alone as noun phrases, as in “dogs,” but singular nouns need a determiner, as in “the dog” or “Kim’s friend’s biggest dog.” Plural nouns can also take a determiner, as in “the dogs.” The category Det is used for determiners, and NP2 is used for the part of a noun phrase after the determiner:

```
(rule (NP (- - - +) ?case -wh ?x ?g1 ?g2 (group ?x ?sem)) ==>
  (:ex "dogs") ; Plural nouns don't need a determiner
  (NP2 (- - - +) ?case ?x ?g1 ?g2 ?sem))

(rule (NP ?agr (common ?) ?wh ?x ?g1 ?g2 ?sem) ==>
  (:ex "Every man" "The dogs on the beach")
  (Det ?agr ?wh ?x ?restriction ?sem)
  (NP2 ?agr (common ?) ?x ?g1 ?g2 ?restriction))
```

Finally, a noun phrase may appear externally to a construction, in which case the noun phrase passed in by the first gap argument will be consumed, but no words from the input will be. An example is the \square in “Whom does Kim like \square ?”

```
(rule (NP ?agr ?case ?wh ?x (gap (NP ?agr ?case ?x)) (gap nil) t)
  ==> ;; Gapped NP
)
```

Now we address the heart of the noun phrase, the NP2 category. The lone rule for NP2 says that it consists of a noun, optionally preceded and followed by modifiers:

```
(rule (NP2 ?agr (common ?) ?x ?g1 ?g2 :sem) ==>
  (modifiers pre noun ?agr () ?x (gap nil) (gap nil) ?pre)
  (noun ?agr ?slots ?x ?noun)
  (modifiers post noun ?agr ?slots ?x ?g1 ?g2 ?post))
```

21.2 Modifiers

Modifiers are split into type types: *Complements* are modifiers that are expected by the head category that is being modified; they cannot stand alone. *Adjuncts* are modifiers that are not required but bring additional information. The distinction is clearest with verb modifiers. In “Kim visited Lee yesterday,” “visited” is the head verb, “Lee” is a complement, and “yesterday” is an adjunct. Returning to nouns, in “the former mayor of Boston,” “mayor” is the head noun, “of Boston” is a complement (although an optional one) and “former” is an adjunct.

The predicate `modifiers` takes eight arguments, so it can be tricky to understand them all. The first two arguments tell if we are before or after the head (`pre` or `post`) and what kind of head we are modifying (`noun`, `verb`, or whatever). Next is an argument that passes along any required information—in the case of nouns, it is the agreement feature. The fourth argument is a list of expected complements, here called `?slots`. Next is the metavariable used to refer to the head. The final three arguments are the two gap accumulators and the semantics, which work the same way here as we have seen before. Notice that the lexicon entry for each Noun can have a list of complements that are considered as postnoun modifiers, but there can be only adjuncts as prenoun modifiers. Also note that gaps can appear in the postmodifiers but not in the premodifiers. For example, we can have “What is Kevin the former mayor of \square ?” where the answer might be “Boston.” But even though we can construct a noun phrase like “the education president,” where “education” is a prenoun modifier of “president,” we cannot construct “* What is George the \square president?” intending that the answer be “education.”

There are four cases for modification. First, a complement is a kind of modifier. Second, if a complement is marked as optional, it can be skipped. Third, an adjunct can appear in the input. Fourth, if there are no complements expected, then there need not be any modifiers at all. The following rules implement these four cases:

```

(rule (modifiers ?pre/post ?cat ?info (?slot . ?slots) ?h
      ?g1 ?g3 :sem) ==>
  (complement ?cat ?info ?slot ?h ?g1 ?g2 ?mod)
  (modifiers ?pre/post ?cat ?info ?slots ?h ?g2 ?g3 ?mods))
(rule (modifiers ?pre/post ?cat ?info ((? (?)) . ?slots) ?h
      ?g1 ?g2 ?mods) ==>
  (modifiers ?pre/post ?cat ?info ?slots ?h ?g1 ?g2 ?mods))
(rule (modifiers ?pre/post ?cat ?info ?slots ?h ?g1 ?g3 :sem) ==>
  (adjunct ?pre/post ?cat ?info ?h ?g1 ?g2 ?adjunct)
  (modifiers ?pre/post ?cat ?info ?slots ?h ?g2 ?g3 ?mods))
(rule (modifiers ? ? ? () ? ?g1 ?g1 t) ==> )

```

We need to say more about the list of complements, or slots, that can be associated with words in the lexicon. Each slot is a list of the form (*role number form*), where the role refers to some semantic relation, the number indicates the ordering of the complements, and the form is the type of constituent expected: noun phrase, verb phrase, or whatever. The details will be covered in the following section on verb phrases, and complement will be covered in the section on XPs. For now, we give a single example. The complement list for one sense of the verb “visit” is:

```
((agt 1 (NP ?)) (obj 2 (NP ?)))
```

This means that the first complement, the subject, is a noun phrase that fills the agent role, and the second complement is also a noun phrase that fills the object role.

21.3 Noun Modifiers

There are two main types of prenoun adjuncts. Most common are adjectives, as in “big slobbery dogs.” Nouns can also be adjuncts, as in “water meter” or “desk lamp.” Here it is clear that the second noun is the head and the first is the modifier: a desk lamp is a lamp, not a desk. These are known as noun-noun compounds. In the following rules, note that we do not need to say that more than one adjective is allowed; this is handled by the rules for modifiers.

```

(rule (adjunct pre noun ?info ?x ?gap ?gap ?sem) ==>
  (adj ?x ?sem))
(rule (adjunct pre noun ?info ?h ?gap ?gap :sem) ==>
  (:sem (noun-noun ?h ?x))
  (noun ?agr () ?x ?sem))

```

After the noun there is a wider variety of modifiers. Some nouns have complements,

which are primarily prepositional phrases, as in "mayor of Boston." These will be covered when we get to the lexical entries for nouns. Prepositional phrases can be adjuncts for nouns or verbs, as in "man in the middle" and "slept for an hour." We can write one rule to cover both cases:

```
(rule (adjunct post ?cat ?info ?x ?g1 ?g2 ?sem) ==>
  (PP ?prep ?prep ?wh ?np ?x ?g1 ?g2 ?sem))
```

Here are the rules for prepositional phrases, which can be either a preposition followed by a noun phrase or can be gapped, as in "to whom are you speaking \square ?" The object of a preposition is always in the objective case: "with him" not "*with he."

```
(rule (PP ?prep ?role ?wh ?np ?x ?g1 ?g2 :sem) ==>
  (prep ?prep t)
  (:sem (?role ?x ?np))
  (NP ?agr (common obj) ?wh ?np ?g1 ?g2 ?np-sem))

(rule (PP ?prep ?role ?wh ?np ?x
  (gap (PP ?prep ?role ?np ?x)) (gap nil) t) ==> )
```

Nouns can be modified by present participles, past participles, and relative clauses. Examples are "the man eating the snack," "the snack eaten by the man," and "the man that ate the snack," respectively. We will see that each verb in the lexicon is marked with an inflection, and that the marker *-ing* is used for present participles while *-en* is used for past participles. The details of the clause will be covered later.

```
(rule (adjunct post noun ?agr ?x ?gap ?gap ?sem) ==>
  (:ex (the man) "visiting me" (the man) "visited by me")
  (:test (member ?infl (-ing passive)))
  (clause ?infl ?x ? ?v (gap (NP ?agr ? ?x)) (gap nil) ?sem))

(rule (adjunct post noun ?agr ?x ?gap ?gap ?sem) ==>
  (rel-clause ?agr ?x ?sem))
```

It is possible to have a relative clause where it is an object, not the subject, that the head refers to: "the snack that the man ate." In this kind of relative clause the relative pronoun is optional: "The snack the man ate was delicious." The following rules say that if the relative pronoun is omitted then the noun that is being modified must be an object, and the relative clause should include a subject internally. The constant *int-subj* indicates this.

```
(rule (rel-clause ?agr ?x :sem) ==>
  (:ex (the man) "that she liked" "that liked her"
  "that I know Lee liked"))
```

```

(opt-rel-pronoun ?case ?x ?int-subj ?rel-sem)
  (clause (finite ? ?) ? ?int-subj ?v
    (gap (NP ?agr ?case ?x)) (gap nil) ?clause-sem))

(rule (opt-rel-pronoun ?case ?x ?int-subj (?type ?x)) ==>
  (:word ?rel-pro)
  (:test (word ?rel-pro rel-pro ?case ?type)))

(rule (opt-rel-pronoun (common obj) ?x int-subj t) ==> )

```

It should be noted that it is rare but not impossible to have names and pronouns with modifiers: “John the Baptist,” “lovely Rita, meter maid,” “Lucy in the sky with diamonds,” “Sylvia in accounting on the 42nd floor,” “she who must be obeyed.” Here and throughout this chapter we will raise the possibility of such rare cases, leaving them as exercises for the reader.

21.4 Determiners

We will cover three kinds of determiners. The simplest is the article: “a dog” or “the dogs.” We also allow genitive pronouns, as in “her dog,” and numbers, as in “three dogs.” The semantic interpretation of a determiner-phrase is of the form (*quantifier variable restriction*). For example, (a ?x (dog ?x)) or ((number 3) ?x (dog ?x)).

```

(rule (Det ?agr ?wh ?x ?restriction (?art ?x ?restriction)) ==>
  (:ex "the" "every")
  (art ?agr ?art)
  (:test (if (= ?art wh) (= ?wh +wh) (= ?wh -wh))))

(rule (Det ?agr ?wh ?x ?r (the ?x ?restriction)) ==>
  (:ex "his" "her")
  (pronoun ?agr gen ?wh ?y ?sem)
  (:test (and* ((genitive ?y ?x) ?sem ?r) ?restriction)))

(rule (Det ?agr -wh ?x ?r ((number ?n) ?x ?r)) ==>
  (:ex "three")
  (cardinal ?n ?agr))

```

These are the most important determiner types, but there are others, and there are pre- and postdeterminers that combine in restricted combinations. Predeterminers include all, both, half, double, twice, and such. Postdeterminers include every, many, several, and few. Thus, we can say “all her many good ideas” or “all the King’s men.” But we can not say “*all much ideas” or “*the our children.” The details are complicated and are omitted from this grammar.

21.5 Verb Phrases

Now that we have defined *modifiers*, verb phrases are easy. In fact, we only need two rules. The first says a verb phrase consists of a verb optionally preceded and followed by modifiers, and that the meaning of the verb phrase includes the fact that the subject fills some role:

```
(rule (VP ?infl ?x ?subject-slot ?v ?g1 ?g2 :sem) ==>
  (:ex "sleeps" "quickly give the dog a bone")
  (modifiers pre verb ? () ?v (gap nil) (gap nil) ?pre-sem)
  (:sem (?role ?v ?x)) (:test (= ?subject-slot (?role 1 ?)))
  (verb ?verb ?infl (?subject-slot . ?slots) ?v ?v-sem)
  (modifiers post verb ? ?slots ?v ?g1 ?g2 ?mod-sem))
```

The VP category takes seven arguments. The first is an inflection, which represents the tense of the verb. To describe the possibilities for this argument we need a quick review of some basic linguistics. A sentence must have a *finite* verb, meaning a verb in the present or past tense. Thus, we say “Kim likes Lee,” not “*Kim liking Lee.” Subject-predicate agreement takes effect for finite verbs but not for any other tense. The other tenses show up as complements to other verbs. For example, the complement to “want” is an infinitive: “Kim wants *to like* Lee” and the complement to the modal auxiliary verb “would” is a nonfinite verb: “Kim would *like* Lee.” If this were in the present tense, it would be “likes,” not “like.” The inflection argument takes on one of the forms in the table here:

Expression	Type	Example
(finite ?agr present)	present tense	eat, eats
(finite ?agr past)	past tense	ate
nonfinite	nonfinite	eat
infinitive	infinitive	to eat
-en	past participle	eaten
-ing	present participle	eating

The second argument is a metavariable that refers to the subject, and the third is the subject’s complement slot. We adopt the convention that the subject slot must always be the first among the verb’s complements. The other slots are handled by the postverb modifiers. The fourth argument is a metavariable indicating the verb phrase itself. The final three are the familiar gap and semantics arguments. As an example, if the verb phrase is the single word “slept,” then the semantics of the verb phrase will be (and (past ?v) (sleep ?v)). Of course, adverbs, complements, and adjuncts will also be handled by this rule.

The second rule for verb phrases handles auxiliary verbs, such as “have,” “is” and “would.” Each auxiliary verb (or aux) produces a verb phrase with a particular

inflection when followed by a verb phrase with the required inflection. To repeat an example, “would” produces a finite phrase when followed by a nonfinite verb. “Have” produces a nonfinite when followed by a past participle. Thus, “would have liked” is a finite verb phrase.

We also need to account for negation. The word “not” can not modify a bare main verb but can follow an auxiliary verb. That is, we can’t say “*Kim not like Lee,” but we can add an auxiliary to get “Kim does not like Lee.”

```
(rule (VP ?infl ?x ?subject-slot ?v ?g1 ?g2 :sem) ==>
  (:ex "is sleeping" "would have given a bone to the dog."
    "did not sleep" "was given a bone by this old man")
  ;; An aux verb, followed by a VP
  (aux ?infl ?needs-infl ?v ?aux)
  (modifiers post aux ? () ?v (gap nil) (gap nil) ?mod)
  (VP ?needs-infl ?x ?subject-slot ?v ?g1 ?g2 ?vp))

(rule (adjunct post aux ? ?v ?gap ?gap (not ?v)) ==>
  (:word not))
```

21.6 Adverbs

Adverbs can serve as adjuncts before or after a verb: “to boldly go,” “to go boldly.” There are some limitations on where they can occur, but it is difficult to come up with firm rules; here we allow any adverb anywhere. We define the category *advp* for adverbial phrase, but currently restrict it to a single adverb.

```
(rule (adjunct ?pre/post verb ?info ?v ?g1 ?g2 ?sem) ==>
  (advp ?wh ?v ?g1 ?g2 ?sem))

(rule (advp ?wh ?v ?gap ?gap ?sem) ==>
  (adverb ?wh ?v ?sem))

(rule (advp ?wh ?v (gap (advp ?v)) (gap nil) t) ==> )
```

21.7 Clauses

A clause consists of a subject followed by a predicate. However, the subject need not be realized immediately before the predicate. For example, in “Alice promised Bob to lend him her car” there is an infinitive clause that consists of the predicate “to lend him her car” and the subject “Alice.” The sentence as a whole is another clause. In

our analysis, then, a clause is a subject followed by a verb phrase, with the possibility that the subject will be instantiated by something from the gap arguments:

```
(rule (clause ?infl ?x ?int-subj ?v ?gap1 ?gap3 :sem) ==>
  (subject ?agr ?x ?subj-slot ?int-subj ?gap1 ?gap2 ?subj-sem)
  (VP ?infl ?x ?subj-slot ?v ?gap2 ?gap3 ?pred-sem)
  (:test (subj-pred-agree ?agr ?infl)))
```

There are now two possibilities for subject. In the first case it has already been parsed, and we pick it up from the gap list. If that is so, then we also need to find the agreement feature of the subject. If the subject was a noun phrase, the agreement will be present in the gap list. If it was not, then the agreement is third-person singular. An example of this is "*That the Red Sox won* surprises me," where the italicized phrase is a non-NP subject. The fact that we need to use "surprises" and not "surprise" indicates that it is third-person singular. We will see that the code (- - + -) is used for this.

```
(rule (subject ?agree ?x ?subj-slot ext-subj
  (gap ?subj) (gap nil) t) ==>
  ;; Externally realized subject (the normal case for S)
  (:test (slot-constituent ?subj-slot ?subj ?x ?)
    (if (= ?subj (NP ?agr ?case ?x))
      (= ?agree ?agr)
      (= ?agree (- - + -)))) ;Non-NP subjects are 3sing
```

In the second case we just parse a noun phrase as the subject. Note that the fourth argument to subject is either ext-subj or int-subj depending on if the subject is realized internally or externally. This will be important when we cover sentences in the next section. In case it was not already clear, the second argument to both clause and subject is the metavariable representing the subject.

```
(rule (subject ?agr ?x (?role 1 (NP ?x)) int-subj ?gap ?gap ?sem)
  ==>
  (NP ?agr (common nom) ?wh ?x (gap nil) (gap nil) ?sem))
```

Finally, the rules for subject-predicate agreement say that only finite predicates need to agree with their subject:

```
(<- (subj-pred-agree ?agr (finite ?agr ?)))
(<- (subj-pred-agree ? ?infl) (atom ?infl))
```


21.8 Sentences

In the previous chapter we allowed only simple declarative sentences. The current grammar supports commands and four kinds of questions in addition to declarative sentences. It also supports *thematic fronting*: placing a nonsubject at the beginning of a sentence to emphasize its importance, as in “*Smith he says his name is*” or “*Murder, she wrote*” or “*In God we trust*.” In the last example it is a prepositional phrase, not a noun phrase, that occurs first. It is also possible to have a subject that is not a noun phrase: “*That the dog didn’t bark* puzzled Holmes.” To support all these possibilities, we introduce a new category, XP, which stands for any kind of phrase. A declarative sentence is then just an XP followed by a clause, where the subject of the clause may or may not turn out to be the XP:

```
(rule (S ?s :sem) ==>
  (:ex "Kim likes Lee" "Lee, I like _" "In god, we trust _"
    "Who likes Lee?" "Kim likes who?")
  (XP ?kind ?constituent ?wh ?x (gap nil) (gap nil) ?topic-sem)
  (clause (finite ? ?) ?x ? ?s (gap ?constituent) (gap nil) ?sem))
```

As it turns out, this rule also serves for two types of questions. The simplest kind of question has an interrogative noun phrase as its subject: “Who likes Lee?” or “What man likes Lee?” Another kind is the so-called *echo question*, which can be used only as a reply to another statement: if I tell you Kim likes Jerry Lewis, you could reasonably reply “Kim likes *who*?” Both these question types have the same structure as declarative sentences, and thus are handled by the same rule.

The following table lists some sentences that can be parsed by this rule, showing the XP and subject of each.

Sentence	XP	Subject
Kim likes Lee	Kim	Kim
Lee, Kim likes	Lee	Kim
In god, we trust	In god	we
That Kim likes Lee amazes	That Kim likes Lee	That Kim likes Lee
Who likes Lee?	Who	Who

The most common type of command has no subject at all: “Be quiet” or “Go to your room.” When the subject is missing, the meaning is that the command refers to *you*, the addressee of the command. The subject can also be mentioned explicitly, and it can be “you,” as in “You be quiet,” but it need not be: “Somebody shut the door” or “Everybody sing along.” We provide a rule only for commands with subject omitted, since it can be difficult to distinguish a command with a subject from a declarative sentence. Note that commands are always nonfinite.

```
(rule (S ?s :sem) ==>
  ;; Commands have implied second-person subject
  (:ex "Give the dog a bone.")
  (:sem (command ?s))
  (:sem (listener ?x))
  (clause nonfinite ?x ext-subj ?s
    (gap (NP ? ? ?x)) (gap nil) ?sem))
```

Another form of command starts with “let,” as in “Let me see what I can do” and “Let us all pray.” The second word is better considered as the object of “let” rather than the subject of the sentence, since the subject would have to be “I” or “we.” This kind of command can be handled with a lexical entry for “let” rather than with an additional rule.

We now consider questions. Questions that can be answered by yes or no have the subject and auxiliary verb inverted: “Did you see him?” or “Should I have been doing this?” The latter example shows that it is only the first auxiliary verb that comes before the subject. The category *aux-inv-S* is used to handle this case:

```
(rule (S ?s (yes-no ?s ?sem)) ==>
  (:ex "Does Kim like Lee?" "Is he a doctor?")
  (aux-inv-S nil ?s ?sem))
```

Questions that begin with a *wh*-phrase also have the auxiliary verb before the subject, as in “Who did you see?” or “Why should I have been doing this?” The first constituent can also be a prepositional phrase: “For whom am I doing this?” The following rule parses an *XP* that must have the *+wh* feature and then parses an *aux-inv-S* to arrive at a question:

```
(rule (S ?s :sem) ==>
  (:ex "Who does Kim like _?" "To whom did he give it _?"
    "What dog does Kim like _?")
  (XP ?slot ?constituent +wh ?x (gap nil) (gap nil) ?subj-sem)
  (aux-inv-S ?constituent ?s ?sem))
```

A question can also be signaled by rising intonation in what would otherwise be a declarative statement: “You want some?” Since we don’t have intonation information, we won’t include this kind of question.

The implementation for *aux-inv-S* is straightforward: parse an auxiliary and then a clause, pausing to look for modifiers in between. (So far, a “not” is the only modifier allowed in that position.)

```
(rule (aux-inv-S ?constituent ?v :sem) ==>
  (:ex "Does Kim like Lee?" (who) "would Kim have liked")
  (aux (finite ?agr ?tense) ?needs-infl ?v ?aux-sem)
  (modifiers post aux ? () ?v (gap nil) (gap nil) ?mod)
  (clause ?needs-infl ?x int-subj ?v (gap ?constituent) (gap nil)
    ?clause-sem))
```

There is one more case to consider. The verb “to be” is the most idiosyncratic in English. It is the only verb that has agreement differences for anything besides third-person singular. And it is also the only verb that can be used in an aux-*inv-S* without a main verb. An example of this is “Is he a doctor?” where “is” clearly is not an auxiliary, because there is no main verb that it could be auxiliary to. Other verb can not be used in this way: “*Seems he happy?” and “*Did they it?” are ungrammatical. The only possibility is “have,” as in “Have you any wool?” but this use is rare.

The following rule parses a verb, checks to see that it is a version of “be,” and then parses the subject and the modifiers for the verb.

```
(rule (aux-inv-S ?ext ?v :sem) ==>
  (:ex "Is he a doctor?")
  (verb ?be (finite ?agr ?) ((?role ?n ?xp) . ?slots) ?v ?sem)
  (:test (word ?be be))
  (subject ?agr ?x (?role ?n ?xp) int-subj
    (gap nil) (gap nil) ?subj-sem)
  (:sem (?role ?v ?x))
  (modifiers post verb ? ?slots ?v (gap ?ext) (gap nil) ?mod-sem))
```

21.9 XPs

All that remains in our grammar is the XP category. XPs are used in two ways: First, a phrase can be extraposed, as in “*In god we trust*,” where “in god” will be parsed as an XP and then placed on the gap list until it can be taken off as an adjunct to “trust.” Second, a phrase can be a complement, as in “He wants *to be a fireman*,” where the infinitive phrase is a complement of “wants.”

As it turns out, the amount of information that needs to appear in a gap list is slightly different from the information that appears in a complement slot. For example, one sense of the verb “want” has the following complement list:

```
((agt 1 (NP ?x)) (con 3 (VP infinitive ?x)))
```

This says that the first complement (the subject) is a noun phrase that serves as the agent of the wanting, and the second is an infinitive verb phrase that is the concept of

the wanting. The subject of this verb phrase is the same as the subject of the wanting, so in "She wants to go home," it is she who both wants and goes. (Contrast this to "He persuaded her to go home," where it is he that persuades, but she that goes.)

But when we put a noun phrase on a gap list, we need to include its number and case as well as the fact that it is an NP and its metavariable, but we don't need to include the fact that it is an agent. This difference means we have two choices: either we can merge the notions of slots and gap lists so that they use a common notation containing all the information that either can use, or we need some way of mapping between them. I made the second choice, on the grounds that each notation was complicated enough without bringing in additional information.

The relation slot-constituent maps between the slot notation used for complements and the constituent notation used in gap lists. There are eight types of complements, five of which can appear in gap lists: noun phrases, clauses, prepositional phrases, the word "it" (as in "it is raining"), and adverbial phrases. The three phrases that are allowed only as complements are verb phrases, particles (such as "up" in "look up the number"), and adjectives. Here is the mapping between the two notations. The *** indicates no mapping:

```
(<- (slot-constituent (?role ?n (NP ?x))
      (NP ?agr ?case ?x) ?x ?h))
(<- (slot-constituent (?role ?n (clause ?word ?infl))
      (clause ?word ?infl ?v) ?v ?h))
(<- (slot-constituent (?role ?n (PP ?prep ?np))
      (PP ?prep ?role ?np ?h) ?np ?h))
(<- (slot-constituent (?role ?n it) (it ? ? ?x) ?x ?))
(<- (slot-constituent (manner 3 (advp ?x)) (advp ?v) ? ?v))
(<- (slot-constituent (?role ?n (VP ?infl ?x)) *** ? ?))
(<- (slot-constituent (?role ?n (Adj ?x)) *** ?x ?))
(<- (slot-constituent (?role ?n (P ?particle)) *** ? ?))
```

We are now ready to define complement. It takes a slot description, maps it into a constituent, and then calls XP to parse that constituent:

```
(rule (complement ?cat ?info (?role ?n ?xp) ?h ?gap1 ?gap2 :sem)
      ==>
      ;; A complement is anything expected by a slot
      (:sem (?role ?h ?x))
      (:test (slot-constituent (?role ?n ?xp) ?constituent ?x ?h))
      (XP ?xp ?constituent ?wh ?x ?gap1 ?gap2 ?sem))
```

The category XP takes seven arguments. The first two are the slot we are trying to fill and the constituent we need to fill it. The third is used for any additional information, and the fourth is the metavariable for the phrase. The last three supply gap and semantic information.

Here are the first five XP categories:

```
(rule (XP (PP ?prep ?np) (PP ?prep ?role ?np ?h) ?wh ?np
      ?gap1 ?gap2 ?sem) ==>
      (PP ?prep ?role ?wh ?np ?h ?gap1 ?gap2 ?sem))

(rule (XP (NP ?x) (NP ?agr ?case ?x) ?wh ?x ?gap1 ?gap2 ?sem) ==>
      (NP ?agr ?case ?wh ?x ?gap1 ?gap2 ?sem))

(rule (XP it (it ? ? ?x) -wh ?x ?gap ?gap t) ==>
      (:word it))

(rule (XP (clause ?word ?infl) (clause ?word ?infl ?v) -wh ?v
      ?gap1 ?gap2 ?sem) ==>
      (:ex (he thinks) "that she is tall")
      (opt-word ?word)
      (clause ?infl ?x int-subj ?v ?gap1 ?gap2 ?sem))

(rule (XP (?role ?n (advp ?v)) (advp ?v) ?wh ?v ?gap1 ?gap2 ?sem)
      ==>
      (advp ?wh ?v ?gap1 ?gap2 ?sem))
```

The category `opt-word` parses a word, which may be optional. For example, one sense of “know” subcategorizes for a clause with an optional “that”: we can say either “I know that he’s here” or “I know he’s here.” The complement list for “know” thus contains the slot `(con 2 (clause (that) (finite ? ?)))`. If the “that” had been obligatory, it would not have parentheses around it.

```
(rule (opt-word ?word) ==> (:word ?word))
(rule (opt-word (?word)) ==> (:word ?word))
(rule (opt-word (?word)) ==>)
```

Finally, here are the three XPs that can not be extraposed:

```
(rule (XP (VP ?infl ?x) *** -wh ?v ?gap1 ?gap2 ?sem) ==>
      (:ex (he promised her) "to sleep")
      (VP ?infl ?x ?subj-slot ?v ?gap1 ?gap2 ?sem))

(rule (XP (Adj ?x) *** -wh ?x ?gap ?gap ?sem) ==>
      (Adj ?x ?sem))

(rule (XP (P ?particle) *** -wh ?x ?gap ?gap t) ==>
      (prep ?particle t))
```

21.10 Word Categories

Each word category has a rule that looks words up in the lexicon and assigns the right features. The relation `word` is used for all lexicon access. We will describe the most complicated word class, verb, and just list the others.

Verbs are complex because they often are *polysemous*—they have many meanings. In addition, each meaning can have several different complement lists. Thus, an entry for a verb in the lexicon will consist of the verb form, its inflection, and a list of senses, where each sense is a semantics followed by a list of possible complement lists. Here is the entry for the verb “sees,” indicating that it is a present-tense verb with three senses. The `understand` sense has two complement lists, which correspond to “He sees” and “He sees that you are right.” The `look` sense has one complement list corresponding to “He sees the picture,” and the `dating` sense, corresponding to “He sees her (only on Friday nights),” has the same complement list.

```
> (?- (word sees verb ?infl ?senses))
?INFL = (FINITE (- - + -) PRESENT)
?SENSES = ((UNDERSTAND ((AGT 1 (NP ?3)))
                    ((EXP 1 (NP ?4))
                     (CON 2 (CLAUSE (THAT) (FINITE ?5 ?6))))))
          (LOOK ((AGT 1 (NP ?7)) (OBJ 2 (NP ?8))))
          (DATING ((AGT 1 (NP ?9)) (OBJ 2 (NP ?10))))))
```

The category `verb` takes five arguments: the verb itself, its inflection, its complement list, its metavariable, and its semantics. The member relations are used to pick a sense from the list of senses and a complement list from the list of lists, and the semantics is built from semantic predicate for the chosen sense and the metavariable for the verb:

```
(rule (verb ?verb ?infl ?slots ?v :sem) ==>
  (:word ?verb)
  (:test (word ?verb verb ?infl ?senses)
    (member (?sem . ?subcats) ?senses)
    (member ?slots ?subcats)
    (tense-sem ?infl ?v ?tense-sem))
  (:sem ?tense-sem)
  (:sem (?sem ?v)))
```

It is difficult to know how to translate tense information into a semantic interpretation. Different applications will have different models of time and thus will want different interpretations. The relation `tense-sem` gives semantics for each tense. Here is a very simple definition of `tense-sem`:

```

(<- (tense-sem (finite ? ?tense) ?v (?tense ?v)))
(<- (tense-sem -ing ?v (progressive ?v)))
(<- (tense-sem -en ?v (past-participle ?v)))
(<- (tense-sem infinitive ?v t))
(<- (tense-sem nonfinite ?v t))
(<- (tense-sem passive ?v (passive ?v)))

```

Auxiliary verbs and modal verbs are listed separately:

```

(rule (aux ?infl ?needs-infl ?v ?tense-sem) ==>
  (:word ?aux)
  (:test (word ?aux aux ?infl ?needs-infl)
    (tense-sem ?infl ?v ?tense-sem)))

(rule (aux (finite ?agr ?tense) nonfinite ?v (?sem ?v)) ==>
  (:word ?modal)
  (:test (word ?modal modal ?sem ?tense)))

```

Nouns, pronouns, and names are also listed separately, although they have much in common. For pronouns we use quantifier *wh* or *pro*, depending on if it is a *wh*-pronoun or not.

```

(rule (noun ?agr ?slots ?x (?sem ?x)) ==>
  (:word ?noun)
  (:test (word ?noun noun ?agr ?slots ?sem)))

(rule (pronoun ?agr ?case ?wh ?x (?quant ?x (?sem ?x))) ==>
  (:word ?pro)
  (:test (word ?pro pronoun ?agr ?case ?wh ?sem)
    (if (= ?wh +wh) (= ?quant wh) (= ?quant pro))))

(rule (name ?agr ?name) ==>
  (:word ?name)
  (:test (word ?name name ?agr)))

```

Here are the rules for the remaining word classes:

```

(rule (adj ?x (?sem ?x)) ==>
  (:word ?adj)
  (:test (word ?adj adj ?sem)))

(rule (adj ?x ((nth ?n) ?x)) ==> (ordinal ?n))

(rule (art ?agr ?quant) ==>
  (:word ?art)
  (:test (word ?art art ?agr ?quant)))

```

```

(rule (prep ?prep t) ==>
  (:word ?prep)
  (:test (word ?prep prep)))

(rule (adverb ?wh ?x ?sem) ==>
  (:word ?adv)
  (:test (word ?adv adv ?wh ?pred)
    (if (= ?wh +wh)
      (= ?sem (wh ?y (?pred ?x ?y)))
      (= ?sem (?pred ?x)))))

(rule (cardinal ?n ?agr) ==>
  (:ex "five")
  (:word ?num)
  (:test (word ?num cardinal ?n ?agr)))

(rule (cardinal ?n ?agr) ==>
  (:ex "5")
  (:word ?n)
  (:test (numberp ?n)
    (if (= ?n 1)
      (= ?agr (- - + -)) ;3sing
      (= ?agr (- - - +))) ;3plur))

(rule (ordinal ?n) ==>
  (:ex "fifth")
  (:word ?num)
  (:test (word ?num ordinal ?n)))

```

21.11 The Lexicon

The lexicon itself consists of a large number of entries in the word relation, and it would certainly be possible to ask the lexicon writer to make a long list of word facts. But to make the lexicon easier to read and write, we adopt three useful tools. First, we introduce a system of abbreviations. Common expressions can be abbreviated with a symbol that will be expanded by word. Second, we provide the macros verb and noun to cover the two most complex word classes. Third, we provide a macro word that makes entries into a hash table. This is more efficient than compiling a word relation consisting of hundreds of Prolog clauses.

The implementation of these tools is left for the next section; here we show the actual lexicon, starting with the list of abbreviations.

The first set of abbreviations defines the agreement features. The obvious way to handle agreement is with two features, one for person and one for number. So first-person singular might be represented (1 sing). A problem arises when we want

to describe verbs. Every verb except “be” makes the distinction only between third-person singular and all the others. We don’t want to make five separate entries in the lexicon to represent all the others. One alternative is to have the agreement feature be a set of possible values, so all the others would be a single set of five values rather than five separate values. This makes a big difference in cutting down on backtracking. The problem with this approach is keeping track of when to intersect sets. Another approach is to make the agreement feature be a list of four binary features, one each for first-person singular, first-person plural, third-person singular, and third-person plural. Then “all the others” can be represented by the list that is negative in the third feature and unknown in all the others. There is no way to distinguish second-person singular from plural in this scheme, but English does not make that distinction. Here are the necessary abbreviations:

```
(abbrev 1sing      (+ - - -))
(abbrev 1plur     (- + - -))
(abbrev 3sing     (- - + -))
(abbrev 3plur     (- - - +))
(abbrev 2pers     (- - - -))
(abbrev ~3sing    (? ? - ?))
```

The next step is to provide abbreviations for some of the common verb complement lists:

```
(abbrev v/intrans ((agt 1 (NP ?)))
(abbrev v/trans   ((agt 1 (NP ?)) (obj 2 (NP ?)))
(abbrev v/ditrans ((agt 1 (NP ?)) (goal 2 (NP ?)) (obj 3 (NP ?)))
(abbrev v/trans2  ((agt 1 (NP ?)) (obj 2 (NP ?)) (goal 2 (PP to ?)))
(abbrev v/trans4  ((agt 1 (NP ?)) (obj 2 (NP ?)) (ben 2 (PP for ?)))
(abbrev v/it-null ((nil 1 it))
(abbrev v/opt-that ((exp 1 (NP ?)) (con 2 (clause (that) (finite ? ?))))
(abbrev v/subj-that ((con 1 (clause that (finite ? ?)) (exp 2 (NP ?)))
(abbrev v/it-that  ((nil 1 it) (exp 2 (NP ?))
                    (con 3 (clause that (finite ? ?))))
(abbrev v/inf      ((agt 1 (NP ?x)) (con 3 (VP infinitive ?x)))
(abbrev v/promise  ((agt 1 (NP ?x)) (goal 2 (NP ?y))
                    (con 3 (VP infinitive ?x)))
(abbrev v/persuade ((agt 1 (NP ?x)) (goal 2 (NP ?y))
                    (con 3 (VP infinitive ?y)))
(abbrev v/want     ((agt 1 (NP ?x)) (con 3 (VP infinitive ?x)))
(abbrev v/p-p-up   ((agt 1 (NP ?)) (pat 2 (NP ?)) (nil 3 (P up)))
(abbrev v/pp-for   ((agt 1 (NP ?)) (pat 2 (PP for ?)))
(abbrev v/pp-after ((agt 1 (NP ?)) (pat 2 (PP after ?)))
```

Verbs

The macro verb allows us to list verbs in the form below, where the spellings of each tense can be omitted if the verb is regular:

(verb (*base past-tense past-participle present-participle present-plural*)
(*semantics complement-list...*) ...)

For example, in the following list "ask" is regular, so only its base-form spelling is necessary. "Do," on the other hand, is irregular, so each form is spelled out. The haphazard list includes verbs that are either useful for examples or illustrate some unusual complement list.

(verb (ask) (query v/ditrans))
 (verb (delete) (delete v/trans))
 (verb (do did done doing does) (perform v/trans))
 (verb (eat ate eaten) (eat v/trans))
 (verb (give gave given giving) (give-1 v/trans2 v/ditrans)
 (donate v/trans v/intrans))
 (verb (go went gone going goes))
 (verb (have had had having has) (possess v/trans))
 (verb (know knew known) (know-that v/opt-that) (know-of v/trans))
 (verb (like) (like-1 v/trans))
 (verb (look) (look-up v/p-up) (search v/pp-for)
 (take-care v/pp-after) (look v/intrans))
 (verb (move moved moved moving moves)
 (self-propel v/intrans) (transfer v/trans2))
 (verb (persuade) (persuade v/persuade))
 (verb (promise) (promise v/promise))
 (verb (put put put putting))
 (verb (rain) (rain v/it-null))
 (verb (saw) (cut-with-saw v/trans v/intrans))
 (verb (see saw seen seeing) (understand v/intrans v/opt-that)
 (look v/trans) (dating v/trans))
 (verb (sleep slept) (sleep v/intrans))
 (verb (surprise) (surprise v/subj-that v/it-that))
 (verb (tell told) (tell v/persuade))
 (verb (trust) (trust v/trans ((agt 1 (NP ?)) (obj 2 (PP in ?))))))
 (verb (try tried tried trying tries) (attempt v/inf))
 (verb (visit) (visit v/trans))
 (verb (want) (desire v/want v/persuade))

Auxiliary Verbs

Auxiliary verbs are simple enough to be described directly with the word macro. Each entry lists the auxiliary itself, the tense it is used to construct, and the tense it must be followed by. The auxiliaries “have” and “do” are listed, along with “to,” which is used to construct infinitive clauses and thus can be treated as if it were an auxiliary.

```
(word have    aux nonfinite -en)
(word have    aux (finite ~3sing present) -en)
(word has     aux (finite 3sing present) -en)
(word had     aux (finite ? past) -en)
(word having  aux -ing -en)

(word do      aux (finite ~3sing present) nonfinite)
(word does    aux (finite 3sing present) nonfinite)
(word did     aux (finite ? past) nonfinite)

(word to      aux infinitive nonfinite)
```

The auxiliary “be” is special: in addition to its use as both an auxiliary and main verb, it also is used in passives and as the main verb in aux-inverted sentences. The function copula is used to keep track of all these uses. It will be defined in the next section, but you can see it takes two arguments, a list of senses for the main verb, and a list of entries for the auxiliary verb. The three senses correspond to the examples “He is a fool,” “He is a Republican,” and “He is in Indiana,” respectively.

```
(copula
  '((nil      ((nil 1 (NP ?x)) (nil 2 (Adj ?x))))
    (is-a     ((exp 1 (NP ?x)) (arg2 2 (NP ?y))))
    (is-loc   ((exp 1 (NP ?x)) (?prep 2 (PP ?prep ?))))))
  '((be      nonfinite -ing)
    (been    -en -ing)
    (being   -ing -en)
    (am      (finite 1sing present) -ing)
    (is      (finite 3sing present) -ing)
    (are     (finite 2pers present) -ing)
    (were    (finite (- - ? ?) past) -ing) ; 2nd sing or pl
    (was     (finite (? - ? -) past) -ing))) ; 1st or 3rd sing
```

Following are the modal auxiliary verbs. Again, it is difficult to specify semantics for them. The word “not” is also listed here; it is not an auxiliary, but it does modify them.

(word can modal able past)
 (word could modal able present)
 (word may modal possible past)
 (word might modal possible present)
 (word shall modal mandatory past)
 (word should modal mandatory present)
 (word will modal expected past)
 (word would modal expected present)
 (word must modal necessary present)
 (word not not)

Nouns

No attempt has been made to treat nouns seriously. We list enough nouns here to make some of the examples work. The first noun shows a complement list that is sufficient to parse "the destruction of the city by the enemy."

(noun destruction * destruction
 (pat (2) (PP of ?)) (agt (2) (PP by ?)))
 (noun beach)
 (noun bone)
 (noun box boxes)
 (noun city cities)
 (noun color)
 (noun cube)
 (noun doctor)
 (noun dog dogs)
 (noun enemy enemies)
 (noun file)
 (noun friend friends friend (friend-of (2) (PP of ?)))
 (noun furniture *)
 (noun hat)
 (noun man men)
 (noun saw)
 (noun woman women)

Pronouns

Here we list the nominative, objective, and genitive pronouns, followed by interrogative and relative pronouns. The only thing missing are reflexive pronouns, such as "myself."

(word I pronoun 1sing (common nom) -wh speaker)
 (word we pronoun 1plur (common nom) -wh speaker+other)
 (word you pronoun 2pers (common ?) -wh listener)
 (word he pronoun 3sing (common nom) -wh male)
 (word she pronoun 3sing (common nom) -wh female)
 (word it pronoun 3sing (common ?) -wh anything)
 (word they pronoun 3plur (common nom) -wh anything)

(word me pronoun 1sing (common obj) -wh speaker)
 (word us pronoun 1plur (common obj) -wh speaker+other)
 (word him pronoun 3sing (common obj) -wh male)
 (word her pronoun 3sing (common obj) -wh female)
 (word them pronoun 3plur (common obj) -wh anything)

(word my pronoun 1sing gen -wh speaker)
 (word our pronoun 1plur gen -wh speaker+other)
 (word your pronoun 2pers gen -wh listener)
 (word his pronoun 3sing gen -wh male)
 (word her pronoun 3sing gen -wh female)
 (word its pronoun 3sing gen -wh anything)
 (word their pronoun 3plur gen -wh anything)
 (word whose pronoun 3sing gen +wh anything)

(word who pronoun ? (common ?) +wh person)
 (word whom pronoun ? (common obj) +wh person)
 (word what pronoun ? (common ?) +wh thing)
 (word which pronoun ? (common ?) +wh thing)

(word who rel-pro ? person)
 (word which rel-pro ? thing)
 (word that rel-pro ? thing)
 (word whom rel-pro (common obj) person)

Names

The following names were convenient for one example or another:

(word God name 3sing) (word Lynn name 3sing)
 (word Jan name 3sing) (word Mary name 3sing)
 (word John name 3sing) (word NY name 3sing)
 (word Kim name 3sing) (word LA name 3sing)
 (word Lee name 3sing) (word SF name 3sing)

Adjectives

Here are a few adjectives:

(word big adj big) (word bad adj bad)
 (word old adj old) (word smart adj smart)
 (word green adj green) (word red adj red)
 (word tall adj tall) (word fun adj fun)

Adverbs

The adverbs covered here include interrogatives:

(word quickly adv -wh quickly)
 (word slowly adv -wh slowly)
 (word where adv +wh loc)
 (word when adv +wh time)
 (word why adv +wh reason)
 (word how adv +wh manner)

Articles

The common articles are listed here:

(word the art 3sing the)
 (word the art 3plur group)
 (word a art 3sing a)
 (word an art 3sing a)
 (word every art 3sing every)
 (word each art 3sing each)
 (word all art 3sing all)
 (word some art ? some)
 (word this art 3sing this)
 (word that art 3sing that)
 (word these art 3plur this)
 (word those art 3plur that)
 (word what art ? wh)
 (word which art ? wh)

Cardinal and Ordinal Numbers

We can take advantage of format's capabilities to fill up the lexicon. To go beyond 20, we would need a subgrammar of numbers.

```
;; This puts in numbers up to twenty, as if by
;; (word five cardinal 5 3plur)
;; (word fifth ordinal 5)

(dotimes (i 21)
  (add-word (read-from-string (format nil "~r" i))
            'cardinal i (if (= i 1) '3sing '3plur))
  (add-word (read-from-string (format nil "~:r" i)) 'ordinal i))
```

Prepositions

Here is a fairly complete list of prepositions:

```
(word above prep) (word about prep) (word around prep)
(word across prep) (word after prep) (word against prep)
(word along prep) (word at prep) (word away prep)
(word before prep) (word behind prep) (word below prep)
(word beyond prep) (word by prep) (word down prep)
(word for prep) (word from prep) (word in prep)
(word of prep) (word off prep) (word on prep)
(word out prep) (word over prep) (word past prep)
(word since prep) (word through prep) (word throughout prep)
(word till prep) (word to prep) (word under prep)
(word until prep) (word up prep) (word with prep)
(word without prep)
```

21.12 Supporting the Lexicon

This section describes the implementation of the macros `word`, `verb`, `noun`, and `abbrev`. Abbreviations are stored in a hash table. The macro `abbrev` and the functions `get-abbrev` and `clear-abbrevs` define the interface. We will see how to expand abbreviations later.

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

(defmacro abbrev (symbol definition)
  "Make symbol be an abbreviation for definition."
  `(setf (gethash ',symbol *abbrevs*) ',definition))

(defun clear-abbrevs () (clrhash *abbrevs*))
(defun get-abbrev (symbol) (gethash symbol *abbrevs*))
```

Words are also stored in a hash table. Currently, words are symbols, but it might be a better idea to use strings for words, since then we could maintain capitalization information. The macro `word` or the function `add-word` adds a word to the lexicon. When used as an index into the hash table, each word returns a list of entries, where the first element of each entry is the word's category, and the other elements depend on the category.

```
(defvar *words* (make-hash-table :size 500))

(defmacro word (word cat &rest info)
  "Put word, with category and subcat info, into lexicon."
  `(add-word ',word ',cat .,(mapcar #'kwote info)))

(defun add-word (word cat &rest info)
  "Put word, with category and other info, into lexicon."
  (push (cons cat (mapcar #'expand-abbrevs-and-variables info))
        (gethash word *words*)))
  word)

(defun kwote (x) (list 'quote x))
```

The function `expand-abbrevs-and-variables` expands abbreviations and substitutes variable structures for symbols beginning with `?`. This makes it easier to make a copy of the structure, which will be needed later.

```
(defun expand-abbrevs-and-variables (exp)
  "Replace all variables in exp with vars, and expand abbrevs."
  (let ((bindings nil))
    (labels
      ((expand (exp)
         (cond
          ((lookup exp bindings))
          ((eq exp '?) (?))
          ((variable-p exp)
           (let ((var (?)))
             (push (cons exp var) bindings)
             var)))
         ((consp exp)
          (reuse-cons (expand (first exp))
```



```

                (expand (rest exp))
                exp))
      (t (multiple-value-bind (expansion found?)
        (get-abbrev exp)
        (if found?
            (expand-abbrevs-and-variables expansion)
            exp))))))
    (expand exp))))

```

Now we can store words in the lexicon, but we need some way of getting them out. The function `word/n` takes a word (which must be instantiated to a symbol) and a category and optional additional information and finds the entries in the lexicon for that word that unify with the category and additional information. For each match, it calls the supplied continuation. This means that `word/n` is a replacement for a long list of word facts. There are three differences: `word/n` hashes, so it will be faster; it is incremental (you can add a word at a time without needing to recompile); and it can not be used when the word is unbound. (It is not difficult to change it to handle an unbound word using `maphash`, but there are better ways of addressing that problem.)

```

(defun word/n (word cat cont &rest info)
  "Retrieve a word from the lexicon."
  (unless (unbound-var-p (deref word))
    (let ((old-trail (fill-pointer *trail*)))
      (dolist (old-entry (gethash word *words*))
        (let ((entry (deref-copy old-entry)))
          (when (and (consp entry)
                     (unify! cat (first entry))
                     (unify! info (rest entry)))
            (funcall cont)))
          (undo-bindings! old-trail))))))

```

Note that `word/n` does not follow our convention of putting the continuation last. Therefore, we will need the following additional functions:

```

(defun word/2 (w cat cont) (word/n w cat cont))
(defun word/3 (w cat a cont) (word/n w cat cont a))
(defun word/4 (w cat a b cont) (word/n w cat cont a b))
(defun word/5 (w cat a b c cont) (word/n w cat cont a b c))
(defun word/6 (w cat a b c d cont) (word/n w cat cont a b c d))

```

We could create the whole lexicon with the macro `word`, but it is convenient to create specific macros for some classes. The macro `noun` is used to generate two entries, one for the singular and one for the plural. The arguments are the base noun, optionally followed by the plural (which defaults to the base plus "s"), the semantics (which

defaults to the base), and a list of complements. Mass nouns, like "furniture," have only one entry, and are marked by an asterisk where the plural would otherwise be.

```
(defmacro noun (base &rest args)
  "Add a noun and its plural to the lexicon."
  '(add-noun-form ',base ,@(mapcar #'kwote args)))

(defun add-noun-form (base &optional (plural (symbol base 's))
                    (sem base) &rest slots)
  (if (eq plural '*)
      (add-word base 'noun '? slots sem)
      (progn
        (add-word base 'noun '3sing slots sem)
        (add-word plural 'noun '3plur slots sem))))
```

Verbs are more complex. Each verb has seven entries: the base or nonfinite, the present tense singular and plural, the past tense, the past-participle, the present-participle, and the passive. The macro `verb` automatically generates all seven entries. Verbs that do not have all of them can be handled by individual calls to `word`. We automatically handle the spelling for the simple cases of adding "s," "ing," and "ed," and perhaps stripping a trailing vowel. More irregular spellings have to be specified explicitly. Here are three examples of the use of `verb`:

```
(verb (do did done doing does) (perform v/trans))
(verb (eat ate eaten) (eat v/trans))
(verb (trust) (trust v/trans ((agt 1 (NP ?)) (obj 2 (PP in ?))))
```

And here is the macro definition:

```
(defmacro verb ((base &rest forms) &body senses)
  "Enter a verb into the lexicon."
  '(add-verb ',senses ',base ,@(mapcar #'kwote (mklist forms))))

(defun add-verb (senses base &optional
                (past (symbol (strip-vowel base) 'ed))
                (past-part past)
                (pres-part (symbol (strip-vowel base) 'ing))
                (plural (symbol base 's)))
  "Enter a verb into the lexicon."
  (add-word base 'verb 'nonfinite senses)
  (add-word base 'verb '(finite ~3sing present) senses)
  (add-word past 'verb '(finite ? past) senses)
  (add-word past-part 'verb '-en senses)
  (add-word pres-part 'verb '-ing senses)
  (add-word plural 'verb '(finite 3sing present) senses)
  (add-word past-part 'verb 'passive
```

```
(mapcar #'passivize-sense
        (expand-abbrevs-and-variables senses))))
```

This uses a few auxiliary functions. First, `strip-vowel` removes a vowel if it is the last character of the given argument. The idea is that for a verb like “fire,” stripping the vowel yields “fir,” from which we can get “fired” and “firing” automatically.

```
(defun strip-vowel (word)
  "Strip off a trailing vowel from a string."
  (let* ((str (string word))
         (end (- (length str) 1)))
    (if (vowel-p (char str end))
        (subseq str 0 end)
        str)))

(defun vowel-p (char) (find char "aeiou" :test #'char-equal))
```

We also provide a function to generate automatically the passive sense with the proper complement list(s). The idea is that the subject slot of the active verb becomes an optional slot marked by the preposition “by,” and any slot that is marked with number 2 can be promoted to become the subject:

```
(defun passivize-sense (sense)
  ;; The first element of sense is the semantics; rest are slots
  (cons (first sense) (mapcar #'passivize-subcat (rest sense))))

(defun passivize-subcat (slots)
  "Return a list of passivizations of this subcat frame."
  ;; Whenever the 1 slot is of the form (?any 1 (NP ?)),
  ;; demote the 1 to a (3), and promote any 2 to a 1.
  (when (and (eql (slot-number (first slots)) 1)
             (starts-with (third (first slots)) 'NP))
    (let ((old-1 '(,(first (first slots)) (3) (PP by ?))))
      (loop for slot in slots
            when (eql (slot-number slot) 2)
            collect '(,(first slot) 1 ,(third slot))
                    ,@(remove slot (rest slots))
                    ,old-1))))

(defun slot-number (slot) (first-or-self (second slot)))
```

Finally, we provide a special function just to define the copula, “be.”

```
(defun copula (senses entries)
  "Copula entries are both aux and main verb."
  ;; They also are used in passive verb phrases and aux-inv-S
  (dolist (entry entries)
    (add-word (first entry) 'aux (second entry) (third entry))
    (add-word (first entry) 'verb (second entry) senses)
    (add-word (first entry) 'aux (second entry) 'passive)
    (add-word (first entry) 'be)))
```

The remaining functions are used for testing, debugging, and extending the grammar. First, we need functions to clear everything so that we can start over. These functions can be placed at the top of the lexicon and grammar files, respectively:

```
(defun clear-lexicon ()
  (clrhash *words*)
  (clear-abbrevs))

(defun clear-grammar ()
  (clear-examples)
  (clear-db))
```

Testing could be done with `run-examples`, but it is convenient to provide another interface, the macro `try` (and its corresponding function, `try-dcg`). Both macro and function can be invoked three ways. With no argument, all the examples stored by `:ex` are run. When the name of a category is given, all the examples for that category alone are run. Finally, the user can supply both the name of a category and a list of words to test whether those words can be parsed as that category. This option is only available for categories that are listed in the definition:

```
(defmacro try (&optional cat &rest words)
  "Tries to parse WORDS as a constituent of category CAT.
  With no words, runs all the :ex examples for category.
  With no cat, runs all the examples."
  `(try-dcg ',cat ',words))

(defun try-dcg (&optional cat words)
  "Tries to parse WORDS as a constituent of category CAT.
  With no words, runs all the :ex examples for category.
  With no cat, runs all the examples."
  (if (null words)
      (run-examples cat)
      (let ((args '((gap nil) (gap nil) ?sem ,words ())))
        (mapc #'test-unknown-word words)
        (top-level-prove
         (ecase cat
           (np '((np ? ? ?wh ?x ,@args))))
```

```

(vp '((vp ?infl ?x ?sl ?v ,@args)))
(pp '((pp ?prep ?role ?wh ?x ,@args)))
(xp '((xp ?slot ?constituent ?wh ?x ,@args)))
(s '((s ? ?sem ,words ()))
(rel-clause '((rel-clause ? ?x ?sem ,words ())))
(clause '((clause ?infl ?x ?int-subj ?v ?g1 ?g2
               ?sem ,words ())))))

(defun test-unknown-word (word)
  "Print a warning message if this is an unknown word."
  (unless (or (gethash word *words*) (numberp word))
    (warn "~&Unknown word: ~a" word)))

```

21.13 Other Primitives

To support the `:test` predicates made in various grammar rules we need definitions of the Prolog predicates `if`, `member`, `=`, `numberp`, and `atom`. They are repeated here:

```

(<- (if ?test ?then) (if ?then ?else (fail)))
(<- (if ?test ?then ?else) (call ?test) ! (call ?then))
(<- (if ?test ?then ?else) (call ?else))

(<- (member ?item (?item . ?rest)))
(<- (member ?item (?x . ?rest)) (member ?item ?rest))

(<- (= ?x ?x))

(defun numberp/1 (x cont)
  (when (numberp (deref x))
    (funcall cont)))

(defun atom/1 (x cont)
  (when (atom (deref x))
    (funcall cont)))

(defun call/1 (goal cont)
  "Try to prove goal by calling it."
  (deref goal)
  (apply (make-predicate (first goal)
                        (length (args goal)))
         (append (args goal) (list cont))))

```

21.14 Examples

Here are some examples of what the parser can handle. I have edited the output by changing variable names like ?168 to more readable names like ?J. The first two examples show that nested clauses are supported and that we can extract a constituent from a nested clause:

```
> (try S John promised Kim to persuade Lee to sleep)
?SEM = (AND (THE ?J (NAME JOHN ?J)) (AGT ?P ?J)
        (PAST ?P) (PROMISE ?P)
        (GOAL ?P ?K) (THE ?K (NAME KIM ?K))
        (CON ?P ?PER) (PERSUADE ?PER) (GOAL ?PER ?L)
        (THE ?L (NAME LEE ?L)) (CON ?PER ?S) (SLEEP ?S));
```

```
> (try S Who did John promise Kim to persuade to sleep)
?SEM = (AND (WH ?W (PERSON ?W)) (PAST ?P)
        (THE ?J (NAME JOHN ?J)) (AGT ?P ?J)
        (PROMISE ?P) (GOAL ?P ?K)
        (THE ?K (NAME KIM ?K)) (CON ?P ?PER)
        (PERSUADE ?PER) (GOAL ?PER ?W)
        (CON ?PER ?S) (SLEEP ?S));
```

In the next example, the “when” can be interpreted as asking about the time of any of the three events: the promising, the persuading, or the sleeping. The grammar finds all three.

```
> (try S When did John promise Kim to persuade Lee to sleep)
?SEM = (AND (WH ?W (TIME ?S ?W)) (PAST ?P)
        (THE ?J (NAME JOHN ?J)) (AGT ?P ?J)
        (PROMISE ?P) (GOAL ?P ?K)
        (THE ?K (NAME KIM ?K)) (CON ?P ?PER)
        (PERSUADE ?PER) (GOAL ?PER ?L)
        (THE ?L (NAME LEE ?L)) (CON ?PER ?S)
        (SLEEP ?S));
```

```
?SEM = (AND (WH ?W (TIME ?PER ?W)) (PAST ?P)
        (THE ?J (NAME JOHN ?J)) (AGT ?P ?J)
        (PROMISE ?P) (GOAL ?P ?K)
        (THE ?K (NAME KIM ?K)) (CON ?P ?PER)
        (PERSUADE ?PER) (GOAL ?PER ?L)
        (THE ?L (NAME LEE ?L)) (CON ?PER ?S)
        (SLEEP ?S));
```

```
?SEM = (AND (WH ?W (TIME ?P ?W)) (PAST ?P)
            (THE ?J (NAME JOHN ?J)) (AGT ?P ?J)
            (PROMISE ?P) (GOAL ?P ?K)
            (THE ?K (NAME KIM ?K)) (CON ?P ?PER)
            (PERSUADE ?PER) (GOAL ?PER ?L)
            (THE ?L (NAME LEE ?L)) (CON ?PER ?S)
            (SLEEP ?S)).
```

The next example shows auxiliary verbs and negation. It is ambiguous between an interpretation where Kim is searching for Lee and one where Kim is looking at something unspecified, on Lee's behalf.

```
> (try S Kim would not have been looking for Lee)
?SEM = (AND (THE ?K (NAME KIM ?K)) (AGT ?S ?K)
            (EXPECTED ?S) (NOT ?S) (PAST-PARTICIPLE ?S)
            (PROGRESSIVE ?S) (SEARCH ?S) (PAT ?S ?L)
            (PAT ?S ?L) (THE ?L (NAME LEE ?L)));

?SEM = (AND (THE ?K (NAME KIM ?K)) (AGT ?2 ?K)
            (EXPECTED ?2) (NOT ?2) (PAST-PARTICIPLE ?LOOK)
            (PROGRESSIVE ?LOOK) (LOOK ?LOOK) (FOR ?LOOK ?L)
            (THE ?L (NAME LEE ?L)));
```

The next two examples are unambiguous:

```
> (try s It should not surprise you that Kim does not like Lee)
?SEM = (AND (MANDATORY ?2) (NOT ?2) (SURPRISE ?2) (EXP ?2 ?YOU)
            (PRO ?YOU (LISTENER ?YOU)) (CON ?2 ?LIKE)
            (THE ?K (NAME KIM ?K)) (AGT ?LIKE ?K)
            (PRESENT ?LIKE) (NOT ?LIKE) (LIKE-1 ?LIKE)
            (OBJ ?LIKE ?L) (THE ?L (NAME LEE ?L)));

> (try s Kim did not want Lee to know that the man knew her)
?SEM = (AND (THE ?K (NAME KIM ?K)) (AGT ?W ?K) (PAST ?W)
            (NOT ?W) (DESIRE ?W) (GOAL ?W ?L)
            (THE ?L (NAME LEE ?L)) (CON ?W ?KN)
            (KNOW-THAT ?KN) (CON ?KN ?KN2)
            (THE ?M (MAN ?M)) (AGT ?KN2 ?M) (PAST ?KN2)
            (KNOW-OF ?KN2) (OBJ ?KN2 ?HER)
            (PRO ?HER (FEMALE ?HER))).
```

The final example appears to be unambiguous, but the parser finds four separate parses. The first is the obvious interpretation where the looking up is done quickly, and the second has quickly modifying the surprise. The last two interpretations are the same as the first two; they are artifacts of the search process. A disambiguation procedure should be equipped to weed out such duplicates.

```

> (try s That Kim looked her up quickly surprised me)
?SEM = (AND (THE ?K (NAME KIM ?K)) (AGT ?LU1 ?K) (PAST ?LU1)
          (LOOK-UP ?LU1) (PAT ?LU1 ?H) (PRO ?H (FEMALE ?H))
          (QUICKLY ?LU1) (CON ?S ?LU1) (PAST ?S) (SURPRISE ?S)
          (EXP ?S ?ME1) (PRO ?ME1 (SPEAKER ?ME1)));

?SEM = (AND (THE ?K (NAME KIM ?K)) (AGT ?LU2 ?K) (PAST ?LU2)
          (LOOK-UP ?LU2) (PAT ?LU2 ?H) (PRO ?H (FEMALE ?H))
          (CON ?S ?LU2) (QUICKLY ?S) (PAST ?S) (SURPRISE ?S)
          (EXP ?S ?ME2) (PRO ?ME2 (SPEAKER ?ME2)));

?SEM = (AND (THE ?K (NAME KIM ?K)) (AGT ?LU3 ?K) (PAST ?LU3)
          (LOOK-UP ?LU3) (PAT ?LU3 ?H) (PRO ?H (FEMALE ?H))
          (QUICKLY ?LU3) (CON ?S ?LU3) (PAST ?S) (SURPRISE ?S)
          (EXP ?S ?ME3) (PRO ?ME3 (SPEAKER ?ME3)));

?SEM = (AND (THE ?K (NAME KIM ?K)) (AGT ?LU4 ?K) (PAST ?LU4)
          (LOOK-UP ?LU4) (PAT ?LU4 ?H) (PRO ?H (FEMALE ?H))
          (CON ?S ?LU4) (QUICKLY ?S) (PAST ?S) (SURPRISE ?S)
          (EXP ?S ?ME4) (PRO ?ME4 (SPEAKER ?ME4)));

```

21.15 History and References

Chapter 20 provides some basic references on natural language. Here we will concentrate on references that provide:

1. A comprehensive grammar of English.
2. A complete implementation.

There are a few good textbooks that partially address both issues. Both Winograd (1983) and Allen (1987) do a good job of presenting the major grammatical features of English and discuss implementation techniques, but they do not provide actual code.

There are also a few textbooks that concentrate on the second issue. Ramsey and Barrett (1987) and Walker et al. (1990) provide chapter-length implementations at about the same level of detail as this chapter. Both are recommended. Pereira and Shieber 1987 and Gazdar and Mellish 1989 are book-length treatments, but because they cover a variety of parsing techniques rather than concentrating on one in depth, they are actually less comprehensive.

Several linguists have made serious attempts at addressing the first issue. The largest is the aptly named *A Comprehensive Grammar of Contemporary English* by Quirk, Greenbaum, Leech and Svartik (1985). More manageable (although hardly concise) is their abridged edition, *A Concise Grammar of Contemporary English*. Both editions contain a gold mine of examples and facts about the English language, but the authors

do not attempt to write rigorous rules. Harris (1982) and Huddleston (1984) offer less complete grammars with greater linguistic rigor.

Naomi Sager (1981) presents the most complete computerized grammar ever published. The grammar is separated into a simple, neat, context-free component and a rather baroque augmentation that manipulates features.

21.16 Exercises

- ?** **Exercise 21.1 [m]** Change the grammar to account better for *mass nouns*. The current grammar treats mass nouns by making them vague between singular and plural, which is incorrect. They should be treated separately, since there are determiners such as “much” that work only with mass nouns, and other determiners such as “these” that work only with plural count nouns.
- ?** **Exercise 21.2 [m]** Change the grammar to make a distinction between *attributive* and *predicative* adjectives. Most adjectives fall into both classes, but some can be used only attributively, as in “an *utter* fool” but not “*the fool is *utter*.” Other adjectives can only be used predicatively, as in “the woman was *loath* to admit it” but not “*a *loath* (to admit it) woman.”
- ?** **Exercise 21.3 [h]** Implement complement lists for adjectives, so that “loath” would take an obligatory infinitive complement, and “proud” would take an optional (PP of) complement. In connection to the previous exercise, note that it is rare if not impossible for attributive adjectives to take complements: “he is proud,” “he is proud of his country” and “a proud citizen” are all acceptable, but “*a proud of his country citizen” is not.
- ?** **Exercise 21.4 [m]** Add rules to *advp* to allow for adverbs to modify other adverbs, as in “extremely likely” or “very strongly.”
- ?** **Exercise 21.5 [h]** Allow adverbs to modify adjectives, as in “very good” or “really delicious.” The syntax will be easy, but it is harder to get a reasonable semantics. While you’re at it, make sure that you can handle adjectives with so-called *nonintersective* semantics. Some adjectives can be handled by intersective semantics: a red circle is something that is red and is a circle. But for other adjectives, this model does not work: a former senator is not something that is former and is a senator—a former senator is not a senator at all. Similarly, a toy elephant is not an elephant.

The semantics should be represented by something closer to ((toy elephant) ?x) rather than (and (toy ?x) (elephant ?x)).

? **Exercise 21.6 [m]** Write a function that notices punctuation instead of ignoring it. It should work something like this:

```
> (string->words "Who asked Lee, Kim and John?")
(WHO ASKED LEE |,| KIM AND JOHN |?|)
```

? **Exercise 21.7 [m]** Change the grammar to allow optional punctuation marks at the end of sentences and before relative clauses.

? **Exercise 21.8 [m]** Change the grammar to allow conjunction with more than two elements, using commas. Can these rules be generated automatically by conj-rule?

? **Exercise 21.9 [h]** Make a distinction between *restrictive* and *nonrestrictive* relative clauses. In "The truck *that has 4-wheel drive* costs \$5000," the italicized relative clause is restrictive. It serves to identify the truck and thus would be part of the quantifier's restriction. The complete sentence might be interpreted as:

```
(and (the ?x (and (truck ?x) (4-wheel-drive ?x)))
      (costs ?x $5000))
```

Contrast this to "The truck, which has 4-wheel drive, costs \$5000." Here the relative clause is nonrestrictive and thus belongs outside the quantifier's restriction:

```
(and (the ?x (truck ?x))
      (4-wheel-drive ?x) (costs ?x $5000))
```

CHAPTER 22

Scheme: An Uncommon Lisp

The best laid schemes o' mice an' men

—Robert Burns (1759–1796)

This chapter presents the Scheme dialect of Lisp and an interpreter for it. While it is not likely that you would use this interpreter for any serious programming, understanding how the interpreter works can give you a better appreciation of how Lisp works, and thus make you a better programmer. A Scheme interpreter is used instead of a Common Lisp one because Scheme is simpler, and also because Scheme is an important language that is worth knowing about.

Scheme is the only dialect of Lisp besides Common Lisp that is currently flourishing. Where Common Lisp tries to standardize all the important features that are in current use by Lisp programmers, Scheme tries to give a minimal set of very powerful features that can be used to implement the others. It is interesting that among all the programming languages in the world, Scheme is one of the smallest, while Common Lisp is one of the largest. The Scheme manual is only 45 pages (only 38 if you omit the example, bibliography, and index), while *Common Lisp the Language*, 2d edition, is 1029 pages. Here is a partial list of the ways Scheme is simpler than Common Lisp:

1. Scheme has fewer built-in functions and special forms.
2. Scheme has no special variables, only lexical variables.
3. Scheme uses the same name space for functions and variables (and everything else).
4. Scheme evaluates the function part of a function call in exactly the same way as the arguments.
5. Scheme functions can not have optional and keyword parameters. However, they can have the equivalent of a `&rest` parameter.
6. Scheme has no `block`, `return`, `go`, or `throw`; a single function (`call/cc`) replaces all of these (and does much more).
7. Scheme has no packages. Lexical variables can be used to implement package-like structures.
8. Scheme, as a standard, has no macros, although most implementations provide macros as an extension.
9. Scheme has no special forms for looping; instead it asks the user to use recursion and promises to implement the recursion efficiently.

The five main special forms in Scheme are `quote` and `if`, which are just as in Common Lisp; `begin` and `set!`, which are just different spellings for `progn` and `setq`; and `lambda`, which is as in Common Lisp, except that it doesn't require a `#'` before it. In addition, Scheme allows variables, constants (numbers, strings, and characters), and function calls. The function call is different because the function itself is evaluated in the same way as the arguments. In Common Lisp, `(f x)` means to look up the function binding of `f` and apply that to the value of `x`. In Scheme, `(f x)` means to evaluate `f` (in this case by looking up the value of the variable `f`), evaluate `x` (by looking up the value of the variable in exactly the same way) and then apply the function to the argument. Any expression can be in the function position, and it is evaluated just like the arguments. Another difference is that Scheme uses `#t` and `#f` for true and false, instead of `t` and `nil`. The empty list is denoted by `()`, and it is distinct from the false value, `#f`. There are also minor lexical differences in the conventions for complex numbers and numbers in different bases, but these can be ignored for all the programs in this book. Also, in Scheme a single macro, `define`, serves to define both variables and functions.

Scheme	Common Lisp
<i>var</i>	<i>var</i>
<i>constant</i>	<i>constant</i>
(quote <i>x</i>) or 'x	(quote <i>x</i>) or 'x
(begin <i>x</i> ...)	(progn <i>x</i> ...)
(set! <i>var x</i>)	(setq <i>var x</i>)
(if <i>p a b</i>)	(if <i>p a b</i>)
(lambda <i>parms x</i> ...)	#'(lambda <i>parms x</i> ...)
(<i>fn arg</i> ...)	(<i>fn arg</i> ...) or (funcall <i>fn arg</i> ...)
#t	t
#f	nil
()	nil
(define <i>var exp</i>)	(defparameter <i>var exp</i>)
(define (<i>fn parm</i> ...) <i>body</i>)	(defun <i>fn (parm</i> ...) <i>body</i>)

? **Exercise 22.1 [s]** What does the following expression evaluate to in Scheme? How many errors does it have as a Common Lisp expression?

```
((if (= (+ 2 2) 4)
      (lambda (x y) (+ (* x y) 12))
      cons)
 5
 6)
```

A great many functions, such as `car`, `cdr`, `cons`, `append`, `+`, `*`, and `list` are the same (or nearly the same) in both dialects. However, Scheme has some spelling conventions that are different from Common Lisp. Most Scheme mutators, like `set!`, end in '!'. Common Lisp has no consistent convention for this; some mutators start with `n` (`nreverse`, `nsubst`, `nintersection`) while others have idiosyncratic names (`delete` versus `remove`). Scheme would use consistent names—`reverse!` and `remove!`—if these functions were defined at all (they are not defined in the standard). Most Scheme predicates end in '?', not 'p'. This makes predicates more obvious and eliminates the complicated conventions for adding a hyphen before the `p`.¹ The only problem with this convention is in spoken language: is `equal?` pronounced “equal-question-mark” or “equal-q” or perhaps `equal`, with rising intonation? This would make Scheme a tone language, like Chinese.

¹One writes `numberp` because there is no hyphen in `number` but `random-state-p` because there is a hyphen in `random-state`. However, `defstruct` concatenates `-p` in all its predicates, regardless of the presence of a hyphen in the structure's name.

In Scheme, it is an error to apply `car` or `cdr` to the empty list. Despite the fact that Scheme has `cons`, it calls the result a `pair` rather than a `cons cell`, so the predicate is `pair?`, not `consp`.

Scheme recognizes not all lambda expressions will be “functions” according to the mathematical definition of function, and so it uses the term “procedure” instead. Here is a partial list of correspondences between the two dialects:

Scheme Procedure	Common Lisp Function
<code>char-ready?</code>	<code>listen</code>
<code>char?</code>	<code>characterp</code>
<code>eq?</code>	<code>eq</code>
<code>equal?</code>	<code>equal</code>
<code>equiv?</code>	<code>eql</code>
<code>even?</code>	<code>evenp</code>
<code>for-each</code>	<code>mapc</code>
<code>integer?</code>	<code>integerp</code>
<code>list->string</code>	<code>coerce</code>
<code>list->vector</code>	<code>coerce</code>
<code>list-ref</code>	<code>nth</code>
<code>list-tail</code>	<code>nthcdr</code>
<code>map</code>	<code>mapcar</code>
<code>negative?</code>	<code>minusp</code>
<code>pair?</code>	<code>consp</code>
<code>procedure?</code>	<code>functionp</code>
<code>set!</code>	<code>setq</code>
<code>set-car!</code>	<code>replaca</code>
<code>vector-set!</code>	<code>setf</code>
<code>string-set!</code>	<code>setf</code>

22.1 A Scheme Interpreter

As we have seen, an interpreter takes a program (or expression) as input and returns the value computed by that program. The Lisp function `eval` is thus an interpreter, and that is essentially the function we are trying to write in this section. We have to be careful, however, in that it is possible to confuse the notions of interpreter and compiler. A compiler takes a program as input and produces as output a translation of that program into some other language—usually a language that can be directly (or more easily) executed on some machine. So it is also possible to write `eval` by compiling the argument and then interpreting the resulting machine-level program. Most modern Lisp systems support both possibilities, although some only interpret

code directly, and others compile all code before executing it. To make the distinction clear, we will not write a function called `eval`. Instead, we will write versions of two functions: `interp`, a Scheme interpreter, and, in the next chapter, `comp`, a Scheme compiler.

An interpreter that handles the Scheme primitives is easy to write. In the interpreter `interp`, the main conditional has eight cases, corresponding to the five special forms, symbols, other atoms, and procedure applications (otherwise known as function calls). For the moment we will stick with `t` and `nil` instead of `#t` and `#f`. After developing a simple interpreter, we will add support for macros, then develop a tail-recursive interpreter, and finally a continuation-passing interpreter. (These terms will be defined when the time comes.). The glossary for `interp` is in figure 22.1.

	Top-Level Functions
<code>scheme</code>	A Scheme read- <code>interp</code> -print loop.
<code>interp</code>	Interpret (evaluate) an expression in an environment.
<code>def-scheme-macro</code>	Define a Scheme macro.
	Special Variables
<code>*scheme-procs*</code>	Some procedures to store in the global environment.
	Auxiliary Functions
<code>set-var!</code>	Set a variable to a value.
<code>get-var</code>	Get the value of a variable in an environment.
<code>set-global-var!</code>	Set a global variable to a value.
<code>get-global-var</code>	Get the value of a variable from the global environment.
<code>extend-env</code>	Add some variables and values to an environment.
<code>init-scheme-interp</code>	Initialize some global variables.
<code>init-scheme-proc</code>	Define a primitive Scheme procedure.
<code>scheme-macro</code>	Retrieve the Scheme macro for a symbol.
<code>scheme-macro-expand</code>	Macro-expand a Scheme expression.
<code>maybe-add</code>	Add an element to the front of a non-singleton list.
<code>print-proc</code>	Print a procedure.
	Data Type (tail-recursive version only)
<code>proc</code>	A Scheme procedure.
	Functions (continuation version only)
<code>interp-begin</code>	Interpret a <code>begin</code> expression.
<code>interp-call</code>	Interpret a function application.
<code>map-interp</code>	Map <code>interp</code> over a list.
<code>call/cc</code>	call with current continuation.
	Previously Defined Functions
<code>last1</code>	Select the last element of a list.
<code>length=1</code>	Is this a list of length 1?

Figure 22.1: Glossary for the Scheme Interpreter

The simple interpreter has eight cases to worry about: (1) If the expression is a symbol, look up its value in the environment. (2) If it is an atom that is not a symbol (such as a number), just return it. Otherwise, the expression must be a list. (3) If it starts with `quote`, return the quoted expression. (4) If it starts with `begin`, interpret each subexpression, and return the last one. (5) If it starts with `set!`, interpret the value and then set the variable to that value. (6) If it starts with `if`, then interpret the conditional, and depending on if it is true or not, interpret the then-part or the else-part. (7) If it starts with `lambda`, build a new procedure—a closure over the current environment. (8) Otherwise, it must be a procedure application. Interpret the procedure and all the arguments, and apply the procedure value to the argument values.

```
(defun interp (x &optional env)
  "Interpret (evaluate) the expression x in the environment env."
  (cond
    ((symbolp x) (get-var x env))
    ((atom x) x)
    ((case (first x)
      (QUOTE (second x))
      (BEGIN (last1 (mapcar #'(lambda (y) (interp y env))
                          (rest x))))
      (SET! (set-var! (second x) (interp (third x) env) env))
      (IF (if (interp (second x) env)
              (interp (third x) env)
              (interp (fourth x) env)))
      (LAMBDA (let ((parms (second x))
                    (code (maybe-add 'begin (rest2 x))))
                #'(lambda (&rest args)
                    (interp code (extend-env parms args env))))))
    (t ;; a procedure application
      (apply (interp (first x) env)
              (mapcar #'(lambda (v) (interp v env))
                      (rest x))))))
```

An environment is represented as an association list of variable/value pairs, except for the global environment, which is represented by values on the `global-val` property of symbols. It would be simpler to represent the global environment in the same way as local environments, but it is more efficient to use property lists than one big global a-list. Furthermore, the global environment is distinct in that every symbol is implicitly defined in the global environment, while local environments only contain variables that are explicitly mentioned (in a lambda expression).

As an example, suppose we interpret the function call `(f 1 2 3)`, and that the functions `f` has been defined by the Scheme expression:

```
(set! f (lambda (a b c) (+ a (g b c))))
```

Then we will interpret `(f 1 2 3)` by interpreting the body of `f` with the environment:

```
((a 1) (b 2) (c 3))
```

Scheme procedures are implemented as Common Lisp functions, and in fact all the Scheme data types are implemented by the corresponding Common Lisp types. I include the function `init-scheme-interp` to initialize a few global values and repeat the definitions of `last1` and `length=1`:

```
(defun set-var! (var val env)
  "Set a variable to a value, in the given or global environment."
  (if (assoc var env)
      (setf (second (assoc var env)) val)
      (set-global-var! var val))
  val)

(defun get-var (var env)
  "Get the value of a variable, from the given or global environment."
  (if (assoc var env)
      (second (assoc var env))
      (get-global-var var)))

(defun set-global-var! (var val)
  (setf (get var 'global-val) val))

(defun get-global-var (var)
  (let* ((default "unbound")
         (val (get var 'global-val default)))
    (if (eq val default)
        (error "Unbound scheme variable: ~a" var)
        val)))

(defun extend-env (vars vals env)
  "Add some variables and values to an environment."
  (nconc (mapcar #'list vars vals) env))

(defparameter *scheme-procs*
  '(+ - * / = < > <= >= cons car cdr not append list read member
    (null? null) (eq? eq) (equal? equal) (eql? eql)
    (write prinl) (display princ) (newline terpri)))
```

```

(defun init-scheme-interp ()
  "Initialize the scheme interpreter with some global variables."
  ;; Define Scheme procedures as CL functions:
  (mapc #'init-scheme-proc *scheme-procs*)
  ;; Define the Boolean 'constants'. Unfortunately, this won't
  ;; stop someone from saying: (set! t nil)
  (set-global-var! t t)
  (set-global-var! nil nil))

(defun init-scheme-proc (f)
  "Define a Scheme procedure as a corresponding CL function."
  (if (listp f)
      (set-global-var! (first f) (symbol-function (second f)))
      (set-global-var! f (symbol-function f))))

(defun maybe-add (op exps &optional if-nil)
  "For example, (maybe-add 'and exps t) returns
  t if exps is nil, exps if there is only one,
  and (and exp1 exp2...) if there are several exps."
  (cond ((null exps) if-nil)
        ((length=1 exps) (first exps))
        (t (cons op exps))))

(defun length=1 (x)
  "Is x a list of length 1?"
  (and (consp x) (null (cdr x))))

(defun last1 (list)
  "Return the last element (not last cons cell) of list"
  (first (last list)))

```

To test the interpreter, we add a simple read-eval-print loop:

```

(defun scheme ()
  "A Scheme read-eval-print loop (using interp)"
  (init-scheme-interp)
  (loop (format t "~&==> ")
        (print (interp (read) nil))))

```

And now we're ready to try out the interpreter. Note the Common Lisp prompt is ">," while the Scheme prompt is "==>."

```

> (scheme)
==> (+ 2 2)
4
==> ((if (= 1 2) * +) 3 4)
7

```

```
=> ((if (= 1 1) * +) 3 4)
12

=> (set! fact (lambda (n)
               (if (= n 0) 1
                   (* n (fact (- n 1))))))
#<DTP-LEXICAL-CLOSURE 36722615>

=> (fact 5)
120

=> (set! table (lambda (f start end)
                (if (<= start end)
                    (begin
                      (write (list start (f start)))
                      (newline)
                      (table f (+ start 1) end))))))
#<DTP-LEXICAL-CLOSURE 41072172>

=> (table fact 1 10)
(1 1)
(2 2)
(3 6)
(4 24)
(5 120)
(6 720)
(7 5040)
(8 40320)
(9 362880)
(10 3628800)
NIL

=> (table (lambda (x) (* x x x)) 5 10)
(5 125)
(6 216)
(7 343)
(8 512)
(9 729)
(10 1000)
NIL

=> [ABORT]
```

22.2 Syntactic Extension with Macros

Scheme has a number of other special forms that were not listed above. Actually, Scheme uses the term “syntax” where we have been using “special form.” The remaining syntax can be defined as “derived expressions” in terms of the five primitives. The Scheme standard does not recognize a concept of macros, but it is clear that a “derived expression” is like a macro, and we will implement them using macros. The following forms are used (nearly) identically in Scheme and Common Lisp:

```
let  let*  and  or  do  cond  case
```

One difference is that Scheme is less lenient as to what counts as a binding in `let`, `let*` and `do`. Every binding must be *(var init)*; just *(var)* or *var* is not allowed. In `do`, a binding can be either *(var init step)* or *(var init)*. Notice there is no `do*`. The other difference is in `case` and `cond`. Where Common Lisp uses the symbol `t` or `otherwise` to mark the final case, Scheme uses `else`. The final three syntactic extensions are unique to Scheme:

```
(define var val)    or    (define (proc-name arg...) body...)
(delay expression)
(letrec ((var init)...) body...)
```

`define` is a combination of `defun` and `defparameter`. In its first form, it assigns a value to a variable. Since there are no special variables in Scheme, this is no different than using `set!`. (There is a difference when the `define` is nested inside another definition, but that is not yet considered.) In the second form, it defines a function. `delay` is used to delay evaluation, as described in section 9.3, page 281. `letrec` is similar to `let`. The difference is that all the *init* forms are evaluated in an environment that includes all the *vars*. Thus, `letrec` can be used to define local recursive functions, just as `labels` does in Common Lisp.

The first step in implementing these syntactic extensions is to change `interp` to allow macros. Only one clause has to be added, but we’ll repeat the whole definition:

```
(defun interp (x &optional env)
  "Interpret (evaluate) the expression x in the environment env.
  This version handles macros."
  (cond
    ((symbolp x) (get-var x env))
    ((atom x) x)
    ((scheme-macro (first x)) ;***
     (interp (scheme-macro-expand x) env)) ;***
    ((case (first x)
         (QUOTE (second x))
```

```

(BEGIN (last1 (mapcar #'(lambda (y) (interp y env))
                (rest x))))
(SET! (set-var! (second x) (interp (third x) env) env))
(IF (if (interp (second x) env)
        (interp (third x) env)
        (interp (fourth x) env)))
(LAMBDA (let ((parms (second x))
              (code (maybe-add 'begin (rest2 x))))
          #'(lambda (&rest args)
              (interp code (extend-env parms args) env))))))
(t ;; a procedure application
  (apply (interp (first x) env)
          (mapcar #'(lambda (v) (interp v env))
                  (rest x))))))

```

Now we provide a mechanism for defining macros. The macro definitions can be in any convenient language; the easiest choices are Scheme itself or Common Lisp. I have chosen the latter. This makes it clear that macros are not part of Scheme itself but rather are used to implement Scheme. If we wanted to offer the macro facility to the Scheme programmer, we would make the other choice. (But then we would be sure to add the backquote notation, which is so useful in writing macros.) `def-scheme-macro` (which happens to be a macro itself) provides a way of adding new Scheme macros. It does that by storing a Common Lisp function on the `scheme-macro` property of a symbol. This function, when given a list of arguments, returns the code that the macro call should expand into. The function `scheme-macro` tests if a symbol has a macro attached to it, and `scheme-macro-expand` does the actual macro-expansion:

```

(defun scheme-macro (symbol)
  (and (symbolp symbol) (get symbol 'scheme-macro)))

(defmacro def-scheme-macro (name parmlist &body body)
  "Define a Scheme macro."
  '(setf (get ',name 'scheme-macro)
        #'(lambda ,parmlist .,body)))

(defun scheme-macro-expand (x)
  "Macro-expand this Scheme expression."
  (if (and (listp x) (scheme-macro (first x)))
      (scheme-macro-expand
       (apply (scheme-macro (first x)) (rest x)))
      x))

```

Here are the definitions of nine important macros in Scheme:

```
(def-scheme-macro let (bindings &rest body)
  '((lambda ,(mapcar #'first bindings) . ,body)
    .,(mapcar #'second bindings)))

(def-scheme-macro let* (bindings &rest body)
  (if (null bindings)
      '(begin .,body)
      '(let (,(first bindings))
        (let* ,(rest bindings) . ,body))))

(def-scheme-macro and (&rest args)
  (cond ((null args) 'T)
        ((length=1 args) (first args))
        (t '(if ,(first args)
                 (and . ,(rest args))))))

(def-scheme-macro or (&rest args)
  (cond ((null args) 'nil)
        ((length=1 args) (first args))
        (t (let ((var (gensym)))
              '(let ((,var ,(first args))
                    (if ,var ,var (or . ,(rest args))))))))

(def-scheme-macro cond (&rest clauses)
  (cond ((null clauses) nil)
        ((length=1 (first clauses))
         '(or ,(first clauses) (cond .,(rest clauses))))
        ((starts-with (first clauses) 'else)
         '(begin .,(rest (first clauses))))
        (t '(if ,(first (first clauses))
                 (begin .,(rest (first clauses)))
                 (cond .,(rest clauses))))))

(def-scheme-macro case (key &rest clauses)
  (let ((key-val (gensym "KEY")))
    '(let ((,key-val ,key))
      (cond ,@(mapcar
                #'(lambda (clause)
                    (if (starts-with clause 'else)
                        clause
                        '((member ,key-val ',(first clause))
                          .,(rest clause))))
                clauses))))))

(def-scheme-macro define (name &rest body)
  (if (atom name)
      '(begin (set! ,name . ,body) ',name)
      '(define ,(first name)
        (lambda ,(rest name) . ,body)))
```

```
(def-scheme-macro delay (computation)
  '(lambda () ,computation))

(def-scheme-macro letrec (bindings &rest body)
  '(let ,(mapcar #'(lambda (v) (list (first v) nil)) bindings)
    ,@(mapcar #'(lambda (v) '(set! .,v)) bindings)
    .,body))
```

We can test out the macro facility:

```
> (scheme-macro-expand '(and p q)) => (IF P (AND Q))
> (scheme-macro-expand '(and q)) => Q
> (scheme-macro-expand '(let ((x 1) (y 2)) (+ x y))) =>
((LAMBDA (X Y) (+ X Y)) 1 2)
> (scheme-macro-expand
  '(letrec
    ((even? (lambda (x) (or (= x 0) (odd? (- x 1)))))
    (odd? (lambda (x) (even? (- x 1)))))
    (even? z))) =>
(LET ((EVEN? NIL)
      (ODD? NIL))
  (SET! EVEN? (LAMBDA (X) (OR (= X 0) (ODD? (- X 1)))))
  (SET! ODD? (LAMBDA (X) (EVEN? (- X 1))))
  (EVEN? Z))

> (scheme)
==> (define (reverse l)
      (if (null? l) nil
          (append (reverse (cdr l)) (list (car l)))))
REVERSE
==> (reverse '(a b c d))
(D C B A)
==> (let* ((x 5) (y (+ x x)))
      (if (or (= x 0) (and (< 0 y) (< y 20)))
          (list x y)
          (+ y x)))
(5 10)
```

The macro `define` is just like `set!`, except that it returns the symbol rather than the value assigned to the symbol. In addition, `define` provides an optional syntax for defining functions—it serves the purposes of both `defun` and `defvar`. The syntax `(define (fn . args) . body)` is an abbreviation for `(define fn (lambda args . body))`.

In addition, Scheme provides a notation where `define` can be used inside a function definition in a way that makes it work like `let` rather than `set!`.

The advantage of the macro-based approach to special forms is that we don't have to change the interpreter to add new special forms. The interpreter remains simple, even while the language grows. This also holds for the compiler, as we see in the next section.

22.3 A Properly Tail-Recursive Interpreter

Unfortunately, the interpreter presented above can not lay claim to the name Scheme, because a true Scheme must be properly tail-recursive. Our interpreter is tail-recursive only when run in a Common Lisp that is tail-recursive. To see the problem, consider the following Scheme procedure:

```
(define (traverse lyst)
  (if lyst (traverse (cdr lyst))))
```

Trace the function `interp` and execute `(interp '(traverse '(a b c d)))`. The nested calls to `interp` go 16 levels deep. In general, the level of nesting is 4 plus 3 times the length of the list. Each call to `interp` requires Common Lisp to allocate some storage on the stack, so for very long lists, we will eventually run out of storage. To earn the name Scheme, a language must guarantee that such a program does not run out of storage.

The problem, in this example, lies in two places. Everytime we interpret an `if` form or a procedure call, we descend another recursive level into `interp`. But that extra level is not necessary. Consider the `if` form. It is certainly necessary to call `interp` recursively to decide if the test is true or not. For the sake of argument, let's say the test is true. Then we call `interp` again on the *then* part. This recursive call will return a value, which will then be immediately returned as the value of the original call as well.

The alternative is to replace the recursive call to `interp` with a renaming of variables, followed by a `goto` statement. That is, instead of calling `interp` and thereby binding a new instance of the variable `x` to the *then* part, we just assign the *then* part to `x`, and branch to the top of the `interp` routine. This works because we know we have no more use for the old value of `x`. A similar technique is used to eliminate the recursive call for the last expression in a `begin` form. (Many programmers have been taught the "structured programming" party line that `goto` statements are harmful. In this case, the `goto` is necessary to implement a low-level feature efficiently.)


```

(t      ;; a procedure application
  (let ((proc (interp (first x) env))
        (args (mapcar #'(lambda (v) (interp v env))
                       (rest x))))
    (if (proc-p proc)
        ;; Execute procedure with rename+goto
        (progn
         (setf x (proc-code proc))
         (setf env (extend-env (proc-parms proc) args
                              (proc-env proc)))

         (GO :INTERP))
        ;; else apply primitive procedure
        (apply proc args))))))

(defun print-proc (proc &optional (stream *standard-output*) depth)
  (declare (ignore depth))
  (format stream "~a" (or (proc-name proc) '???)))

```

By tracing the tail-recursive version of `interp`, you can see that calls to `traverse` descend only three recursive levels of `interp`, regardless of the length of the list traversed.

Note that we are not claiming that this interpreter allocates no storage when it makes tail-recursive calls. Indeed, it wastes quite a bit of storage in evaluating arguments and building environments. The claim is that since the storage is allocated on the heap rather than on the stack, it can be reclaimed by the garbage collector. So even if `traverse` is applied to an infinitely long list (i.e., a circular list), the interpreter will never run out of space—it will always be able to garbage-collect and continue.

There are many improvements that could be made to this interpreter, but effort is better spent in improving a compiler rather than an interpreter. The next chapter does just that.

22.4 Throw, Catch, and Call/cc

Tail-recursion is crucial to Scheme. The idea is that when the language is guaranteed to optimize tail-recursive calls, then there is no need for special forms to do iteration. All loops can be written using recursion, without any worry of overflowing the runtime stack. This helps keep the language simple and rules out the `goto` statement, the scourge of the structured programming movement. However, there are cases where some kind of nonlocal exit is the best alternative. Suppose that some unexpected event happens deep inside your program. The best action is to print an error message and pop back up to the top level of your program. This could be done trivially with a `goto`-like statement. Without it, every function along the calling path would have to

be altered to accept either a valid result or an indication of the exceptional condition, which just gets passed up to the next level.

In Common Lisp, the functions `throw` and `catch` are provided for this kind of nonlocal exit. Scott Zimmerman, the perennial world Frisbee champion, is also a programmer for a Southern California firm. He once told me, "I'm starting to learn Lisp, and it must be a good language because it's got `throw` and `catch` in it." Unfortunately for Scott, `throw` and `catch` don't refer to Frisbees but to transfer of control. They are both special forms, with the following syntax:

```
(catch tag body...)
(throw tag value)
```

The first argument to `catch` is a tag, or label. The remaining arguments are evaluated one at a time, and the last one is returned. Thus, `catch` is much like `progn`. The difference is that if any code in the dynamic extent of the body of the `catch` evaluates the special form `throw`, then control is immediately passed to the enclosing `catch` with the same tag.

For example, the form

```
(catch 'tag
      (print 1) (throw 'tag 2) (print 3))
```

prints 1 and returns 2, without going on to print 3. A more representative example is:

```
(defun print-table (l)
  (catch 'not-a-number (mapcar #'print-sqrt-abs l)))

(defun print-sqrt-abs (x)
  (print (sqrt (abs (must-be-number x)))))

(defun must-be-number (x)
  (if (numberp x) x
      (throw 'not-a-number "huh?")))

> (print-table '(1 4 -9 x 10 20))
1
2
3
"huh?"
```

Here `print-table` calls `print-sqrt-abs`, which calls `must-be-number`. The first three times all is fine and the values 1,2,3 get printed. The next time `x` is not a number, so the value "huh?" gets thrown to the tag `not-a-number` established by `catch` in `f`. The

throw bypasses the pending calls to abs, sqrt, and print, as well as the rest of the call to mapcar.

This kind of control is provided in Scheme with a very general and powerful procedure, call-with-current-continuation, which is often abbreviated call/cc. call/cc is a normal procedure (not a special form like throw and catch) that takes a single argument. Let's call the argument computation. computation must be a procedure of one argument. When call/cc is invoked, it calls computation, and whatever computation returns is the value of the call to call/cc. The trick is that the procedure computation also takes an argument (which we'll call cc) that is another procedure representing the current continuation point. If cc is applied to some value, that value is returned as the value of the call to call/cc. Here are some examples:

```
> (scheme)
=> (+ 1 (call/cc (lambda (cc) (+ 20 300))))
321
```

This example ignores cc and just computes (+ 1 (+ 20 300)). More precisely, it is equivalent to:

```
((lambda (val) (+ 1 val))
 (+ 20 300))
```

The next example does make use of cc:

```
=> (+ 1 (call/cc (lambda (cc) (+ 20 (cc 300)))))
301
```

This passes 300 to cc, thus bypassing the addition of 20. It effectively throws 300 out of the computation to the catch point established by call/cc. It is equivalent to:

```
((lambda (val) (+ 1 val))
 300)
```

or to:

```
((lambda (val) (+ 1 val))
 (catch 'cc
  ((lambda (v) (+ 20 v))
   (throw 'cc 300))))
```

Here's how the throw/catch mechanism would look in Scheme:

```
(define (print-table l)
  (call/cc
   (lambda (escape)
     (set! not-a-number escape)
     (map print-sqrt-abs l))))

(define (print-sqrt-abs x)
  (write (sqrt (abs (must-be-number x)))))

(define (must-be-number x)
  (if (numberp x) x
      (not-a-number "huh?")))

(define (map fn l)
  (if (null? l)
      '()
      (cons (fn (first l))
            (map fn (rest l)))))
```

The ability to return to a pending point in the computation is useful for this kind of error and interrupt handling. However, the truly amazing, wonderful thing about call/cc is the ability to return to a continuation point more than once. Consider a slight variation:

```
=> (+ 1 (call/cc (lambda (cc)
                  (set! old-cc cc)
                  (+ 20 (cc 300)))))
301
=> (old-cc 500)
501
```

Here, we first computed 301, just as before, but along the way saved cc in the global variable old-cc. Afterward, calling (old-cc 500) returns (for the second time) to the point in the computation where 1 is added, this time returning 501. The equivalent Common Lisp code leads to an error:

```
> (+ 1 (catch 'tag (+ 20 (throw 'tag 300))))
301
> (throw 'tag 500)
Error: there was no pending CATCH for the tag TAG
```

In other words, call/cc's continuations have indefinite extent, while throw/catch tags only have dynamic extent.

We can use `call/cc` to implement automatic backtracking (among other things). Suppose we had a special form, `amb`, the “ambiguous” operator, which returns one of its arguments, chosen at random. We could write:

```
(define (integer) (amb 1 (+ 1 (integer))))
```

and a call to `integer` would return some random positive integer. In addition, suppose we had a function, `fail`, which doesn’t return at all but instead causes execution to continue at a prior `amb` point, with the other choice taken. Then we could write succinct² backtracking code like the following:

```
(define (prime)
  (let ((n (integer)))
    (if (prime? n) n (fail))))
```

If `prime?` is a predicate that returns true only when its argument is a prime number, then `prime` will always return some prime number, decided by generating random integers. While this looks like a major change to the language—adding backtracking and nondeterminism—it turns out that `amb` and `fail` can be implemented quite easily with `call/cc`. First, we need to make `amb` be a macro:

```
(def-scheme-macro amb (x y)
  '(random-choice (lambda () ,x) (lambda () ,y)))
```

The rest is pure Scheme. We maintain a list of `backtrack-points`, which are implemented as functions of no arguments. To backtrack, we just call one of these functions. That is what `fail` does. The function `choose-first` takes two functions and pushes the second, along with the proper continuation, on `backtrack-points`, and then calls the first, returning that value. The function `random-choice` is what `amb` expands into: it decides which choice is first, and which is second. (Note that the convention in Scheme is to write global variables like `backtrack-points` without asterisks.)

```
(define backtrack-points nil)

(define (fail)
  (let ((last-choice (car backtrack-points)))
    (set! backtrack-points (cdr backtrack-points))
    (last-choice)))
```

²although inefficient

```

(define (random-choice f g)
  (if (= 1 (random 2))
      (choose-first f g)
      (choose-first g f)))

(define (choose-first f g)
  (call/cc
   (lambda (k)
     (set! backtrack-points
              (cons (lambda () (k g)) backtrack-points))
      (f))))

```


This implements chronological backtracking, as in Prolog. However, we actually have the freedom to do other kinds of backtracking as well. Instead of having `fail` take the first element of `backtrack-points`, we could choose a random element instead. Or, we could do some more complex analysis to choose a good backtrack point.


`call/cc` can be used to implement a variety of control structures. As another example, many Lisp implementations provide a `reset` function that aborts the current computation and returns control to the top-level read-eval-print loop. `reset` can be defined quite easily using `call/cc`. The trick is to capture a continuation that is at the top level and save it away for future use. The following expression, evaluated at the top level, saves the appropriate continuation in the value of `reset`:


```

(call/cc (lambda (cc) (set! reset (lambda ()
                                   (cc "Back to top level")))))

```

 **Exercise 22.2 [m]** Can you implement `call/cc` in Common Lisp?

 **Exercise 22.3 [s]** Can you implement `amb` and `fail` in Common Lisp?

 **Exercise 22.4 [m]** `fail` could be written `(define (fail) ((pop backtrack-points)))` if we had the `pop` macro in Scheme. Write `pop`.

22.5 An Interpreter Supporting Call/cc

It is interesting that the more a host language has to offer, the easier it is to write an interpreter. Perhaps the hardest part of writing a Lisp interpreter (or compiler) is garbage collection. By writing our interpreter in Lisp, we bypassed the problem

all together—the host language automatically collects garbage. Similarly, if we are using a Common Lisp that is properly tail-recursive, then our interpreter will be too, without taking any special steps. If not, the interpreter must be rewritten to take care of tail-recursion, as we have seen above.

It is the same with `call/cc`. If our host language provides continuations with indefinite extent, then it is trivial to implement `call/cc`. If not, we have to rewrite the whole interpreter, so that it explicitly handles continuations. The best way to do this is to make `interp` a function of three arguments: an expression, an environment, and a continuation. That means the top level will have to change too. Rather than having `interp` return a value that gets printed, we just pass it the function `print` as a continuation:

```
(defun scheme ()
  "A Scheme read-eval-print loop (using interp).
  Handles call/cc by explicitly passing continuations."
  (init-scheme-interp)
  (loop (format t "~&=> ")
        (interp (read) nil #'print)))
```

Now we are ready to tackle `interp`. For clarity, we will base it on the non-tail-recursive version. The cases for symbols, atoms, macros, and quote are almost the same as before. The difference is that the result of each computation gets passed to the continuation, `cc`, rather than just being returned.

The other cases are all more complex, because they all require explicit representation of continuations. That means that calls to `interp` cannot be nested. Instead, we call `interp` with a continuation that includes another call to `interp`. For example, to interpret `(if p x y)`, we first call `interp` on the second element of the form, the predicate `p`. The continuation for this call is a function that tests the value of `p` and interprets either `x` or `y` accordingly, using the original continuation for the recursive call to `interp`. The other cases are similar. One important change is that Scheme procedures are implemented as Lisp functions where the first argument is the continuation:

```
(defun interp (x env cc)
  "Evaluate the expression x in the environment env,
  and pass the result to the continuation cc."
  (cond
    ((symbolp x) (funcall cc (get-var x env)))
    ((atom x) (funcall cc x))
    ((scheme-macro (first x))
     (interp (scheme-macro-expand x) env cc))
    ((case (first x)
        (QUOTE (funcall cc (second x)))
        (BEGIN (interp-begin (rest x) env cc))
```


Because Scheme procedures expect a continuation as the first argument, we need to redefine `init-scheme-proc` to install procedures that accept and apply the continuation:

```
(defun init-scheme-proc (f)
  "Define a Scheme primitive procedure as a CL function."
  (if (listp f)
      (set-global-var! (first f)
                      #'(lambda (cont &rest args)
                          (funcall cont (apply (second f) args))))
      (init-scheme-proc (list f f))))
```

We also need to define `call/cc`. Think for a moment about what `call/cc` must do. Like all Scheme procedures, it takes the current continuation as its first argument. The second argument is a procedure—a computation to be performed. `call/cc` performs the computation by calling the procedure. This is just a normal call, so it uses the current continuation. The tricky part is what `call/cc` passes the computation as its argument. It passes an escape procedure, which can be invoked to return to the same point that the original call to `call/cc` would have returned to. Once the working of `call/cc` is understood, the implementation is obvious:

```
(defun call/cc (cc computation)
  "Make the continuation accessible to a Scheme procedure."
  (funcall computation cc
            ;; Package up CC into a Scheme function:
            #'(lambda (cont val)
                (declare (ignore cont))
                (funcall cc val))))

;; Now install call/cc in the global environment
(set-global-var! 'call/cc #'call/cc)
(set-global-var! 'call-with-current-continuation #'call/cc)
```

22.6 History and References

Lisp interpreters and AI have a long history together. MIT AI Lab Memo No. 1 (McCarthy 1958) was the first paper on Lisp. McCarthy's students were working on a Lisp compiler, had written certain routines—read, print, etc.—in assembly

language, and were trying to develop a full Lisp interpreter in assembler. Sometime around the end of 1958, McCarthy wrote a theoretical paper showing that Lisp was powerful enough to write the universal function, `eval`. A programmer on the project, Steve Russell, saw the paper, and, according to McCarthy:

Steve Russell said, look, why don't I program this eval and—you remember the interpreter—and I said to him, ho, ho, you're confusing theory with practice, this eval is intended for reading not for computing. But he went ahead and did it. That is, he compiled the eval in my paper into 704 machine code fixing bugs and then advertised this as a Lisp interpreter, which it certainly was.³

So the first Lisp interpreter was the result of a programmer ignoring his boss's advice. The first compiler was for the Lisp 1.5 system (McCarthy et al. 1962). The compiler was written in Lisp; it was probably the first compiler written in its own language.

Allen's *Anatomy of Lisp* (1978) was one of the first overviews of Lisp implementation techniques, and it remains one of the best. However, it concentrates on the dynamic-scoping Lisp dialects that were in use at the time. The more modern view of a lexically scoped Lisp was documented in an influential pair of papers by Guy Steele (1976a,b). His papers "Lambda: the ultimate goto" and "Compiler optimization based on viewing lambda as rename plus goto" describe properly tail-recursive interpreters and compilers.

The Scheme dialect was invented by Gerald Sussman and Guy Steele around 1975 (see their MIT AI Memo 349). The *Revised⁴ Report on the Algorithmic Language Scheme* (Clinger et al. 1991) is the definitive reference manual for the current version of Scheme.

Abelson and Sussman (1985) is probably the best introduction to computer science ever written. It may or may not be a coincidence that it uses Scheme as the programming language. It includes a Scheme interpreter. Winston and Horn's *Lisp* (1989) also develops a Lisp interpreter.

The `amb` operator for nondeterministic choice was proposed by John McCarthy (1963) and used in SCHEMER (Zabih et al. 1987), a nondeterministic Lisp. Ruf and Weise (1990) present another implementation of backtracking in Scheme that incorporates all of logic programming.

³McCarthy's words from a talk on the history of Lisp, 1974, recorded by Stoyan (1984).

22.7 Exercises

- ?** **Exercise 22.5 [m]** While Scheme does not provide full-blown support for optional and keyword arguments, it does support rest parameters. Modify the interpreter to support the Scheme syntax for rest parameters:

Scheme	Common Lisp
<code>(lambda x <i>body</i>)</code>	<code>(lambda (&rest x) <i>body</i>)</code>
<code>(lambda (x y . z) <i>body</i>)</code>	<code>(lambda (x y &rest z) <i>body</i>)</code>

- ?** **Exercise 22.6 [h]** The representation of environments is somewhat wasteful. Currently it takes $3n$ cons cells to represent an environment with n variables. Change the representation to take less space.

- ?** **Exercise 22.7 [m]** As we've implemented macros, they need to be expanded each time they are encountered. This is not so bad for the compiler—you expand the source code and compile it, and then never refer to the source code again. But for the interpreter, this treatment of macros is most unsatisfactory: the work of macro-expansion must be done again and again. How can you eliminate this duplicated effort?

- ?** **Exercise 22.8 [m]** It turns out Scheme allows some additional syntax in `let` and `cond`. First, there is the “named-let” expression, which binds initial values for variables but also defines a local function that can be called within the body of the `let`. Second, `cond` recognizes the symbol `=>` when it is the second element of a `cond` clause, and treats it as a directive to pass the value of the test (when it is not false) to the third element of the clause, which must be a function of one argument. Here are two examples:

```
(define (fact n)
  ;; Iterative factorial; does not grow the stack
  (let loop ((result 1) (i n))
    (if (= i 0) result (loop (* result i) (- i 1)))))

(define (lookup key alist)
  ;; Find key's value in alist
  (cond ((assoc key alist) => cdr)
        (else #f)))
```

These are equivalent to:

```

(define (fact n)
  (letrec
    ((loop (lambda (result i)
             (if (= i 0)
                 result
                 (loop (* result i) (- i 1)))))
    (loop 1 n)))

(define (lookup key alist)
  (let ((g0030 (assoc key alist)))
    (if g0030
        (cdr g0030)
        #f)))

```

Write macro definitions for `let` and `cond` allowing these variations.

- ?** **Exercise 22.9 [h]** Some Scheme implementations permit `define` statements inside the body of a `lambda` (and thus of a `define`, `let`, `let*`, or `letrec` as well). Here is an example:

```

(define (length l)
  (define (len l n)
    (if (null? l) n (len (cdr l) (+ n 1))))
  (len l 0))

```

The internal definition of `len` is interpreted not as defining a global name but rather as defining a local name as if with `letrec`. The above definition is equivalent to:

```

(define (length l)
  (letrec ((len (lambda (l n)
                  (if (null? l) n (len (cdr l) (+ n 1)))))
    (len l 0)))

```

Make changes to the interpreter to allow this kind of internal definition.

- ?** **Exercise 22.10** Scheme programmers are often disdainful of the `function` or `#'` notation in Common Lisp. Is it possible (without changing the compiler) to make Common Lisp accept `(lambda () ...)` instead of `#'(lambda () ...)` and `fn` instead of `#'fn`?

- ?** **Exercise 22.11 [m]** The top level of the continuation-passing version of scheme includes the call: `(interp (read) nil #'print)`. Will this always result in some

value being printed? Or is it possible that the expression read might call some escape function that ignores the value without printing anything?

? **Exercise 22.12 [h]** What would have to be added or changed to turn the Scheme interpreter into a Common Lisp interpreter?

? **Exercise 22.13 [h]** How would you change the interpreter to allow for multiple values? Explain how this would be done both for the first version of the interpreter and for the continuation-passing version.

22.8 Answers

Answer 22.2 There is no way to implement a full `call/cc` to Common Lisp, but the following works for cases where the continuation is only used with dynamic extent:

```
(defun call/cc (computation)
  "Call computation, passing it the current continuation.
  The continuation has only dynamic extent."
  (funcall computation #'(lambda (x) (return-from call/cc x))))
```

Answer 22.3 No. `fail` requires continuations with dynamic extent.

Answer 22.5 We need only modify `extend-env` to know about an atomic vars list. While we're at it, we might as well add some error checking:

```
(defun extend-env (vars vals env)
  "Add some variables and values to an environment."
  (cond ((null vars)
         (assert (null vals) () "Too many arguments supplied")
         env)
        ((atom vars)
         (cons (list vars vals) env))
        (t (assert (rest vals) () "Too few arguments supplied")
            (cons (list (first vars) (first vals))
                  (extend-env (rest vars) (rest vals) env)))))
```

Answer 22.6 Storing the environment as an association list, $((var\ val)\dots)$, makes it easy to look up variables with `assoc`. We could save one `cons` cell per variable just by changing to $((var . val)\dots)$. But even better is to switch to a different representation, one presented by Steele and Sussman in *The Art of the Interpreter* (1978). In this representation we switch from a single list of `var/val` pairs to a list of frames, where each frame is a `var-list/val-list` pair. It looks like this:

```
((var...) . (val...))
((var...) . (val...))
...)
```

Now `extend-env` is trivial:

```
(defun extend-env (vars vals env)
  "Add some variables and values to an environment."
  (cons (cons vars vals) env))
```

The advantage of this approach is that in most cases we already have a list of variables (the procedure's parameter list) and values (from the `mapcar` of `interp` over the arguments). So it is cheaper to just `cons` these two lists together, rather than arranging them into pairs. Of course, `get-var` and `set-var!` become more complex.

Answer 22.7 One answer is to destructively alter the source code as it is macro-expanded, so that the next time the source code is interpreted, it will already be expanded. The following code takes care of that:

```
(defun scheme-macro-expand (x)
  (displace x (apply (scheme-macro (first x)) (rest x))))

(defun displace (old new)
  "Destructively change old cons-cell to new value."
  (if (consp new)
      (progn (setf (car old) (car new))
             (setf (cdr old) (cdr new))
             old)
      (displace old '(begin .new))))
```

One drawback to this approach is that the user's source code is actually changed, which may make debugging confusing. An alternative is to expand into something that keeps both the original and macro-expanded code around:

```
(defun displace (old new)
  "Destructively change old to a DISPLACED structure."
  (setf (car old) 'DISPLACED)
  (setf (cdr old) (list new old))
  old)
```

This means that DISPLACED is a new special form, and we need a clause for it in the interpreter. It would look something like this:

```
(case (first x)
  ...
  (DISPLACED (interp (second x) env))
  ...)
```

We'd also need to modify the printing routines to print just old whenever they see (displaced old new).

Answer 22.8

```
(def-scheme-macro let (vars &rest body)
  (if (symbolp vars)
      ;; named let
      (let ((f vars) (vars (first body)) (body (rest body)))
        '(letrec ((,f (lambda ,(mapcar #'first vars) .,body))
                  (,f ,(mapcar #'second vars))))
      ;; "regular" let
      '((lambda ,(mapcar #'first vars) . ,body)
        . ,(mapcar #'second vars))))

(def-scheme-macro cond (&rest clauses)
  (cond ((null clauses) nil)
        ((length=1 (first clauses))
         '(or ,(first clauses) (cond .,(rest clauses))))
        ((starts-with (first clauses) 'else)
         '(begin .,(rest (first clauses))))
        ((eq (second (first clauses)) '=>)
         (assert (= (length (first clauses)) 3))
         (let ((var (gensym)))
           '(let ((,var ,(first (first clauses)))
                 (if ,var ,(third (first clauses)) ,var)
                 (cond .,(rest clauses))))))
        (t '(if ,(first (first clauses))
                (begin .,(rest (first clauses)))
                (cond .,(rest clauses))))))
```


Answer 22.10 It is easy to define `lambda` as a macro, eliminating the need for `#'(lambda ...)`:

```
(defmacro lambda (args &rest body)
  '(function (lambda ,args ,@body)))
```

If this were part of the Common Lisp standard, I would gladly use it. But because it is not, I have avoided it, on the grounds that it can be confusing.

It is also possible to write a new function-defining macro that would do the following type of expansion:

```
(defn double (x) (* 2 x)) ⇒
(defparameter double (defun double (x) (* 2 x)))
```

This makes `double` a special variable, so we can write `double` instead of `#'double`. But this approach is not recommended—it is dangerous to define special variables that violate the asterisk convention, and the Common Lisp compiler may not be able to optimize special variable references the way it can function special forms. Also, this approach would not interact properly with `flet` and `labels`.

CHAPTER 23

Compiling Lisp

Many textbooks show simple interpreters for Lisp, because they are simple to write, and because it is useful to know how an interpreter works. Unfortunately, not as many textbooks show how to write a compiler, even though the same two reasons hold. The simplest compiler need not be much more complex than an interpreter.

One thing that makes a compiler more complex is that we have to describe the output of the compiler: the instruction set of the machine we are compiling for. For the moment let's assume a stack-based machine. The calling sequence on this machine for a function call with n arguments is to push the n arguments onto the stack and then push the function to be called. A "CALL n " instruction saves the return point on the stack and goes to the first instruction of the called function. By convention, the first instruction of a function will always be "ARGS n ", which pops n arguments off the stack, putting them in the new function's environment, where they can be accessed by LVAR and LSET instructions. The function should return with a RETURN instruction, which resets the program counter and the environment to the point of the original CALL instruction.

In addition, our machine has three JUMP instructions; one that branches unconditionally, and two that branch depending on if the top of the stack is nil or non-nil. There is also an instruction for popping unneeded values off the stack, and for accessing and altering global variables. The instruction set is shown in figure 23.1. A glossary for the compiler program is given in figure 23.2. A summary of a more complex version of the compiler appears on page 795.

opcode	args	description
CONST	x	push a constant on the stack
LVAR	i,j	push a local variable's value
GVAR	sym	push a global variable's value
LSET	i,j	store top-of-stack in a local variable
GSET	sym	store top-of-stack in a global variable
POP		pop the stack
TJUMP	label	go to label if top-of-stack is non-nil; pop stack
FJUMP	label	go to label if top-of-stack is nil; pop stack
JUMP	label	go to label (don't pop stack)
RETURN		go to last return point
ARGS	n	move n arguments from stack to environment
CALL	n	go to start of function, saving return point n is the number of arguments passed
FN	fn	create a closure from argument and current environment and push it on the stack

Figure 23.1: Instruction Set for Hypothetical Stack Machine

As an example, the procedure

```
(lambda () (if (= x y) (f (g x)) (h x y (h 1 2))))
```

should compile into the following instructions:

```

      ARGS    0
      GVAR    X
      GVAR    Y
      GVAR    =
      CALL    2
      FJUMP   L1
      GVAR    X
      GVAR    G
      CALL    1
      GVAR    F
      CALL    1
      JUMP    L2
L1:   GVAR    X
      GVAR    Y
      CONST   1
      CONST   2
      GVAR    H
      CALL    2

```

```

          GVAR   H
          CALL   3
L2:      RETURN

```

	Top-Level Functions
comp-show	Compile an expression and show the resulting code.
compiler	Compile an expression as a parameterless function.
	Special Variables
label-num	Number for the next assembly language label.
primitive-fns	List of built-in Scheme functions.
	Data Types
fn	A Scheme function.
	Major Functions
comp	Compile an expression into a list of instructions.
comp-begin	Compile a sequence of expressions.
comp-if	Compile a conditional (if) expression.
comp-lambda	Compile a lambda expression.
	Auxiliary Functions
gen	Generate a single instruction.
seq	Generate a sequence of instructions.
gen-label	Generate an assembly language label.
gen-var	Generate an instruction to reference a variable.
gen-set	Generate an instruction to set a variable.
name!	Set the name of a function to a given value.
print-fn	Print a Scheme function (just the name).
show-fn	Print the instructions in a Scheme function.
label-p	Is the argument a label?
in-env-p	Is the symbol in the environment? If so, where?

Figure 23.2: Glossary for the Scheme Compiler

The first version of the Scheme compiler is quite simple. It mimics the structure of the Scheme evaluator. The difference is that each case generates code rather than evaluating a subexpression:

```

(defun comp (x env)
  "Compile the expression x into a list of instructions."
  (cond
    ((symbolp x) (gen-var x env))
    ((atom x) (gen 'CONST x))
    ((scheme-macro (first x)) (comp (scheme-macro-expand x) env))
    ((case (first x)

```

```

(QUOTE (gen 'CONST (second x)))
(BEGIN (comp-begin (rest x) env))
(SET! (seq (comp (third x) env) (gen-set (second x) env)))
(IF (comp-if (second x) (third x) (fourth x) env))
(LAMBDA (gen 'FN (comp-lambda (second x) (rest (rest x)) env)))
;; Procedure application:
;; Compile args, then fn, then the call
(t (seq (mappend #'(lambda (y) (comp y env)) (rest x))
        (comp (first x) env)
        (gen 'call (length (rest x))))))

```

The compiler `comp` has the same nine cases—in fact the exact same structure—as the interpreter `interp` from chapter 22. Each case is slightly more complex, so the three main cases have been made into separate functions: `comp-begin`, `comp-if`, and `comp-lambda`. A `begin` expression is compiled by compiling each argument in turn but making sure to pop each value but the last off the stack after it is computed. The last element in the `begin` stays on the stack as the value of the whole expression. Note that the function `gen` generates a single instruction (actually a list of one instruction), and `seq` makes a sequence of instructions out of two or more subsequences.

```

(defun comp-begin (exps env)
  "Compile a sequence of expressions, popping all but the last."
  (cond ((null exps) (gen 'CONST nil))
        ((length=1 exps) (comp (first exps) env))
        (t (seq (comp (first exps) env)
                 (gen 'POP)
                 (comp-begin (rest exps) env)))))

```

An `if` expression is compiled by compiling the predicate, then part, and else part, and by inserting appropriate branch instructions.

```

(defun comp-if (pred then else env)
  "Compile a conditional expression."
  (let ((L1 (gen-label))
        (L2 (gen-label)))
    (seq (comp pred env) (gen 'FJUMP L1)
          (comp then env) (gen 'JUMP L2)
          (list L1) (comp else env)
          (list L2))))

```

Finally, a `lambda` expression is compiled by compiling the body, surrounding it with one instruction to set up the arguments and another to return from the function, and

then storing away the resulting compiled code, along with the environment. The data type `fn` is implemented as a structure with slots for the body of the code, the argument list, and the name of the function (for printing purposes only).

```
(defstruct (fn (:print-function print-fn))
  code (env nil) (name nil) (args nil))

(defun comp-lambda (args body env)
  "Compile a lambda form into a closure with compiled code."
  (assert (and (listp args) (every #'symbolp args)) ())
  "Lambda arglist must be a list of symbols, not ~a" args)
;; For now, no &rest parameters.
;; The next version will support Scheme's version of &rest
(make-fn
 :env env :args args
 :code (seq (gen 'ARGS (length args))
            (comp-begin body (cons args env))
            (gen 'RETURN))))
```

The advantage of compiling over interpreting is that much can be decided at compile time. For example, the compiler can determine if a variable reference is to a global or lexical variable, and if it is to a lexical variable, exactly where that lexical variable is stored. This computation is done only once by the compiler, but it has to be done each time the expression is encountered by the interpreter. Similarly, the compiler can count up the number of arguments once and for all, while the interpreter must go through a loop, counting up the number of arguments, and testing for the end of the arguments after each one is interpreted. So it is clear that the compiler can be more efficient than the interpreter.

Another advantage is that the compiler can be more robust. For example, in `comp-lambda`, we check that the parameter list of a lambda expression is a list containing only symbols. It would be too expensive to make such checks in an interpreter, but in a compiler it is a worthwhile trade-off to check once at compile time for error conditions rather than checking repeatedly at run time.

Before we show the rest of the compiler, here's a useful top-level interface to `comp`:

```
(defvar *label-num* 0)

(defun compiler (x)
  "Compile an expression as if it were in a parameterless lambda."
  (setf *label-num* 0)
  (comp-lambda '() (list x) nil))
```

```
(defun comp-show (x)
  "Compile an expression and show the resulting code"
  (show-fn (compiler x))
  (values))
```

Now here's the code to generate individual instructions and sequences of instructions. A sequence of instructions is just a list, but we provide the function `seq` rather than using `append` directly for purposes of data abstraction. A label is just an atom.

```
(defun gen (opcode &rest args)
  "Return a one-element list of the specified instruction."
  (list (cons opcode args)))

(defun seq (&rest code)
  "Return a sequence of instructions"
  (apply #'append code))

(defun gen-label (&optional (label 'L))
  "Generate a label (a symbol of the form Lnnn)"
  (intern (format nil "~a~d" label (incf *label-num*))))
```

Environments are now represented as lists of frames, where each frame is a sequence of variables. Local variables are referred to not by their name but by two integers: the index into the list of frames and the index into the individual frame. As usual, the indexes are zero-based. For example, given the code:

```
(let ((a 2.0)
      (b 2.1))
  (let ((c 1.0)
        (d 1.1))
    (let ((e 0.0)
          (f 0.1))
      (+ a b c d e f))))
```

the innermost environment is `((e f) (c d) (a b))`. The function `in-env-p` tests if a variable appears in an environment. If this environment were called `env`, then `(in-env-p 'f env)` would return `(2 1)` and `(in-env-p 'x env)` would return `nil`.

```

(defun gen-var (var env)
  "Generate an instruction to reference a variable's value."
  (let ((p (in-env-p var env)))
    (if p
        (gen 'LVAR (first p) (second p) ";" var)
        (gen 'GVAR var))))

(defun gen-set (var env)
  "Generate an instruction to set a variable to top-of-stack."
  (let ((p (in-env-p var env)))
    (if p
        (gen 'LSET (first p) (second p) ";" var)
        (gen 'GSET var))))

```

Finally, we have some auxiliary functions to print out the results, to distinguish between labels and instructions, and to determine the index of a variable in an environment. Scheme functions now are implemented as structures, which must have a field for the code, and one for the environment. In addition, we provide a field for the name of the function and for the argument list; these are used only for debugging purposes. We'll adopt the convention that the `define` macro sets the function's name field, by calling `name!` (which is not part of standard Scheme).

```

(def-scheme-macro define (name &rest body)
  (if (atom name)
      '(name! (set! ,name . ,body) ',name)
      (scheme-macro-expand
        '(define ,(first name)
              (lambda ,(rest name) . ,body))))))

(defun name! (fn name)
  "Set the name field of fn, if it is an un-named fn."
  (when (and (fn-p fn) (null (fn-name fn)))
    (setf (fn-name fn) name)
    name))

;; This should also go in init-scheme-interp:
(set-global-var! 'name! #'name!)

(defun print-fn (fn &optional (stream *standard-output*) depth)
  (declare (ignore depth))
  (format stream "{~a}" (or (fn-name fn) '???)))

```



```

(defun show-fn (fn &optional (stream *standard-output*) (depth 0))
  "Print all the instructions in a function.
  If the argument is not a function, just princ it,
  but in a column at least 8 spaces wide."
  (if (not (fn-p fn))
      (format stream "~8a" fn)
      (progn
        (fresh-line)
        (incf depth 8)
        (dolist (instr (fn-code fn))
          (if (label-p instr)
              (format stream "~a:" instr)
              (progn
                (format stream "~VT" depth)
                (dolist (arg instr)
                  (show-fn arg stream depth))
                (fresh-line)))))))

(defun label-p (x) "Is x a label?" (atom x))

(defun in-env-p (symbol env)
  "If symbol is in the environment, return its index numbers."
  (let ((frame (find symbol env :test #'find)))
    (if frame (list (position frame env) (position symbol frame)))))

```

Now we are ready to show the compiler at work:

```

> (comp-show '(if (= x y) (f (g x)) (h x y) (h 1 2)))
  ARGS      0
  GVAR      X
  GVAR      Y
  GVAR      =
  CALL      2
  FJUMP     L1
  GVAR      X
  GVAR      G
  CALL      1
  GVAR      F
  CALL      1
  JUMP      L2
L1:  GVAR      X
     GVAR      Y
     CONST     1
     CONST     2
     GVAR      H
     CALL      2
     GVAR      H
     CALL      3
L2:  RETURN

```

This example should give the reader a feeling for the code generated by the compiler.

Another reason a compiler has an advantage over an interpreter is that the compiler can afford to spend some time trying to find a more efficient encoding of an expression, while for the interpreter, the overhead of searching for a more efficient interpretation usually offsets any advantage gained. Here are some places where a compiler could do better than an interpreter (although our compiler currently does not):

```
> (comp-show '(begin "doc" (write x) y))
  ARGS    0
  CONST   doc
  POP
  GVAR    X
  GVAR    WRITE
  CALL    1
  POP
  GVAR    Y
  RETURN
```

In this example, code is generated to push the constant "doc" on the stack and then immediately pop it off. If we have the compiler keep track of what expressions are compiled "for value"—as *y* is the value of the expression above—and which are only compiled "for effect," then we can avoid generating any code at all for a reference to a constant or variable for effect. Here's another example:

```
> (comp-show '(begin (+ (* a x) (f x)) x))
  ARGS    0
  GVAR    A
  GVAR    X
  GVAR    *
  CALL    2
  GVAR    X
  GVAR    F
  CALL    1
  GVAR    +
  CALL    2
  POP
  GVAR    X
  RETURN
```

In this expression, if we can be assured that `+` and `*` refer to the normal arithmetic functions, then we can compile this as if it were `(begin (f x) x)`. Furthermore, it is reasonable to assume that `+` and `*` will be instructions in our machine that can be invoked inline, rather than having to call out to a function. Many compilers spend a significant portion of their time optimizing arithmetic operations, by taking into account associativity, commutativity, distributivity, and other properties.

Besides arithmetic, compilers often have expertise in conditional expressions. Consider the following:

```
> (comp-show '(if (and p q) x y))
      ARGS    0
      GVAR    P
      FJUMP   L3
      GVAR    Q
      JUMP    L4
L3:   GVAR    NIL
L4:   FJUMP   L1
      GVAR    X
      JUMP    L2
L1:   GVAR    Y
L2:   RETURN
```

Note that `(and p q)` macro-expands to `(if p q nil)`. The resulting compiled code is correct, but inefficient. First, there is an unconditional jump to L4, which labels a conditional jump to L1. This could be replaced with a conditional jump to L1. Second, at L3 we load NIL and then jump on nil to L1. These two instructions could be replaced by an unconditional jump to L1. Third, the FJUMP to L3 could be replaced by an FJUMP to L1, since we now know that the code at L3 unconditionally goes to L1.

Finally, some compilers, particularly Lisp compilers, have expertise in function calling. Consider the following:

```
> (comp-show '(f (g x y)))
      ARGS    0
      GVAR    X
      GVAR    Y
      GVAR    G
      CALL    2
      GVAR    F
      CALL    1
      RETURN
```

Here we call *g* and when *g* returns we call *f*, and when *f* returns we return from this function. But this last return is wasteful; we push a return address on the stack, and then pop it off, and return to the next return address. An alternative function-calling protocol involves pushing the return address before calling *g*, but then not pushing a return address before calling *f*; when *f* returns, it returns directly to the calling function, whatever that is.

Such an optimization looks like a small gain; we basically eliminate a single instruction. In fact, the implications of this new protocol are enormous: we can now invoke a recursive function to an arbitrary depth without growing the stack at all—as long as the recursive call is the last statement in the function (or in a branch of the function when there are conditionals). A function that obeys this constraint on its recursive calls is known as a *properly tail-recursive* function. This subject was discussed in section 22.3.

All the examples so far have only dealt with global variables. Here's an example using local variables:

```
> (comp-show '((lambda (x) ((lambda (y z) (f x y z)) 3 x)) 4))
  ARGS  0
  CONST 4
  FN
    ARGS  1
    CONST 3
    LVAR  0 0 ; X
    FN
      ARGS  2
      LVAR  1 0 ; X
      LVAR  0 0 ; Y
      LVAR  0 1 ; Z
      GVAR  F
      CALL  3
      RETURN
    CALL  2
    RETURN
  CALL  1
  RETURN
```

The code is indented to show nested functions. The top-level function loads the constant 4 and an anonymous function, and calls the function. This function loads the constant 3 and the local variable *x*, which is the first (0th) element in the top (0th) frame. It then calls the double-nested function on these two arguments. This function loads *x*, *y*, and *z*: *x* is now the 0th element in the next-to-top (1st) frame, and *y* and *z* are the 0th and 1st elements of the top frame. With all the arguments in

place, the function f is finally called. Note that no continuations are stored— f can return directly to the caller of this function.

However, all this explicit manipulation of environments is inefficient; in this case we could have compiled the whole thing by simply pushing 4, 3, and 4 on the stack and calling f .

	Top-Level Functions
scheme	A read-compile-execute-print loop.
comp-go	Compile and execute an expression.
machine	Run the abstract machine.
	Data Types
prim	A Scheme primitive function.
ret-addr	A return address (function, program counter, environment).
	Auxiliary Functions
arg-count	Report an error for wrong number of arguments.
comp-list	Compile a list of expressions onto the stack.
comp-const	Compile a constant expression.
comp-var	Compile a variable reference.
comp-funcall	Compile a function application.
primitive-p	Is this function a primitive?
init-scheme-comp	Initialize primitives used by compiler.
gen-args	Generate code to load arguments to a function.
make-true-list	Convert a dotted list to a nondotted one.
new-fn	Build a new function.
is	Predicate is true if instructions opcode matches.
optimize	A peephole optimizer.
gen1	Generate a single instruction.
target	The place a branch instruction branches to,
next-instr	The next instruction in a sequence.
quasi-q	Expand a quasiquote form into append, cons, etc.
	Functions for the Abstract Machine
assemble	Turn a list of instructions into a vector.
asm-first-pass	Find labels and length of code.
asm-second-pass	Put code into the code vector.
opcode	The opcode of an instruction.
args	The arguments of an instruction.
argi	For $i = 1, 2, 3$ — select i th argument of instruction.

Figure 23.3: Glossary of the Scheme Compiler, Second Version

23.1 A Properly Tail-Recursive Lisp Compiler

In this section we describe a new version of the compiler, first by showing examples of its output, and then by examining the compiler itself, which is summarized in figure 23.3. The new version of the compiler also makes use of a different function calling sequence, using two new instructions, CALLJ and SAVE. As the name implies, SAVE saves a return address on the stack. The CALLJ instruction no longer saves anything; it can be seen as an unconditional jump—hence the J in its name.

First, we see how nested function calls work:

```
> (comp-show '(f (g x)))
      ARGS  0
      SAVE  K1
      GVAR  X
      GVAR  G
      CALLJ 1
K1:   GVAR  F
      CALLJ 1
```

The continuation point K1 is saved so that g can return to it, but then no continuation is saved for f, so f returns to whatever continuation is on the stack. Thus, there is no need for an explicit RETURN instruction. The final CALL is like an unconditional branch.

The following example shows that all functions but the last (f) need a continuation point:

```
> (comp-show '(f (g (h x) (h y))))
      ARGS  0
      SAVE  K1
      SAVE  K2
      GVAR  X
      GVAR  H
      CALLJ 1
K2:   SAVE  K3
      GVAR  Y
      GVAR  H
      CALLJ 1
K3:   GVAR  G
      CALLJ 2
K1:   GVAR  F
      CALLJ 1
```

This code first computes (h x) and returns to K2. Then it computes (h y) and returns to K3. Next it calls g on these two values, and returns to K1 before transferring to f. Since whatever f returns will also be the final value of the function we are compiling, there is no need to save a continuation point for f to return to.

In the next example we see that unneeded constants and variables in begin expressions are ignored:

```
> (comp-show '(begin "doc" x (f x) y))
      ARGS    0
      SAVE    K1
      GVAR    X
      GVAR    F
      CALLJ   1
K1:   POP
      GVAR    Y
      RETURN
```

One major flaw with the first version of the compiler is that it could pass data around, but it couldn't actually *do* anything to the data objects. We fix that problem by augmenting the machine with instructions to do arithmetic and other primitive operations. Unneeded primitive operations, like variables constants, and arithmetic operations are ignored when they are in the nonfinal position within begins. Contrast the following two expressions:

```
> (comp-show '(begin (+ (* a x) (f x)) x))
      ARGS    0
      SAVE    K1
      GVAR    X
      GVAR    F
      CALLJ   1
K1:   POP
      GVAR    X
      RETURN

> (comp-show '(begin (+ (* a x) (f x))))
      ARGS    0
      GVAR    A
      GVAR    X
      *
      SAVE    K1
      GVAR    X
      GVAR    F
      CALLJ   1
K1:   +
      RETURN
```

The first version of the compiler was context-free, in that it compiled all equivalent expressions equivalently, regardless of where they appeared. A properly tail-recursive compiler needs to be context-sensitive: it must compile a call that is the final value of a function differently than a call that is used as an intermediate value, or one whose value is ignored. In the first version of the compiler, `comp-lambda` was responsible for generating the RETURN instruction, and all code eventually reached that instruction. To make sure the RETURN was reached, the code for the two branches of `if` expressions had to rejoin at the end.

In the tail-recursive compiler, each piece of code is responsible for inserting its own RETURN instruction or implicitly returning by calling another function without saving a continuation point.

We keep track of these possibilities with two flags. The parameter `val?` is true when the expression we are compiling returns a value that is used elsewhere. The parameter `more?` is false when the expression represents the final value, and it is true when there is more to compute. In summary, there are three possibilities:

val?	more?	example: the X in:
true	true	<code>(if X y z) or (f X y)</code>
true	false	<code>(if p X z) or (begin y X)</code>
false	true	<code>(begin X y)</code>
false	false	<i>impossible</i>

The code for the compiler employing these conventions follows:

```
(defun comp (x env val? more?)
  "Compile the expression x into a list of instructions."
  (cond
    ((member x '(t nil)) (comp-const x val? more?))
    ((symbolp x) (comp-var x env val? more?))
    ((atom x) (comp-const x val? more?))
    ((scheme-macro (first x)) (comp (scheme-macro-expand x) env val? more?))
    ((case (first x)
      (QUOTE (arg-count x 1)
        (comp-const (second x) val? more?))
      (BEGIN (comp-begin (rest x) env val? more?))
      (SET! (arg-count x 2)
        (assert (symbolp (second x)) (x)
          "Only symbols can be set!, not ~a in ~a"
          (second x) x)
        (seq (comp (third x) env t t)
          (gen-set (second x) env)
          (if (not val?) (gen 'POP))
          (unless more? (gen 'RETURN))))))
```



```

(IF      (arg-count x 2 3)
         (comp-if (second x) (third x) (fourth x)
                  env val? more?))
(LAMBDA (when val?
         (let ((f (comp-lambda (second x) (rest2 x) env)))
               (seq (gen 'FN f) (unless more? (gen 'RETURN))))))
(t      (comp-funcall (first x) (rest x) env val? more?))))

```

Here we've added one more case: `t` and `nil` compile directly into primitive instructions, rather than relying on them being bound as global variables. (In real Scheme, the Boolean values are `#t` and `#f`, which need not be quoted, the empty list is `()`, which must be quoted, and `t` and `nil` are ordinary symbols with no special significance.)

I've also added some error checking for the number of arguments supplied to `quote`, `set!` and `if`. Note that it is reasonable to do more error checking in a compiler than in an interpreter, since the checking need be done only once, not each time through. The function to check arguments is as follows:

```

(defun arg-count (form min &optional (max min))
  "Report an error if form has wrong number of args."
  (let ((n-args (length (rest form))))
    (assert (<= min n-args max) (form)
            "Wrong number of arguments for ~a in ~a:
             ~d supplied, ~d@[ to ~d~] expected"
            (first form) form n-args min (if (/= min max) max))))

```

? **Exercise 23.1 [m]** Modify the compiler to check for additional compile-time errors suggested by the following erroneous expression:

```
(cdr (+ (list x y) 'y (3 x) (car 3 x)))
```

The tail-recursive compiler still has the familiar nine cases, but I have introduced `comp-var`, `comp-const`, `comp-if`, and `comp-funcall` to handle the increased complexity introduced by the `var?` and `more?` parameters.

Let's go through the `comp-` functions one at a time. First, `comp-begin` and `comp-list` just handle and pass on the additional parameters. `comp-list` will be used in `comp-funcall`, a new function that will be introduced to compile a procedure application.

```

(defun comp-begin (exps env val? more?)
  "Compile a sequence of expressions,
  returning the last one as the value."
  (cond ((null exps) (comp-const nil val? more?))
        ((length=1 exps) (comp (first exps) env val? more?))
        (t (seq (comp (first exps) env nil t)
                 (comp-begin (rest exps) env val? more?))))))

(defun comp-list (exps env)
  "Compile a list, leaving them all on the stack."
  (if (null exps) nil
      (seq (comp (first exps) env t t)
            (comp-list (rest exps) env))))

```

Then there are two trivial functions to compile variable access and constants. If the value is not needed, these produce no instructions at all. If there is no more to be done, then these functions have to generate the return instruction. This is a change from the previous version of `comp`, where the caller generated the return instruction. Note I have extended the machine to include instructions for the most common constants: `t`, `nil`, and some small integers.

```

(defun comp-const (x val? more?)
  "Compile a constant expression."
  (if val? (seq (if (member x '(t nil -1 0 1 2))
                  (gen x)
                  (gen 'CONST x))
                (unless more? (gen 'RETURN))))))

(defun comp-var (x env val? more?)
  "Compile a variable reference."
  (if val? (seq (gen-var x env) (unless more? (gen 'RETURN))))))

```

The remaining two functions are more complex. First consider `comp-if`. Rather than blindly generating code for the predicate and both branches, we will consider some special cases. First, it is clear that `(if t x y)` can reduce to `x` and `(if nil x y)` can reduce to `y`. It is perhaps not as obvious that `(if p x x)` can reduce to `(begin p x)`, or that the comparison of equality between the two branches should be done on the object code, not the source code. Once these trivial special cases have been considered, we're left with three more cases: `(if p x nil)`, `(if p nil y)`, and `(if p x y)`. The pattern of labels and jumps is different for each.

```

(defun comp-if (pred then else env val? more?)
  "Compile a conditional (IF) expression."
  (cond
    ((null pred)           ; (if nil x y) ==> y
     (comp else env val? more?))
    ((constantp pred)     ; (if t x y) ==> x
     (comp then env val? more?))
    ((and (listp pred)    ; (if (not p) x y) ==> (if p y x)
          (length=1 (rest pred))
          (primitive-p (first pred) env 1)
          (eq (prim-opcode (primitive-p (first pred) env 1)) 'not))
     (comp-if (second pred) else then env val? more?))
    (t (let ((pcode (comp pred env t t))
              (tcode (comp then env val? more?))
              (ecode (comp else env val? more?)))
         (cond
          ((equal tcode ecode) ; (if p x x) ==> (begin p x)
           (seq (comp pred env nil t) ecode))
          ((null tcode) ; (if p nil y) ==> p (TJUMP L2) y L2:
           (let ((L2 (gen-label)))
              (seq pcode (gen 'TJUMP L2) ecode (list L2)
                   (unless more? (gen 'RETURN))))))
          ((null ecode) ; (if p x) ==> p (FJUMP L1) x L1:
           (let ((L1 (gen-label)))
              (seq pcode (gen 'FJUMP L1) tcode (list L1)
                   (unless more? (gen 'RETURN))))))
          (t
           ; (if p x y) ==> p (FJUMP L1) x L1: y
           ; or p (FJUMP L1) x (JUMP L2) L1: y L2:
           (let ((L1 (gen-label))
                 (L2 (if more? (gen-label))))
              (seq pcode (gen 'FJUMP L1) tcode
                   (if more? (gen 'JUMP L2))
                   (list L1) ecode (if more? (list L2))))))))))

```

Here are some examples of if expressions. First, a very simple example:

```

> (comp-show '(if p (+ x y) (* x y)))
  ARGS    0
  GVAR    P
  FJUMP   L1
  GVAR    X
  GVAR    Y
  +
  RETURN
L1:  GVAR    X
     GVAR    Y
     *
     RETURN

```

Each branch has its own RETURN instruction. But note that the code generated is sensitive to its context. For example, if we put the same expression inside a begin expression, we get something quite different:

```
> (comp-show '(begin (if p (+ x y) (* x y)) z))
      ARGS  0
      GVAR  Z
      RETURN
```

What happens here is that $(+ x y)$ and $(* x y)$, when compiled in a context where the value is ignored, both result in no generated code. Thus, the if expression reduces to $(if p nil nil)$, which is compiled like $(begin p nil)$, which also generates no code when not evaluated for value, so the final code just references z . The compiler can only do this optimization because it knows that $+$ and $*$ are side-effect-free operations. Consider what happens when we replace $+$ with f :

```
> (comp-show '(begin (if p (f x) (* x x)) z))
      ARGS  0
      GVAR  P
      FJUMP L2
      SAVE  K1
      GVAR  X
      GVAR  F
      CALLJ 1
K1:   POP
L2:   GVAR  Z
      RETURN
```

Here we have to call $(f x)$ if p is true (and then throw away the value returned), but we don't have to compute $(* x x)$ when p is false.

These examples have inadvertently revealed some of the structure of `comp-funcall`, which handles five cases. First, it knows some primitive functions that have corresponding instructions and compiles these instructions inline when their values are needed. If the values are not needed, then the function can be ignored, and just the arguments can be compiled. This assumes true functions with no side effects. If there are primitive operations with side effects, they too can be compiled inline, but the operation can never be ignored. The next case is when the function is a lambda expression of no arguments. We can just compile the body of the lambda expression as if it were a begin expression. Nonprimitive functions require a function call. There are two cases: when there is more to compile we have to save a continuation

point, and when we are compiling the final value of a function, we can just branch to the called function. The whole thing looks like this:

```
(defun comp-funcall (f args env val? more?)
  "Compile an application of a function to arguments."
  (let ((prim (primitive-p f env (length args))))
    (cond
      (prim ; function compilable to a primitive instruction
        (if (and (not val?) (not (prim-side-effects prim)))
            ;; Side-effect free primitive when value unused
            (comp-begin args env nil more?)
            ;; Primitive with value or call needed
            (seq (comp-list args env)
                 (gen (prim-opcode prim))
                 (unless val? (gen 'POP))
                 (unless more? (gen 'RETURN))))))
      ((and (starts-with f 'lambda) (null (second f)))
        ;; ((lambda () body)) => (begin body)
        (assert (null args) () "Too many arguments supplied")
        (comp-begin (rest2 f) env val? more?))
      (more? ; Need to save the continuation point
        (let ((k (gen-label 'k)))
          (seq (gen 'SAVE k)
                (comp-list args env)
                (comp f env t t)
                (gen 'CALLJ (length args))
                (list k)
                (if (not val?) (gen 'POP))))))
      (t ; function call as rename plus goto
        (seq (comp-list args env)
              (comp f env t t)
              (gen 'CALLJ (length args))))))
```

The support for primitives is straightforward. The `prim` data type has five slots. The first holds the name of a symbol that is globally bound to a primitive operation. The second, `n-args`, is the number of arguments that the primitive requires. We have to take into account the number of arguments to each function because we want `(+ x y)` to compile into a primitive addition instruction, while `(+ x y z)` should not. It will compile into a call to the `+` function instead. The `opcode` slot gives the opcode that is used to implement the primitive. The `always` field is true if the primitive always returns non-nil, false if it always returns nil, and nil otherwise. It is used in exercise 23.6. Finally, the `side-effects` field says if the function has any side effects, like doing I/O or changing the value of an object.

```

(defstruct (prim (:type list))
  symbol n-args opcode always side-effects)

(defparameter *primitive-fns*
  '((+ 2 + true) (- 2 - true) (* 2 * true) (/ 2 / true)
    (< 2 <) (> 2 >) (<= 2 <=) (>= 2 >=) (/= 2 /=) (= 2 =)
    (eq? 2 eq) (equal? 2 equal) (equiv? 2 eql)
    (not 1 not) (null? 1 not)
    (car 1 car) (cdr 1 cdr) (cadr 1 cadr) (cons 2 cons true)
    (list 1 list1 true) (list 2 list2 true) (list 3 list3 true)
    (read 0 read nil t) (write 1 write nil t) (display 1 display nil t)
    (newline 0 newline nil t) (compiler 1 compiler t)
    (name! 2 name! true t) (random 1 random true nil)))

(defun primitive-p (f env n-args)
  "F is a primitive if it is in the table, and is not shadowed
  by something in the environment, and has the right number of args."
  (and (not (in-env-p f env))
        (find f *primitive-fns*
              :test #'(lambda (f prim)
                        (and (eq f (prim-symbol prim))
                             (= n-args (prim-n-args prim)))))))

(defun list1 (x) (list x))
(defun list2 (x y) (list x y))
(defun list3 (x y z) (list x y z))
(defun display (x) (princ x))
(defun newline () (terpri))

```

These optimizations only work if the symbols are permanently bound to the global values given here. We can enforce that by altering `gen-set` to preserve them as constants:

```

(defun gen-set (var env)
  "Generate an instruction to set a variable to top-of-stack."
  (let ((p (in-env-p var env)))
    (if p
        (gen 'LSET (first p) (second p) ";" var)
        (if (assoc var *primitive-fns*)
            (error "Can't alter the constant ~a" var)
            (gen 'GSET var)))))

```

Now an expression like `(+ x 1)` will be properly compiled using the `+` instruction rather than a subroutine call, and an expression like `(set! + *)` will be flagged as an error when `+` is a global variable, but allowed when it has been locally bound. However, we still need to be able to handle expressions like `(set! add +)` and then `(add x y)`. Thus, we need some function object that `+` will be globally bound to, even if the compiler normally optimizes away references to that function. The function `init-scheme-comp` takes care of this requirement:

```
(defun init-scheme-comp ()
  "Initialize the primitive functions."
  (dolist (prim *primitive-fns*)
    (setf (get (prim-symbol prim) 'global-val)
          (new-fn :env nil :name (prim-symbol prim)
                  :code (seq (gen 'PRIM (prim-symbol prim))
                              (gen 'RETURN))))))
```

There is one more change to make—rewriting `comp-lambda`. We still need to get the arguments off the stack, but we no longer generate a `RETURN` instruction, since that is done by `comp-begin`, if necessary. At this point we'll provide a hook for a peephole optimizer, which will be introduced in section 23.4, and for an assembler to convert the assembly language to machine code. `new-fn` provides this interface, but for now, `new-fn` acts just like `make-fn`.

We also need to account for the possibility of rest arguments in a lambda list. A new function, `gen-args`, generates the single instruction to load the arguments of the stack. It introduces a new instruction, `ARGS.`, into the abstract machine. This instruction works just like `ARGS`, except it also conses any remaining arguments on the stack into a list and stores that list as the value of the rest argument. With this innovation, the new version of `comp-lambda` looks like this:

```

(defun comp-lambda (args body env)
  "Compile a lambda form into a closure with compiled code."
  (new-fn :env env :args args
         :code (seq (gen-args args 0)
                    (comp-begin body
                               (cons (make-true-list args) env)
                               t nil))))

(defun gen-args (args n-so-far)
  "Generate an instruction to load the arguments."
  (cond ((null args) (gen 'ARGS n-so-far))
        ((symbolp args) (gen 'ARGS. n-so-far))
        ((and (consp args) (symbolp (first args))))
         (gen-args (rest args) (+ n-so-far 1)))
        (t (error "Illegal argument list"))))

(defun make-true-list (dotted-list)
  "Convert a possibly dotted list into a true, non-dotted list."
  (cond ((null dotted-list) nil)
        ((atom dotted-list) (list dotted-list))
        (t (cons (first dotted-list)
                  (make-true-list (rest dotted-list))))))

(defun new-fn (&key code env name args)
  "Build a new function."
  (assemble (make-fn :env env :name name :args args
                   :code (optimize code))))

```

`new-fn` includes calls to an assembler and an optimizer to generate actual machine code. For the moment, both will be identity functions:

```

(defun optimize (code) code)
(defun assemble (fn) fn)

```

Here are some more examples of the compiler at work:

```

> (comp-show '(if (null? (car l)) (f (+ (* a x) b))
              (g (/ x 2))))
      ARGS    0
      GVAR    L
      CAR
      FJUMP   L1
      GVAR    X
      2
      /
      GVAR    G
      CALLJ   1
L1:   GVAR    A

```



```

GVAR  X
*
GVAR  B
+
GVAR  F
CALLJ 1

```

There is no need to save any continuation points in this code, because the only calls to nonprimitive functions occur as the final values of the two branches of the function.

```

> (comp-show '(define (last1 l)
              (if (null? (cdr l)) (car l)
                  (last1 (cdr l)))))
ARGS  0
FN
  ARGS  1
  LVAR  0    0    ;    L
  CDR
  FJUMP L1
  LVAR  0    0    ;    L
  CDR
  GVAR  LAST1
  CALLJ 1
L1:   LVAR  0    0    ;    L
      CAR
      RETURN
GSET  LAST1
CONST LAST1
NAME!
RETURN

```

The top-level function just assigns the nested function to the global variable `last1`. Since `last1` is tail-recursive, it has only one return point, for the termination case, and just calls itself without saving continuations until that case is executed.

Contrast that to the non-tail-recursive definition of `length` below. It is not tail-recursive because before it calls `length` recursively, it must save a continuation point, `K1`, so that it will know where to return to to add 1.

```

> (comp-show '(define (length l)
              (if (null? l) 0 (+ 1 (length (cdr l))))))
  ARGS  0
  FN
    ARGS  1
    LVAR  0      0      ;      L
    FJUMP L2
    1
    SAVE  K1
    LVAR  0      0      ;      L
    CDR
    GVAR  LENGTH
    CALLJ 1
K1:    +
    RETURN
L2:    0
    RETURN
    GSET  LENGTH
    CONST LENGTH
    NAME!
    RETURN

```

Of course, it is possible to write length in tail-recursive fashion:

```

> (comp-show '(define (length l)
              (letrec ((len (lambda (l n)
                              (if (null? l) n
                                  (len (rest l) (+ n 1))))))
                (len l 0))))
  ARGS  0
  FN
    ARGS  1
    NIL
    FN
      ARGS  1
      FN
        ARGS  2
        LVAR  0      0      ;      L
        FJUMP L2
        SAVE  K1
        LVAR  0      0      ;      L
        GVAR  REST
        CALLJ 1
K1:    LVAR  0      1      ;      N
        1
        +
        LVAR  1      0      ;      LEN

```

```

                                CALLJ 2
L2:                                LVAR 0 1 ; N
                                RETURN
                                LSET 0 0 ; LEN
                                POP
                                LVAR 1 0 ; L
                                0
                                LVAR 0 0 ; LEN
                                CALLJ 2
                                CALLJ 1
GSET LENGTH
CONST LENGTH
NAME!
RETURN

```

Let's look once again at an example with nested conditionals:

```

> (comp-show '(if (not (and p q (not r))) x y))
  ARGS 0
  GVAR P
  FJUMP L3
  GVAR Q
  FJUMP L1
  GVAR R
  NOT
  JUMP L2
L1:  NIL
L2:  JUMP L4
L3:  NIL
L4:  FJUMP L5
     GVAR Y
     RETURN
L5:  GVAR X
     RETURN

```

Here the problem is with multiple JUMPs and with not recognizing negation. If *p* is false, then the *and* expression is false, and the whole predicate is true, so we should return *x*. The code does in fact return *x*, but it first jumps to L3, loads NIL, and then does an FJUMP that will always jump to L5. Other branches have similar inefficiencies. A sufficiently clever compiler should be able to generate the following code:

```

        ARGS  0
        GVAR  P
        FJUMP L1
        GVAR  Q
        FJUMP L1
        GVAR  R
        TJUMP L1
        GVAR  Y
        RETURN
L1:    GVAR  X
        RETURN

```

23.2 Introducing Call/cc

Now that the basic compiler works, we can think about how to implement `call/cc` in our compiler. First, remember that `call/cc` is a normal function, not a special form. So we could define it as a primitive, in the manner of `car` and `cons`. However, primitives as they have been defined only get to see their arguments, and `call/cc` will need to see the run-time stack, in order to save away the current continuation. One choice is to install `call/cc` as a normal Scheme nonprimitive function but to write its body in assembly code ourselves. We need to introduce one new instruction, `CC`, which places on the stack a function (to which we also have to write the assembly code by hand) that saves the current continuation (the stack) in its environment, and, when called, fetches that continuation and installs it, by setting the stack back to that value. This requires one more instruction, `SET-CC`. The details of this, and of all the other instructions, are revealed in the next section.

23.3 The Abstract Machine

So far we have defined the instruction set of a mythical abstract machine and generated assembly code for that instruction set. It's now time to actually execute the assembly code and hence have a useful compiler. There are several paths we could pursue: we could implement the machine in hardware, software, or microcode, or we could translate the assembly code for our abstract machine into the assembly code of some existing machine. Each of these approaches has been taken in the past.

Hardware. If the abstract machine is simple enough, it can be implemented directly in hardware. The Scheme-79 and Scheme-81 Chips (Steele and Sussman 1980; Batali et al. 1982) were VLSI implementations of a machine designed specifically to run Scheme.

Macro-Assembler. In the translation or macro-assembler approach, each instruction in the abstract machine language is translated into one or more instructions in the host computer's instruction set. This can be done either directly or by generating assembly code and passing it to the host computer's assembler. In general this will lead to code expansion, because the host computer probably will not provide direct support for Scheme's data types. Thus, whereas in our abstract machine we could write a single instruction for addition, with native code we might have to execute a series of instructions to check the type of the arguments, do an integer add if they are both integers, a floating-point add if they are both floating-point numbers, and so on. We might also have to check the result for overflow, and perhaps convert to bignum representation. Compilers that generate native code often include more sophisticated data-flow analysis to know when such checks are required and when they can be omitted.

Microcode. The MIT Lisp Machine project, unlike the Scheme Chip, actually resulted in working machines. One important decision was to go with microcode instead of a single chip. This made it easy to change the system as experienced was gained, and as the host language was changed from ZetaLisp to Common Lisp. The most important architectural feature of the Lisp Machine was the inclusion of tag bits on each word to specify data types. Also important was microcode to implement certain frequently used generic operations. For example, in the Symbolics 3600 Lisp Machine, the microcode for addition simultaneously did an integer add, a floating-point add, and a check of the tag bits. If both arguments turned out to be either integers or floating-point numbers, then the appropriate result was taken. Otherwise, a trap was signaled, and a conversion routine was entered. This approach makes the compiler relatively simple, but the trend in architecture is away from highly microcoded processors toward simpler (RISC) processors.

Software. We can remove many of these problems with a technique known as *byte-code assembly*. Here we translate the instructions into a vector of bytes and then interpret the bytes with a byte-code interpreter. This gives us (almost) the machine we want; it solves the code expansion problem, but it may be slower than native code compilation, because the byte-code interpreter is written in software, not hardware or microcode.

Each opcode is a single byte (we have less than 256 opcodes, so this will work). The instructions with arguments take their arguments in the following bytes of the instruction stream. So, for example, a CALL instruction occupies two bytes; one for the opcode and one for the argument count. This means we have imposed a limit of 256 arguments to a function call. An LVAR instruction would take three bytes; one for the opcode, one for the frame offset, and one for the offset within the frame. Again, we have imposed 256 as the limit on nesting level and variables per frame. These limits seem high enough for any code written by a human, but remember, not only humans write code. It is possible that some complex macro may expand into something with more than 256 variables, so a full implementation would have

some way of accounting for this. The GVAR and CONST instructions have to refer to an arbitrary object; either we can allocate enough bytes to fit a pointer to this object, or we can add a constants field to the fn structure, and follow the instructions with a single-byte index into this vector of constants. This latter approach is more common.

We can now handle branches by changing the program counter to an index into the code vector. (It seems severe to limit functions to 256 bytes of code; a two-byte label allows for 65536 bytes of code per function.) In summary, the code is more compact, branching is efficient, and dispatching can be fast because the opcode is a small integer, and we can use a branch table to go to the right piece of code for each instruction.

Another source of inefficiency is implementing the stack as a list, and consing up new cells every time something is added to the stack. The alternative is to implement the stack as a vector with a fill-pointer. That way a push requires no consing, only a change to the pointer (and a check for overflow). The check is worthwhile, however, because it allows us to detect infinite loops in the user's code.

Here follows an assembler that generates a sequence of instructions (as a vector). This is a compromise between byte codes and the assembly language format. First, we need some accessor functions to get at parts of an instruction:

```
(defun opcode (instr) (if (label-p instr) :label (first instr)))
(defun args (instr) (if (listp instr) (rest instr)))
(defun arg1 (instr) (if (listp instr) (second instr)))
(defun arg2 (instr) (if (listp instr) (third instr)))
(defun arg3 (instr) (if (listp instr) (fourth instr)))

(defsetf arg1 (instr) (val) '(setf (second ,instr) ,val))
```

Now we write the assembler, which already is integrated into the compiler with a hook in new-fn.

```
(defun assemble (fn)
  "Turn a list of instructions into a vector."
  (multiple-value-bind (length labels)
    (asm-first-pass (fn-code fn))
    (setf (fn-code fn)
          (asm-second-pass (fn-code fn)
                           length labels))
    fn))

(defun asm-first-pass (code)
  "Return the labels and the total code length."
  (let ((length 0)
        (labels nil))
    (dolist (instr code)
      (if (label-p instr)
```

```

        (push (cons instr length) labels)
        (incf length)))
(values length labels)))

(defun asm-second-pass (code length labels)
  "Put code into code-vector, adjusting for labels."
  (let ((addr 0)
        (code-vector (make-array length)))
    (dolist (instr code)
      (unless (label-p instr)
        (if (is instr '(JUMP TJUMP FJUMP SAVE))
            (setf (arg1 instr)
                  (cdr (assoc (arg1 instr) labels))))
          (setf (aref code-vector addr) instr)
              (incf addr)))
      code-vector))

```

If we want to be able to look at assembled code, we need a new printing function:

```

(defun show-fn (fn &optional (stream *standard-output*) (indent 2))
  "Print all the instructions in a function.
  If the argument is not a function, just princ it,
  but in a column at least 8 spaces wide."
  ;; This version handles code that has been assembled into a vector
  (if (not (fn-p fn))
      (format stream "~8a" fn)
      (progn
        (fresh-line)
        (dotimes (i (length (fn-code fn)))
          (let ((instr (elt (fn-code fn) i)))
            (if (label-p instr)
                (format stream "~a:" instr)
                (progn
                  (format stream "~VT~2d: " indent i)
                  (dolist (arg instr)
                    (show-fn arg stream (+ indent 8)))
                  (fresh-line))))))))))

```

```

(defstruct ret-addr fn pc env)

```

```

(defun is (instr op)
  "True if instr's opcode is OP, or one of OP when OP is a list."
  (if (listp op)
      (member (opcode instr) op)
      (eq (opcode instr) op)))

```

```

(defun top (stack) (first stack))

```

```

(defun machine (f)
  "Run the abstract machine on the code for f."
  (let* ((code (fn-code f))
         (pc 0)
         (env nil)
         (stack nil)
         (n-args 0)
         (instr))
    (loop
      (setf instr (elt code pc))
      (incf pc)
      (case (opcode instr)

        ;; Variable/stack manipulation instructions:
        (LVAR (push (elt (elt env (arg1 instr)) (arg2 instr))
                    stack))
        (LSET (setf (elt (elt env (arg1 instr)) (arg2 instr))
                    (top stack)))
        (GVAR (push (get (arg1 instr) 'global-val) stack))
        (GSET (setf (get (arg1 instr) 'global-val) (top stack)))
        (POP (pop stack))
        (CONST (push (arg1 instr) stack))

        ;; Branching instructions:
        (JUMP (setf pc (arg1 instr)))
        (FJUMP (if (null (pop stack)) (setf pc (arg1 instr))))
        (TJUMP (if (pop stack) (setf pc (arg1 instr))))

        ;; Function call/return instructions:
        (SAVE (push (make-ret-addr :pc (arg1 instr)
                                  :fn f :env env)
                    stack))
        (RETURN ;; return value is top of stack; ret-addr is second
                (setf f (ret-addr-fn (second stack))
                      code (fn-code f)
                      env (ret-addr-env (second stack))
                      pc (ret-addr-pc (second stack)))
                ;; Get rid of the ret-addr, but keep the value
                (setf stack (cons (first stack) (rest2 stack))))
        (CALLJ (pop env) ; discard the top frame
                (setf f (pop stack)
                      code (fn-code f)
                      env (fn-env f)
                      pc 0
                      n-args (arg1 instr)))
        (ARGS (assert (= n-args (arg1 instr)) ()))

```



```

        "Wrong number of arguments:~
        ~d expected, ~d supplied"
        (arg1 instr) n-args)
      (push (make-array (arg1 instr)) env)
      (loop for i from (- n-args 1) downto 0 do
        (setf (elt (first env) i) (pop stack))))
(ARGS. (assert (>= n-args (arg1 instr)) ()
  "Wrong number of arguments:~
  ~d or more expected, ~d supplied"
  (arg1 instr) n-args)
  (push (make-array (+ 1 (arg1 instr))) env)
  (loop repeat (- n-args (arg1 instr)) do
    (push (pop stack) (elt (first env) (arg1 instr))))
  (loop for i from (- (arg1 instr) 1) downto 0 do
    (setf (elt (first env) i) (pop stack))))
(FN   (push (make-fn :code (fn-code (arg1 instr))
  :env env) stack))
(PRIM (push (apply (arg1 instr)
  (loop with args = nil repeat n-args
    do (push (pop stack) args)
    finally (return args)))
  stack))

;; Continuation instructions:
(SET-CC (setf stack (top stack)))
(CC     (push (make-fn
  :env (list (vector stack))
  :code '((ARGS 1) (LVAR 1 0 ";" stack) (SET-CC)
  (LVAR 0 0) (RETURN)))
  stack))

;; Nullary operations:
((SCHEME-READ NEWLINE)
 (push (funcall (opcode instr)) stack))

;; Unary operations:
((CAR CDR CADR NOT LIST1 COMPILER DISPLAY WRITE RANDOM)
 (push (funcall (opcode instr) (pop stack)) stack))

;; Binary operations:
((+ - * / < > <= >= /= = CONS LIST2 NAME! EQ EQUAL EQL)
 (setf stack (cons (funcall (opcode instr) (second stack)
  (first stack))
  (rest2 stack))))

```

```

;; Ternary operations:
(LIST3
  (setf stack (cons (funcall (opcode instr) (third stack)
                           (second stack) (first stack))
                   (rest3 stack))))

;; Constants:
((T NIL -1 0 1 2)
 (push (opcode instr) stack))

;; Other:
((HALT) (RETURN (top stack)))
(otherwise (error "Unknown opcode: ~a" instr))))))

(defun init-scheme-comp ()
  "Initialize values (including call/cc) for the Scheme compiler."
  (set-global-var! 'exit
    (new-fn :name 'exit :args '(val) :code '((HALT))))
  (set-global-var! 'call/cc
    (new-fn :name 'call/cc :args '(f)
            :code '((ARGS 1) (CC) (LVAR 0 0 ":" f) (CALLJ 1))))
  (dolist (prim *primitive-fns*)
    (setf (get (prim-symbol prim) 'global-val)
          (new-fn :env nil :name (prim-symbol prim)
                  :code (seq (gen 'PRIM (prim-symbol prim))
                             (gen 'RETURN))))))

```

Here's the Scheme top level. Note that it is written in Scheme itself; we compile the definition of the read-eval-print loop,¹ load it into the machine, and then start executing it. There's also an interface to compile and execute a single expression, `comp-go`.

```

(defconstant scheme-top-level
  '(begin (define (scheme)
           (newline)
           (display "=> ")
           (write ((compiler (read))))
           (scheme))
         (scheme)))

(defun scheme ()
  "A compiled Scheme read-eval-print loop"
  (init-scheme-comp)
  (machine (compiler scheme-top-level)))

```

¹Strictly speaking, this is a read-compile-funcall-write loop.

```
(defun comp-go (exp)
  "Compile and execute the expression."
  (machine (compiler '(exit ,exp))))
```

- ?** **Exercise 23.2 [m]** This implementation of the machine is wasteful in its representation of environments. For example, consider what happens in a tail-recursive function. Each ARG instruction builds a new frame and pushes it on the environment. Then each CALL pops the latest frame off the environment. So, while the stack does not grow with tail-recursive calls, the heap certainly does. Eventually, we will have to garbage-collect all those unused frames (and the cons cells used to make lists out of them). How could we avoid or limit this garbage collection?

23.4 A Peephole Optimizer

In this section we investigate a simple technique that will generate slightly better code in cases where the compiler gives inefficient sequences of instructions. The idea is to look at short sequences of instructions for prespecified patterns and replace them with equivalent but more efficient instructions.

In the following example, `comp-if` has already done some source-level optimization, such as eliminating the `(f x)` call.

```
> (comp-show '(begin (if (if t 1 (f x)) (set! x 2)) x))
0: ARGS 0
1: 1
2: FJUMP 6
3: 2
4: GSET X
5: POP
6: GVAR X
7: RETURN
```

But the generated code could be made much better. This could be done with more source-level optimizations to transform the expression into `(set! x 2)`. Alternatively, it could also be done by looking at the preceding instruction sequence and transforming local inefficiencies. The optimizer presented in this section is capable of generating the following code:

```
> (comp-show '(begin (if (if t 1 (f x)) (set! x 2)) x))
0: ARGS  0
1: 2
2: GSET  X
3: RETURN
```

The function `optimize` is implemented as a data-driven function that looks at the opcode of each instruction and makes optimizations based on the following instructions. To be more specific, `optimize` takes a list of assembly language instructions and looks at each instruction in order, trying to apply an optimization. If any changes at all are made, then `optimize` will be called again on the whole instruction list, because further changes might be triggered by the first round of changes.

```
(defun optimize (code)
  "Perform peephole optimization on assembly code."
  (let ((any-change nil))
    ;; Optimize each tail
    (loop for code-tail on code do
      (setf any-change (or (optimize-1 code-tail code)
                          any-change)))
    ;; If any changes were made, call optimize again
    (if any-change
        (optimize code)
        code)))
```

The function `optimize-1` is responsible for each individual attempt to optimize. It is passed two arguments: a list of instructions starting at the current one and going to the end of the list, and a list of all the instructions. The second argument is rarely used. The whole idea of a peephole optimizer is that it should look at only a few instructions following the current one. `optimize-1` is data-driven, based on the opcode of the first instruction. Note that the optimizer functions do their work by destructively modifying the instruction sequence, *not* by consing up and returning a new sequence.

```
(defun optimize-1 (code all-code)
  "Perform peephole optimization on a tail of the assembly code.
  If a change is made, return true."
  ;; Data-driven by the opcode of the first instruction
  (let* ((instr (first code))
         (optimizer (get-optimizer (opcode instr))))
    (when optimizer
      (funcall optimizer instr code all-code))))
```

We need a table to associate the individual optimizer functions with the opcodes. Since opcodes include numbers as well as symbols, an eql hash table is an appropriate choice:

```
(let ((optimizers (make-hash-table :test #'eql)))

  (defun get-optimizer (opcode)
    "Get the assembly language optimizer for this opcode."
    (gethash opcode optimizers))

  (defun put-optimizer (opcode fn)
    "Store an assembly language optimizer for this opcode."
    (setf (gethash opcode optimizers) fn)))
```

We could now build a table with `put-optimizer`, but it is worth defining a macro to make this a little neater:

```
(defmacro def-optimizer (opcodes args &body body)
  "Define assembly language optimizers for these opcodes."
  (assert (and (listp opcodes) (listp args) (= (length args) 3)))
  `(dolist (op ',opcodes)
    (put-optimizer op #'(lambda ,args .body))))
```

Before showing example optimizer functions, we will introduce three auxiliary functions. `gen1` generates a single instruction, `target` finds the code sequence that a jump instruction branches to, and `next-instr` finds the next actual instruction in a sequence, skipping labels.

```
(defun gen1 (&rest args) "Generate a single instruction" args)
(defun target (instr code) (second (member (arg1 instr) code)))
(defun next-instr (code) (find-if (complement #'label-p) code))
```

Here are six optimizer functions that implement a few important peephole optimizations.

```
(def-optimizer (:LABEL) (instr code all-code)
  ;; ... L ... => ... ... ;if no reference to L
  (when (not (find instr all-code :key #'arg1))
    (setf (first code) (second code)
          (rest code) (rest2 code))
    t))
```

```

(def-optimizer (GSET LSET) (instr code all-code)
  ;; ex: (begin (set! x y) (if x z))
  ;; (SET X) (POP) (VAR X) ==> (SET X)
  (when (and (is (second code) 'POP)
             (is (third code) '(GVAR LVAR))
             (eq (arg1 instr) (arg1 (third code)))))
    (setf (rest code) (nthcdr 3 code))
    t))

(def-optimizer (JUMP CALL CALLJ RETURN) (instr code all-code)
  ;; (JUMP L1) ...dead code... L2 ==> (JUMP L1) L2
  (setf (rest code) (member-if #'label-p (rest code))))
  ;; (JUMP L1) ... L1 (JUMP L2) ==> (JUMP L2) ... L1 (JUMP L2)
  (when (and (is instr 'JUMP)
             (is (target instr code) '(JUMP RETURN))
             (setf (first code) (copy-list (target instr code)))))
    t)))

(def-optimizer (TJUMP FJUMP) (instr code all-code)
  ;; (FJUMP L1) ... L1 (JUMP L2) ==> (FJUMP L2) ... L1 (JUMP L2)
  (when (is (target instr code) 'JUMP)
    (setf (second instr) (arg1 (target instr code)))
    t))

(def-optimizer (T -1 0 1 2) (instr code all-code)
  (case (opcode (second code))
    (NOT ;; (T) (NOT) ==> NIL
     (setf (first code) (gen1 'NIL)
           (rest code) (rest2 code))
     t)
    (FJUMP ;; (T) (FJUMP L) ... => ...
     (setf (first code) (third code)
           (rest code) (rest3 code))
     t)
    (TJUMP ;; (T) (TJUMP L) ... => (JUMP L) ...
     (setf (first code) (gen1 'JUMP (arg1 (next-instr code)))))
    t)))

```

```

(def-optimizer (NIL) (instr code all-code)
  (case (opcode (second code))
    (NOT ;; (NIL) (NOT) ==> T
     (setf (first code) (gen1 'T)
           (rest code) (rest2 code))
     t)
    (TJUMP ;; (NIL) (TJUMP L) ... => ...
     (setf (first code) (third code)
           (rest code) (rest3 code))
     t)
    (FJUMP ;; (NIL) (FJUMP L) ==> (JUMP L)
     (setf (first code) (gen1 'JUMP (arg1 (next-instr code))))
     t)))

```

23.5 Languages with Different Lexical Conventions

This chapter has shown how to evaluate a language with Lisp-like syntax, by writing a read-eval-print loop where only the `eval` needs to be replaced. In this section we see how to make the read part slightly more general. We still read Lisp-like syntax, but the lexical conventions can be slightly different.

The Lisp function `read` is driven by an object called the *readtable*, which is stored in the special variable `*readtable*`. This table associates some action to take with each of the possible characters that can be read. The entry in the readtable for the character `#\()`, for example, would be directions to read a list. The entry for `#\;` would be directions to ignore every character up to the end of the line.

Because the readtable is stored in a special variable, it is possible to alter completely the way read works just by dynamically rebinding this variable.

The new function `scheme-read` temporarily changes the readtable to a new one, the Scheme readtable. It also accepts an optional argument, the stream to read from, and it returns a special marker on end of file. This can be tested for with the predicate `eof-object?`. Note that once `scheme-read` is installed as the value of the Scheme symbol `read` we need do no more—`scheme-read` will always be called when appropriate (by the top level of Scheme, and by any user Scheme program).

```

(defconstant eof "EoF")
(defun eof-object? (x) (eq x eof))
(defvar *scheme-readtable* (copy-readtable))

```

```
(defun scheme-read (&optional (stream *standard-input*))
  (let ((*readtable* *scheme-readtable*))
    (read stream nil eof)))
```

The point of having a special eof constant is that it is unforgeable. The user cannot type in a sequence of characters that will be read as something eq to eof. In Common Lisp, but not Scheme, there is an escape mechanism that makes eof forgable. The user can type #.eof to get the effect of an end of file. This is similar to the ^D convention in UNIX systems, and it can be quite handy.

So far the Scheme readtable is just a copy of the standard readtable. The next step in implementing scheme-read is to alter *scheme-readtable*, adding read macros for whatever characters are necessary. Here we define macros for #t and #f (the true and false values), for #d (decimal numbers) and for the backquote read macro (called quasiquote in Scheme). Note that the backquote and comma characters are defined as read macros, but the @ in ,@ is processed by reading the next character, not by a read macro on @.

```
(set-dispatch-macro-character #\# #\t
  #'(lambda (&rest ignore) t)
  *scheme-readtable*)

(set-dispatch-macro-character #\# #\f
  #'(lambda (&rest ignore) nil)
  *scheme-readtable*)

(set-dispatch-macro-character #\# #\d
  ;; In both Common Lisp and Scheme,
  ;; #x, #o and #b are hexadecimal, octal, and binary,
  ;; e.g. #xff = #o377 = #b11111111 = 255
  ;; In Scheme only, #d255 is decimal 255.
  #'(lambda (stream &rest ignore)
    (let ((*read-base* 10)) (scheme-read stream)))
  *scheme-readtable*)

(set-macro-character #\'
  #'(lambda (s ignore) (list 'quasiquote (scheme-read s)))
  nil *scheme-readtable*)

(set-macro-character #\,
  #'(lambda (stream ignore)
    (let ((ch (read-char stream)))
      (if (char= ch #\@)
          (list 'unquote-splicing (read stream))
          (progn (unread-char ch stream)
                 (list 'unquote (read stream))))))
  nil *scheme-readtable*)
```

Finally, we install scheme-read and eof-object? as primitives:


```
(defparameter *primitive-fns*
  '((+ 2 + true nil) (- 2 - true nil) (* 2 * true nil) (/ 2 / true nil)
    (< 2 < nil nil) (> 2 > nil nil) (<= 2 <= nil nil) (>= 2 >= nil nil)
    (/= 2 /= nil nil) (= 2 = nil nil)
    (eq? 2 eq nil nil) (equal? 2 equal nil nil) (equiv? 2 eql nil nil)
    (not 1 not nil nil) (null? 1 not nil nil) (cons 2 cons true nil)
    (car 1 car nil nil) (cdr 1 cdr nil nil) (cadr 1 cadr nil nil)
    (list 1 list1 true nil) (list 2 list2 true nil) (list 3 list3 true nil)
    (read 0 read nil t) (write 1 write nil t) (display 1 display nil t)
    (newline 0 newline nil t) (compiler 1 compiler t nil)
    (name! 2 name! true t) (random 1 random true nil)))
```

Here we test `scheme-read`. The characters in italics were typed as a response to the `scheme-read`.

```
> (scheme-read) #t
T
> (scheme-read) #f
NIL
> (scheme-read) '(a,b,@c d)
(QUASIQUOTE (A (UNQUOTE B) (UNQUOTE-SPLICING C) D))
```

The final step is to make `quasiquote` a macro that expands into the proper sequence of calls to `cons`, `list`, and `append`. The careful reader will keep track of the difference between the form returned by `scheme-read` (something starting with `quasiquote`), the expansion of this form with the Scheme macro `quasiquote` (which is implemented with the Common Lisp function `quasi-q`), and the eventual evaluation of the expansion. In an environment where `b` is bound to the number 2 and `c` is bound to the list (`c1 c2`), we might have:

Typed:	'(a ,b ,@c d)
Read:	(quasiquote (a (unquote b) (unquote-splicing c) d))
Expanded:	(cons 'a (cons b (append c '(d))))
Evaluated:	(a 2 c1 c2 d)

The implementation of the `quasiquote` macro is modeled closely on the one given in Charniak et al.'s *Artificial Intelligence Programming*. I added support for vectors. In `combine-quasiquote` I add the trick of reusing the old `cons` cell `x` rather than consing together `left` and `right` when that is possible. However, the implementation still wastes `cons` cells—a more efficient version would pass back multiple values rather than consing quote onto a list, only to strip it off again.

```

(setf (scheme-macro 'quasiquote) 'quasi-q)

(defun quasi-q (x)
  "Expand a quasiquote form into append, list, and cons calls."
  (cond
    ((vectorp x)
     (list 'apply 'vector (quasi-q (coerce x 'list))))
    ((atom x)
     (if (constantp x) x (list 'quote x)))
    ((starts-with x 'unquote)
     (assert (and (rest x) (null (rest2 x))))
     (second x))
    ((starts-with x 'quasiquote)
     (assert (and (rest x) (null (rest2 x))))
     (quasi-q (quasi-q (second x))))
    ((starts-with (first x) 'unquote-splicing)
     (if (null (rest x))
         (second (first x))
         (list 'append (second (first x)) (quasi-q (rest x)))))
    (t (combine-quasiquote (quasi-q (car x))
                           (quasi-q (cdr x))
                           x))))

(defun combine-quasiquote (left right x)
  "Combine left and right (car and cdr), possibly re-using x."
  (cond ((and (constantp left) (constantp right))
         (if (and (eql (eval left) (first x))
                  (eql (eval right) (rest x)))
             (list 'quote x)
             (list 'quote (cons (eval left) (eval right)))))
        ((null right) (list 'list left))
        ((starts-with right 'list)
         (list* 'list left (rest right)))
        (t (list 'cons left right))))

```

Actually, there is a major problem with the quasiquote macro, or more accurately, in the entire approach to macro-expansion based on textual substitution. Suppose we wanted a function that acted like this:

```

> (extrema '(3 1 10 5 20 2))
((max 20) (min 1))

```

We could write the Scheme function:

```
(define (extrema list)
  ;; Given a list of numbers, return an a-list
  ;; with max and min values
  '((max ,(apply max list)) (min ,(apply min list))))
```

After expansion of the quasiquote, the definition of `extrema` will be:

```
(define extrema
  (lambda (list)
    (list (list 'max (apply max list))
          (list 'min (apply min list)))))
```

The problem is that `list` is an argument to the function `extrema`, and the argument shadows the global definition of `list` as a function. Thus, the function will fail. One way around this dilemma is to have the macro-expansion use the global value of `list` rather than the symbol `list` itself. In other words, replace the `'list` in `quasi-q` with `(get-global-var 'list)`. Then the expansion can be used even in an environment where `list` is locally bound. One has to be careful, though: if this tack is taken, then `comp-funcall` should be changed to recognize function constants, and to do the right thing with respect to primitives.

It is problems like these that made the designers of Scheme admit that they don't know the best way to specify macros, so there is no standard macro definition mechanism in Scheme. Such problems rarely come up in Common Lisp because functions and variables have different name spaces, and because local function definitions (with `flet` or `labels`) are not widely used. Those who do define local functions tend not to use already established names like `list` and `append`.

23.6 History and References

Guy Steele's 1978 MIT master's thesis on the language Scheme, rewritten as Steele 1983, describes an innovative and influential compiler for Scheme, called RABBIT.² A good article on an "industrial-strength" Scheme compiler based on this approach is described in Kranz et al.'s 1986 paper on ORBIT, the compiler for the T dialect of Scheme.

Abelson and Sussman's *Structure and Interpretation of Computer Programs* (1985) contains an excellent chapter on compilation, using slightly different techniques and compiling into a somewhat more confusing machine language. Another good text

²At the time, the MacLisp compiler dealt with something called "lisp assembly code" or LAP. The function to input LAP was called `lapin`. Those who know French will get the pun.

is John Allen's *Anatomy of Lisp* (1978). It presents a very clear, simple compiler, although it is for an older, dynamically scoped dialect of Lisp and it does not address tail-recursion or `call/cc`.

The peephole optimizer described here is based on the one in Masinter and Deutsch 1980.

23.7 Exercises

? **Exercise 23.3 [h]** Scheme's syntax for numbers is slightly different from Common Lisp's. In particular, complex numbers are written like `3+4i` rather than `#c(3 4)`. How could you make `scheme-read` account for this?

? **Exercise 23.4 [m]** Is it possible to make the core Scheme language even smaller, by eliminating any of the five special forms (`quote`, `begin`, `set!`, `if`, `lambda`) and replacing them with macros?

? **Exercise 23.5 [m]** Add the ability to recognize internal `defines` (see page 779).

? **Exercise 23.6 [h]** In `comp-if` we included a special case for `(if t x y)` and `(if nil x y)`. But there are other cases where we know the value of the predicate. For example, `(if (* a b) x y)` can also reduce to `x`. Arrange for these optimizations to be made. Note the `prim-always` field of the `prim` structure has been provided for this purpose.

? **Exercise 23.7 [m]** Consider the following version of the quicksort algorithm for sorting a vector:

```
(define (sort-vector vector test)
  (define (sort lo hi)
    (if (>= lo hi)
        vector
        (let ((pivot (partition vector lo hi test)))
          (sort lo pivot)
          (sort (+ pivot 1) hi))))
    (sort 0 (- (vector-length vector) 1))))
```

Here the function `partition` takes a vector, two indices into the vector, and a comparison function, `test`. It modifies the vector and returns an index, `pivot`, such that all elements of the vector below `pivot` are less than all elements at `pivot` or above.

It is well known that quicksort takes time proportional to $n \log n$ to sort a vector of n elements, if the pivots are chosen well. With poor pivot choices, it can take time proportional to n^2 .

The question is, what is the space required by quicksort? Besides the vector itself, how much additional storage must be temporarily allocated to sort a vector?

Now consider the following modified version of quicksort. What time and space complexity does it have?

```
(define (sort-vector vector test)
  (define (sort lo hi)
    (if (>= lo hi)
        vector
        (let ((pivot (partition vector lo hi)))
          (if (> (- hi pivot) (- pivot lo))
              (begin (sort lo pivot)
                     (sort (+ pivot 1) hi))
              (begin (sort (+ pivot 1) hi)
                     (sort lo pivot))))))
    (sort 0 (- (vector-length vector) 1))))
```

The next three exercises describe extensions that are not part of the Scheme standard.

? **Exercise 23.8 [h]** The `set!` special form is defined only when its first argument is a symbol. Extend `set!` to work like `setf` when the first argument is a list. That is, `(set! (car x) y)` should expand into something like `((setter car) y x)`, where `(setter car)` evaluates to the primitive procedure `set-car!`. You will need to add some new primitive functions, and you should also provide a way for the user to define new `set!` procedures. One way to do that would be with a `setter` function for `set!`, for example:

```
(set! (setter third)
      (lambda (val list) (set-car! (cdr (cdr list)) val)))
```

? **Exercise 23.9 [m]** It is a curious asymmetry of Scheme that there is a special notation for lambda expressions within `define` expressions, but not within `let`. Thus, we see the following:

```
(define square (lambda (x) (* x x)))      ; is the same as
(define (square x) (* x x))
```

```
(let ((square (lambda (x) (* x x)))) ...) ; is not the same as
(let (((square x) (* x x)) ...)          ; ← illegal!
```

Do you think this last expression should be legal? If so, modify the macros for `let`, `let*`, and `letrec` to allow the new syntax. If not, explain why it should not be included in the language.

? **Exercise 23.10 [m]** Scheme does not define `funcall`, because the normal function-call syntax does the work of `funcall`. This suggests two problems. (1) Is it possible to define `funcall` in Scheme? Show a definition or explain why there can't be one. Would you ever have reason to use `funcall` in a Scheme program? (2) Scheme does define `apply`, as there is no syntax for an application. One might want to extend the syntax to make `(+ . numbers)` equivalent to `(apply + numbers)`. Would this be a good idea?

? **Exercise 23.11 [d]** Write a compiler that translates Scheme to Common Lisp. This will involve changing the names of some procedures and special forms, figuring out a way to map Scheme's single name space into Common Lisp's distinct function and variable name spaces, and dealing with Scheme's continuations. One possibility is to translate a `call/cc` into a `catch` and `throw`, and disallow dynamic continuations.

23.8 Answers

Answer 23.2 We can save frames by making a resource for frames, as was done on page 337. Unfortunately, we can't just use the `defresource` macro as is, because we need a separate resource for each size frame. Thus, a two-dimensional array or a vector of vectors is necessary. Furthermore, one must be careful in determining when a frame is no longer needed, and when it has been saved and may be used again. Some compilers will generate a special calling sequence for a tail-recursive call where the environment can be used as is, without discarding and then creating a new frame for the arguments. Some compilers have varied and advanced representations for environments. An environment may never be represented explicitly as a list of frames; instead it may be represented implicitly as a series of values in registers.

Answer 23.3 We could read in Scheme expressions as before, and then convert any symbols that looked like complex numbers into numbers. The following routines do this without consing.

```
(defun scheme-read (&optional (stream *standard-input*))
  (let ((*readtable* *scheme-readtable*))
    (convert-numbers (read stream nil eof))))

(defun convert-numbers (x)
  "Replace symbols that look like Scheme numbers with their values."
  ;; Don't copy structure, make changes in place.
  (typecase x
    (cons (setf (car x) (convert-numbers (car x)))
          (setf (cdr x) (convert-numbers (cdr x)))
          x)
    (symbol (or (convert-number x) x))
    (vector (dotimes (i (length x))
                    (setf (aref x i) (convert-numbers (aref x i))))
            x)
    (t x)))

(defun convert-number (symbol)
  "If str looks like a complex number, return the number."
  (let* ((str (symbol-name symbol))
        (pos (position-if #'sign-p str))
        (end (- (length str) 1)))
    (when (and pos (char-equal (char str end) #\i))
      (let ((re (read-from-string str nil nil :start 0 :end pos))
            (im (read-from-string str nil nil :start pos :end end)))
        (when (and (numberp re) (numberp im))
          (complex re im))))))

(defun sign-p (char) (find char "+-"))
```

Actually, that's not quite good enough, because a Scheme complex number can have multiple signs in it, as in $3.4e-5+6.7e+8i$, and it need not have two numbers, as in $3i$ or $4+i$ or just $+i$. The other problem is that complex numbers can only have a lowercase i , but `read` does not distinguish between the symbols $3+4i$ and $3+4I$.

Answer 23.4 Yes, it is possible to implement `begin` as a macro:

```
(setf (scheme-macro 'begin)
      #'(lambda (&rest exps) '((lambda () .,exps))))
```

With some work we could also eliminate `quote`. Instead of `'x`, we could use `(string->symbol "X")`, and instead of `'(1 2)`, we could use something like `(list 1 2)`. The problem is in knowing when to reuse the same list. Consider:

```
=> (define (one-two) '(1 2))
ONE-TWO
=> (eq? (one-two) (one-two))
T
=> (eq? '(1 2) '(1 2))
NIL
```

A clever memoized macro for `quote` could handle this, but it would be less efficient than having `quote` as a special form. In short, what's the point?

It is also (nearly) possible to replace `if` with alternate code. The idea is to replace:

```
(if test then-part else-part)
```

with

```
(test (delay then-part) (delay else-part))
```

Now if we are assured that any `test` returns either `#t` or `#f`, then we can make the following definitions:

```
(define #t (lambda (then-part else-part) (force then-part)))
(define #f (lambda (then-part else-part) (force else-part)))
```

The only problem with this is that any value, not just `#t`, counts as true.

This seems to be a common phenomenon in Scheme compilers: translating everything into a few very general constructs, and then recognizing special cases of these constructs and compiling them specially. This has the disadvantage (compared to explicit use of many special forms) that compilation may be slower, because all macros have to be expanded first, and then special cases have to be recognized. It has the advantage that the optimizations will be applied even when the user did not have a special construct in mind. Common Lisp attempts to get the advantages of both by allowing implementations to play loose with what they implement as macros and as special forms.

Answer 23.6 We define the predicate `always` and install it in two places in `comp-if`:

```
(defun always (pred env)
  "Does predicate always evaluate to true or false?"
  (cond ((eq pred t) 'true)
        ((eq pred nil) 'false)
        ((symbolp pred) nil)
        ((atom pred) 'true)
        ((scheme-macro (first pred))
         (always (scheme-macro-expand pred) env))
        ((case (first pred)
             (QUOTE (if (null (second pred)) 'false 'true))
             (BEGIN (if (null (rest pred)) 'false
                        (always (last1 pred) env)))
             (SET! (always (third pred) env))
             (IF (let ((test (always (second pred) env))
                       (then (always (third pred) env))
                       (else (always (fourth pred) env)))
                  (cond ((eq test 'true) then)
                        ((eq test 'false) else)
                        ((eq then else) then))))
             (LAMBDA 'true)
             (t (let ((prim (primitive-p (first pred) env)
                       (length (rest pred))))
                  (if prim (prim-always prim))))))))))

(defun comp-if (pred then else env val? more?)
  (case (always pred env)
        (true      ; (if nil x y) ==> y ; ***
         (comp then env val? more?)) ; ***
        (false     ; (if t x y) ==> x   ; ***
         (comp else env val? more?)) ; ***
        (otherwise
         (let ((pcode (comp pred env t t))
               (tcode (comp then env val? more?))
               (ecode (comp else env val? more?)))
             (cond
              ((and (listp pred) ; (if (not p) x y) ==> (if p y x)
                    (length=1 (rest pred))
                    (primitive-p (first pred) env 1)
                    (eq (prim-opcode (primitive-p (first pred) env 1))
                        'not))
               (comp-if (second pred) else then env val? more?))
              ((equal tcode ecode) ; (if p x x) ==> (begin p x)
               (seq (comp pred env nil t) ecode))
              ((null tcode) ; (if p nil y) ==> p (TJUMP L2) y L2:
               (let ((L2 (gen-label)))
                 (seq pcode (gen 'TJUMP L2) ecode (list L2)))))))))
```

```

        (unless more? (gen 'RETURN))))))
((null ecode)      ; (if p x) ==> p (FJUMP L1) x L1:
  (let ((L1 (gen-label)))
    (seq pcode (gen 'FJUMP L1) tcode (list L1)
          (unless more? (gen 'RETURN))))))
(t                ; (if p x y) ==> p (FJUMP L1) x L1: y
  ; or p (FJUMP L1) x (JUMP L2) L1: y L2:
  (let ((L1 (gen-label))
        (L2 (if more? (gen-label))))
    (seq pcode (gen 'FJUMP L1) tcode
          (if more? (gen 'JUMP L2))
          (list L1) ecode (if more? (list L2))))))))))

```

Development note: originally, I had coded `always` as a predicate that took a Boolean value as input and returned true if the expression always had that value. Thus, you had to ask first if the predicate was always true, and then if it was always false. Then I realized this was duplicating much effort, and that the duplication was exponential, not just linear: for a triply-nested conditional I would have to do eight times the work, not twice the work. Thus I switched to the above formulation, where `always` is a three-valued function, returning true, false, or nil for none-of-the-above. But to demonstrate that the right solution doesn't always appear the first time, I give my original definition as well:

```

(defun always (boolean pred env)
  "Does predicate always evaluate to boolean in env?"
  (if (atom pred)
      (and (constantp pred) (equiv boolean pred))
      (case (first pred)
          (QUOTE (equiv boolean pred))
          (BEGIN (if (null (rest pred)) (equiv boolean nil)
                    (always boolean (last1 pred) env)))
          (SET! (always boolean (third pred) env))
          (IF (or (and (always t (second pred) env)
                    (always boolean (third pred) env))
                (and (always nil (second pred) env)
                    (always boolean (fourth pred) env))
                (and (always boolean (third pred) env)
                    (always boolean (fourth pred) env))))
              (LAMBDA (equiv boolean t)
                (t (let ((prim (primitive-p (first pred) env)
                       (length (rest pred))))
                    (and prim
                        (eq (prim-always prim)
                            (if boolean 'true 'false))))))))))
      (and prim
            (eq (prim-always prim)
                (if boolean 'true 'false))))))
(defun equiv (x y) "Boolean equivalence" (eq (not x) (not y)))

```

Answer 23.7 The original version requires $O(n)$ stack space for poorly chosen pivots. Assuming a properly tail-recursive compiler, the modified version will never require more than $O(\log n)$ space, because at each step at least half of the vector is being sorted tail-recursively.

Answer 23.10 (1) `(defun (funcall fn . args) (apply fn args))`
(2) Suppose you changed the piece of code `(+ . numbers)` to `(+ . (map sqrt numbers))`. The latter is the same expression as `(+ map sqrt numbers)`, which is not the intended result at all. So there would be an arbitrary restriction: the last argument in an `apply` form would have to be an atom. This kind of restriction goes against the grain of Scheme.

CHAPTER 24

ANSI Common Lisp

This chapter briefly covers some advanced features of Common Lisp that were not used in the rest of the book. The first topic, *packages*, is crucial in building large systems but was not covered in this book, since the programs are concise. The next four topics—error handling, pretty printing, series, and the loop macro—are covered in *Common Lisp the Language*, 2d edition, but not in the first edition of the book. Thus, they may not be applicable to your Lisp compiler. The final topic, *sequence functions*, shows how to write efficient functions that work for either lists or vectors.

24.1 Packages

A *package* is a symbol table that maps from strings to symbols named by those strings. When read is confronted with a sequence of characters like `list`, it uses the symbol table to determine that this refers to the symbol `list`. The important point is that every use of the symbol name `list` refers to the same symbol. That makes it easy to refer to predefined symbols, but it also makes it easy to introduce unintended name conflicts. For example, if I wanted to hook up the `emycin` expert system from chapter 16 with the parser from chapter 19, there would be a conflict because both programs use the symbol `defrule` to mean different things.

Common Lisp uses the package system to help resolve such conflicts. Instead of a single symbol table, Common Lisp allows any number of packages. The function `read` always uses the current package, which is defined to be the value of the special variable `*package*`. By default, Lisp starts out in the `common-lisp-user` package.¹ That means that if we type a new symbol, like `zxv@!?!+qw`, it will be entered into that package. Converting a string to a symbol and placing it in a package is called *interning*. It is done automatically by `read`, and can be done by the function `intern` if necessary. Name conflicts arise when there is contention for names within the `common-lisp-user` package.

To avoid name conflicts, simply create your new symbols in another package, one that is specific to your program. The easiest way to implement this is to split each system into at least two files—one to define the package that the system resides in, and the others for the system itself. For example, the `emycin` system should start with a file that defines the `emycin` package. The following form defines the `emycin` package to use the `lisp` package. That means that when the current package is `emycin`, you can still refer to all the built-in Lisp symbols.

```
(make-package "EMYCIN" :use '("LISP"))
```

The file containing the package definition should always be loaded before the rest of the system. Those files should start with the following call, which insures that all new symbols will be interned in the `emycin` package:

```
(in-package "EMYCIN")
```

Packages are used for information-hiding purposes as well as for avoiding name clashes. A distinction is made between *internal* and *external* symbols. External symbols are those that a user of a system would want to refer to, while internal symbols are those that help implement the system but are not needed by a user of the system. The symbol `rule` would probably be internal to both the `emycin` and `parser` package, but `defrule` would be external, because a user of the `emycin` system uses `defrule` to define new rules. The designer of a system is responsible for advertising which symbols are external. The proper call is:

```
(export '(emycin defrule defcontext defparm yes/no yes no is))
```

Now the user who wants to refer to symbols in the `emycin` package has four choices. First, he or she can use the *package prefix* notation. To refer to the symbol `defrule` in the `emycin` package, type `emycin:defrule`. Second, the user can make `emycin` be the current package with `(in-package "EMYCIN")`. Then, of course, we need

¹Or in the user package in non-ANSI systems.

only type `defrule`. Third, if we only need part of the functionality of a system, we can import specific symbols into the current package. For example, we could call `(import 'emycin:defrule)`. From then on, typing `defrule` (in the current package) will refer to `emycin:defrule`. Fourth, if we want the full functionality of the system, we call `(use-package "EMYCIN")`. This makes all the external symbols of the `emycin` package accessible in the current package.

While packages help eliminate name conflicts, `import` and `use-package` allow them to reappear. The advantage is that there will only be conflicts between external symbols. Since a carefully designed package should have far fewer external than internal symbols, the problem has at least been reduced. But if two packages both have an external `defrule` symbol, then we cannot `use-package` both these packages, nor `import` both symbols without producing a genuine name conflict. Such conflicts can be resolved by *shadowing* one symbol or the other; see *Common Lisp the Language* for details.

The careful reader may be confused by the distinction between "EMYCIN" and `emycin`. In *Common Lisp the Language*, it was not made clear what the argument to package functions must be. Thus, some implementations signal an error when given a symbol whose print name is a package. In ANSI Common Lisp, all package functions are specified to take either a package, a package name (a string), or a symbol whose print name is a package name. In addition, ANSI Common Lisp adds the convenient `defpackage` macro. It can be used as a replacement for separate calls to `make-package`, `use-package`, `import`, and `export`. Also note that ANSI renames the `lisp` package as `common-lisp`.

```
(defpackage emycin
  (:use common-lisp)
  (:export emycin defrule defcontext defparm yes/no yes no is))
```

For more on packages and building systems, see section 25.16 or *Common Lisp the Language*.

The Seven Name Spaces

One important fact to remember about packages is that they deal with symbols, and only indirectly deal with the uses those symbols might have. For example, you may think of `(export 'parse)` as exporting the function `parse`, but really it is exporting the symbol `parse`, which may happen to have a function definition associated with it. However, if the symbol is put to another use—perhaps as a variable or a data type—then those uses are made accessible by the `export` statement as well.

Common Lisp has at least seven name spaces. The two we think of most often are (1) for functions and macros and (2) for variables. We have seen that Scheme

conflates these two name spaces, but Common Lisp keeps them separate, so that in a function application like `(f)` the function/macro name space is consulted for the value of `f`, but in `(+ f)`, `f` is treated as a variable name. Those who understand the scope and extent rules of Common Lisp know that (3) special variables form a distinct name space from lexical variables. So the `f` in `(+ f)` is treated as either a special or lexical variable, depending on if there is an applicable special declaration. There is also a name space (4) for data types. Even if `f` is defined as a function and/or a variable, it can also be defined as a data type with `defstruct`, `deftype`, or `defclass`. It can also be defined as (5) a label for `go` statements within a `tagbody` or (6) a block name for `return-from` statements within a block. Finally, symbols inside a quoted expression are treated as constants, and thus form name space (7). These symbols are often used as keys in user-defined tables, and in a sense each such table defines a new name space. One example is the *tag* name space, used by `catch` and `throw`. Another is the package name space.

It is a good idea to limit each symbol to only one name space. Common Lisp will not be confused if a symbol is used in multiple ways, but the poor human reader probably will be.

In the following example `f`, can you identify which of the twelve uses of `f` refer to which name spaces?

```
(defun f (f)
  (block f
    (tagbody
      f (catch 'f
          (if (typep f 'f)
              (throw 'f (go f)))
              (funcall #'f (get (symbol-value 'f) 'f)))))))
```

24.2 Conditions and Error Handling

An extraordinary feature of ANSI Common Lisp is the facility for handling errors. In most languages it is very difficult for the programmer to arrange to recover from an error. Although Ada and some implementations of C provide functions for error recovery, they are not generally part of the repertoire of most programmers. Thus, we find C programs that exit with the ungraceful message `Segmentation violation: core dumped`.

Common Lisp provides one of the most comprehensive and easy-to-use error-handling mechanism of any programming language, which leads to more robust programs. The process of error handling is divided into two parts: signaling an error, and handling it.

Signaling Errors

An *error* is a condition that the program does not know how to handle. Since the program does not know what to do, its only recourse is to announce the occurrence of the error, with the hope that some other program or user will know what to do. This announcement is called *signaling* an error. An error can be signaled by a Common Lisp built-in function, as when `(/ 3 0)` signals a divide-by-zero error. Errors can also be signaled explicitly by the programmer, as in a call to `(error "Illegal value.")`.

Actually, it is a bit of a simplification to talk only of *signaling errors*. The precise term is *signaling a condition*. Some conditions, like end-of-file, are not considered errors, but nevertheless they are unusual conditions that must be dealt with. The condition system in Common Lisp allows for the definition of all kinds of conditions, but we will continue to talk about errors in this brief discussion, since most conditions are in fact error conditions.

Handling Errors

By default, signaling an error invokes the debugger. In the following example, the `>>` prompt means that the user is in the debugger rather than at the top level.

```
> (/ 3 0)
Error: An attempt was made to divide by zero.
>>
```

ANSI Common Lisp provides ways of changing this default behavior. Conceptually, this is done by setting up an *error handler* which handles the error in some way. Error handlers are bound dynamically and are used to process signaled errors. An error handler is much like a `catch`, and signaling an error is like a `throw`. In fact, in many systems `catch` and `throw` are implemented with the error-condition system.

The simplest way of handling an error is with the macro `ignore-errors`. If no error occurs, `ignore-errors` is just like `progn`. But if an error does occur, `ignore-errors` will return `nil` as its first value and `t` as its second, to indicate that an error has occurred but without doing anything else:

```
> (ignore-errors (/ 3 1)) ⇒ 3 NIL
> (ignore-errors (/ 3 0)) ⇒ NIL T
```

`ignore-errors` is a very coarse-grain tool. In an interactive interpreter, `ignore-errors` can be used to recover from any and all errors in the response to one input and get back to the read-process-print loop for the next input. If the errors that are ignored are not serious ones, this can be a very effective way of transforming a buggy program into a useful one.

But some errors are too important to ignore. If the error is running out of memory, then ignoring it will not help. Instead, we need to find some way of freeing up memory and continuing.

The condition-handling system can be used to handle only certain errors. The macro `handler-case`, is a convenient way to do this. Like `case`, its first argument is evaluated and used to determine what to do next. If no error is signaled, then the value of the expression is returned. But if an error does occur, the following clauses are searched for one that matches the type of the error. In the following example, `handler-case` is used to handle division by zero and other arithmetic errors (perhaps floating-point underflow), but it allows all other errors to pass unhandled.

```
(defun div (x y)
  (handler-case (/ x y)
    (division-by-zero () most-positive-fixnum)
    (arithmetic-error () 0)))

> (div 8 2) ⇒ 4
> (div 3 0) ⇒ 16777215
> (div 'xyzy 1)
Error: The value of NUMBER, XYZZY, should be a number
```

Through judicious use of `handler-case`, the programmer can create robust code that reacts well to unexpected situations. For more details, see chapter 29 of *Common Lisp the Language*, 2d edition.

24.3 Pretty Printing

ANSI Common Lisp adds a facility for user-controlled pretty printing. In general, *pretty printing* refers to the process of printing complex expressions in a format that uses indentation to improve readability. The function `pprint` was always available, but before ANSI Common Lisp it was left unspecified, and it could not be extended by the user. Chapter 27 of *Common Lisp the Language*, 2d edition presents a pretty-printing facility that gives the user fine-grained control over the printing of all types of objects. In addition, the facility is integrated with the `format` function.

24.4 Series

The functional style of programming with higher-order functions is one of the attractions of Lisp. The following expression to sum the square roots of the positive numbers in the list `nums` is clear and concise:

```
(reduce #'+ (mapcar #'sqrt (find-all-if #'plusp nums)))
```

Unfortunately, it is inefficient: both `find-all-if` and `mapcar` cons up intermediate lists that are not needed in the final sum. The following two versions using `loop` and `dolist` are efficient but not as pretty:

```
;; Using Loop
(loop for num in nums
      when (plusp num)
      sum (sqrt num))

;; Using dolist
(let ((sum 0))
  (dolist (num nums sum)
    (when (plusp num)
      (incf sum num))))
```

A compromise between the two approaches is provided by the *series* facility, defined in appendix A of *Common Lisp the Language*, 2d edition. The example using `series` would look like:

```
(collect-sum (#Msqrt (choose-if #'plusp nums)))
```

This looks very much like the functional version: only the names have been changed. However, it compiles into efficient iterative code very much like the `dolist` version.

Like pipes (see section 9.3), elements of a series are only evaluated when they are needed. So we can write `(scan-range :from 0)` to indicate the infinite series of integers starting from 0, but if we only use, say, the first five elements of this series, then only the first five elements will be generated.

The `series` facility offers a convenient and efficient alternative to iterative loops and sequence functions. Although the `series` proposal has not yet been adopted as an official part of ANSI Common Lisp, its inclusion in the reference manual has made it increasingly popular.

24.5 The Loop Macro

The original specification of Common Lisp included a simple `loop` macro. The body of the loop was executed repeatedly, until a `return` was encountered. ANSI Common Lisp officially introduces a far more complex `loop` macro, one that had been used in ZetaLisp and its predecessors for some time. This book has occasionally used the complex `loop` in place of alternatives such as `do`, `dotimes`, `dolist`, and the mapping functions.

If your Lisp does not include the complex `loop` macro, this chapter gives a definition that will run all the examples in this book, although it does not support all the features of `loop`. This chapter also serves as an example of a complex macro. As with

any macro, the first thing to do is to look at some macro calls and what they might expand into. Here are two examples:

```
(loop for i from 1 to n do (print (sqrt i))) ≡
(LET* ((I 1)
      (TEMP N))
  (TAGBODY
    LOOP
    (IF (> I TEMP)
      (GO END))
    (PRINT (SQRT I))
    (SETF I (+ I 1))
    (GO LOOP)
  END))

(loop for v in list do (print v)) ≡
(LET* ((IN LIST)
      (V (CAR IN)))
  (TAGBODY
    LOOP
    (IF (NULL IN)
      (GO END))
    (PRINT V)
    (SETF IN (CDR IN))
    (SETF V (CAR IN))
    (GO LOOP)
  END))
```

Each loop initializes some variables, then enters a loop with some exit tests and a body. So the template is something like:

```
(let* (variables...)
  (tagbody
    loop
    (if exit-tests
      (go end))
    body
    (go loop)
  end))
```

Actually, there's more we might need in the general case. There may be a prologue that appears before the loop but after the variable initialization, and similarly there may be an epilogue after the loop. This epilogue may involve returning a value, and since we want to be able to return from the loop in any case, we need to wrap a block around it. So the complete template is:

```
(let* (variables...)
  (block name
    prologue
    (tagbody
      loop
      body
      (go loop)
    end
    epilogue
    (return result))))
```

To generate this template from the body of a loop form, we will employ a structure with fields for each of the parts of the template:

```
(defstruct loop
  "A structure to hold parts of a loop as it is built."
  (vars nil) (prologue nil) (body nil) (steps nil)
  (epilogue nil) (result nil) (name nil))
```

Now the loop macro needs to do four things: (1) decide if this is a use of the simple, non-keyword loop or the complex ANSI loop. If it is the latter, then (2) make an instance of the loop structure, (3) process the body of the loop, filling in appropriate fields of the structure, and (4) place the filled fields into the template. Here is the loop macro:

```
(defmacro loop (&rest exps)
  "Supports both ANSI and simple LOOP.
  Warning: Not every loop keyword is supported."
  (if (every #'listp exps)
      ;; No keywords implies simple loop:
      `(block nil (tagbody loop ,@exps (go loop)))
      ;; otherwise process loop keywords:
      (let ((l (make-loop)))
        (parse-loop-body l exps)
        (fill-loop-template l))))

(defun fill-loop-template (l)
  "Use a loop-structure instance to fill the template."
  `(let* ,(nreverse (loop-vars l))
    (block ,(loop-name l)
      ,@(nreverse (loop-prologue l))
      (tagbody
        loop
        ,@(nreverse (loop-body l))
        ,@(nreverse (loop-steps l))
        (go loop))
```

```

end
,@(nreverse (loop-epilogue l))
(return ,(loop-result l))))))

```

Most of the work is in writing `parse-loop-body`, which takes a list of expressions and parses them into the proper fields of a loop structure. It will use the following auxiliary functions:

```

(defun add-body (l exp) (push exp (loop-body l)))
(defun add-test (l test)
  "Put in a test for loop termination."
  (push '(if ,test (go end)) (loop-body l)))
(defun add-var (l var init &optional (update nil update?))
  "Add a variable, maybe including an update step."
  (unless (assoc var (loop-vars l))
    (push (list var init) (loop-vars l)))
  (when update?
    (push '(setq ,var ,update) (loop-steps l))))

```

There are a number of alternative ways of implementing this kind of processing. One would be to use special variables: `*prologue*`, `*body*`, `*epilogue*`, and so on. This would mean we wouldn't have to pass around the loop structure `l`, but there would be significant clutter in having seven new special variables. Another possibility is to use local variables and close the definitions of `loop`, along with the `add-` functions in that local environment:

```

(let (body prologue epilogue steps vars name result)
  (defmacro loop ...)
  (defun add-body ...)
  (defun add-test ...)
  (defun add-var ...))

```

This is somewhat cleaner style, but some early Common Lisp compilers do not support embedded defuns, so I chose to write in a style that I knew would work in all implementations. Another design choice would be to return multiple values for each of the components and have `parse-loop-body` put them all together. This is in fact done in one of the Lisp Machine implementations of `loop`, but I think it is a poor decision: seven components are too many to keep track of by positional notation.

Anatomy of a Loop

All this has just been to set up for the real work: parsing the expressions that make up the loop with the function `parse-loop-body`. Every loop consists of a sequence of

clauses, where the syntax of each clause is determined by the first expression of the clause, which should be a known symbol. These symbols are called *loop keywords*, although they are not in the keyword package.

The loop keywords will be defined in a data-driven fashion. Every keyword has a function on its property list under the `loop-fn` indicator. The function takes three arguments: the loop structure being built, the very next expression in the loop body, and a list of the remaining expressions after that. The function is responsible for updating the loop structure (usually by making appropriate calls to the `add-` functions) and then returning the unparsed expressions. The three-argument calling convention is used because many of the keywords only look at one more expression. So those functions see that expression as their first argument, and they can conveniently return their second argument as the unparsed remainder. Other functions will want to look more carefully at the second argument, parsing some of it and returning the rest.

The macro `defloop` is provided to add new loop keywords. This macro enforces the three-argument calling convention. If the user supplies only two arguments, then a third argument is automatically added and returned as the remainder. Also, if the user specifies another symbol rather than a list of arguments, this is taken as an alias, and a function is constructed that calls the function for that keyword:

```
(defun parse-loop-body (l exps)
  "Parse the exps based on the first exp being a keyword.
  Continue until all the exps are parsed."
  (unless (null exps)
    (parse-loop-body
     1 (call-loop-fn 1 (first exps) (rest exps)))))

(defun call-loop-fn (l key exps)
  "Return the loop parsing function for this keyword."
  (if (and (symbolp key) (get key 'loop-fn))
      (funcall (get key 'loop-fn) 1 (first exps) (rest exps))
      (error "Unknown loop key: ~a" key)))

(defmacro defloop (key args &rest body)
  "Define a new LOOP keyword."
  ;; If the args do not have a third arg, one is supplied.
  ;; Also, we can define an alias with (defloop key other-key)
  '(setf (get ',key 'loop-fn)
        ,(cond ((and (symbolp args) (null body))
                 #'(lambda (l x y)
                     (call-loop-fn l ',args (cons x y))))
              ((and (listp args) (= (length args) 2))
                 #'(lambda (,@args -exps-) ,@body -exps-))
              (t #'(lambda ,args ,@body)))))
```

Now we are ready to define some loop keywords. Each of the following sections

refers to (and implements the loop keywords in) a section of chapter 26 of *Common Lisp the Language*, 2d edition.

Iteration Control (26.6)

Here we define keywords for iterating over elements of a sequence and for stopping the iteration. The following cases are covered, where uppercase words represent loop keywords:

```
(LOOP REPEAT n ...)
(Loop FOR i FROM s TO e BY inc ...)
(Loop FOR v IN l ...)
(Loop FOR v ON l ...)
(Loop FOR v = expr [THEN step] ...)
```

The implementation is straightforward, although somewhat tedious for complex keywords like `for`. Take the simpler keyword, `repeat`. To handle it, we generate a new variable that will count down the number of times to repeat. We call `add-var` to add that variable, with its initial value, to the loop structure. We also give this variable an update expression, which decrements the variable by one each time through the loop. Then all we need to do is call `add-test` to insert code that will exit the loop when the variable reaches zero:

```
(defloop repeat (l times)
  "(LOOP REPEAT n ...) does loop body n times."
  (let ((i (gensym "REPEAT")))
    (add-var l i times '(- ,i 1))
    (add-test l '(<= ,i 0))))
```

The loop keyword `for` is more complicated, but each case can be analyzed in the same way as `repeat`:

```
(defloop as for) ;; AS is the same as FOR

(defloop for (l var exps)
  "4 of the 7 cases for FOR are covered here:
  (LOOP FOR i FROM s TO e BY inc ...) does arithmetic iteration
  (LOOP FOR v IN l ...) iterates for each element of l
  (LOOP FOR v ON l ...) iterates for each tail of l
  (LOOP FOR v = expr [THEN step]) initializes and iterates v"
  (let ((key (first exps))
        (source (second exps))
        (rest (rest2 exps)))
    (ecase key
```

```

((from downfrom upfrom to downto upto by)
 (loop-for-arithmetic 1 var exps))
(in (let ((v (gensym "IN")))
      (add-var 1 v source '(cdr ,v))
      (add-var 1 var '(car ,v) '(car ,v))
      (add-test 1 '(null ,v))
      rest))
 (on (add-var 1 var source '(cdr ,var))
      (add-test 1 '(null ,var))
      rest)
 (= (if (eq (first rest) 'then)
        (progn
          (pop rest)
          (add-var 1 var source (pop rest)))
        (progn
          (add-var 1 var nil)
          (add-body 1 '(setq ,var ,source))))
      rest)
 ;; ACROSS, BEING clauses omitted
 )))

(defun loop-for-arithmetic (1 var exps)
  "Parse loop expressions of the form:
 (LOOP FOR var [FROM|DOWNFROM|UPFROM exp1] [TO|DOWNTO|UPTO exp2]
  [BY exp3]"
  ;; The prepositions BELOW and ABOVE are omitted
  (let ((exp1 0)
        (exp2 nil)
        (exp3 1)
        (down? nil))
    ;; Parse the keywords:
    (when (member (first exps) '(from downfrom upfrom))
      (setf exp1 (second exps)
            down? (eq (first exps) 'downfrom)
            exps (rest2 exps)))
    (when (member (first exps) '(to downto upto))
      (setf exp2 (second exps)
            down? (or down? (eq (first exps) 'downto))
            exps (rest2 exps)))
    (when (eq (first exps) 'by)
      (setf exp3 (second exps)
            exps (rest2 exps)))
    ;; Add variables and tests:
    (add-var 1 var exp1
              '(,(if down? '- '+) ,var ,(maybe-temp 1 exp3)))
    (when exp2
      (add-test 1 '(,(if down? '<'>) ,var ,(maybe-temp 1 exp2))))
    ;; and return the remaining expressions:

```



```

    exp))

(defun maybe-temp (l exp)
  "Generate a temporary variable, if needed."
  (if (constantp exp)
      exp
      (let ((temp (gensym "TEMP")))
        (add-var l temp exp)
        temp)))

```

End-Test Control (26.7)

In this section we cover the following clauses:

```

(LLOOP UNTIL test ...)
(LLOOP WHILE test ...)
(LLOOP ALWAYS condition ...)
(LLOOP NEVER condition ...)
(LLOOP THEREIS condition ...)
(LLOOP ... (LOOP-FINISH) ...)

```

Each keyword is quite simple:

```

(defun loop-until (l test) (add-test l test))

(defun loop-while (l test) (add-test l '(not ,test)))

(defun loop-always (l test)
  (setf (loop-result l) t)
  (add-body l '(if (not ,test) (return nil))))

(defun loop-never (l test)
  (setf (loop-result l) t)
  (add-body l '(if ,test (return nil))))

(defun loop-thereis (l test) (add-body l '(return-if ,test)))

(defmacro return-if (test)
  "Return TEST if it is non-nil."
  (once-only (test)
    '(if ,test (return ,test))))

(defmacro loop-finish () '(go end))

```

Value Accumulation (26.8)

The `collect` keyword poses another challenge. How do you collect a list of expressions presented one at a time? The answer is to view the expressions as a queue, one where we add items to the rear but never remove them from the front of the queue. Then we can use the queue functions defined in section 10.5.

Unlike the other clauses, value accumulation clauses can communicate with each other. There can be, say, two `collect` and an `append` clause in the same loop, and they all build onto the same list. Because of this, I use the same variable name for the accumulator, rather than gensyming a new variable for each use. The name chosen is stored in the global variable `*acc*`. In the official loop standard it is possible for the user to specify the variable with an `into` modifier, but I have not implemented that option. The clauses covered are:

```
(LOOP COLLECT item ...)
(LLOOP NCONC item ...)
(LLOOP APPEND item ...)
(LLOOP COUNT item ...)
(LLOOP SUM item ...)
(LLOOP MAXIMIZE item ...)
(LLOOP MINIMIZE item ...)
```

The implementation is:

```
(defconstant *acc* (gensym "ACC")
  "Variable used for value accumulation in LOOP.")

;;; INTO preposition is omitted

(defloop collect (l exp)
  (add-var l *acc* '(make-queue))
  (add-body l '(enqueue ,exp ,*acc*))
  (setf (loop-result l) '(queue-contents ,*acc*)))

(defloop nconc (l exp)
  (add-var l *acc* '(make-queue))
  (add-body l '(queue-nconc ,*acc* ,exp))
  (setf (loop-result l) '(queue-contents ,*acc*)))

(defloop append (l exp exps)
  (call-loop-fn l 'nconc '((copy-list ,exp) .,exps)))

(defloop count (l exp)
  (add-var l *acc* 0)
  (add-body l '(when ,exp (incf ,*acc*)))
  (setf (loop-result l) *acc*))
```

```


(defloop sum (l exp)
  (add-var l *acc* 0)
  (add-body l '(incf ,*acc* ,exp))
  (setf (loop-result l) *acc*))

(defloop maximize (l exp)
  (add-var l *acc* nil)
  (add-body l '(setf ,*acc*
                    (if ,*acc*
                        (max ,*acc* ,exp)
                        ,exp)))
  (setf (loop-result l) *acc*))

(defloop minimize (l exp)
  (add-var l *acc* nil)
  (add-body l '(setf ,*acc*
                    (if ,*acc*
                        (min ,*acc* ,exp)
                        ,exp)))
  (setf (loop-result l) *acc*))

(defloop collecting collect)
(defloop nconcing nconc)
(defloop appending append)
(defloop counting count)
(defloop summing sum)
(defloop maximizing maximize)
(defloop minimizing minimize)

```

 **Exercise 24.1** loop lets us build aggregates (lists, maximums, sums, etc.) over the body of the loop. Sometimes it is inconvenient to be restricted to a single-loop body. For example, we might want a list of all the nonzero elements of a two-dimensional array. One way to implement this is with a macro, `with-collection`, that sets up and returns a queue structure that is built by calls to the function `collect`. For example:

```

> (let ((A '#2a((1 0 0) (0 2 4) (0 0 3))))
  (with-collection
    (loop for i from 0 to 2 do
      (loop for j from 0 to 2 do
        (if (> (aref a i j) 0)
            (collect (aref A i j)))))))
(1 2 4 3)

```

Implement `with-collection` and `collect`.

Variable Initialization (26.9)

The `with` clause allows local variables—I have included it, but recommend using a `let` instead. I have not included the `and` preposition, which allows the variables to nest at different levels.

```
;;; 26.9. Variable Initializations ("and" omitted)

(defun with (l var exps)
  (let ((init nil))
    (when (eq (first exps) '=)
      (setf init (second exps)
            exps (rest2 exps)))
    (add-var l var init)
    exps))
```

Conditional Execution (26.10)

`loop` also provides forms for conditional execution. These should be avoided whenever possible, as Lisp already has a set of perfectly good conditional macros. However, sometimes you want to make, say, a `collect` conditional on some test. In that case, `loop` conditionals are acceptable. The clauses covered here are:

```
(LOOP WHEN test ... [ELSE ...]) ; IF is a synonym for WHEN
(LUMP UNLESS test ... [ELSE ...])
```

Here is an example of `when`:

```
> (loop for x from 1 to 10
      when (oddp x)
        collect x
      else collect (- x))
(1 -2 3 -4 5 -6 7 -8 9 -10)
```

Of course, we could have said `collect (if (oddp x) x (- x))` and done without the conditional. There is one extra feature in `loop`'s conditionals: the value of the test is stored in the variable `it` for subsequent use in the `THEN` or `ELSE` parts. (This is just the kind of feature that makes some people love `loop` and others throw up their hands in despair.) Here is an example:

```
> (loop for x from 1 to 10
      when (second (assoc x '((1 one) (3 three) (5 five))))
      collect it)
(ONE THREE FIVE)
```

The conditional clauses are a little tricky to implement, since they involve parsing other clauses. The idea is that `call-loop-fn` parses the THEN and ELSE parts, adding whatever is necessary to the body and to other parts of the loop structure. Then `add-body` is used to add labels and `go` statements that branch to the labels as needed. This is the same technique that is used to compile conditionals in chapter 23; see the function `comp-if` on page 787. Here is the code:

```
(defloop when (l test exps)
  (loop-unless l '(not ,(maybe-set-it test exps)) exps))

(defloop unless (l test exps)
  (loop-unless l (maybe-set-it test exps) exps))

(defun maybe-set-it (test exps)
  "Return value, but if the variable IT appears in exps,
  then return code that sets IT to value."
  (if (find-anywhere 'it exps)
      '(setq it ,test)
      test))

(defloop if when)

(defun loop-unless (l test exps)
  (let ((label (gensym "L")))
    (add-var l 'it nil)
    ;; Emit code for the test and the THEN part
    (add-body l '(if ,test (go ,label)))
    (setf exps (call-loop-fn l (first exps) (rest exps)))
    ;; Optionally emit code for the ELSE part
    (if (eq (first exps) 'else)
        (progn
          (let ((label2 (gensym "L")))
            (add-body l '(go ,label2))
            (add-body l label)
            (setf exps (call-loop-fn l (second exps) (rest2 exps)))
            (add-body l label2)))
          (add-body l label)))
    exps)
```

Unconditional Execution (26.11)

The unconditional execution keywords are `do` and `return`:

```
(defloop do (l exp exps)
  (add-body l exp)
  (loop (if (symbolp (first exps)) (RETURN exps))
        (add-body l (pop exps))))

(defloop return (l exp) (add-body l '(return ,exp)))
```

Miscellaneous Features (26.12)

Finally, the miscellaneous features include the keywords `initially` and `finally`, which define the loop prologue and epilogue, and the keyword `named`, which gives a name to the loop for use by a `return-from` form. I have omitted the data-type declarations and destructuring capabilities.

```
(defloop initially (l exp exps)
  (push exp (loop-prologue l))
  (loop (if (symbolp (first exps)) (RETURN exps))
        (push (pop exps) (loop-prologue l))))

(defloop finally (l exp exps)
  (push exp (loop-epilogue l))
  (loop (if (symbolp (first exps)) (RETURN exps))
        (push (pop exps) (loop-epilogue l))))

(defloop named (l exp) (setf (loop-name l) exp))
```

24.6 Sequence Functions

Common Lisp provides sequence functions to make the programmer's life easier: the same function can be used for lists, vectors, and strings. However, this ease of use comes at a cost. Sequence functions must be written very carefully to make sure they are efficient. There are three main sources of indeterminacy that can lead to inefficiency: (1) the sequences can be of different types; (2) some functions have keyword arguments; (3) some functions have a `&rest` argument. Careful coding can limit or eliminate these sources of inefficiency, by making as many choices as possible at compile time and making the remaining choices outside of the main loop.

In this section we see how to implement the new ANSI sequence function `map-into` and the updated function `reduce` efficiently. This is essential for those without an ANSI compiler. Even those who do have access to an ANSI compiler will benefit from seeing the efficiency techniques used here.

Before defining the sequence functions, the macro `once-only` is introduced.

Once-only: A Lesson in Macrology

The macro `once-only` has been around for a long time on various systems, although it didn't make it into the Common Lisp standard. I include it here for two reasons: first, it is used in the following `funcall-if` macro, and second, if you can understand how to write and when to use `once-only`, then you truly understand macro.

First, you have to understand the problem that `once-only` addresses. Suppose we wanted to have a macro that multiplies its input by itself:²

```
(defmacro square (x) '(* ,x ,x))
```

This definition works fine in the following case:

```
> (macroexpand '(square z)) ⇒ (* Z Z)
```

But it doesn't work as well here:

```
> (macroexpand '(square (print (incf i))))
(* (PRINT (INCF I)) (PRINT (INCF I)))
```

The problem is that `i` will get incremented twice, not once, and two different values will get printed, not one. We need to bind `(print (incf i))` to a local variable before doing the multiplication. On the other hand, it would be superfluous to bind `z` to a local variable in the previous example. This is where `once-only` comes in. It allows us to write macro definitions like this:

```
(defmacro square (x) (once-only (x) '(* ,x ,x)))
```

and have the generated code be just what we want:

```
> (macroexpand '(square z))
(* Z Z)
```

²As was noted before, the proper way to do this is to proclaim `square` as an inline function, not a macro, but please bear with the example.

```
> (macroexpand '(square (print (incf i))))
(LET ((G3811 (PRINT (INCF I))))
  (* G3811 G3811))
```

You have now learned lesson number one of once-only: you know how macros differ from functions when it comes to arguments with side effects, and you now know how to handle this. Lesson number two comes when you try to write (or even understand) a definition of once-only—only when you truly understand the nature of macros will you be able to write a correct version. As always, the first thing to determine is what a call to once-only should expand into. The generated code should test the variable to see if it is free of side effects, and if so, generate the body as is; otherwise it should generate code to bind a new variable, and use that variable in the body of the code. Here's roughly what we want:

```
> (macroexpand '(once-only (x) '(* ,x ,x)))
(if (side-effect-free-p x)
    '(* ,x ,x)
    '(let ((g001 ,x))
      ,(let ((x 'g001))
        '(* ,x ,x))))
```

where g001 is a new symbol, to avoid conflicts with the x or with symbols in the body. Normally, we generate macro bodies using backquotes, but if the macro body itself has a backquote, then what? It is possible to nest backquotes (and appendix C of *Common Lisp the Language*, 2d edition has a nice discussion of doubly and triply nested backquotes), but it certainly is not trivial to understand. I recommend replacing the inner backquote with its equivalent using list and quote:

```
(if (side-effect-free-p x)
    '(* ,x ,x)
    (list 'let (list (list 'g001 x))
          (let ((x 'g001))
            '(* ,x ,x))))
```

Now we can write once-only. Note that we have to account for the case where there is more than one variable and where there is more than one expression in the body.

```
(defmacro once-only (variables &rest body)
  "Returns the code built by BODY. If any of VARIABLES
  might have side effects, they are evaluated once and stored
  in temporary variables that are then passed to BODY."
  (assert (every #'symbolp variables))
  (let ((temps (loop repeat (length variables) collect (gensym))))
    '(if (every #'side-effect-free-p (list .,variables))
```



```

      (progn .,body)
      (list 'let
            ,(list ,@(mapcar #'(lambda (tmp var)
                                '(list ',tmp ,var))
                            temps variables))
            (let ,(mapcar #'(lambda (var tmp) '(,var ',tmp))
                    variables temps)
                .,body))))))
(defun side-effect-free-p (exp)
  "Is exp a constant, variable, or function,
  or of the form (THE type x) where x is side-effect-free?"
  (or (constantp exp) (atom exp) (starts-with exp 'function)
      (and (starts-with exp 'the)
           (side-effect-free-p (third exp)))))

```

Here we see the expansion of the call to `once-only` and a repeat of the expansions of two calls to `square`:

```

> (macroexpand '(once-only (x)>(* ,x ,x)))
(IF (EVERY #'SIDE-EFFECT-FREE-P (LIST X))
    (PROGN
     (* ,X ,X))
    (LIST 'LET (LIST (LIST 'G3763 X))
          (LET ((X 'G3763))
              (* ,X ,X))))))

> (macroexpand '(square z))
(* Z Z)

> (macroexpand '(square (print (incf i))))
(LET ((G3811 (PRINT (INCF I))))
    (* G3811 G3811))

```

This output was produced with `*print-gensym*` set to `nil`. When this variable is non-`nil`, uninterned symbols are printed with a prefix `#:`, as in `#:G3811`. This insures that the symbol will not be interned by a subsequent read.

It is worth noting that Common Lisp automatically handles problems related to multiple evaluation of subforms in `setf` methods. See page 884 for an example.

Avoid Overusing Macros

A word to the wise: don't get carried away with macros. Use macros freely to represent your *problem*, but shy away from new macros in the implementation of your *solution*, unless absolutely necessary. So, it is good style to introduce a macro,

say, `defrule`, which defines rules for your application, but adding macros to the code itself may just make things harder for others to use.

Here is a story. Before `if` was a standard part of Lisp, I defined my own version of `if`. Unlike the simple `if`, my version took any number of test/result pairs, followed by an optional `else` result. In general, the expansion was:

```
(if abcd...x) => (cond (ab) (cd) ... (Tx))
```

My `if` also had one more feature: the symbol 'that' could be used to refer to the value of the most recent test. For example, I could write:

```
(if (assoc item a-list)
    (process (cdr that)))
```

which would expand into:

```
(LET (THAT)
  (COND
    ((SETQ THAT (ASSOC ITEM A-LIST)) (PROCESS (CDR THAT))))))
```

This was a convenient feature (compare it to the `=>` feature of Scheme's `cond`, as discussed on page 778), but it backfired often enough that I eventually gave up on my version of `if`. Here's why. I would write code like this:

```
(if (total-score x)
    (print (/ that number-of-trials))
    (error "No scores"))
```

and then make a small change:

```
(if (total-score x)
    (if *print-scores* (print (/ that number-of-trials)))
    (error "No scores"))
```

The problem is that the variable that now refers to `*print-scores*`, not `(total-score x)`, as it did before. My macro violates referential transparency. In general, that's the whole point of macros, and it is why macros are sometimes convenient. But in this case, violating referential transparency can lead to confusion.

MAP-INTO

The function `map-into` is used on page 632. This function, added for the ANSI version of Common Lisp, is like `map`, except that instead of building a new sequence, the first argument is changed to hold the results. This section describes how to write a fairly efficient version of `map-into`, using techniques that are applicable to any sequence function. We'll start with a simple version:

```
(defun map-into (result-sequence function &rest sequences)
  "Destructively set elements of RESULT-SEQUENCE to the results
  of applying FUNCTION to respective elements of SEQUENCES."
  (replace result-sequence (apply #'map 'list function sequences)))
```

This does the job, but it defeats the purpose of `map-into`, which is to avoid generating garbage. Here's a version that generates less garbage:

```
(defun map-into (result-sequence function &rest sequences)
  "Destructively set elements of RESULT-SEQUENCE to the results
  of applying FUNCTION to respective elements of SEQUENCES."
  (let ((n (loop for seq in (cons result-sequence sequences)
                 minimize (length seq))))
    (dotimes (i n)
      (setf (elt result-sequence i)
            (apply function
                   (mapcar #'(lambda (seq) (elt seq i))
                           sequences))))))
```

There are three problems with this definition. First, it wastes space: `mapcar` creates a new argument list each time, only to have the list be discarded. Second, it wastes time: doing a `setf` of the *i*th element of a list makes the algorithm $O(n^2)$ instead of $O(n)$, where *n* is the length of the list. Third, it is subtly wrong: if `result-sequence` is a vector with a fill pointer, then `map-into` is supposed to ignore `result-sequence`'s current length and extend the fill pointer as needed. The following version fixes those problems:

```
(defun map-into (result-sequence function &rest sequences)
  "Destructively set elements of RESULT-SEQUENCE to the results
  of applying FUNCTION to respective elements of SEQUENCES."
  (let ((arglist (make-list (length sequences)))
        (n (if (listp result-sequence)
                most-positive-fixnum
                (array-dimension result-sequence 0))))
    ;; arglist is made into a list of args for each call
    ;; n is the length of the longest vector
```

```

(when sequences
  (setf n (min n (loop for seq in sequences
                      minimize (length seq))))))
;; Define some shared functions:
(flet
  ((do-one-call (i)
    (loop for seq on sequences
          for arg on arglist
          do (if (listp (first seq))
                (setf (first arg)
                      (pop (first seq)))
                (setf (first arg)
                      (aref (first seq) i))))
    (apply function arglist))
   (do-result (i)
    (if (and (vectorp result-sequence)
            (array-has-fill-pointer-p result-sequence))
        (setf (fill-pointer result-sequence)
              (max i (fill-pointer result-sequence))))))
  (declare (inline do-one-call))
  ;; Decide if the result is a list or vector,
  ;; and loop through each element
  (if (listp result-sequence)
      (loop for i from 0 to (- n 1)
            for r on result-sequence
            do (setf (first r)
                    (do-one-call i)))
      (loop for i from 0 to (- n 1)
            do (setf (aref result-sequence i)
                    (do-one-call i))
            finally (do-result n))))
result-sequence))

```

There are several things worth noticing here. First, I split the main loop into two versions, one where the result is a list, and the other where it is a vector. Rather than duplicate code, the local functions `do-one-call` and `do-result` are defined. The former is declared inline because it is called often, while the latter is not. The arguments are computed by looking at each sequence in turn, taking the *i*th element if it is a vector, and popping the sequence if it is a list. The arguments are stored into the list `arglist`, which has been preallocated to the correct size. All in all, we compute the answer fairly efficiently, without generating unnecessary garbage.

The application could be done more efficiently, however. Think what `apply` must do: scan down the argument list, and put each argument into the location expected by the function-calling conventions, and then branch to the function. Some implementations provide a better way of doing this. For example, the TI Lisp Machine provides two low-level primitive functions, `%push` and `%call`, that compile into single

instructions to put the arguments into the right locations and branch to the function. With these primitives, the body of `do-one-call` would be:

```
(loop for seq on sequences
      do (if (listp (first seq))
            (%push (pop (first seq)))
            (%push (aref (first seq) i))))
(%call function length-sequences)
```

There is a remaining inefficiency, though. Each sequence is type-checked each time through the loop, even though the type remains constant once it is determined the first time. Theoretically, we could code separate loops for each combination of types, just as we coded two loops depending on the type of the result sequence. But that would mean 2^n loops for n sequences, and there is no limit on how large n can be.

It might be worth it to provide specialized functions for small values of n , and dispatch to the appropriate function. Here's a start at that approach:

```
(defun map-into (result function &rest sequences)
  (apply
   (case (length sequences)
     (0 (if (listp result) #'map-into-list-0 #'map-into-vect-0))
     (1 (if (listp result)
            (if (listp (first sequences))
                #'map-into-list-1-list #'map-into-list-1-vect)
            (if (listp (first sequences))
                #'map-into-vect-1-list #'map-into-vect-1-vect)))
     (2 (if (listp result)
            (if (listp (first sequences))
                (if (listp (second sequences))
                    #'map-into-list-2-list-list
                    #'map-into-list-2-list-vect)
                ...)
            (t (if (listp result) #'map-into-list-n #'map-into-vect-n)))
         result function sequences))
```

The individual functions are not shown. This approach is efficient in execution time, but it takes up a lot of space, considering that `map-into` is a relatively obscure function. If `map-into` is declared `inline` and the compiler is reasonably good, then it will produce code that just calls the appropriate function.

REDUCE with :key

Another change in the ANSI proposal is to add a `:key` keyword to `reduce`. This is a useful addition—in fact, for years I had been using a `reduce-by` function that provided

just this functionality. In this section we see how to add the `:key` keyword.

At the top level, I define `reduce` as an interface to the keywordless function `reduce*`. They are both proclaimed inline, so there will be no overhead for the keywords in normal uses of `reduce`.

```
(proclaim '(inline reduce reduce*))

(defun reduce* (fn seq from-end start end key init init-p)
  (funcall (if (listp seq) #'reduce-list #'reduce-vect)
           fn seq from-end (or start 0) end key init init-p))

(defun reduce (function sequence &key from-end start end key
              (initial-value nil initial-value-p))
  (reduce* function sequence from-end start end
           key initial-value initial-value-p))
```

The easier case is when the sequence is a vector:

```
(defun reduce-vect (fn seq from-end start end key init init-p)
  (when (null end) (setf end (length seq)))
  (assert (<= 0 start end (length seq)) (start end)
          "Illegal subsequence of ~a --- :start ~d :end ~d"
          seq start end)
  (case (- end start)
    (0 (if init-p init (funcall fn)))
    (1 (if init-p
          (funcall fn init (funcall-if key (aref seq start)))
          (funcall-if key (aref seq start))))
    (t (if (not from-end)
          (let ((result)
                (if init-p
                    (funcall
                     (funcall
                      fn init
                      (funcall-if key (aref seq start)))
                     (funcall
                      fn
                      (funcall-if key (aref seq start))
                      (funcall-if key (aref seq (+ start 1)))))))
            (loop for i from (+ start (if init-p 1 2))
                  to (- end 1)
                  do (setf result
                          (funcall
                           (funcall
                            fn result
                            (funcall-if key (aref seq i))))))
              result)
          (let ((result)
                (if init-p
```

```

      (funcall
       fn
       (funcall-if key (aref seq (- end 1)))
       init)
      (funcall
       fn
       (funcall-if key (aref seq (- end 2)))
       (funcall-if key (aref seq (- end 1))))))
(loop for i from (- end (if init-p 2 3)) downto start
  do (setf result
        (funcall
         fn
         (funcall-if key (aref seq i))
         result)))
result))))

```

When the sequence is a list, we go to some trouble to avoid computing the length, since that is an $O(n)$ operation on lists. The hardest decision is what to do when the list is to be traversed from the end. There are four choices:

- **recurse.** We could recursively walk the list until we hit the end, and then compute the results on the way back up from the recursions. However, some implementations may have fairly small bounds on the depths of recursive calls, and a system function like `reduce` should never run afoul of such limitations. In any event, the amount of stack space consumed by this approach would normally be more than the amount of heap space consumed in the next approach.
- **reverse.** We could reverse the list and then consider `from-end` true. The only drawback is the time and space needed to construct the reversed list.
- **nreverse.** We could destructively reverse the list in place, do the reduce computation, and then destructively reverse the list back to its original state (perhaps with an `unwind-protect` added). Unfortunately, this is just incorrect. The list may be bound to some variable that is accessible to the function used in the reduction. If that is so, the function will see the reversed list, not the original list.
- **coerce.** We could convert the list to a vector, and then use `reduce-vect`. This has an advantage over the reverse approach in that vectors generally take only half as much storage as lists. Therefore, this is the approach I adopt.

```

(defmacro funcall-if (fn arg)
  (once-only (fn)
    '(if ,fn (funcall ,fn ,arg) ,arg)))

```

```

(defun reduce-list (fn seq from-end start end key init init-p)
  (when (null end) (setf end most-positive-fixnum))
  (cond ((> start 0)
         (reduce-list fn (nthcdr start seq) from-end 0
                      (- end start) key init init-p))
        ((or (null seq) (eql start end))
         (if init-p init (funcall fn)))
        ((= (- end start) 1)
         (if init-p
             (funcall fn init (funcall-if key (first seq)))
             (funcall-if key (first seq))))
        (from-end
         (reduce-vect fn (coerce seq 'vector) t start end
                      key init init-p))
        ((null (rest seq))
         (if init-p
             (funcall fn init (funcall-if key (first seq)))
             (funcall-if key (first seq))))
        (t (let ((result
                  (if init-p
                      (funcall
                       fn init
                       (funcall-if key (pop seq)))
                      (funcall
                       fn
                       (funcall-if key (pop seq))
                       (funcall-if key (pop seq))))))
            (if end
                (loop repeat (- end (if init-p 1 2)) while seq
                     do (setf result
                             (funcall
                              fn result
                              (funcall-if key (pop seq))))))
                (loop while seq
                     do (setf result
                             (funcall
                              fn result
                              (funcall-if key (pop seq))))))
            result))))))

```


24.7 Exercises

- ?** **Exercise 24.2 [m]** The function `reduce` is a very useful one, especially with the key keyword. Write nonrecursive definitions for `append` and `length` using `reduce`. What other common functions can be written with `reduce`?
- ?** **Exercise 24.3** The so-called loop keywords are not symbols in the keyword package. The preceding code assumes they are all in the current package, but this is not quite right. Change the definition of `loop` so that any symbol with the same name as a loop keyword acts as a keyword, regardless of the symbol's package.
- ?** **Exercise 24.4** Can there be a value for *exp* for which the following expressions are not equivalent? Either demonstrate such an *exp* or argue why none can exist.

```
(loop for x in list collect exp)
(mapcar #'(lambda (x) exp) list))
```

- ?** **Exercise 24.5** The object-oriented language Eiffel provides two interesting loop keywords: `invariant` and `variant`. The former takes a Boolean-valued expression that must remain true on every iteration of the loop, and the latter takes an integer-valued expression that must decrease on every iteration, but never becomes negative. Errors are signaled if these conditions are violated. Use `defloop` to implement these two keywords. Make them generate code conditionally, based on a global flag.

24.8 Answers

Answer 24.1

```
(defvar *queue*)
(defun collect (item) (enqueue item *queue*))
(defmacro with-collection (&body body)
  '(let ((*queue* (make-queue)))
    ,@body
    (queue-contents *queue*)))
```

Here's another version that allows the collection variable to be named. That way, more than one collection can be going on at the same time.

```
(defun collect (item &optional (queue *queue*))
  (enqueue item queue))

(defmacro with-collection ((&optional (queue '*queue*)
                             &body body)
  '(let ((,queue (make-queue)))
    ,@body
    (queue-contents ,queue)))
```

Answer 24.2

```
(defun append-r (x y)
  (reduce #'cons x :initial-value y :from-end t))

(defun length-r (list)
  (reduce #'+ list :key #'(lambda (x) 1)))
```

Answer 24.4 The difference between `loop` and `mapcar` is that the former uses only one variable `x`, while the latter uses a different `x` each time. If `x`'s extent is no bigger than its scope (as it is in most expressions) then this makes no difference. But if any `x` is captured, giving it a longer extent, then a difference shows up. Consider `exp = #'(lambda () x)`.

```
> (mapcar #'funcall (loop for x in '(1 2 3) collect
                        #'(lambda () x)))
(3 3 3)

> (mapcar #'funcall (mapcar #'(lambda (x) #'(lambda () x))
                          '(1 2 3)))
(1 2 3)
```

Answer 24.5

```
(defvar *check-invariants* t
  "Should VARIANT and INVARIANT clauses in LOOP be checked?")

(defloop invariant (l exp)
  (when *check-invariants*
    (add-body l '(assert ,exp () "Invariant violated."))))

(defloop variant (l exp)
  (when *check-invariants*
    (let ((var (gensym "INV")))
      (add-var l var nil)
      (add-body l '(setf ,var (update-variant ,var ,exp))))))
```

```
(defun update-variant (old new)
  (assert (or (null old) (< new old)) ()
          "Variant is not monotonically decreasing")
  (assert (> new 0) () "Variant is no longer positive")
  new)
```

Here's an example:

```
(defun gcd2 (a b)
  "Greatest common divisor. For two positive integer arguments."
  (check-type a (integer 1))
  (check-type b (integer 1))
  (loop with x = a with y = b
        invariant (and (> x 0) (> y 0)) ;; (= (gcd x y) (gcd a b))
        variant (max x y)
        until (= x y)
        do (if (> x y) (decf x y) (decf y x))
        finally (return x)))
```

Here the invariant is written semi-informally. We could include the calls to `gcd`, but that seems to be defeating the purpose of `gcd2`, so that part is left as a comment. The idea is that the comment should help the reader prove the correctness of the code, and the executable part serves to notify the lazy reader when something is demonstrably wrong at run time.

CHAPTER 25

Troubleshooting

*Perhaps if we wrote programs from childhood on,
as adults we'd be able to read them.*

—Alan Perlis

When you buy a new appliance such as a television, it comes with an instruction booklet that lists troubleshooting hints in the following form:

PROBLEM: Nothing works.

Diagnosis: Power is off.

Remedy: Plug in outlet and turn on power switch.

If your Lisp compiler came without such a handy instruction booklet, this chapter may be of some help. It lists some of the most common difficulties that Lisp programmers encounter.

25.1 Nothing Happens

PROBLEM: You type an expression to Lisp's read-eval-print loop and get no response—no result, no prompt.

Diagnosis: There are two likely reasons why output wasn't printed: either Lisp is still doing read or it is still doing eval. These possibilities can be broken down further into four cases:

Diagnosis: If the expression you type is incomplete, Lisp will wait for more input to complete it. An expression can be incomplete because you have left off a right parenthesis (or inserted an extra left parenthesis). Or you may have started a string, atom, or comment without finishing it. This is particularly hard to spot when the error spans multiple lines. A string begins and ends with double-quotes: "string"; an atom containing unusual characters can be delimited by vertical bars: |AN ATOM|; and a comment can be of the form #| a comment |#. Here are four incomplete expressions:

```
(+ (* 3 (sqrt 5) 1)
  (format t "~&X=~a, Y=~a. x y)
  (get '|strange-atom 'prop)
  (if (= x 0) #| test if x is zero
      y
      x)
```

Remedy: Add a), ", |, and |#, respectively. Or hit the interrupt key and type the input again.

Diagnosis: Your program may be waiting for input.

Remedy: Never do a (read) without first printing a prompt of some kind. If the prompt does not end with a newline, a call to finish-output is also in order. In fact, it is a good idea to call a function that is at a higher level than read. Several systems define the function prompt-and-read. Here is one version:

```
(defun prompt-and-read (ctl-string &rest args)
  "Print a prompt and read a reply."
  (apply #'format t ctl-string args)
  (finish-output)
  (read))
```

Diagnosis: The program may be caught in an infinite loop, either in an explicit loop or in a recursive function.

Remedy: Interrupt the computation, get a back trace, and see what functions are active. Check the base case and loop variant on active functions and loops.

Diagnosis: Even a simple expression like `(mapc #'sqrt list)` or `(length list)` will cause an infinite loop if `list` is an infinite list—that is, a list that has some tail that points back to itself.

Remedy: Be very careful any time you modify a structure with `nconc`, `delete`, `setf`, and so forth.

PROBLEM: You get a new prompt from the read-eval-print loop, but no output was printed.

Diagnosis: The expression you evaluated must have returned no values at all, that is, the result `(values)`.

25.2 Change to Variable Has No Effect

PROBLEM: You redefined a variable, but the new value was ignored.

Diagnosis: Altering a variable by editing and re-evaluating a `defvar` form will not change the variable's value. `defvar` only assigns an initial value when the variable is unbound.

Remedy: Use `setf` to update the variable, or change the `defvar` to a `defparameter`.

Diagnosis: Updating a locally bound variable will not affect a like-named variable outside that binding. For example, consider:

```
(defun check-ops (*ops*)
  (if (null *ops*)
      (setf *ops* *default-ops*))
  (mapcar #'check-op *ops*))
```

If `check-ops` is called with a null argument, the `*ops*` that is a parameter of `check-ops` will be updated, but the global `*ops*` will not be, even if it is declared special.

Remedy: Don't shadow variables you want to update. Use a different name for the local variable. It is important to distinguish special and local variables. Stick to the naming convention for special variables: they should begin and end with asterisks. Don't forget to introduce a binding for all local variables. The following excerpt from a recent textbook is an example of this error:

```
(defun test ()
  (setq x 'test-data)           ; Warning!
  (solve-problem x))           ; Don't do this.
```

This function should have been written:

```
(defun test ()
  (let ((x 'test-data))         ; Do this instead.
    (solve-problem x)))
```

25.3 Change to Function Has No Effect

PROBLEM: You redefined a function, but the change was ignored.

Diagnosis: When you change a macro, or a function that has been declared inline, the change will not necessarily be seen by users of the changed function. (It depends on the implementation.)

Remedy: Recompile after changing a macro. Don't use inline functions until everything is debugged. (Use `(declare (notinline f))` to cancel an inline declaration).

Diagnosis: If you change a normal (non-inline) function, that change *will* be seen by code that refers to the function by *name*, but not by code that refers to the old value of the function itself. Consider:

```
(defparameter *scorer* #'score-fn)
(defparameter *printer* 'print-fn)

(defun show (values)
  (funcall *printer*
    (funcall *scorer* values)
    (reduce #'better values)))
```

Now suppose that the definitions of `score-fn`, `print-fn`, and `better` are all changed. Does any of the prior code have to be recompiled? The variable `*printer*` can stay as is. When it is funcalled, the symbol `print-fn` will be consulted for the current functional value. Within `show`, the expression `#'better` is compiled into code that will get the current version of `better`, so it too is safe. However, the variable `*scorer*` must be changed. Its value is the old definition of `score-fn`.

Remedy: Re-evaluate the definition of `*scorer*`. It is unfortunate, but this problem encourages many programmers to use symbols where they really mean functions. Symbols will be coerced to the global function they name when passed to `funcall`

or `apply`, but this can be the source of another error. In the following example, the symbol `local-fn` will not refer to the locally bound function. One needs to use `#'local-fn` to refer to it.

```
(flet ((local-fn (x) ...))
  (mapcar 'local-fn list))
```

Diagnosis: If you changed the name of a function, did you change the name everywhere? For example, if you decide to change the name of `print-fn` to `print-function` but forget to change the value of `*printer*`, then the old function will be called.

Remedy: Use your editor's global replace command. To be even safer, redefine obsolete functions to call `error`. The following function is handy for this purpose:

```
(defun make-obsolete (fn-name)
  "Print an error if an obsolete function is called."
  (setf (symbol-function fn-name)
        #'(lambda (&rest args)
            (declare (ignore args))
            (error "Obsolete function."))))
```

Diagnosis: Are you using `labels` and `flet` properly? Consider again the function `replace-?-vars`, which was defined in section 11.3 to replace an anonymous logic variable with a unique new variable.

```
(defun replace-?-vars (exp)
  "Replace any ? within exp with a var of the form ?123."
  (cond ((eq exp '?) (gensym "?"))
        ((atom exp) exp)
        (t (cons (replace-?-vars (first exp))
                  (replace-?-vars (rest exp))))))
```

It might occur to the reader that gensyming a different variable each time is wasteful. The variables must be unique in each clause, but they can be shared across clauses. So we could generate variables in the sequence `?1, ?2, ...`, intern them, and thus reuse these variables in the next clause (provided we warn the user never to use such variable names). One way to do that is to introduce a local variable to hold the variable number, and then a local function to do the computation:


```
(defun replace-?-vars (exp)
  "Replace any ? within exp with a var of the form ?123."
  ;; *** Buggy Version ***
  (let ((n 0))
    (flet
      ((replace-?-vars (exp)
         (cond ((eq exp '?) (symbol '? (incf n)))
               ((atom exp) exp)
               (t (cons (replace-?-vars (first exp))
                        (replace-?-vars (rest exp)))))))
      (replace-?-vars exp))))
```

This version doesn't work. The problem is that `flet`, like `let`, defines a new function within the body of the `flet` but not within the new function's definition. So two lessons are learned here: use `labels` instead of `flet` to define recursive functions, and don't shadow a function definition with a local definition of the same name (this second lesson holds for variables as well). Let's fix the problem by changing `labels` to `flet` and naming the local function `recurse`:

```
(defun replace-?-vars (exp)
  "Replace any ? within exp with a var of the form ?123."
  ;; *** Buggy Version ***
  (let ((n 0))
    (labels
      ((recurse (exp)
         (cond ((eq exp '?) (symbol '? (incf n)))
               ((atom exp) exp)
               (t (cons (replace-?-vars (first exp))
                        (replace-?-vars (rest exp)))))))
      (recurse exp))))
```

Annoyingly, this version still doesn't work! This time, the problem is carelessness; we changed the `replace-?-vars` to `recurse` in two places, but not in the two calls in the body of `recurse`.

Remedy: In general, the lesson is to make sure you call the right function. If there are two functions with similar effects and you call the wrong one, it can be hard to see. This is especially true if they have similar names.

PROBLEM: Your closures don't seem to be working.

Diagnosis: You may be erroneously creating a lambda expression by consing up code. Here's an example from a recent textbook:

```
(defun make-specialization (c)
  (let (pred newc)
    ...
    (setf (get newc 'predicate)
          '(lambda (obj)
              (and ,(cons pred '(obj))
                    (apply ',(get c 'predicate) (list obj))))))
    ...))
```

Strictly speaking, this is legal according to *Common Lisp the Language*, although in ANSI Common Lisp it will *not* be legal to use a list beginning with `lambda` as a function. But in either version, it is a bad idea to do so. A list beginning with `lambda` is just that: a list, not a closure. Therefore, it cannot capture lexical variables the way a closure does.

Remedy: The correct way to create a closure is to evaluate a call to the special form function, or its abbreviation, `#'`. Here is a replacement for the code beginning with `'(lambda ...`. Note that it is a closure, closed over `pred` and `c`. Also note that it gets the predicate each time it is called; thus, it is safe to use even when predicates are being changed dynamically. The previous version would not work when a predicate is changed.

```
#'(lambda (obj)
      (and (funcall pred obj)
            (funcall (get c 'predicate) obj))) ; Do this instead.
```

It is important to remember that function (and thus `#'`) is a special form, and thus only returns the right value when it is evaluated. A common error is to use `#'` notation in positions that are not evaluated:

```
(defvar *obscure-fns* '(#'cis #'cosh #'ash #'bit-orc2)) ; wrong
```

This does not create a list of four functions. Rather, it creates a list of four sublists; the first sublist is `(function cis)`. It is an error to `funcall` or `apply` such an object. The two correct ways to create a list of functions are shown below. The first assures that each function special form is evaluated, and the second uses function names instead of functions, thus relying on `funcall` or `apply` to coerce the names to the actual functions.

```
(defvar *obscure-fns* (list #'cis #'cosh #'ash #'bit-orc2))
(defvar *obscure-fns* '(cis cosh ash bit-orc2))
```

Another common error is to expect `#'if` or `#'or` to return a function. This is an error

because special forms are just syntactic markers. There is no function named `if` or `or`; they should be thought of as directives that tell the compiler what to do with a piece of code.

By the way, the function `make-specialization` above is bad not only for its lack of function but also for its use of backquote. The following is a better use of backquote:

```
(lambda (obj)
  (and (,pred obj)
       (,(get c 'predicate) obj)))
```

25.4 Values Change “by Themselves”

PROBLEM: You deleted/removed something, but it didn’t take effect. For example:

```
> (setf numbers '(1 2 3 4 5)) ⇒ (1 2 3 4 5)
> (remove 4 numbers) ⇒ (1 2 3 5)
> numbers ⇒ (1 2 3 4 5)
> (delete 1 numbers) ⇒ (2 3 4 5)
> numbers ⇒ (1 2 3 4 5)
```

Remedy: Use `(setf numbers (delete 1 numbers))`. Note that `remove` is a non-destructive function, so it will never alter its arguments. `delete` is destructive, but when asked to delete the first element of a list, it returns the rest of the list, and thus does not alter the list itself. That is why `setf` is necessary. Similar remarks hold for `nconc`, `sort`, and other destructive operations.

PROBLEM: You created a hundred different structures and changed a field in one of them. Suddenly, all the other ones magically changed!

Diagnosis: Different structures may share identical subfields. For example, suppose you had:

```
(defstruct block
  (possible-colors '(red green blue))
  ...)
```

```
(setf b1 (make-block))
(setf b2 (make-block))
...
(delete 'green (block-possible-colors b1))
```

Both `b1` and `b2` share the initial list of possible colors. The `delete` function modifies this shared list, so `green` is deleted from `b2`'s possible colors list just as surely as it is deleted from `b1`'s.

Remedy: Don't share pieces of data that you want to alter individually. In this case, either use `remove` instead of `delete`, or allocate a different copy of the list to each instance:

```
(defstruct block
  (possible-colors (list 'red 'green 'blue))
  ...)
```

Remember that the initial value field of a `defstruct` is an expression that is evaluated anew each time `make-block` is called. It is incorrect to think that the initial form is evaluated once when the `defstruct` is defined.

25.5 Built-In Functions Don't Find Elements

PROBLEM: You tried `(find item list)`, and you know it is there, but it wasn't found.

Diagnosis: By default, many built-in functions use `eq` as an equality test. `find` is one of them. If `item` is, say, a list that is `equal` but not `eq` to one of the elements of `list`, it will not be found.

Remedy: Use `(find item list :test #'equal)`

Diagnosis: If the `item` is `nil`, then `nil` will be returned whether it is found or not.

Remedy: Use `member` or `position` instead of `find` whenever the item can be `nil`.

25.6 Multiple Values Are Lost

PROBLEM: You only get one of the multiple values you were expecting.

Diagnosis: In certain contexts where a value must be tested by Lisp, multiple values are discarded. For example, consider:

```
(or (mv-1 x) (mv-2 x))
(and (mv-1 x) (mv-2 x))
(cond ((mv-1 x))
      (t (mv-2 x)))
```

In each case, if `mv-2` returns multiple values, they will all be passed on. But if `mv-1` returns multiple values, only the first value will be passed on. This is true even in the last clause of a `cond`. So, while the final clause `(t (mv-2 x))` passes on multiple values, the final clause `((mv-2 x))` would not.

Diagnosis: Multiple values can be inadvertently lost in debugging as well. Suppose I had:

```
(multiple-value-bind (a b c)
  (mv-1 x)
  ...)
```

Now, if I become curious as to what `mv-1` returns, I might change this code to:

```
(multiple-value-bind (a b c)
  (print (mv-1 x)) ;*** debugging output
  ...)
```

Unfortunately, `print` will see only the first value returned by `mv-1`, and will return only that one value to be bound to the variable `a`. The other values will be discarded, and `b` and `c` will be bound to `nil`.

25.7 Declarations Are Ignored

PROBLEM: Your program uses 1024×1024 arrays of floating-point numbers. But you find that it takes 15 seconds just to initialize such an array to zeros! Imagine how inefficient it is to actually do any computation! Here is your function that zeroes an array:

```
(defun zero-array (arr)
  "Set the 1024x1024 array to all zeros."
  (declare (type (array float) arr))
  (dotimes (i 1024)
    (dotimes (j 1024)
      (setf (aref arr i j) 0.0))))
```

Diagnosis: The main problem here is an ineffective declaration. The type `(array`

`float`) does not help the compiler, because the array could be displaced to an array of another type, and because `float` encompasses both single- and double-precision floating-point numbers. Thus, the compiler is forced to allocate storage for a new copy of the number 0.0 for each of the million elements of the array. The function is slow mainly because it generates so much garbage.

Remedy: The following version uses a much more effective type declaration: a simple array of single-precision numbers. It also declares the size of the array and turns safety checks off. It runs in under a second on a SPARCstation, which is slower than optimized C, but faster than unoptimized C.

```
(defun zero-array (arr)
  "Set the array to all zeros."
  (declare (type (simple-array single-float (1024 1024)) arr)
           (optimize (speed 3) (safety 0)))
  (dotimes (i 1024)
    (dotimes (j 1024)
      (setf (aref arr i j) 0.0))))
```

Another common error is to use something like `(simple-vector fixnum)` as a type specifier. It is a quirk of Common Lisp that the `simple-vector` type specifier only accepts a size, not a type, while the array, vector and `simple-array` specifiers all accept an optional type followed by an optional size or list of sizes. To specify a simple vector of fixnums, use `(simple-array fixnum (*))`.

To be precise, `simple-vector` means `(simple-array t (*))`. This means that `simple-vector` cannot be used in conjunction with any other type specifier. A common mistake is to think that the type (and `simple-vector (vector fixnum)`) is equivalent to `(simple-array fixnum (*))`, a simple, one-dimensional vector of fixnums. Actually, it is equivalent to `(simple-array t (*))`, a simple one-dimensional array of any type elements. To eliminate this problem, avoid `simple-vector` altogether.

25.8 My Lisp Does the Wrong Thing

When all else fails, it is tempting to shift the blame for an error away from your own code and onto the Common Lisp implementation. It is certainly true that errors are found in existing implementations. But it is also true that most of the time, Common Lisp is merely doing something the user did not expect rather than something that is in error.

For example, a common "bug report" is to complain about `read-from-string`. A user might write:

```
(read-from-string "a b c" :start 2)
```

expecting the expression to start reading at position 2 and thus return b. In fact, this expression returns a. The angry user thinks the implementation has erroneously ignored the `:start` argument and files a bug report,¹ only to get back the following explanation:

The function `read-from-string` takes two optional arguments, `eof-errorp` and `eof-value`, in addition to the keyword arguments. Thus, in the expression above, `:start` is taken as the value of `eof-errorp`, with 2 as the value of `eof-value`. The correct answer is in fact to read from the start of the string and return the very first form, a.

The functions `read-from-string` and `parse-namestring` are the only built-in functions that have this problem, because they are the only ones that have both optional and keyword arguments, with an even number of optional arguments. The functions `write-line` and `write-string` have keyword arguments and a single optional argument (the stream), so if the stream is accidentally omitted, an error will be signaled. (If you type `(write-line str :start 4)`, the system will complain either that `:start` is not a stream or that 4 is not a keyword.)

The moral is this: functions that have both optional and keyword arguments are confusing. Take care when using existing functions that have this problem, and abstain from using both in your own functions.

25.9 How to Find the Function You Want

Veteran Common Lisp programmers often experience a kind of software *déjà vu*: they believe that the code they are writing could be done by a built-in Common Lisp function, but they can't remember the name of the function.

Here's an example: while coding up a problem I realized I needed a function that, given the lists `(a b c d)` and `(c d)`, would return `(a b)`, that is, the part of the first list without the second list. I thought that this was the kind of function that might be in the standard, but I didn't know what it would be called. The desired function is similar to `set-difference`, so I looked that up in the index of *Common Lisp the Language* and was directed to page 429. I browsed through the section on "using lists as sets" but found nothing appropriate. However, I was reminded of the function `butlast`, which is also similar to the desired function. The index directed me to page 422 for `butlast`, and on the same page I found `ldiff`, which was exactly the desired function. It might have been easier to find (and remember) if it were called `list-difference`, but the methodology of browsing near similar functions paid off.

¹This misunderstanding has shown up even in published articles, such as Baker 1991.

If you think you know part of the name of the desired function, then you can use `apropos` to find it. For example, suppose I thought there was a function to push a new element onto the front of an array. Looking under `array`, `push-array`, and `array-push` in the index yields nothing. But I can turn to Lisp itself and ask:

```
> (apropos "push")
PUSH           Macro (VALUE PLACE), plist
PUSHNEW        Macro (VALUE PLACE &KEY ...), plist
VECTOR-PUSH     function (NEW-ELEMENT VECTOR), plist
VECTOR-PUSH-EXTEND function (DATA VECTOR &OPTIONAL ...), plist
```

This should be enough to remind me that `vector-push` is the answer. If not, I can get more information from the manual or from the online functions documentation or describe:

```
> (documentation 'vector-push 'function)
"Add NEW-ELEMENT as an element at the end of VECTOR.
The fill pointer (leader element 0) is the index of the next
element to be added. If the array is full, VECTOR-PUSH returns
NIL and the array is unaffected; use VECTOR-PUSH-EXTEND instead
if you want the array to grow automatically."
```

Another possibility is to browse through existing code that performs a similar purpose. That way, you may find the exact function you want, and you may get additional ideas on how to do things differently.

25.10 Syntax of LOOP

`loop` by itself is a powerful programming language, one with a syntax quite different from the rest of Lisp. It is therefore important to exercise restraint in using `loop`, lest the reader of your program become lost. One simple rule for limiting the complexity of loops is to avoid the `with` and `and` keywords. This eliminates most problems dealing with binding and scope.

When in doubt, macro-expand the loop to see what it actually does. But if you need to macro-expand, then perhaps it would be clearer to rewrite the loop with more primitive constructs.

25.11 Syntax of COND

For many programmers, the special form `cond` is responsible for more syntax errors than any other, with the possible exception of `loop`. Because most `cond`-clause start

with two left parentheses, beginners often come to the conclusion that every clause must. This leads to errors like the following:

```
(let ((entry (assoc item list)))
  (cond ((entry (process entry))
        ...))
```

Here `entry` is a variable, but the urge to put in an extra parenthesis means that the `cond`-clause attempts to call `entry` as a function rather than testing its value as a variable.

The opposite problem, leaving out a parenthesis, is also a source of error:

```
(cond (lookup item list)
      (t nil))
```

In this case, `lookup` is accessed as a variable, when the intent was to call it as a function. In Common Lisp this will usually lead to an unbound variable error, but in Scheme this bug can be very difficult to pin down: the value of `lookup` is the function itself, and since this is not null, the test will succeed, and the expression will return `list` without complaining.

The moral is to be careful with `cond`, especially when using Scheme. Note that `if` is much less error prone and looks just as nice when there are no more than two branches.

25.12 Syntax of CASE

In a `case` special form, each clause consists of a key or list of keys, followed by the value of that case. The thing to watch out for is when the key is `t`, otherwise, or `nil`. For example:

```
(case letter
  (s ...)
  (t ...)
  (u ...))
```

Here the `t` is taken as the default clause; it will always succeed, and all subsequent clauses will be ignored. Similarly, using a `()` or `nil` as a key will not have the desired effect: it will be interpreted as an empty key list. If you want to be completely safe, you can use a list of keys for every clause.² This is a particularly good idea when you

²Scheme requires a list of keys in each clause. Now you know why.

write a macro that expands into a case. The following code correctly tests for t and nil keys:

```
(case letter
  ((s) ...)
  ((t) ...)
  ((u) ...)
  ((nil) ...))
```

25.13 Syntax of LET and LET*

A common error is leaving off a layer of parentheses in let, just like in cond. Another error is to refer to a variable that has not yet been bound in a let. To avoid this problem, use let* whenever a variable's initial binding refers to a previous variable.

25.14 Problems with Macros

In section 3.2 we described a four-part approach to the design of macros:

- Decide if the macro is really necessary.
- Write down the syntax of the macro.
- Figure out what the macro should expand into.
- Use defmacro to implement the syntax/expansion correspondence.

This section shows the problems that can arise in each part, starting with the first:

- Decide if the macro is really necessary.

Macros extend the rules for evaluating an expression, while function calls obey the rules. Therefore, it can be a mistake to define too many macros, since they can make it more difficult to understand a program. A common mistake is to define macros that *do not* violate the usual evaluation rules. One recent book on AI programming suggests the following:

```
(defmacro binding-of (binding)           ; Warning!
  '(cadr ,binding))                    ; Don't do this.
```

The only possible reason for this macro is an unfounded desire for efficiency. Always use an inline function instead of a macro for such cases. That way you get the

efficiency gain, you have not introduced a spurious macro, and you gain the ability to apply or map the function #'binding-of, something you could not do with a macro:

```
(proclaim '(inline binding-of))
(defun binding-of (binding)           ; Do this instead.
  (second binding))
```

- Write down the syntax of the macro.

Try to make your macro follow conventions laid down by similar macros. For example, if your macro defines something, it should obey the conventions of `defvar`, `defstruct`, `defmacro`, and the rest: start with the letters `def`, take the name of the thing to be defined as the first argument, then a lambda-list if appropriate, then a value or body. It would be nice to allow for optional declarations and documentation strings.

If your macro binds some variables or variablelike objects, use the conventions laid down by `let`, `let*`, and `labels`: allow for a list of variable or (*variable init-val*) pairs. If you are iterating over some kind of sequence, follow `dotimes` and `dolist`. For example, here is the syntax of a macro to iterate over the leaves of a tree of conses:

```
(defmacro dotree ((var tree &optional result) &body body)
  "Perform body with var bound to every leaf of tree,
  then return result. Return and Go can be used in body."
  ...)
```

- Figure out what the macro should expand into.
- Use `defmacro` to implement the syntax/expansion correspondence.

There are a number of things to watch out for in figuring out how to expand a macro. First, make sure you don't shadow local variables. Consider the following definition for `pop-end`, a function to pop off and return the last element of a list, while updating the list to no longer contain the last element. The definition uses `last1`, which was defined on page 305 to return the last element of a list, and the built-in function `nbutlast` returns all but the last element of a list, destructively altering the list.

```
(defmacro pop-end (place)           ; Warning! Buggy!
  "Pop and return last element of the list in PLACE."
  '(let ((result (last1 ,place)))
    (setf ,place (nbutlast ,place))
    result))
```

This will do the wrong thing for `(pop-end result)`, or for other expressions that mention the variable `result`. The solution is to use a brand new local variable that could not possibly be used elsewhere:

```
(defmacro pop-end (place) ; Less buggy
  "Pop and return last element of the list in PLACE."
  (let ((result (gensym)))
    `(let ((,result (last1 ,place)))
      (setf ,place (nbutlast ,place))
      ,result)))
```

There is still the problem of shadowing local *functions*. For example, a user who writes:

```
(flet ((last1 (x) (sqrt x)))
  (pop-end list)
  ...)
```

will be in for a surprise. `pop-end` will expand into code that calls `last1`, but since `last1` has been locally defined to be something else, the code won't work. Thus, the expansion of the macro violates referential transparency. To be perfectly safe, we could try:

```
(defmacro pop-end (place) ; Less buggy
  "Pop and return last element of the list in PLACE."
  (let ((result (gensym)))
    `(let ((,result (funcall ,#'last1 ,place)))
      (setf ,place (funcall ,#'nbutlast ,place))
      ,result)))
```

This approach is sometimes used by Scheme programmers, but Common Lisp programmers usually do not bother, since it is rarer to define local functions in Common Lisp. Indeed, in *Common Lisp the Language*, 2d edition, it was explicitly stated (page 260) that a user function cannot redefine or even bind any built-in function, variable, or macro. Even if it is not prohibited in your implementation, redefining or binding a built-in function is confusing and should be avoided.

Common Lisp programmers expect that arguments will be evaluated in left-to-right order, and that no argument is evaluated more than once. Our definition of `pop-end` violates the second of these expectations. Consider:

```
(pop-end (aref lists (incf i))) ≡
(LET ((#:G3096 (LAST1 (AREF LISTS (INCF I))))
      (SETF (AREF LISTS (INCF I)) (NBUTLAST (AREF LISTS (INCF I))))
      #:G3096)
```

This increments `i` three times, when it should increment it only once. We could fix this by introducing more local variables into the expansion:

```
(let* ((temp1 (incf i))
      (temp2 (AREF LISTS temp1))
      (temp3 (LAST1 temp2)))
  (setf (aref lists temp1) (nbutlast temp2))
  temp3)
```

This kind of left-to-right argument processing via local variables is done automatically by the Common Lisp `setf` mechanism. Fortunately, the mechanism is easy to use. We can redefine `pop-end` to call `pop` directly:

```
(defmacro pop-end (place)
  "Pop and return last element of the list in PLACE."
  `(pop (last ,place)))
```

Now all we need to do is define the `setf` method for `last`. Here is a simple definition. It makes use of the function `last2`, which returns the last two elements of a list. In ANSI Common Lisp we could use `(last list 2)`, but with a pre-ANSI compiler we need to define `last2`:

```
(defsetf last (place) (value)
  `(setf (cdr (last2 ,place)) ,value))

(defun last2 (list)
  "Return the last two elements of a list."
  (if (null (rest2 list))
      list
      (last2 (rest list))))
```

Here are some macro-expansions of calls to `pop-end` and to the `setf` method for `last`. Different compilers will produce different code, but they will always respect the left-to-right, one-evaluation-only semantics:

```
> (pop-end (aref (foo lists) (incf i))) ≡
(LET ((G0128 (AREF (FOO LISTS) (SETQ I (+ I 1)))))
  (PROG1
    (CAR (LAST G0128))
    (SYS:SETCDR (LAST2 G0128) (CDR (LAST G0128)))))

> (setf (last (append x y)) 'end) ≡
(SYS:SETCDR (LAST2 (APPEND X Y)) 'END)
```

Unfortunately, there is an error in the `setf` method for `last`. It assumes that the list will have at least two elements. If the list is empty, it is probably an error, but if a list has exactly one element, then `(setf (last list) val)` should have the same effect as `(setf list val)`. But there is no way to do that with `defsetf`, because the

`setf` method defined by `defsetf` never sees *list* itself. Instead, it sees a local variable that is automatically bound to the value of *list*. In other words, `defsetf` evaluates the *list* and *val* for you, so that you needn't worry about evaluating the arguments out of order, or more than once.

To solve the problem we need to go beyond the simple `defsetf` macro and delve into the complexities of `define-setf-method`, one of the trickiest macros in all of Common Lisp. `define-setf-method` defines a `setf` method not by writing code directly but by specifying five values that will be used by Common Lisp to write the code for a call to `setf`. The five values give more control over the exact order in which expressions are evaluated, variables are bound, and results are returned. The five values are: (1) a list of temporary, local variables used in the code; (2) a list of values these variables should be bound to; (3) a list of one variable to hold the value specified in the call to `setf`; (4) code that will store the value in the proper place; (5) code that will access the value of the place. This is necessary for variations of `setf` like `incf` and `pop`, which need to both access and store.

In the following `setf` method for `last`, then, we are defining the meaning of `(setf (last place) value)`. We keep track of all the variables and values needed to evaluate `place`, and add to that three more local variables: `last2-var` will hold the last two elements of the list, `last2-p` will be true only if there are two or more elements in the list, and `last-var` will hold the form to access the last element of the list. We also make up a new variable, `result`, to hold the value. The code to store the value either modifies the `cdr` of `last2-var`, if the list is long enough, or it stores directly into `place`. The code to access the value just retrieves `last-var`.

```
(define-setf-method last (place)
  (multiple-value-bind (temps vals stores store-form access-form)
    (get-setf-method place)
    (let ((result (gensym))
          (last2-var (gensym))
          (last2-p (gensym))
          (last-var (gensym)))
      ;; Return 5 vals: temps vals stores store-form access-form
      (values
        '(@temps ,last2-var ,last2-p ,last-var)
        '(@vals (last2 ,access-form)
              (= (length ,last2-var) 2)
              (if ,last2-p (rest ,last2-var) ,access-form))
        (list result)
        '(if ,last2-p
            (setf (cdr ,last2-var) ,result)
            (let ((,(first stores) ,result))
              ,store-form))
        last-var))))
```

It should be mentioned that `setf` methods are very useful and powerful things. It is often better to provide a `setf` method for an arbitrary function, `f`, than to define a special setting function, say, `set-f`. The advantage of the `setf` method is that it can be used in idioms like `incf` and `pop`, in addition to `setf` itself. Also, in ANSI Common Lisp, it is permissible to name a function with `#'` (`setf f`), so you can also use `map` or apply the `setf` method. Most `setf` methods are for functions that just access data, but it is permissible to define `setf` methods for functions that do any computation whatsoever. As a rather fanciful example, here is a `setf` method for the square-root function. It makes `(setf (sqrt x) 5)` be almost equivalent to `(setf x (* 5 5))`; the difference is that the first returns 5 while the second returns 25.

```
(define-setf-method sqrt (num)
  (multiple-value-bind (temps vals stores store-form access-form)
    (get-setf-method num)
    (let ((store (gensym)))
      (values temps
              vals
              (list store)
              '(let ((,(first stores) (* ,store ,store))
                  ,store-form
                  ,store)
                '(sqrt ,access-form))))))
```

Turning from `setf` methods back to macros, another hard part about writing portable macros is anticipating what compilers might warn about. Let's go back to the `dotree` macro. Its definition might look in part like this:

```
(defmacro dotree ((var tree &optional result) &body body)
  "Perform body with var bound to every leaf of tree,
  then return result. Return and Go can be used in body."
  `(let ((,var))
      ...
      ,@body))
```

Now suppose a user decides to count the leaves of a tree with:

```
(let ((count 0))
  (dotree (leaf tree count)
    (incf count)))
```

The problem is that the variable `leaf` is not used in the body of the macro, and a compiler may well issue a warning to that effect. To make matters worse, a conscientious user might write:

```
(let ((count 0))
  (dotree (leaf tree count)
    (declare (ignore leaf))
    (incf count)))
```

The designer of a new macro must decide if declarations are allowed and must make sure that compiler warnings will not be generated unless they are warranted.

Macros have the full power of Lisp at their disposal, but the macro designer must remember the purpose of a macro is to translate macro code into primitive code, and not to do any computations. Consider the following macro, which assumes that `translate-rule-body` is defined elsewhere:

```
(defmacro defrule (name &body body) ; Warning! buggy!
  "Define a new rule with the given name."
  (setf (get name 'rule)
    #'(lambda () ,(translate-rule-body body))))
```

The idea is to store a function under the `rule` property of the rule's name. But this definition is incorrect because the function is stored as a side effect of expanding the macro, rather than as an effect of executing the expanded macro code. The correct definition is:

```
(defmacro defrule (name &body body)
  "Define a new rule with the given name."
  '(setf (get ',name 'rule)
    #'(lambda () ,(translate-rule-body body))))
```

Beginners sometimes fail to see the difference between these two approaches, because they both have the same result when interpreting a file that makes use of `defrule`. But when the file is compiled and later loaded into a different Lisp image, the difference becomes clear: the first definition erroneously stores the function in the compiler's image, while the second produces code that correctly stores the function when the code is loaded.

Beginning macro users have asked, "How can I have a macro that expands into code that does more than one thing? Can I splice in the results of a macro?"

If by this the beginner wants a macro that just *does* two things, the answer is simply to use a `progn`. There will be no efficiency problem, even if the `progn` forms are nested. That is, if macro-expansion results in code like:

```
(progn (progn (progn a b) c) (progn d e))
```

the compiler will treat it the same as `(progn a b c d e)`.

On the other hand, if the beginner wants a macro that *returns* two values, the proper form is `values`, but it must be understood that the calling function needs to arrange specially to see both values. There is no way around this limitation. That is, there is no way to write a macro—or a function for that matter—that will “splice in” its results to an arbitrary call. For example, the function `floor` returns two values (the quotient and remainder), as does `intern` (the symbol and whether or not the symbol already existed). But we need a special form to capture these values. For example, compare:

```
> (list (floor 11 5) (intern 'x)) ⇒ (2 X)
> (multiple-value-call #'list
   (floor 11 5) (intern 'x)) ⇒ (2 1 X :INTERNAL)
```

25.15 A Style Guide to Lisp

In a sense, this whole book is a style guide to writing quality Lisp programs. But this section attempts to distill some of the lessons into a set of guidelines.

When to Define a Function

Lisp programs tend to consist of many short functions, in contrast to some languages that prefer a style using fewer, longer functions. New functions should be introduced for any of the following reasons:

1. For a specific, easily stated purpose.
2. To break up a function that is too long.
3. When the name would be useful documentation.
4. When it is used in several places.

In (2), it is interesting to consider what “too long” means. Charniak et al. (1987) suggested that 20 lines is the limit. But now that large bit-map displays have replaced 24-line terminals, function definitions have become longer. So perhaps one screenful is a better limit than 20 lines. The addition of `flet` and `labels` also contributes to longer function definitions.

When to Define a Special Variable

In general, it is a good idea to minimize the use of special variables. Lexical variables are easier to understand, precisely because their scope is limited. Try to limit special variables to one of the following uses:

1. For parameters that are used in many functions spread throughout a program.
2. For global, persistent, mutable data, such as a data base of facts.
3. For infrequent but deeply nested use.

An example of (3) might be a variable like `*standard-output*`, which is used by low-level printing functions. It would be confusing to have to pass this variable around among all your high-level functions just to make it available to `print`.

When to Bind a Lexical Variable

In contrast to special variables, lexical variables are encouraged. You should feel free to introduce a lexical variable (with a `let`, `lambda` or `defun`) for any of the following reasons:

1. To avoid typing in the same expression twice.
2. To avoid computing the same expression twice.
3. When the name would be useful documentation.
4. To keep the indentation manageable.

How to Choose a Name

Your choice of names for functions, variables, and other objects should be clear, meaningful, and consistent. Some of the conventions are listed here:

1. Use mostly letters and hyphens, and use full words: `delete-file`.
2. You can introduce an abbreviation if you are consistent: `get-dtree`, `dtree-fetch`. For example, this book uses `fn` consistently as the abbreviation for "function."
3. Predicates end in `-p` (or `?` in Scheme), unless the name is already a predicate: `variable-p`, `occurs-in`.
4. Destructive functions start with `n` (or end in `!` in Scheme): `nreverse`.

5. Generalized variable-setting macros end in *f*: *setf*, *incf*. (Push is an exception.)
6. Slot selectors created by *defstruct* are of the form *type-slot*. Use this for non-*defstruct* selectors as well: *char-bits*.
7. Many functions have the form *action-object*: *copy-list*, *delete-file*.
8. Other functions have the form *object-modifier*: *list-length*, *char-lessp*. Be consistent in your choice between these two forms. Don't have *print-edge* and *vertex-print* in the same system.
9. A function of the form *modulename-functionname* is an indication that packages are needed. Use *parser:print-tree* instead of *parser-print-tree*.
10. Special variables have asterisks: **db**, **print-length**.
11. Constants do not have asterisks: *pi*, *most-positive-fixnum*.
12. Parameters are named by type: (*defun length (sequence) ...*) or by purpose: (*defun subsetp (subset superset) ...*) or both: (*defun / (number &rest denominator-numbers) ...*)
13. Avoid ambiguity. A variable named *last-node* could have two meanings; use *previous-node* or *final-node* instead.
14. A name like *propagate-constraints-to-neighboring-vertexes* is too long, while *prp-con* is too short. In deciding on length, consider how the name will be used: *propagate-constraints* is just right, because a typical call will be (*propagate-constraints vertex*), so it will be obvious what the constraints are propagating to.

Deciding on the Order of Parameters

Once you have decided to define a function, you must decide what parameters it will take, and in what order. In general,

1. Put important parameters first (and optional ones last).
2. Make it read like prose if possible: (*push element stack*).
3. Group similar parameters together.

Interestingly, the choice of a parameter list for top-level functions (those that the user is expected to call) depends on the environment in which the user will function. In many systems the user can type a keystroke to get back the previous input to the top

level, and can then edit that input and re-execute it. In these systems it is preferable to have the parameters that are likely to change be at the end of the parameter list, so that they can be easily edited. On systems that do not offer this kind of editing, it is better to either use keyword parameters or make the highly variable parameters first in the list (with the others optional), so that the user will not have to type as much.

Many users want to have *required* keyword parameters. It turns out that all keyword parameters are optional, but the following trick is equivalent to a required keyword parameter. First we define the function `required` to signal an error, and then we use a call to `required` as the default value for any keyword that we want to make required:

```
(defun required ()
  (error "A required keyword argument was not supplied."))

(defun fn (x &key (y (required)))
  ...)
```

25.16 Dealing with Files, Packages, and Systems

While this book has covered topics that are more advanced than any other Lisp text available, it is still concerned only with programming in the small: a single project at a time, capable of being implemented by a single programmer. More challenging is the problem of programming in the large: building multiproject, multiprogrammer systems that interact well.

This section briefly outlines an approach to organizing a larger project into manageable components, and how to place those components in files.

Every system should have a separate file that defines the other files that comprise the system. I recommend defining any packages in that file, although others put package definitions in separate files.

The following is a sample file for the mythical system Project-X. Each entry in the file is discussed in turn.

1. The first line is a comment known as the *mode line*. The text editor emacs will parse the characters between `-*-` delimiters to discover that the file contains Lisp code, and thus the Lisp editing commands should be made available. The dialect of Lisp and the package are also specified. This notation is becoming widespread as other text editors emulate emacs's conventions.
2. Each file should have a description of its contents, along with information on the authors and what revisions have taken place.

3. Comments with four semicolons (; ; ; ;) denote header lines. Many text editors supply a command to print all such lines, thus achieving an outline of the major parts of a file.
4. The first executable form in every file should be an `in-package`. Here we use the user package. We will soon create the `project-x` package, and it will be used in all subsequent files.
5. We want to define the Project-X system as a collection of files. Unfortunately, Common Lisp provides no way to do that, so we have to load our own system-definition functions explicitly with a call to `load`.
6. The call to `define-system` specifies the files that make up Project-X. We provide a name for the system, a directory for the source and object files, and a list of *modules* that make up the system. Each module is a list consisting of the module name (a symbol) followed by a one or more files (strings or pathnames). We have used keywords as the module names to eliminate any possible name conflicts, but any symbol could be used.
7. The call to `defpackage` defines the package `project-x`. For more on packages, see section 24.1.
8. The final form prints instructions on how to load and run the system.

```

;;; -*- Mode: Lisp; Syntax: Common-Lisp; Package: User -*-
;;; (Brief description of system here.)
;;; Define the Project-X system.
(in-package "USER")

(load "/usr/norvig/defsys.lisp") ; load define-system

(define-system ;; Define the system Project-X
  :name :project-x
  :source-dir "/usr/norvig/project-x/*.lisp"
  :object-dir "/usr/norvig/project-x/*.bin"
  :modules '(:macros "header" "macros")
            (:main "parser" "transformer" "optimizer"
                  "commands" "database" "output")
            (:windows "xwindows" "clx" "client")))

(defpackage :project-x ;; Define the package Project-X
  (:export "DEFINE-X" "DO-X" "RUN-X")
  (:nicknames "PX")
  (:use common-lisp))

```

```
(format *debug-io* "~& To load the Project-X system, type
  (make-system :name :project-x)
To run the system, type
  (project-x:run-x)")
```

Each of the files that make up the system will start like this:

```
;;; -*- Mode: Lisp; Syntax: Common-Lisp; Package: Project-X -*-

(in-package "PROJECT-X")
```

Now we need to provide the system-definition functions, `define-system` and `make-system`. The idea is that `define-system` is used to define the files that make up a system, the modules that the system is comprised of, and the files that make up each module. It is necessary to group files into modules because some files may depend on others. For example, all macros, special variables, constants, and inline functions need to be both compiled and loaded before any other files that reference them are compiled. In Project-X, all `defvar`, `defparameter`, `defconstant`, and `defstruct`³ forms are put in the file header, and all `defmacro` forms are put in the file macros. Together these two files form the first module, named `:macros`, which will be loaded before the other two modules (`:main` and `:windows`) are compiled and loaded.

`define-system` also provides a place to specify a directory where the source and object files will reside. For larger systems spread across multiple directories, `define-system` will not be adequate.

Here is the first part of the file `defsys.lisp`, showing the definition of `define-system` and the structure `sys`.

```
;;; -*- Mode: Lisp; Syntax: Common-Lisp; Package: User -*-

;;; A Facility for Defining Systems and their Components

(in-package "USER")

(defvar *systems* nil "List of all systems defined.")

(defstruct sys
  "A system containing a number of source and object files."
  name source-dir object-dir modules)
```

³`defstruct` forms are put here because they may create inline functions.

```
(defun define-system (&key name source-dir object-dir modules)
  "Define a new system."
  ;; Delete any old system of this name, and add the new one.
  (setf *systems* (delete name *systems* :test #'string-equal
                          :key #'sys-name))

  (push (make-sys
        :name (string name)
        :source-dir (pathname source-dir)
        :object-dir (pathname object-dir)
        :modules '((:all .,(mapcar #'first modules)) .,modules))
        *systems*)
  name)
```

The function `make-system` is used to compile and/or load a previously defined system. The name supplied is used to look up the definition of a system, and one of three actions is taken on the system. The keyword `:cload` means to compile and then load files. `:load` means to load files; if there is an object (compiled) file and it is newer than the source file, then it will be loaded, otherwise the source file will be loaded. Finally, `:update` means to compile just those source files that have been changed since their corresponding source files were last altered, and to load the new compiled version.

```
(defun make-system (&key (module :all) (action :cload)
                  (name (sys-name (first *systems*))))
  "Compile and/or load a system or one of its modules."
  (let ((system (find name *systems* :key #'sys-name
                    :test #'string-equal)))
    (check-type system (not null))
    (check-type action (member :cload :update :load))
    (with-compilation-unit () (sys-action module system action)))

  (defun sys-action (x system action)
    "Perform the specified action to x in this system.
X can be a module name (symbol), file name (string)
or a list."
    (typecase x
      (symbol (let ((files (rest (assoc x (sys-modules system))))
                    (if (null files)
                        (warn "No files for module ~a" x)
                        (sys-action files system action))))
      (list (dolist (file x)
             (sys-action file system action)))
      ((string pathname)
       (let ((source (merge-pathnames
                      x (sys-source-dir system)))
             (object (merge-pathnames
                      x (sys-object-dir system))))
         (case action
```

```

(:cload (compile-file source) (load object))
(:update (unless (newer-file-p object source)
                (compile-file source)
                (load object))
          (load object)
          (load source))))
(t (warn "Don't know how to ~a ~a in system ~a"
        action x system)))

```

To support this, we need to be able to compare the write dates on files. This is not hard to do, since Common Lisp provides the function `file-write-date`.

```

(defun newer-file-p (file1 file2)
  "Is file1 newer than (written later than) file2?"
  (>-num (if (probe-file file1) (file-write-date file1)
             (if (probe-file file2) (file-write-date file2))))

(defun >-num (x y)
  "True if x and y are numbers, and x > y."
  (and (numberp x) (numberp y) (> x y)))

```

25.17 Portability Problems

Programming is difficult. All programmers know the frustration of trying to get a program to work according to the specification. But one thing that really defines the professional programmer is the ability to write portable programs that will work on a variety of systems. A portable program not only must work on the computer it was tested on but also must anticipate the difference between your computer and other ones. To do this, you must understand the Common Lisp specification in the abstract, not just how it is implemented on your particular machine.

There are three ways in which Common Lisp systems can vary: in the treatment of “is an error” situations, in the treatment of unspecified results, and in extensions to the language.

Common Lisp the Language specifies that it “is an error” to pass a non-number to an arithmetic function. For example, it is an error to evaluate `(+ nil 1)`. However, it is not specified what should be done in this situation. Some implementations may signal an error, but others may not. An implementation would be within its right to return 1, or any other number or non-number as the result.

An unsuspecting programmer may code an expression that is an error but still computes reasonable results in his or her implementation. A common example is applying `get` to a non-symbol. This is an error, but many implementations will

just return nil, so the programmer may write `(get x 'prop)` when `(if (symbolp x) (get x 'prop) nil)` is actually needed for portable code. Another common problem is with `subseq` and the sequence functions that take `:end` keywords. It is an error if the `:end` parameter is not an integer less than the length of the sequence, but many implementations will not complain if `:end` is nil or is an integer greater than the length of the sequence.

The Common Lisp specification often places constraints on the result that a function must compute, without fully specifying the result. For example, both of the following are valid results:

```
> (union '(a b c) '(b c d)) => (A B C D)
> (union '(a b c) '(b c d)) => (D A B C)
```

A program that relies on one order or the other will not be portable. The same warning applies to `intersection` and `set-difference`. Many functions do not specify how much the result shares with the input. The following computation has only one possible printed result:



```
> (remove 'x '(a b c d)) => (A B C D)
```

However, it is not specified whether the output is `eq` or only `equal` to the second input.


Input/output is particularly prone to variation, as different operating systems can have very different conceptions of how I/O and the file system works. Things to watch out for are whether `read-char` echoes its input or not, the need to include `finish-output`, and variation in where newlines are needed, particularly with respect to the top level.

Finally, many implementations provide extensions to Common Lisp, either by adding entirely new functions or by modifying existing functions. The programmer must be careful not to use such extensions in portable code.

25.18 Exercises


-  **Exercise 25.1 [h]** On your next programming project, keep a log of each bug you detect and its eventual cause and remedy. Classify each one according to the taxonomy given in this chapter. What kind of mistakes do you make most often? How could you correct that?
-  **Exercise 25.2 [s-d]** Take a Common Lisp program and get it to work with a different compiler on a different computer. Make sure you use conditional compilation read

macros (#+ and #-) so that the program will work on both systems. What did you have to change?

 **Exercise 25.3 [m]** Write a `setf` method for `if` that works like this:

```
(setf (if test (first x) y) (+ 2 3)) ≡
(let ((temp (+ 2 3)))
  (if test
    (setf (first x) temp)
    (setf y temp)))
```

You will need to use `define-setf-method`, not `defsetf`. (Why?) Make sure you handle the case where there is no else part to the `if`.

 **Exercise 25.4 [h]** Write a `setf` method for `lookup`, a function to get the value for a key in an association list.

```
(defun lookup (key alist)
  "Get the cdr of key's entry in the association list."
  (cdr (assoc key alist)))
```

25.19 Answers

Answer 25.4 Here is the `setf` method for `lookup`. It looks for the key in the a-list, and if the key is there, it modifies the `cdr` of the pair containing the key; otherwise it adds a new key/value pair to the front of the a-list.

```
(define-setf-method lookup (key alist-place)
  (multiple-value-bind (temps vals stores store-form access-form)
    (get-setf-method alist-place)
    (let ((key-var (gensym))
          (pair-var (gensym))
          (result (gensym)))
      (values
       '(,key-var ,@temps ,pair-var)
       '(,key ,@vals (assoc ,key-var ,access-form))
       '(,result)
       '(if ,pair-var
           (setf (cdr ,pair-var) ,result)
           (let ((,(first stores)
                  (acons ,key-var ,result ,access-form)))
             ,store-form
             ,result)))
       '(cdr ,pair-var))))))
```

Appendix: Obtaining the Code in this Book

FTP: The File Transfer Protocol

FTP is a file transfer protocol that is widely accepted by computers around the world. FTP makes it easy to transfer files between two computers on which you have accounts. But more importantly, it also allows a user on one computer to access files on a computer on which he or she does not have an account, as long as both computers are connected to the Internet. This is known as *anonymous FTP*.

All the code in this book is available for anonymous FTP from the computer `mkp.com` in files in the directory `pub/norvig`. The file `README` in that directory gives further instructions on using the files.

In the session below, the user `smith` retrieves the files from `mkp.com`. Smith's input is in *slanted font*. The login name must be *anonymous*, and Smith's own mail address is used as the password. The command `cd pub/norvig` changes to that directory, and the command `ls` lists all the files. The command `mget *` retrieves all files (the *m* stands for "multiple"). Normally, there would be a prompt before each file asking if you do indeed want to copy it, but the *prompt* command disabled this. The command `bye` ends the FTP session.

```
% ftp mkp.com (or ftp 199.182.55.2)
Name (mkp.com:smith): anonymous
331 Guest login ok, send ident as password
Password: smith@cs.stateu.edu
230 Guest login ok, access restrictions apply
ftp> cd pub/norvig
```

```
250 CWD command successful.
ftp> ls
...
ftp> prompt
Interactive mode off.
ftp> mget *
...
ftp> bye
%
```

Anonymous FTP is a privilege, not a right. The site administrators at `mkp.com` and at other sites below have made their systems available out of a spirit of sharing, but there are real costs that must be paid for the connections, storage, and processing that makes this sharing possible. To avoid overloading these systems, do not FTP from 7:00 a.m. to 6:00 p.m. local time. This is especially true for sites not in your country. If you are using this book in a class, ask your professor for a particular piece of software before you try to FTP it; it would be wasteful if everybody in the class transferred the same thing. Use common sense and be considerate: none of us want to see sites start to close down because a few are abusing their privileges.

If you do not have FTP access to the Internet, you can still obtain the files from this book by contacting Morgan Kaufmann at the following:

Morgan Kaufmann Publishers, Inc.
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205
USA
Telephone 415/392-2665
Facsimile 415/982-2665
Internet `mkp@mkp.com`
(800) 745-7323

Make sure to specify which format you want:

Macintosh diskette ISBN 1-55860-227-5
DOS 5.25 diskette ISBN 1-55860-228-3
DOS 3.5 diskette ISBN 1-55860-229-1

Available Software

In addition to the program from this book, a good deal of other software is available. The tables below list some of the relevant AI/Lisp programs. Each entry lists the name of the system, an address, and some comments. The address is either a computer from which you can FTP, or a mail address of a contact. Unless it is stated that distribution is by *email* or *Floppy* or requires a *license*, then you can FTP from the contact's home computer. In some cases the host computer and/or directory have

been provided in italics in the comments field. However, in most cases it should be obvious what files to transfer. First do an `ls` command to see what files and directories are available. If there is a file called `README`, follow its advice: do a `get README` and then look at the file. If you still haven't found what you are looking for, be aware that most hosts keep their public software in the directory `pub`. Do a `cd pub` and then another `ls`, and you should find the desired files.

If a file ends in the suffix `.Z`, then you should give the FTP command `binary` before transferring it, and then give the UNIX command `uncompress` to recover the original file. Files with the suffix `.tar` contain several files that can be unpacked with the `tar` command. If you have problems, consult your local documentation or system administrator.

Knowledge Representation

System	Address	Comments
Babbler	rsfl@ra.msstate.edu	<i>email</i> ; Markov chains/NLP
BACK	peltason@tubvm.cs.tu-berlin.de	3.5" <i>floppy</i> ; KL-ONE family
Belief	almond@stat.washington.edu	belief networks
Classic	d1m@research.att.com	<i>license</i> ; KL-ONE family
Fol Getfol	fausto@irst.it	<i>tape</i> ; Weyrauch's FOL system
Framekit	ehnt+@cs.cmu.edu	<i>floppy</i> ; frames
FrameWork	mkant+@cs.cmu.edu	<i>a.gp.cs.cmu.edu:/usr/mkant/Public</i> ; frames
Frobs	kessler@cs.utah.edu	frames
Knowbel	kramer@ai.toronto.edu	sorted/temporal logic
MVL	ginsberg@t.stanford.edu	multivalued logics
OPS	slisp-group@b.gp.cs.cmu.edu	Forgy's OPS-5 language
PARKA	spector@cs.umd.edu	frames (designed for connection machine)
Parmenides	pshell@cs.cmu.edu	frames
Rhetorical	miller@cs.rochester.edu	planning, time logic
SB-ONE	kobsa@cs.uni-sb.de	<i>license</i> ; in German; KL-ONE family
SNePS	shapiro@cs.buffalo.edu	<i>license</i> ; semantic net/NLP
SPI	cs.orst.edu	Probabilistic inference
YAK	franconi@irst.it	KL-ONE family

Planning and Learning

System	Address	Comments
COBWEB/3	cobweb@ptolemy.arc.nasa.gov	<i>email</i> ; concept formation
MATS	kautz@research.att.com	<i>license</i> ; temporal constraints
MICRO-xxx	waander@cs.ume.edu	case-based reasoning
Nonlin	nonlin-users-request@cs.umd.edu	Tate's planner in Common Lisp
Prodigy	prodigy@cs.cmu.edu	<i>license</i> ; planning and learning
PROTOS	porter@cs.utexas.edu	knowledge acquisition
SNLP	weld@cs.washington.edu	nonlinear planner
SOAR	soar-requests/@cs.cmu.edu	<i>license</i> ; integrated architecture
THEO	tom.mitchell@cs.cmu.edu	frames, learning
Tileworld	pollack@ai.sri.com	planning testbed
TileWorld	tileworld@ptolemy.arc.nasa.gov	planning testbed

Mathematics

System	Address	Comments
JACAL	jaffer@altdorf.ai.mit.edu	algebraic manipulation
Maxima	rascal.ics.utexas.edu	version of Macsyma; also proof-checker, nqthm
MMA	fateman@cs.berkeley.edu	<i>peoplesparc.berkeley.edu:pub/mma.*</i> ; algebra
XLispStat	umnstat.stat.umn.edu	Statistics; also S Bayes

Compilers and Utilities

System	Address	Comments
AKCL	rascal.ics.utexas.edu	Austin Koyoto Common Lisp
CLX, CLUE	export.lcs.mit.edu	Common Lisp interface to X Windows
Gambit	gambit@cs.brandeis.edu	<i>acorn.cs.brandeis.edu:dist/gambit*</i> ; Scheme compiler
ISI Grapher	isi.edu	Graph displayer; also NLP word lists
PCL	arisia.xerox.com	Implementation of CLOS
Prolog	aisun1.ai.uga.edu	Prolog-based utilities and NLP programs
PYTHON	ram+@cs.cmu.edu	<i>a.gp.cs.cmu.edu</i> : Common Lisp Compiler and tools
SBProlog	arizona.edu	Stony Brook Prolog, Icon, Snobol
Scheme	altdorf.ai.mit.edu	Scheme utilities and compilers
Scheme	scheme@nexus.yorku.ca	Scheme utilities and programs
SIOD	bu.edu	<i>users/gjc</i> ; small scheme interpreter
Utilities	a.gp.cs.cmu.edu	<i>/usr/mkant/Public</i> ; profiling, defsystem, etc.
XLisp	cs.orst.edu	Lisp interpreter
XScheme	tut.cis.ohio-state.edu	Also mitscheme compiler; sbprolog

Bibliography

- Abelson, Harold, and Gerald J. Sussman, with Julie Sussman. (1985) *Structure and Interpretation of Computer Programs*. MIT Press.
- Aho, A. V., and J. D. Ullman. (1972) *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall.
- Ait-Kaci, Hassan. (1991) *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press. An earlier version was published as "The WAM: A (Real) Tutorial." Digital Equipment Corporation Paris Research Lab, Report no. 5.
- Ait-Kaci, Hassan, Patrick Lincoln, and Roger Nasr. (1987) "Le Fun: Logic, Equations and Functions." *Proceedings of the IEEE*, CH2472-9/87.
- Allen, James. (1987) *Natural Language Understanding*. Benjamin/Cummings.
- Allen, James, James Hendler, and Austin Tate. (1990) *Readings in Planning*. Morgan Kaufmann.
- Allen, John. (1978) *Anatomy of Lisp*. McGraw-Hill.
- Amarel, Saul. (1968) "On Representation of Problems of Reasoning about Actors." In *Machine Intelligence 3*, ed. Donald Michie. Edinburgh University Press.
- Anderson, James A. D. W. (1989) *Pop-11 Comes of Age: the advancement of an AI programming language*. Ellis Horwood.
- Anderson, John Robert. (1976) *Language, Memory, and Thought*. Lawrence Erlbaum.
- Baker, Henry G. (1991) "Pragmatic Parsing in Common Lisp; or, Putting defmacro on Steroids." *Lisp Pointers* 4, no. 2.
- Barr, Avron, and Edward A. Feigenbaum. (1981) *The Handbook of Artificial Intelligence*. 3 vols. Morgan Kaufmann.

- Batali, John, Edmund Goodhue, Chris Hanson, Howie Shrobe, Richard M. Stallman, and Gerald Jay Sussman. (1982) "The Scheme-81 Architecture—System and Chip." In *Proceedings, Conference on Advanced Research in VLSI*, 69–77.
- Bennett, James S. (1985) "Roget: A Knowledge-Based System for Acquiring the Conceptual Structure of a Diagnostic Expert System." *Journal of Automated Reasoning* 1: 49–74.
- Berlekamp, E. R., J. H. Conway, and R. K. Guy. (1982) *Winning Ways*. 2 vols. Academic Press.
- Berlin, Andrew, and Daniel Weise. (1990) "Compiling scientific code using partial evaluation." *IEEE Computer*, 25–37.
- Bobrow, Daniel G. (1968) "Natural Language Input for a Computer Problem-Solving System." In Minsky 1968.
- Bobrow, Daniel G. (1982) *LOOPS: An Object-Oriented Programming System for Interlisp*. Xerox PARC.
- Bobrow, Daniel G. (1985) "If Prolog is the Answer, What is the Question? or What It Takes to Support AI Programming Paradigms." *IEEE Transactions on Software Engineering*, SE-11.
- Bobrow, Daniel G., Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. (1986) "Common Loops: Merging Lisp and Object-Oriented Programming." *Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications*.
- Boyer, R. S., and J. S. Moore. (1972) "The Sharing of Structure in Theorem Proving Programs." In *Machine Intelligence 7*, ed. B. Meltzer and D. Michie. Wiley.
- Brachman, Ronald J., and Hector J. Levesque. (1985) *Readings in Knowledge Representation*. Morgan Kaufmann.
- Brachman, Ronald J., Richard E. Fikes, and Hector J. Levesque. (1983) "KRYPTON: A Functional Approach to Knowledge Representation," FLAIR Technical Report no. 16, Fairchild Laboratory for Artificial Intelligence. Reprinted in Brachman and Levesque 1985.
- Bratko, Ivan. (1990) *Prolog Programming for Artificial Intelligence*. Addison-Wesley.
- Bromley, Hank, and Richard Lamson. (1987) *A Guide to Programming the Lisp Machine*. 2d ed. Kluwer Academic.
- Brooks, Rodney A. (1985) *Programming in Common Lisp*. Wiley.

- Brownston, L., R. Farrell, E. Kant, and N. Martin. (1985) *Programming Expert Systems in OPS5*. Addison-Wesley.
- Buchanan, Bruce G., and Edward Hance Shortliffe. (1984) *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley.
- Bundy, Alan. (1984) *Catalogue of Artificial Intelligence Tools*. Springer-Verlag.
- Cannon, Howard I. (1980) "Flavors." AI Lab Technical Report, MIT.
- Carbonell, Jamie A. (1981) *Subjective Understanding: Computer Models of Belief Systems*. UMI Research Press.
- Cardelli, Luca, and Peter Wegner. (1986) "On Understanding Types, Data Abstraction and Polymorphism." *ACM Computing Surveys* 17.
- Chapman, David. (1987) "Planning for Conjunctive Goals." *Artificial Intelligence* 32:333–377. Reprinted in Allen, Hendler, and Tate 1990.
- Charniak, Eugene, and Drew McDermott. (1985) *Introduction to Artificial Intelligence*. Addison-Wesley.
- Charniak, Eugene, Christopher Riesbeck, Drew McDermott, and James Meehan. (1987) *Artificial Intelligence Programming*. 2d ed. Lawrence Erlbaum.
- Cheeseman, Peter. (1985) "In Defense of Probability." In *Proceedings of the Ninth IJCAI* 1002–1009.
- Chomsky, Noam. (1972) *Language and Mind*. Harcourt Brace Jovanovich.
- Church, Alonzo. (1941) "The Calculi of Lambda-Conversion." *Annals of Mathematical Studies*. Vol. 6, Princeton University Press.
- Church, Kenneth, and Ramesh Patil. (1982) "Coping with Syntactic Ambiguity, or How to Put the Block in the Box on the Table." *American Journal of Computational Linguistics* 8, nos. 3-4:139–149.
- Clinger, William, and Jonathan Rees. (1991) *Revised⁴ Report on the Algorithmic Language Scheme*. Unpublished document available online on cs.voregin.edu.
- Clocksinn, William F., and Christopher S. Mellish. (1987) *Programming in Prolog*. 3d ed. Springer-Verlag.
- Clowes, Maxwell B. (1971) "On Seeing Things." *Artificial Intelligence* 2:79–116.
- Coelho, Helder, and Jose C. Cotta. (1988) *Prolog by Example*. Springer-Verlag.

- Cohen, Jacques. (1985) "Describing Prolog by its interpretation and compilation." *Communications of the ACM* 28, no. 12:1311–1324.
- Cohen, Jacques. (1990) "Constraint Logic Programming Languages." *Communications of the ACM* 33, no. 7:52–68.
- Colby, Kenneth. (1975) *Artificial Paranoia*. Pergamon.
- Collins, Allan. (1978) "Fragments of a Theory of Human Plausible Reasoning. *Theoretical Issues in Natural Language Processing*. David Waltz, ed. ACM. Reprinted in Shafer and Pearl 1990.
- Colmerauer, Alain. (1985) "Prolog in 10 figures." *Communications of the ACM* 28, no. 12:1296–1310.
- Colmerauer, Alain. (1990) "An Introduction to Prolog III." *Communications of the ACM* 33, no. 7:69–90.
- Colmerauer, Alain, Henri Kanoui, Robert Pasero, and Phillippe Roussel. (1973) *Un Système de Communication Homme-Machine en Français*. Rapport, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II.
- Cooper, Thomas A., and Nancy Wogrin. (1988) *Rule-Based Programming with OPS5*. Morgan Kaufmann.
- Dahl, Ole-Johan, and Kristen Nygaard. (1966) "SIMULA—An Algol-based Simulation Language." *Communications of the ACM* 9, no. 9:671–678.
- Davenport, J. H., Y. Siret, and E. Tournier. (1988) *Computer Algebra: Systems and Algorithms for Algebraic Computation*. Academic Press.
- Davis, Ernest. (1990) *Representations of Commonsense Reasoning*. Morgan Kaufmann.
- Davis, Lawrence. (1987) *Genetic Algorithms and Simulated Annealing*. Morgan Kaufmann.
- Davis, Lawrence. (1991) *Handbook of Genetic Algorithms*. van Nostrand Reinhold.
- Davis, Randall. (1977) "Meta-Level Knowledge." *Proceedings of the Fifth IJCAI*, 920–928. Reprinted in Buchanan and Shortliffe 1984.
- Davis, Randall. (1979) "Interactive Transfer of Expertise." *Artificial Intelligence* 12:121–157. Reprinted in Buchanan and Shortliffe 1984.
- Davis, Randall, and Douglas B. Lenat. (1982) *Knowledge-Based Systems in Artificial Intelligence*. McGraw-Hill.
- DeGroot, A. D. (1965) *Thought and Choice in Chess*. Mouton. (English translation, with additions, of the Dutch edition, 1946.)

- DeGroot, A. D., (1966) "Perception and Memory versus Thought: Some Old Ideas and Recent Findings." In *Problem Solving*, ed. B. Kleinmuntz. Wiley.
- de Kleer, Johan. (1986a) "An Assumption-Based Truth Maintenance System." *Artificial Intelligence* 28:127-162. Reprinted in Ginsberg 1987.
- de Kleer, Johan. (1986b) "Extending the ATMS." *Artificial Intelligence* 28:163-196.
- de Kleer, Johan. (1986c) "Problem-Solving with the ATMS." *Artificial Intelligence* 28:197-224.
- de Kleer, Johan. (1988) "A General Labelling Algorithm for Assumption-Based Truth Maintenance." *Proceedings of the AAAI*, 188-192.
- Dowty, David R., Robert E. Wall, and Stanley Peters. (1981) *Introduction to Montague Semantics*. Synthese Language Library, vol. 11. D. Reidel.
- Doyle, Jon. (1979) "A Truth Maintenance System." *Artificial Intelligence* 12:231-272.
- Doyle, Jon. (1983) "The Ins and Outs of Reason Maintenance." *Proceedings of the Eighth IJCAI* 349-351.
- Dubois, Didier, and Henri Prade. (1988) "An Introduction to Possibilistic and Fuzzy Logics." *Non-Standard Logics for Automated Reasoning*. Academic Press. Reprinted in Shafer and Pearl 1990.
- Earley, Jay. (1970) "An Efficient Context-Free Parsing Algorithm." *CACM* 6, no. 2:451-455. Reprinted in Grosz et al. 1986.
- Elcock, E. W., and P. Hoddinott. (1986) "Comments on Kornfeld's 'Equality for Prolog': E-Unification as a Mechanism for Augmenting the Prolog Search Strategy." *Proceedings of the AAAI*, 766-775.
- Emanuelson, P., and A. Haraldsson. (1980) "On Compiling Embedded Languages in Lisp." *Lisp Conference*, Stanford, Calif., 208-215.
- Ernst, G. W., and Newell, Alan. (1969) *GPS: A Case Study in Generality and Problem Solving*. Academic Press.
- Fateman, Richard J. (1973) "Reply to an Editorial." *ACM SIGSAM Bulletin* 25 (March):9-11.
- Fateman, Richard J. (1974) "Polynomial Multiplication, Powers and Asymptotic Analysis: Some Comments," *SIAM Journal of Computation* no. 3, 3:196-213.
- Fateman, Richard J. (1979) "MACSYMA's general simplifier: philosophy and operation." In *Proceedings of the 1979 MACSYMA Users' Conference (MUC-79)*, ed. V. E. Lewis 563-582. Lab for Computer Science, MIT.

- Fateman, Richard J. (1991) "FRPOLY: A Benchmark Revisited." *Lisp and Symbolic Computation* 4:155-164.
- Feigenbaum, Edward A. and Julian Feldman. (1963) *Computers and Thought*. McGraw-Hill.
- Field, A.J., and P. G. Harrison. (1988) *Functional Programming*. Addison-Wesley.
- Fikes, Richard E., and Nils J. Nilsson. (1971) "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence* 2:189-208. Reprinted in Allen, Hendler, and Tate 1990.
- Fodor, Jerry A. (1975) *The Language of Thought*. Harvard University Press.
- Forgy, Charles L. (1981) "OPS5 User's Manual." Report CMU-CS-81-135, Carnegie Mellon University.
- Forgy, Charles L. (1982) "RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem." *Artificial Intelligence* 19:17-37.
- Franz Inc. (1988) *Common Lisp: the Reference*. Addison-Wesley.
- Gabriel, Richard P. (1985) *Performance and evaluation of Lisp systems*. MIT Press.
- Gabriel, Richard P. (1990) "Lisp." In *Encyclopedia of Artificial Intelligence*, ed. Stuart C. Shapiro. Wiley.
- Galler, B. A., and M. J. Fisher. (1964) "An Improved Equivalence Algorithm." *Communications of the ACM* 7, no. 5:301-303.
- Gazdar, Richard, and Chris Mellish. (1989) *Natural Language Processing in Lisp*. Addison-Wesley. Also published simultaneously: *Natural Language Processing in Prolog*.
- Genesereth, Michael R., and Matthew L. Ginsberg. (1985) "Logic Programming." *Communications of the ACM* 28, no. 9:933-941.
- Genesereth, Michael R., and Nils J. Nilsson. (1987) *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann.
- Giannesini, Francis, H. Kanoui, R. Pasero, and M. van Caneghem. (1986) *Prolog*. Addison-Wesley.
- Ginsberg, Matthew L. (1987) *Readings in NonMonotonic Reasoning*. Morgan Kaufmann.
- Ginsberg, Matthew L., and William D. Harvey. (1990) "Iterative Broadening." *Proceedings, Eighth National Conference on AI*, 216-220.

- Goldberg, Adele, and David Robinson. (1983) *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.
- Goldberg, David E. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.
- Gordon, Jean, and Edward H. Shortliffe. (1984) "The Dempster-Shafer Theory of Evidence." In Buchanan and Shortliffe 1984.
- Green, Cordell. (1968) "Theorem-proving by resolution as a basis for question-answering systems." In *Machine Intelligence 4*, ed. Bernard Meltzer and Donald Michie. 183–205. Edinburgh University Press.
- Grosz, Barbara J., Karen Sparck-Jones, and Bonnie Lynn Webber. (1986) *Readings in Natural Language Processing*. Morgan Kaufmann.
- Guzman, Adolfo. (1968) "Computer Recognition of Three-Dimensional Objects in a Visual Scene." Ph.D. thesis, MAC-TR-59, Project MAC, MIT.
- Hafner, Carole, and Bruce Wilcox. (1974) *LISP/MTS Programmer's Guide*. Mental Health Research Institute Communication no. 302, University of Michigan.
- Harris, Zellig S. (1982) *A Grammar of English on Mathematical Principles*. Wiley.
- Hasemer, Tony, and John Domingue. (1989) *Common Lisp Programming for Artificial Intelligence*. Addison-Wesley.
- Hayes, Patrick. "Naive Physics I: Ontology for Liquids." In Hobbs and Moore 1985.
- Heckerman, David. (1986) "Probabilistic Interpretations for Mycin's Certainty Factors." In *Uncertainty in Artificial Intelligence*, ed. L. N. Kanal and J. F. Lemmer. Elsevier (North-Holland). Reprinted in Shafer and Pearl 1990.
- Hennessey, Wade L. (1989) *Common Lisp*. McGraw-Hill.
- Hewitt, Carl. (1977) "Viewing Control Structures as Patterns of Passing Messages." *Artificial Intelligence* 8, no. 3:323–384.
- Hobbs, Jerry R., and Robert C. Moore. (1985) *Formal Theories of the Commonsense World*. Ablex.
- Hofstadter, Douglas R. (1979) *Godel, Escher, Bach: An Eternal Golden Braid*. Vintage.
- Hölldobler, Steffen. (1987) *Foundations of Equational Logic Programming*, Springer-Verlag Lecture Notes in Artificial Intelligence.
- Huddleston, Rodney. (1984) *Introduction to the Grammar of English*. Cambridge University Press.

- Huffman, David A. (1971) "Impossible Objects as Nonsense Pictures." 295–323. In *Machine Intelligence 6*, ed. B. Meltzer and D. Michie. Edinburgh University Press.
- Hughes, R. J. M. (1985) "Lazy Memo Functions." In *Proceedings of the Conference on Functional Programming and Computer Architecture*, Nancy, 129–146. Springer-Verlag.
- Ingerman, Peter Z. (1961) "Thunks." *Communications of the ACM* 4, no. 1:55–58.
- Jaffar, Joxan, Jean-Louis Lassez, and Michael J. Maher. (1984) "A Theory of Complete Logic Programs with Equality." *Journal of Logic Programming* 3:211–223.
- Jackson, Peter. (1990) *Introduction to Expert Systems*. 2d ed. Addison-Wesley.
- James, Glenn, and Robert C. James. (1949) *Mathematics Dictionary*. Van Nostrand.
- Kanal, L. N., and J. F. Lemmer. (1986) *Uncertainty in Artificial Intelligence*. North-Holland.
- Kanal, L. N., and J. F. Lemmer. (1988) *Uncertainty in Artificial Intelligence 2*. North-Holland.
- Kay, Alan. (1969) "The Reactive Engine." Ph.D. thesis, University of Utah.
- Kay, Martin. (1980) *Algorithm schemata and data structures in syntactic processing*. Xerox Palo Alto Research Center Report CSL-80-12. Reprinted in Grosz et al. 1986.
- Kernighan, B. W., and P. J. Plauger. (1974) *The Elements of Programming Style*. McGraw-Hill.
- Kernighan, B. W., and P. J. Plauger. (1981) *Software Tools in Pascal*. Addison-Wesley.
- Keene, Sonya. (1989) *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley.
- Knight, K. (1989) "Unification: A Multidisciplinary Survey." *ACM Computing Surveys*, 21, no. 1:93–121.
- Knuth, Donald E., and Robert W. Moore. (1975) "An Analysis of Alpha-Beta Pruning." *Artificial Intelligence*, 6, no. 4:293–326.
- Kohlbecker, Eugene Edmund, Jr. (1986) "Syntactic Extensions in the Programming Language Lisp." Ph.D. thesis, Indiana University.
- Korf, R. E. (1985) "Depth-first Iterative Deepening: an Optimal Admissible Tree Search." *Artificial Intelligence* 27:97–109.

- Kornfeld, W. A. (1983) "Equality for Prolog." *Proceedings of the Seventh IJCAI*, 514–519.
- Koschman, Timothy. (1990) *The Common Lisp Companion*. Wiley.
- Kowalski, Robert. (1974) "Predicate logic as a programming language." In *Proceedings of the IFIP-74 Congress*, 569–574. North-Holland.
- Kowalski, Robert. (1979) "Algorithm = Logic + Control." *Communications of the ACM* 22:424–436.
- Kowalski, Robert. (1980) *Logic for Problem Solving*. North-Holland.
- Kowalski, Robert. (1988) "The Early Years of Logic Programming." *Communications of the ACM* 31:38–43.
- Kranz, David, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. (1986) "ORBIT: An optimizing compiler for Scheme." SIGPLAN Compiler Construction Conference.
- Kreutzer, Wolfgang, and Bruce McKenzie. (1990) *Programming for Artificial Intelligence: Methods, Tools and Applications*. Addison-Wesley.
- Lakoff, George. (1987) *Women, Fire and Dangerous Things: What Categories Reveal About the Mind*. University of Chicago Press.
- Landin, Peter. (1965) "A Correspondence Between Algol 60 and Church's Lambda Notation." *Communications of the ACM* 8, no. 2:89–101.
- Lang, Kevin J., and Barak A. Perlmutter. (1988) "Oaklisp: An Object-Oriented Dialect of Scheme." *Lisp and Symbolic Computing* 1:39–51.
- Langacker, Ronald W. (1967) *Language and its Structure*. Harcourt, Brace & World.
- Lassez, J.-L., M. J. Maher, and K. Marriott. (1988) "Unification Revisited." In *Foundations of Deductive Databases and Logic Programming*, ed. J. Minker, 587–625. Morgan Kaufmann.
- Lee, Kai-Fu, and Sanjoy Mahajan. (1986) "Bill: A Table-Based, Knowledge-Intensive Othello Program." Technical Report CMU-CS-86-141, Carnegie Mellon University.
- Lee, Kai-Fu, and Sanjoy Mahajan. (1990) "The Development of a World Class Othello Program." *Artificial Intelligence* 43:21–36.
- Levesque, Hector. (1986) "Making Believers out of Computers." *Artificial Intelligence* 30:81–108.

- Levy, David N. L. (1976) *Computer Chess*. Batsford.
- Levy, David N. L. (1988) *Computer Games*. Springer-Verlag.
- Levy, David N. L. (1988) *Computer Chess Compendium*. Springer-Verlag.
- Levy, David N. L. (1990) *Heuristic Programming in Artificial Intelligence: the First Computer Olympiad*. Ellis Horwood.
- Lloyd, J. W. (1987) *Foundations of Logic Programming*. Springer-Verlag.
- Loomis, Lynn. (1974) *Calculus*. Addison-Wesley.
- Loveland, D. W. (1987) "Near-Horn Prolog." *Proceedings of the Fourth International Conference on Logic Programming*, 456-469.
- Luger, George F., and William A. Stubblefield, (1989) *Artificial Intelligence and the Design of Expert Systems*. Benjamin/Cummings.
- Maier, David, and David S. Warren. (1988) *Computing with Logic*. Benjamin/Cummings
- Marsland, T. A. (1990) "Computer Chess Methods." Entry in *Encyclopedia of Artificial Intelligence*, ed. Stuart C. Shapiro. Wiley.
- Martin, William A., and Richard J. Fateman. (1971) "The MACSYMA System." *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, 59-75, ACM SIGSAM.
- Masinter, Larry, and Peter Deutsch, (1980) "Local Optimization in a Compiler for Stack-Based Lisp Machines." *Proceedings of the Lisp and Functional Programming Conference*.
- McAllester, David. (1982) "Reasoning Utility Package User's Manual." AI Memo 667, AI Lab, MIT.
- McCarthy, John. (1958) "An Algebraic Language for the Manipulation of Symbolic Expressions." AI Lab Memo no. 1, MIT.
- McCarthy, John. (1960) "Recursive functions of symbolic expressions and their computation by machine." *Communications of the ACM* 3, no 3:184-195.
- McCarthy, John. (1963) "A basis for a mathematical theory of computation." In *Computer Programming and Formal Systems*, ed. P. Braffort and D. Hirschberg. North-Holland.
- McCarthy, John. (1968) "Programs with Common Sense." In Minsky 1968. Reprinted in Brachman and Levesque 1985.

- McCarthy, John. (1978) "History of Lisp." In *History of Programming Languages*, ed. Richard W. Wexelblat. Academic Press. Also in *ACM SIGPLAN Notices* 13, no. 8.
- McCarthy, John, P. W. Abrahams, D. J. Edwards, P. A. Fox, T. P. Hart, and M. J. Levin. (1962) *Lisp 1.5 Programmer's Manual*. MIT Press.
- McDermott, Drew. (1978) "Tarskian Semantics, or No Notation without Denotation!" *Cognitive Science*, 2:277-282. Reprinted in Grosz, Sparck-Jones and Webber 1986.
- McDermott, Drew. (1987) "A Critique of Pure Reason." *Computational Intelligence* 3:151-160.
- Meyer, Bertrand. (1988) *Object-oriented Software Construction*. Prentice-Hall.
- Michie, Donald. (1968) "Memo Functions and Machine Learning." *Nature* 218:19-22.
- Miller, Molly M., and Eric Benson. (1990) *Lisp Style & Design*. Digital Press.
- Minsky, Marvin. (1968) *Semantic Information Processing*. MIT Press.
- Miranker, Daniel. (1990) *TREAT: A New and Efficient Match Algorithm for AI Production Systems*. Pitman.
- Moon, David. (1986) "Object-Oriented Programming with Flavors." *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications*.
- Moon, David and Richard Stallman and Daniel Weinreb. (1983) *The Lisp Machine Manual*. AI Lab, MIT.
- Moore, Robert C. (1982) "The Role of Logic in Knowledge Representation and Commonsense Reasoning." *Proceedings of the AAAI-82*. Reprinted in Brachman and Levesque 1985.
- Moses, Joel. (1967) "Symbolic Integration." Report no. MAC-TR-47, Project MAC, MIT.
- Moses, Joel. (1975) "AMACSYMA Primer." Mathlab Memo no. 2, Computer Science Lab, MIT.
- Mueller, Robert A., and Rex L. Page. (1988) *Symbolic Computing with Lisp and Prolog*. Wiley.
- Musser, David R., and Alexander A. Stepanov. (1989) *The ADA Generic Library*. Springer-Verlag.

- Naish, Lee. (1986) *Negation and Control in Prolog*. Springer-Verlag Lecture Notes in Computer Science 238.
- Newell, Alan, J. C. Shaw, and Herbert A. Simon. (1963) "Chess-Playing Programs and the Problem of Complexity." In Feigenbaum and Feldman 1963, 39–70.
- Newell, Alan, and Herbert A. Simon. (1963) "GPS, A Program that Simulates Human Thought." In Feigenbaum and Feldman 1963, 279–293. Reprinted in Allen, Hendler, and Tate 1990.
- Newell, Alan, and Herbert A. Simon, (1972) *Human Problem Solving*. Prentice-Hall.
- Nilsson, Nils. (1971) *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill.
- Norvig, Peter. (1991) "Correcting a Widespread Error in Unification Algorithms." *Software Practice and Experience* 21, no. 2:231–233.
- Nygaard, Kristen, and Ole-Johan Dahl. (1981) "SIMULA 67." In *History of Programming Languages*, ed. Richard W. Wexelblat.
- O'Keefe, Richard. (1990) *The Craft of Prolog*. MIT Press.
- Pearl, Judea. (1984) *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.
- Pearl, Judea. (1988) *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.
- Pearl, Judea. (1989) "Bayesian and Belief-Functions Formalisms for Evidential Reasoning: A Conceptual Analysis." *Proceedings, Fifth Israeli Symposium on Artificial Intelligence*. Reprinted in Shafer and Pearl 1990.
- Pereira, Fernando C. N., and Stuart M. Shieber. (1987) *Prolog and Natural-Language Analysis*. Center for the Study of Language and Information, Lecture Notes no. 10.
- Pereira, Fernando C. N., and David H. D. Warren. (1980) "Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks." *Artificial Intelligence* 13:231–278. Reprinted in Grosz et al. 1986.
- Perlis, Alan. (1982) "Epigrams on Programming." *ACM SIGPLAN Notices* 17, no. 9.
- Plaisted, David A. (1988) "Non-Horn Clause Logic Programming Without Contrapositives." *Journal of Automated Reasoning* 4:287–325.
- Quillian, M. Ross. (1967) "Word Concepts: A Theory of Simulation of Some Basic Semantic Capabilities." *Behavioral Science* 12:410–430. Reprinted in Brachman and Levesque 1985.

- Quirk, Randolph, Sidney Greenbaum, Geoffrey Leech, and Jan Svartik. (1985) *A Comprehensive Grammar of the English Language*. Longman.
- Ramsey, Allan, and Rosalind Barrett. (1987) *AI in Practice: Examples in Pop-11*. Halstead Press.
- Rich, Elaine, and Kevin Knight. (1991) *Artificial Intelligence*. McGraw-Hill.
- Risch, R. H. (1969) "The Problem of Integration in Finite Terms." *Translations of the A.M.S.* 139:167-189.
- Risch, R. H. (1979) "Algebraic Properties of the Elementary Functions of Analysis." *American Journal of Mathematics* 101:743-759.
- Robinson, J. A. (1965) "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM* 12, no. 1:23-41.
- Rosenbloom, Paul S. (1982) "A World-Championship-Level Othello Program." *Artificial Intelligence* 19:279-320.
- Roussel, Phillipe. (1975) *Prolog: manual de reference et d'utilisation*. Groupe d'Intelligence Artificielle, Université d'Aix-Marseille.
- Rowe, Neal. (1988) *Artificial Intelligence Through Prolog*. Prentice-Hall.
- Ruf, Erik, and Daniel Weise. (1990) "LogScheme: Integrating Logic Programming into Scheme." *Lisp and Symbolic Computation* 3, no. 3:245-288.
- Russell, Stuart. (1985) "The Compleat Guide to MRS." Computer Science Dept. Report no. STAN-CS-85-1080, Stanford University.
- Russell, Stuart, and Eric Wefald. (1989) "On Optimal Game-Tree Search using Rational Meta-Reasoning." *Proceedings of the International Joint Conference on Artificial Intelligence*, 334-340.
- Sacerdoti, Earl. (1974) "Planning in a Hierarchy of Abstraction Spaces." *Artificial Intelligence* 5:115-135. Reprinted in Allen, Hendler, and Tate 1990.
- Sager, Naomi. (1981) *Natural Language Information Processing*. Addison-Wesley.
- Samuel, A. L. (1959) "Some Studies in Machine Learning Using the Game of Checkers." *IBM Journal of Research and Development* 3:210-229. Reprinted in Feigenbaum and Feldman 1963.
- Sangal, Rajeev. (1991) *Programming Paradigms in Lisp*. McGraw Hill.
- Schank, Roger C., and Kenneth Mark Colby. (1973) *Computer Models of Thought and Language*. Freeman.

- Schank, Roger C., and Christopher Riesbeck. (1981) *Inside Computer Understanding*. Lawrence Erlbaum.
- Schmolze, J. G., and T. A. Lipkis. (1983) "Classification in the KL-ONE Knowledge Representation System." *Proceedings of the Eighth IJCAI*. 330–332.
- Sedgewick, Robert. (1988) *Algorithms*. Addison-Wesley.
- Shannon, Claude E. (1950a) "Programming a Digital Computer for Playing Chess." *Philosophy Magazine* 41:356–375.
- Shannon, Claude E. (1950b) "Automatic Chess Player." *Scientific American*, Feb., 182.
- Shebs, Stan T., and Robert R. Kessler. (1987) "Automatic Design and Implementation of Language Data Types." *SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques (ACM SIGPLAN Notices 22, no. 7:26–37*.
- Shapiro, Stuart C. (ed.). (1990) *Encyclopedia of Artificial Intelligence*. Wiley.
- Shafer, Glenn, and Judea Pearl. (1990) *Readings in Uncertain Reasoning*. Morgan Kaufmann.
- Sheil, B. A. (1983) "Power Tools for Programmers." *Datamation*, Feb., 131–144.
- Shortliffe, Edward H. (1976) *Computer-Based Medical Consultation: MYCIN*. American Elsevier.
- Shortliffe, Edward H., and Bruce G. Buchanan (1975) "A Model of Inexact reasoning in Medicine." *Mathematical Biosciences*, 23:351–379. Reprinted in Shafer and Pearl 1990.
- Slade, Richard. (1987) *The T Programming Language: A Dialect of Lisp*. Prentice Hall.
- Slagle, J. R. (1963) "A heuristic program that solves symbolic integration problems in freshman calculus." In *Computers and Thought*, ed. Feigenbaum and Feldman, 191–203. Also in *Journal of the ACM* 10:507–520.
- Spiegelhalter, David J. (1986) "A Statistical View of Uncertainty in Expert Systems." In *Artificial Intelligence and Statistics*, ed. W. Gale. Addison-Wesley. Reprinted in Shafer and Pearl 1990.
- Staples, John, and Peter J. Robinson. (1988) "Efficient Unification of Quantified Terms." *Journal of Logic Programming* 5:133–149.
- Steele, Guy L., Jr. (1976a) "LAMBDA: The Ultimate Imperative." AI Lab Memo 353, MIT.

- Steele, Guy L., Jr. (1976b) "LAMBDA: The Ultimate Declarative." AI Lab Memo 379, MIT.
- Steele, Guy L., Jr. (1977) "Debunking the 'Expensive Procedure Call' Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO." AI Lab Memo 443, MIT.
- Steele, Guy L., Jr., (1978) "Rabbit: a Compiler for Scheme (A Study in Compiler Optimization)." AI Lab Technical Report 474, MIT.
- Steele, Guy L. Jr., (1983) "Compiler optimization based on viewing lambda as Rename Plus Goto." In *AI: An MIT Perspective*, vol. 2. MIT Press.
- Steele, Guy L. Jr., (1984) *Common Lisp the Language*. Digital Press.
- Steele, Guy L. Jr., (1990) *Common Lisp the Language*, 2d edition. Digital Press.
- Steele, Guy L., Jr., and Gerald J. Sussman. (1978) "The revised report on Scheme, a dialect of Lisp." AI Lab Memo 452, MIT.
- Steele, Guy L., Jr., and Gerald J. Sussman. (1978) "The art of the interpreter, or the modularity complex (parts zero, one, and two)." AI Lab Memo 453, MIT.
- Steele, Guy L., Jr., and Gerald Jay Sussman. (1979) "Design of LISP-Based Processors or, SCHEME: A Dielectric LISP or, Finite Memories Considered Harmful or, LAMBDA: The Ultimate Opcode." AI Lab Memo 379, MIT.
- Steele, Guy L., Jr., and Gerald J. Sussman. (1980) "Design of a Lisp-Based Processor." *Communications of the ACM* 23, no. 11:628-645.
- Stefik, Mark, and Daniel G. Bobrow. (1986) "Object-Oriented Programming: Themes and Variations." *AI Magazine* 6, no. 4.
- Sterling, Leon, and Ehud Shapiro. (1986) *The Art of Prolog*. MIT Press.
- Sterling, L., A. Bundy, L. Byrd, R. O'Keefe and B. Silver. (1982) "Solving Symbolic Equations with PRESS." In *Computer Algebra, Lecture Notes in Computer Science No. 144*, ed. J. Calmet, 109-116. Springer-Verlag. Also in *Journal of Symbolic Computation* 7 (1989):71-84.
- Stickel, Mark. (1988) "A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler." *Journal of Automated Reasoning* 4:353-380.
- Stoyan, Herbert. (1984) "Early Lisp History." *Proceedings of the Lisp and Functional Programming Conference*, 299-310.
- Stroustrup, Bjarne. (1986) *The C++ Programming Language*. Addison-Wesley.

- Sussman, Gerald J. (1973) *A Computer Model of Skill Acquisition*. Elsevier.
- Tanimoto, Steven. (1990) *The Elements of Artificial Intelligence using Common Lisp*. Computer Science Press.
- Tate, Austin. (1977) "Generating Project Networks." IJCAI-77, Boston. Reprinted in Allen, Hendler, and Tate 1990.
- Tater, Deborah G. (1987) *A Programmer's Guide to Common Lisp*. Digital Press.
- Thomason, Richmond. (1974) *Formal Philosophy—Selected Papers of Richard Montague*. Yale University Press.
- Touretzky, David. (1989) *Common Lisp: A Gentle Introduction to Symbolic Computation*. Benjamin/Cummings.
- Tversky, Amos, and Daniel Kahneman. (1974) "Judgement Under Uncertainty: Heuristics and Biases." *Science* 185:1124–1131. Reprinted in Shafer and Pearl 1990.
- Tversky, Amos, and Daniel Kahneman. (1983) "Extensional Versus Intuitive Reasoning: The Conjunction Fallacy in Probability Judgement." *Psychological Review* 90:29–315.
- Tversky, Amos, and Daniel Kahneman. (1986) "Rational Choices and the Framing of Decisions." *Journal of Business* 59:S251-S278. Reprinted in Shafer and Pearl 1990.
- Ungar, David. (1984) "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm." In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pa., April), 157–167. *ACM SIGPLAN Notices* 19, no. 5.
- van Emden, Maarten H., and Keitaro Yukawa. (1987) "Logic Programming with Equations." *Journal of Logic Programming* 4:265–288.
- van Melle, W. J. (1980) *System Aids in Constructing Consultation Programs*. UMI Research Press.
- Van Roy, Peter L., (1990) "Can Logic Programming Execute as Fast as Imperative Programming?" Report UCB/CSD 90/600, University of California, Berkeley.
- Vygotsky, Lev Semenovich. (1962) *Thought and Language*. MIT Press.
- Waibel, Alex, and Kai-Fu Lee (1991) *Readings in Speech Understanding*. Morgan Kaufmann.

- Waldinger, Richard. (1977) "Achieving Several Goals Simultaneously." In *Machine Intelligence 8*. Ellis Horwood Limited.
- Walker, Adrian, Michael McCord, John F. Sowa, and Walter G. Wilson. (1990) *Knowledge Systems and Prolog*. Addison-Wesley.
- Waltz, David I. (1975) "Understanding Line Drawings of Scenes with Shadows." In *The Psychology of Computer Vision*, ed. Patrick H. Winston. McGraw-Hill.
- Waltz, David I. (1990) "Waltz Filtering." In *Encyclopedia of Artificial Intelligence*, ed. Stuart C. Shapiro. Wiley.
- Wand, Mitchell. (1980) "Continuation-Based Program Transformation Strategies." *Journal of the ACM* 27, no. 1:174-180.
- Warren, David H. D. (1974a) "WARPLAN: A System for Generating Plans." Department of Computational Logic Memo 76, AI, Edinburgh University.
- Warren, David H. D. (1974b) "Extract from APIC Studies in Data Processing, No. 24." Reprinted in Allen, Hendler, and Tate, 1990.
- Warren, David H. D. (1979) "Prolog on the DECsystem-10." In *Expert Systems in the Micro-Electronic Age*, ed. Donald Michie. Edinburgh University Press.
- Warren, David H. D. (1983) *An abstract Prolog instruction set*. Technical Note 309, SRI International.
- Warren, David H. D., L. M. Pereira, and Fernando C. N. Pereira. (1977) "Prolog—the Language and its Implementation Compared with Lisp." *Proceedings of the ACM SIGART-SIGPLAN Symposium on AI and Programming Languages*.
- Warren, David H. D., and Fernando C. N. Pereira. (1982) "An Efficient Easily Adaptable System for Interpreting Natural Language Queries." *American Journal of Computational Linguistics*, 8, nos.3-4:110-122.
- Waterman, David A. (1986) *A Guide to Expert Systems*. Addison-Wesley.
- Waters, Richard C. (1991) "Supporting the Regression Testing of Lisp Programs." *Lisp Pointers* 4, no. 2:47-53.
- Wegner, Peter. (1987) "Dimensions of object-based language design." *ACM SIGPLAN Notices*, 168-182.
- Weinreb, Daniel, and David A. Moon (1980) "Flavors: Message Passing in the Lisp Machine." AI Memo no. 602, Project MAC, MIT.
- Weiss, Sholom M., and Casimar A. Kulikowski. (1984) *A Practical Guide to Designing Expert Systems*. Rowman & Allanheld.
- Weissman, Clark. (1967) *Lisp 1.5 Primer*. Dickenson.

- Weizenbaum, Joseph. (1966) "ELIZA—A computer program for the study of natural language communication between men and machines." *Communications of the ACM* 9:36-45.
- Weizenbaum, Joseph. (1976) *Computer Power and Human Reason*. Freeman.
- Whorf, Benjamin Lee. (1956) *Language, Thought, and Reality*. MIT Press.
- Wilensky, Robert. (1986) *Common LISPcraft*. Norton.
- Winograd, Terry. (1983) *Language as a Cognitive Process*. Addison-Wesley.
- Winston, Patrick H. (1975) *The Psychology of Computer Vision*. McGraw-Hill.
- Winston, Patrick H. (1984) *Artificial Intelligence*. Addison-Wesley.
- Winston, Patrick H., and Bertold K. P. Horn. (1988) *Lisp*, 3d ed. Addison-Wesley.
- Wirth, N. (1976) *Algorithms + Data Structures = Programs*. Prentice Hall.
- Wong, Douglas. (1981) "Language Comprehension in a Problem Solver." *Proceedings of the International Joint Conference on Artificial Intelligence*, 7-12.
- Woods, William A. (1970) "Transition Network Grammars for Natural Language Analysis." *Communications of the ACM* 13:591-606. Reprinted in Grosz et al. 1986.
- Woods, William A. (1975) "What's in a Link: Foundations for Semantic Networks." In *Representation and Understanding*, ed. D. G. Bobrow and A. M. Collins. Academic Press.
- Woods, William A. (1977) "Lunar Rocks on Natural English: Explorations in Natural Language Question Answering." In *Linguistic Structures Processing*, ed. A. Zamponi. Elsevier-North-Holland.
- Zabih, Ramin, David McAllester, and David Chapman. (1987) "Non-Deterministic Lisp with Dependency-Directed Backtracking." *Proceedings of the AAI*.
- Zadeh, Lotfi. (1978) "Fuzzy Sets as a Basis for a Theory of Possibility." *Fuzzy Sets Systems*, 1:3-28.
- Zucker, S. W. (1990) "Vision, Early." In *Encyclopedia of Artificial Intelligence*, ed. Stuart C. Shapiro. Wiley.

Index

!, 420
&allow-other-keys, 101
&aux, 102
&body, 102
&key, 98
&optional, 98
&rest, 101
(), 10
abbrevs, 740
acc, 848
bigger-grammar, 43
bindings, 300
board, 623, 624
cities, 197
clock, 623, 624
db-predicates, 360, 361
dbg-ids, 124
debug-io, 124
depth-incr, 484
depth-max, 484
depth-start, 484
edge-table, 639
examples, 708
grammar, 657
grammar1, 657
grammar3, 657
grammar4, 661
infix->prefix-rules, 249
label-num, 786, 788
maze-ops, 134
move-number, 623, 624
occurs-check, 356, 361
open-categories, 664
ops, 114, 127
package, 835
ply-boards, 623, 634
predicate, 421
primitive-fns, 786, 823
primitives, 489
print-gensym, 855
print-level, 379
profiled-functions, 290
readtable, 821
rules-for, 297
scheme-procs, 757, 759
scheme-readtable, 821
school-ops, 117
search-cut-off, 483
simple-grammar, 39
simplification-rules, 243, 247, 249
standard-output, 124, 888
state, 114
static-edge-table, 643
student-rules, 221
systems, 892
trail, 379, 391
uncompiled, 408
uniq-atom-table, 334

- *uniq-cons-table*, 334
- *var-counter*, 379
- *vars*, 340
- *weights*, 609
- *world*, 498, 500
- ., 68
- .,@, 68
- >, 690
- if, 61
- if-not, 61
- :-, 690
- :LABEL, 819
- :after, 447
- :before, 447
- :end keywords, 895
- :ex, 708, 744
- :pass, 394
- :print-function, 379, 499
- :sem, 705
- :test, 128
- :test-not, 100
- <-, 351, 360, 361, 373, 399
- =, 374, 395, 406, 745
- =/2, 414
- ==>, 705, 707
- >-num, 894
- ?, 379
- ?*, 183
- ?+, 183
- ?-, 361, 363, 364, 373, 391
- ??, 183, 494
- ?and, 183
- ?if, 183
- ?is, 183
- ?not, 183
- ?or, 183
- #', 14, 92
- #+, 292
- #-, 292
- #., 340, 645
- #d, 822
- #f, 754, 822
- #t, 754, 822
- &rest, 754
- \+, 415
- ^, 243
- ~&, 84
- ~{...~}, 85, 230
- ~^, 85
- ~a, 84, 230
- ~f, 84
- ~r, 84
- ~s, 84
- 10*N+D, 670
- 68000 assembler, 319
- 88->h8, 622, 623

- A, 660
- a, 494
- a*-search, 209
- A+, 660
- a-lists, 74
- Ait-Kaci, Hassan, 385, 426, 504
- abbrev, 740
- abbreviations, 732, 739
- Abelson, Harold, 213, 307, 367, 383, 511, 777, 825
- abstract machine, 810
- abstraction, 423
- ABSTRIPS, 147
- account, 436, 445
- account-deposit, 437
- account-interest, 437
- account-withdraw, 437
- accumulator, 329, 686, 698
- accumulators, 63
- accusative case, 717
- achieve, 114, 140
- achieve-all, 120, 128, 139
- achieve-each, 139
- acons, 50
- action-p, 136
- Actors, 457
- Ada, 27, 459, 837

- add-body, 843
- add-clause, 360, 361, 408
- add-examples, 709
- add-fact, 486, 490
- add-noun-form, 742
- add-test, 843
- add-var, 843
- add-verb, 742
- adder, 92
- Adj, 38
- adj, 716, 731
- Adj*, 38
- adjectives, 738
- adjunct, 716, 719, 720, 723
- adjuncts, 718
- adverb, 716, 732
- adverbial phrase, 723
- adverbs, 723, 738
- advp, 716
- Aho, A. V., 307
- air-distance, 201
 - M, 60, 678
- all-directions, 601, 602
- all-parses, 675
- all-squares, 602, 603, 631
- Allen, James, 748
- Allen, John, 148, 777, 825
- alpha cutoff, 615
- alpha-beta, 602, 616
- alpha-beta-searcher, 602, 616
- alpha-beta-searcher2, 623, 631
- alpha-beta-searcher3, 623, 636
- alpha-beta2, 623, 631
- alpha-beta3, 623, 635
- always, 832, 847
- Amarel, Saul, 132
- ambiguity, 669
- ambiguous-vertex-p, 569, 570
- and, 53, 415, 429, 485, 764
- and*/2, 708
- Anderson, John, 655
- anon-vars-in, 433
- anonymous-variables-in, 391,
 - 400, 433
- antecedent rules, 561
- any-legal-move?, 602, 606
- append, 11, 69, 848
- append-pipes, 285
- append1, 659
- applicable-ops, 213
- apply, 18, 91
- apply-op, 115, 129
- apply-scorer, 672
- apply-semantic, 668
- appropriate-ops, 141
- appropriate-p, 114, 129
- apropos, 86, 878
- arch, 587, 589, 590, 593
- aref, 73
- arg-count, 795, 799
- arg1, 812
- arg2, 674, 812
- arg3, 812
- argi, 795
- ARGS, 785, 805, 814, 815
- args, 390, 391, 795, 812
- args->prefix, 513, 520
- argument
 - keyword, 322, 877
 - optional, 322, 412, 877
 - rest, 322, 805
- Aristotle, 111, 147
- arrow (\Rightarrow), 5
- art, 716, 731
- Article, 36
- articles, 738
- as, 845
- ask-vals, 533, 539
- asm-first-pass, 795, 812
- asm-second-pass, 795, 813
- assemble, 795, 812
- assembler, 805
- assert, 88
- assert-equal, 295

- assoc, 73
- asymptotic complexity, 274
- atom/1, 745
- attributive adjectives, 749
- audited-account, 447
- augmented transition network (ATN), 712
- AutoLisp, ix
- aux, 716, 731
- aux-inv-S, 716, 727
- auxiliary verb, 735
- average, 87

- backquote, 68
- backquote, 822
- backtrack points, 420
- backtrack-points, 772
- backtracking, 349, 367, 372, 772
 - automatic, 349
 - chronological, 773
- Backus-Naur Form (BNF), 678
- backward-chaining, 351, 536, 543
- Bacon, Francis, 460
- bag, 416
- bagof/3, 416
- bananas, 132
- bank-account, 92
- Barrett, Rosalind, xv, 594, 748
- Batali, John, 810
- Bayes's law, 557
- Bayesian classification, 652
- be, 735
- beam-problem, 452
- beam-search, 196
- begin, 754
- belief functions, 557
- benchmark, 295, 411, 522
- Berkeley, California, 633
- Berlin, Andrew, 267
- best-first-search, 194
- best-problem, 452
- beta cutoff, 615

- better-path, 210
- bfs-problem, 450
- Bill, 597, 636, 651
- binary-exp-p, 229, 242
- binary-tree, 192
- binary-tree-eql-best-beam-problem, 452
- binary-tree-problem, 451
- bind-new-variables, 391, 405
- bind-unbound-vars, 391, 398
- bind-variables-in, 391, 404
- binding-val, 157, 391
- binomial theorem, 524
- bit, 73
- bit sequence, 79
- bit vector, 79
- black, 601, 602
- block, 65
- block, 754, 767
- blocks world, 136, 211
- blood test, 558
- BOA constructor, 221
- board, 601, 602
- Bobrow, Daniel, 219, 234, 458
- body, 351
- bound-p, 377
- boundary line, 566
- Boyer, R. S., 425
- Brachman, Ronald J., 503
- bracketing, 675
- Bratko, Ivan, 383
- breadth-first-search, 192
- break, 87
- bref, 601, 602
- Brooks, Rodney A., 259
- Brown, Allen, 142
- Buchanan, Bruce G., 557, 558
- build-cases, 276
- build-code, 276
- build-exp, 302
- butlast, 877
- Butler, Nicholas Murray, 530

- byte-code assembly, 811
- C, ix, 837
- C++, 459
- cache, 536
- calculus, 252
- CALL, 785, 811, 820
- call-loop-fn, 844
- call-next-method, 445
- call-with-current-continuation, 372, 770
- call/1, 414, 745
- call/cc, 425, 754, 757, 776, 780, 810
- CALLJ, 814, 820
- Cannon, Howard, 457
- canon, 513, 521
- canon->prefix, 513, 520
- canon-simplifier, 513, 521
- canonical simplification, 510
- car, 14, 69
- Carbonell, Jamie, 167
- cardinal, 716, 732
- Carlyle, Thomas, 175
- case, 53, 764, 879
- case sensitive, 8
- Cassio, 597
- Catalan Numbers, 663
- catch, 623, 769, 837
- catch point, 410
- categories, 485
 - closed-class, 664
 - open-class, 664
- category names, 660
- CC, 810, 815
- ccase, 88
- cdr, 14, 69
- cdr-coding, 522
- Cerf, Jonathan, 637
- cerrror, 87
- certainty factors, 532
- cf->english, 533, 551
- cf-and, 533, 535
- cf-cut-off, 533, 536
- cf-or, 533, 535
- cf-p, 533, 536
- change a field, 873
- Chapman, David, 148
- char, 73
- char-ready?, 756
- char?, 756
- Charniak, Eugene, xiii, xv, 345, 383, 504, 586, 594, 823, 887
- chart parsers, 679
- Chat-80, 711
- check-conditions, 533, 549
- check-diagram, 569, 588
- check-reply, 533, 540
- check-type, 88
- checkers, 651
- Cheeseman, Peter, 558
- chess, 652
- choice of names, 888
- Chomsky, Noam, 655
- choose-first, 773
- Church, Alonzo, 20
- Church, Kenneth, 663
- city, 197, 198
- class, 436
 - variable, 436
- clause, 350, 723
- clause, 361, 716, 724
- clause-body, 360
- clause-head, 360
- clauses-with-arity, 390, 391
- clear-abbrevs, 740
- clear-db, 361, 362, 533, 537
- clear-dtrees, 476
- clear-examples, 708
- clear-grammar, 744
- clear-lexicon, 744
- clear-m-array, 318
- clear-memoize, 275
- clear-predicate, 361, 362
- clear-rules, 533, 545

- cliche, 60, 176
- Clinger, William, 777
- Clocksin, William F., 382
- CLOS, xii, 30, 435, 439, 445–446, 448, 453–454, 456, 458–459
 - flaw in, 448
- closed world assumption, 466
- closure, 92, 457
- Clowes, Maxwell B., 594
- clrhash, 74
- CMU Lisp, 327
- coef, 512–514
- coefficients of a polynomial, 510
- Coelho, Helder, 147, 383
- Cohen, Jacques, 426, 504
- Colby, Kenneth, 153, 167, 655
- collect, 848, 863
- collect-sems, 707
- collecting, 849
- Collins, Allan, 167
- Colmerauer, Alain, 382, 504, 684, 711
- combine-all, 45
- combine-all-pipes, 286
- combine-edge-moves, 642
- combine-quasiquote, 824
- combine-rules, 304
- command, 725
- comment, 6
- common cases, 717
- Common Lisp, vii–xiv, 4, 7–9, 12, 20, 24, 25, 27, 29, 30, 48–51, 55, 57, 62, 66, 68, 72, 74, 76, 78, 79, 81, 82, 84, 85, 88, 91, 93, 94, 97, 98, 101–103, 106, 110, 112, 113, 115, 120, 122, 155, 156, 161, 165, 178, 182, 203, 245, 246, 266–268, 278, 279, 281, 292, 317, 318, 321, 322, 330, 346, 372, 411, 419, 435, 438, 439, 445, 504, 514, 522, 574, 623, 632, 652, 666, 678, 753–755, 759, 760, 762, 766, 767, 769, 771, 774, 780, 783, 811, 822, 823, 825, 826, 828, 830, 834–840, 843, 852, 853, 855, 857, 872, 876, 877, 879, 882–885, 891, 894, 895, 900
- CommonLoops, 458
- commutative-p, 229
- comp, 786, 798
- comp-begin, 786, 787, 800
- comp-const, 795, 800
- comp-funcall, 795, 803
- comp-go, 795
- comp-if, 786, 787, 801, 831
- comp-lambda, 786, 788
- comp-list, 795, 800
- comp-show, 786, 789
- comp-var, 795, 800
- compact disc player, 665
- compilation, 526
- compile-all-rules-indexed, 306
- compile-arg, 391, 395, 399, 404
- compile-args, 302
- compile-body, 391, 394, 405, 422
- compile-call, 391, 394
- compile-clause, 391, 394, 397, 406
- compile-exp, 301
- compile-file, 645
- compile-if, 391, 403
- compile-indexed-rule, 304
- compile-predicate, 391, 392, 397, 422
- compile-rule, 276, 300
- compile-rule-set, 304
- compile-unify, 391, 395, 402
- compile-unify-variable, 391, 403
- compiled
 - for effect, 792
 - for value, 792
- compiler, 275, 298
 - context-free, 798
 - context-sensitive, 798
 - versus interpreter, 277
- compiler, 786, 788

- complement, 101, 716, 728
- complements, 718
- complete-parses, 658
- compose, 177, 217
- computation on lists, 6
- computer vision, 565
- concat, 411, 686
- concave line, 566
- conclude, 533, 547
- cond, 53, 764, 782, 878
- conj-category, 711
- conj-rule, 710
- conjuncts, 708
- cons, 11, 69, 328
- cons cells, 69
- consistency, 464
- consistency checker, 90
- consistent-labelings, 569, 572
- consp, 69
- CONST, 785, 812, 814
- constants, 889
- constraint propagation, 568
- construct-diagram, 569, 576
- construct-vertex, 569, 576
- context, 533, 542
- contexts, 538, 541
- continuation, 300, 367
- continue-p, 369
- convert-number, 829
- convert-numbers, 829
- convert-op, 126
- convex line, 566
- Cooper, Thomas A., 266
- copula, 735, 744
- copy-board, 602, 603
- copy-poly, 516
- copy-tree, 76
- corner-for, 642
- corner-p, 642
- cost-fn, 452, 453
- Cotta, Jose, 147, 383
- count, 62, 848
- count-difference, 602, 603
- count-edge-neighbors, 643
- count-if, 60
- counting, 849
- create-list-of-equations, 224
- cross-product, 622, 623
- cube, 575
- cube-on-plate, 581
- current-state, 449
- cut, 420

- D, 660
- dag, 345
- Dahl, Ole-Johan, 456
- data-driven dispatch, 394
- Davenport, J. H., 259, 260, 528
- Davis, Ernest, 503
- Davis, Lawrence, 652
- Davis, Randall, 549
- dbg, 124
- dbg-indent, 124
- dcg-normal-goal-p, 691
- dcg-word-list-p, 691
- de Moivre, 310
- debug, 124
- debugging, 85
- decf, 56
- decidability, 464
- declaration, 875
- declarative languages, 435
- declared inline, 869
- def-attached-fn, 490
- def-cons-struct, 347
- def-optimizer, 819
- def-prolog-compiler-macro, 391, 395
- def-scheme-macro, 757, 763
- default rules, 561
- defclass, 445
- defconstant, 51, 157
- defcontext, 533, 542
- defdiagram, 569, 575, 588
- define, 754, 762, 764, 790

- define-class, 440
- define-enumerated-type, 654
- define-setf-method, 514, 884
- define-system, 891, 893
- defining enumerated, 652
- definite clause grammar (DCG),
690, 711
- definite integrals, 519
- defloop, 844, 849
- defmacro, 51, 66
- defmethod, 445, 446
- defpackage, 836, 891
- defparameter, 15, 39, 51
- defparm, 533, 541
- defresource, 337
- defrule, 277, 533, 549, 886
- defsetf, 514
- defstruct, 51
- defun, 12, 51
- defun*, 327
- defun-memo, 273
- defvar, 39, 51
- deg->radians, 201
- degree, 512, 513
- degree of a polynomial, 510
- DeGroot, A. D., 652
- delay, 281, 762, 765
- delay decisions, 25
- delegation, 436, 442
- delete, 62
- delete-trie, 344
- Dempster, 557
- dense polynomials, 511
- depth-first-search, 191
- dequeue, 342
- deref, 378
- deref-copy, 417, 430
- deref-equal, 414
- deref-exp, 410
- deriv, 257
- deriv-divides, 257
- deriv-poly, 513, 518
- derivative-divides technique, 252
- describe, 86, 878
- destructive operations, 328
- Det, 687–689, 693, 695, 697, 701,
716, 721
- determine-winner, 57
- Deutsch, Peter, 826
- developing an AI computer program,
110
- dfs-problem, 450
- diagram, 569, 576
- diff, 194
- difference between relations and
functions, 350
- difference list, 702
- differentiable field, 528
- differentiation, 248
- directed acyclic graph (dag), 345, 649
- disassemble, 318
- disassembled code, 318
- discrimination net, 345
- discrimination tree, 472
- disjunction, 485
- displace, 781
- display, 804
- distance, 201
- distinguish unknown from false, 496
- ditransitive verbs, 712
- div, 839
- divide-factors, 255
- dividing by zero, 233
- do, 59, 852
- documentation, 87, 878
- documentation string, 12
- dolist, 58
- dotimes, 59
- dotted pair notation, 69
- double-float, 320
- Dowty, David R., 711
- Doyle, Jon, 504
- Drosophila melanogaster*, 596
- dtree, 476

- dtree-atom-fetch, 479
- dtree-fetch, 479
- dtree-index, 477, 498
- Dubois, Didier, 558
- dynamic extent, 771
- dynamic typing, 27

- each, 494
- Earley, Jay, 679
- earth-diameter, 201
- eat-porridge, 89
- ecase, 88
- echo question, 725
- edge-and-x-lists, 639
- edge-index, 639
- edge-move-probability, 642
- edge-stability, 639
- Edinburgh Prolog, 425, 690
- efficiency, 461
- efficient-pat-match, 332, 333
- Eiffel, 455, 459, 863
- Elcock, E. W., 504
- ELIZA, xi, 151-154, 159, 163-166, 168, 169, 175, 178, 181, 184, 187, 219-222, 234, 239, 240, 309, 330, 435
- eliza, 164, 177
- elt, 73
- Elvis, 536
- Emanuelson, P., 307
- empty, 601, 602
- empty-pipe, 282
- empty-queue-p, 342
- EMYCIN, xii, 532-534, 536-538, 541, 543, 544, 548-550, 559-563
- emycin, 533, 548
- encapsulate information, 448
- end game, 649
- end of file, 822
- enqueue, 342
- ensure-generic-fn, 440
- ensure-generic-function, 445

- enumerate, 284
- enumerated type, 599
- environment, 758
 - global, 758
- eof, 821
- eof-object?, 821
- eq, 72
- eq?, 756
- eq1, 72
- eq1-problem, 450
- equal, 69, 72
- equal?, 756
- equalp, 72
- equiv, 832
- eqv?, 756
- Eratosthenes, 285
- Ernst, G. W., 147
- error, 838
 - handler, 838
 - signaling, 838
- error, 87
- error in transcribing, 588
- errors
 - handling, 837
- eval, 91, 245
- eval-condition, 533, 546
- evaluable, 245
- evaluation, 24
 - lazy, 307
 - rule for Lisp, 22
- even?, 756
- every, 62
- examples of rules, 705
- executing-p, 126
- exercises
 - level of difficulty, xv
- existentials, 467
- exit
 - nonlocal, 768
- exp, 242
- exp-args, 242
- exp-p, 242

- expand-pat-match-abbrev, 187
- expert system, 461, 530
- expert-system shells, 531
- explanation, 531
- exponent->prefix, 513, 520
- exponentiation, 523
- expression, 5
 - derived, 762
 - lambda, 21
 - rational, 526
 - reading and evaluating, 24
 - special form, 9, 22
- expressiveness, 461, 464
- extend-bindings, 158, 159, 361
- extend-env, 757, 759
- extend-parse, 659, 668, 671, 681, 682
- extensibility, 29
- extent, 93

- fact-present-p, 490
- factorize, 254
- facts, 350
- fail, 157, 361, 430, 772
- failure continuations, 425
- false, 532, 533
- false-p, 533, 536
- fast-time->seconds, 292
- fast-time-difference, 292
- Fateman, Richard, 259, 265, 267, 511, 522, 524
- Fateman, Richard J., 528
- Feigenbaum, Edward, 460
- fetch, 478
- fib, 269
- Fibonacci, 269
- Field, A. J., 307
- Fikes, Richard, 147, 503
- fill pointer, 330
- fill-loop-template, 842
- filler-gap dependency, 702
- filter, 285
- final-value, 602

- finally, 852
- find, 62
- find-all, 101
- find-all-if, 100
- find-anywhere, 255, 391
- find-bracketing-piece, 602, 605
- find-if, 61
- find-labelings, 569, 586
- find-out, 533, 545
- find-path, 210
- find-trie, 344
- find-vertex, 569, 573
- finish-output, 895
- finite verb, 722
- finite-binary-tree, 193
- first, 10, 69
- first-class, 27
- first-match-pos, 185
- first-name, 13, 16
- first-or-nil, 658
- Fischer, Sylvia, 554
- Fisher, M. J., 504
- FJUMP, 785, 814, 820
- flatten, 165, 329, 347
- flavors, 438, 457
- flet, 870
- flexible flow of control, 531
- float, 320
- floating-point numbers, 875
- FN, 815
- fn, 786, 788, 790
- Fodor, Jerry A., 655
- follow-arc, 344
- follow-binding, 391, 404
- for, 845
- for-each, 756
- force, 281
- format, 84, 230, 739, 839
 - option, 625
- format directives, 84
- FORTTRAN, 84, 266, 267, 434, 655
- forward chaining, 351, 485

- forward pruning, 647
- four semicolons, 891
- fourth, 10
- Fowler, Henry Watson, 715
- frames, 493
- Francis, 355
- FRANZ LISP, ix
- free-of, 255
- fresh-line, 84
- Frisbee, 769
- front, 342
- frpoly, 522, 528
- funcall, 91, 693, 828
- funcall-if, 861
- function, 79
 - application, 23
 - data-driven, 818
 - destructive, 80, 888
 - first-class, 27
 - generic, 322, 436, 439
 - higher-order, 18, 194, 839
 - list processing, 10
 - new, 92, 887
 - properly tail-recursive, 794
 - proving correctness, 227
 - recursive, 523
 - sequence, 852
 - short, 887
 - tail-recursive, 63
- function, 92, 872
- functional programming, 435
- functionp, 283
- fuzzy set theory, 461, 558

- Gabriel, Richard, 522
- Galler, B. A., 504
- game playing, 596
- garbage collector, 328
 - ephemeral, 330, 336
 - generational, 330
- Gazdar, Richard, 679, 748
- gen, 786, 789
 - gen-args, 795
 - gen-label, 786, 789
 - gen-set, 786, 790, 804
 - gen-var, 786, 790
 - gen1, 795, 819
 - generate, 40, 41
 - generate-all, 45, 286
 - generate-tree, 44
 - generation scavenging, 336
 - generic function, 436, 439
 - generic operations, 811
 - generic-fn-p, 440
 - genetic learning, 651
 - gensym, 363
 - get, 894
 - get-abbrev, 740
 - get-binding, 157, 361
 - get-cf, 533, 537
 - get-clauses, 360, 361
 - get-context-data, 533, 548
 - get-db, 533, 537
 - get-dtree, 476
 - get-examples, 708
 - get-fast-time, 292
 - get-global-var, 757, 759
 - get-method, 438
 - get-move, 602, 607, 625
 - get-optimizer, 819
 - get-parm, 533, 541
 - get-rules, 533, 545
 - get-trie, 344
 - get-vals, 533, 537
 - get-var, 757, 759
 - get-world, 500
 - gethash, 74
 - Ginsberg, Matthew L., 214
 - go, 754, 837
 - goal-p, 450
 - Goldberg, Adele, 457
 - Goldberg, David E., 652
 - Gordon, , 558
 - goto, 766, 768

- GPS, xi, 109–121, 123, 125, 127, 129, 130, 132, 133, 135, 136, 142, 143, 145–147, 149, 175, 189, 190, 211, 213, 215, 239, 470
- GPS, 114, 127, 130, 135
- gps, 367
- gps-successors, 212
- grammar
 - context-free, 678
 - context-free phrase-structure, 35, 686
 - definite clause (DCG), 690, 711
 - rule, 685
 - unification, 678
- grandfather, 385
- graph-search, 206
- Green, Cordell, 382
- Greenbaum, Sidney, 748
- ground, 569, 579
- grundy, 312
- GSET, 785, 814, 820
- Guzman, Adolfo, 594
- GVAR, 785, 812, 814

- h8->88, 622, 623
- Hölldobler, Steffen, 504
- Hafner, Carole, 30
- HALT, 816
- halting problem, 511
- handle-conj, 711 handler-case, 178, 839
- Haraldsson, A., 307
- Harrell, Steve, 457
- Harris, Zellig S., 749
- Harrison, P. G., 307
- Harvey, William D., 214
- has-variable-p, 391, 396
- hash table, 74, 296, 477
- Hayes, Patrick, 469
- head, 351
- head, 282
- Heckerman, David, 558
- help-string, 538
- Hendler, James, 148
- Hennessey, Wade L., xiv, 259, 383
- Hewitt, Carl, 382, 457
- higher-order predications, 485
- Hoare, C. A. R., 66
- Hockney, David, 509
- Hoddinott, P., 504
- Horn clauses, 684
- Horn, Bertold, xiv, 213, 367, 383, 777
- Huddleston, Rodney, 749
- Huffman, David A., 594
- Hughes, R. J. M., 307
- human, 602, 607, 622
- hungry monkey, 132
- hyphen before the p, 755

- Iago, 597, 652
- Iago, 623, 646
- Iago-eval, 623, 645
- IBM 704, 14
- identity, 669
- idiom, 176
- if, 16, 424, 745, 754, 851
- ignore, 391
- ignore declaration, 410
- ignore-errors, 838
- imperative programming, 434
- import, 836
- impossible diagram, 582
- impossible-diagram-p, 570
- impossible-vertex-p, 570
- in-env-p, 786, 791
- in-exp, 228
- in-integral-table?, 258
- in-package, 835, 891
- inc-profile-time, 294
- incf, 56
- ind, 485, 490
- indefinite extent, 771
- index, 477, 481, 498
- index-new-fact, 492

- index-rules, 298
- indexing, 297, 335, 526
- individuals, 485
- infectious blood disease, 552
- infinite set, 280
- infix notation, 240
- infix->prefix, 240, 241
- infix-funcall, 667
- infix-scorer, 674
- inflection, 722
- information hiding, 436, 454, 835
- Ingalls, Daniel, 457
- Ingerman, Peter Z., 307
- inheritance, 436, 499
 - data-driven, 443
 - for classes, 444
 - generic, 443
 - multiple, 436, 457
- init-edge-table, 640
- init-scheme-comp, 795, 805, 816
- init-scheme-interp, 757, 760
- init-scheme-proc, 757, 776
- initial-board, 602, 603
- initially, 852
- inline, 293
- insert-path, 210
- inspect, 87
- inst-name, 533, 540
- instance, 436
- instance variable, 436
- instrument, 265
- instrumentation, 268
- integer, 772
- integer?, 756
- integers, 282, 667, 674
- integrals, 252
- integrate, 256
- integrate-from-table, 258
- integrating polynomials, 519
- integration by parts, 260
- integration-table, 257
- interactive environment, 28
- interactive-interpreter, 177,
178, 216
- INTERLISP, ix
- intern, 835
- internal definition, 779
- interning, 835
- interp, 757, 758, 762, 767, 774
- interp-begin, 757, 775
- interp-call, 757, 775
- interpretation
 - declarative, 351
 - procedural, 351
- interpreter, 275
 - tail-recursive, 766
 - versus compiler, 277
- intersperse, 520
- intractable, 461
- intransitive verbs, 693
- inv-span, 674
- inverse-op, 228
- IPL, 110
- irev, 412
- iright, 374
- is, 192, 203, 533, 547, 795, 813
- is/2, 418
- isolate, 227
- iter-wide-search, 204
- iterative deepening, 205, 482, 646
- iterative widening, 204

- Jackson, Peter, 558
- Jaffar, Joxan, 504
- James, Glenn, 239
- James, Robert, 239
- JUMP, 785, 814, 820

- k*poly, 513, 517
- k+poly, 513, 516
- Kahneman, Daniel, 558
- Kay, Alan, 457
- Kay, Martin, 679
- KCL, 428
- Keene, Sonya, 458

- Kernighan, B. W., viii
- keyword, 98
- killer heuristic, 634
- Kinski, Natassja, 700
- KL-ONE, 462, 503
- Kleene star, 37
- Kleene, Stephen Cole, 38
- Klier, Peter, 511
- Knight, Kevin, 383, 594
- knowledge
 - technical compendium, vii
- knowledge engineer, 548
- knowledge representation, 461
- knowledge-based system, 530
- Knuth, Donald E., 652
- Korf, R. E., 214
- Kornfeld, W. A., 504
- Kowalski, Robert, 382, 465
- Kranz, David, 825
- Kreutzer, Wolfgang, xv, 213
- KRYPTON, 503
- Kulikowski, Casimir A., 558

- label-p, 786, 791
- labels, 762, 870
- labels-for, 569, 573
- lambda, 20
- lambda, 754, 783
- lambda expression, 21
- Lang, Kevin J., 458
- Langacker, Ronand, 655
- language
 - declarative, 435
 - frame, 462
 - hybrid representation, 462
 - network-based, 462
 - object-oriented, 462
 - procedural, 462
- Lassez, Jean-Louis, 383, 504
- last, 12, 69, 884
- last-name, 12
- last1, 305, 757, 760
- last2, 883
- ldiff, 877
- leaping before you look, 121
- learning, 651
- Lee, Kai-Fu, 636, 637, 651
- Leech, Geoffrey, 748
- left-recursive rules, 681, 705
- legal-moves, 602, 607
- legal-nodes, 623, 632
- legal-p, 602, 604
- len, 455
- length, 69, 370
- length1, 58
- length1.1, 58
- length10, 63
- length11, 63
- length12, 64
- length2, 58
- length3, 59
- length4, 60
- length5, 60
- length6, 60
- length7, 60
- length8, 61
- length9, 62
- length=1, 255, 276, 496, 757, 760
- let, 41, 764, 782
- let*, 56, 764
- letrec, 762, 765
- Levesque, Hector J., 503, 504
- Levy, David, 652
- lexical closure, 92
- lexical-rules, 658, 664
- lexicon, 732
- likes/2, 389
- limited-account, 442, 444, 446
- Lincoln, Abraham, 75
- line-diagram labeling problem, 565
- linear equations, 234
- Lipkis, T. A., 503
- LIPS, 376
- Lisp

- evaluation rule for, 22
- lexical rules for, 5
- Lisp 1.5, 777
- lisp/2, 418
- list
 - association, 73, 74, 343, 476
 - difference, 702
 - processing function, 10
 - property, 74, 476
- list, 11, 69
- list*, 67, 69
- list->string, 756
- list->vector, 756
- list-ref, 756
- list-tail, 756
- list1, 804
- list2, 804
- list3, 804
- listp, 69
- Lloyd, J. W., 383, 415
- load, 645
- local maximum, 197
- logic programming, 435
- logic puzzle, 373
- long-distance dependencies, 702
- lookup, 157, 361, 896
- loop, 842, 864, 878
- LOOP FOR, 845
- loop keywords, 844
 - data-driven, 844
- loop macro, 840
- LOOP REPEAT, 845
- loop-finish, 847
- loop-for-arithmetic, 846
- loop-unless, 851
- losing-value, 602, 613
- loss, 311, 312
- Loveland, D. W., 504
- LSET, 785, 814, 820
- Luger, George F., 558
- LVAR, 785, 811, 814
- machine, 795, 814
- MacLachlan, Rob, 327
- MACLISP, ix
- macro, 66, 853, 760
 - conditional read, 292
 - defining, 763
 - design, 880
- macro-expansion, 778
- MACSYMA, xi, xii, 151, 239, 259, 260, 297, 522, 528
- Mahajan, Sanjoy, 636, 651
- Maher, Michael J., 504
- Maier, David, 383
- main variable of a polynomial, 510
- main-op, 297
- main-var, 512-514
- maintenance, 177
- make=, 391, 394
- make-anonymous, 391, 399
- make-augmented-dcg, 707
- make-block-ops, 137
- make-clause, 440
- make-copy-diagram, 569, 577
- make-dcg, 691
- make-dcg-body, 692
- make-empty-nlist, 476
- make-flips, 602, 605
- make-instance, 445
- make-maze-op, 134
- make-maze-ops, 134
- make-move, 602, 604
- make-moves, 312
- make-obsolete, 870
- make-parameters, 391, 392
- make-pipe, 282, 283
- make-poly, 513, 514
- make-predicate, 391, 392
- make-queue, 342
- make-rat, 526
- make-system, 893
- make-true-list, 795
- make-variable, 225, 340

- map, 756, 771
- map-edge-n-pieces, 640
- map-interp, 757, 775
- map-into, 632, 857
- map-path, 204
- map-pipe, 285
- mapc, 62
- mapc-retrieve, 480, 488
- mapc-retrieve-in-world, 501
- mapcar, 14, 62, 864
- maphash, 74
- mappend, 19, 165, 171
- mappend-pipe, 286
- Marsland, T. A., 652
- Martin, William, 259, 522, 528
- Masinter, Larry, 826
- mass nouns, 749
- match-and, 184
- match-if, 186
- match-is, 184
- match-not, 184
- match-or, 184
- match-var, 332, 333
- match-variable, 158
- matching-ifs, 305
- math-quiz, 97, 98
- matrix-transpose, 569, 574
- max, 420
- maximize, 849 -
- maximize-difference, 602, 608
- maximizer, 602, 608
- maximizing, 849
- maybe-add, 496, 757, 760
- maybe-add-undo-bindings, 391, 398
- maybe-set-it, 851
- maybe-temp, 847
- McAllester, Davic, 504
- McCarthy, John, 20, 248, 259, 503, 652, 776, 777
- McCord, Michael, 711
- McDermott, Drew, xv, 147, 383, 503, 586, 594
- McKenzie, Bruce, xv, 213
- meaning, 676
- meanings, 669
- Meehan, James, xv
- Mellish, Chris, 382, 679, 748
- member, 16, 62, 327, 358, 374, 745
- member-equal, 129
- memo, 270, 274
- memoization, 270, 296, 526, 662
- memoize, 271, 275, 662
- message, 436
- metamorphosis grammar, 711
- metareasoning, 650
- metavariable, 697
- method, 436, 438
- method combinations, 458
- Meyer, Bertrand, 455, 459
- Michie, Donald, 307, 652
- microcode for addition, 811
- minimax, 612
- minimax, 602, 613
- minimax-searcher, 602, 614
- minimize, 849
- minimizing, 849
- Minsky, Marvin, 234
- mix-ins, 457
- mklist, 165
- mobility, 623, 629, 637
- modal auxiliary verbs, 735
- modified-weighted-squares, 602, 621
- modifiers, 718
- modifiers, 716, 719
- Modula, 27, 459
- monitoring function, 599
- monotonicity, 464
- Montague, Richard, 711
- Moon, David A., 457
- Moore, J. S., 425
- Moore, Robert, 466, 652
- Moses, Joel, 239, 259
- most-negative-fixnum, 613
- most-positive-fixnum, 195

- most-positive-fixnum, 613
- move-ons, 137
- move-op, 137
- moves, 311
- MRS, 504
- MU-Prolog, 383
- multimethod, 436, 458
- multiple goals, 145
- multiple values, 96, 685
- multiple-value-bind, 96, 875
- multiple-value-call, 887
- Musser, David R., 27
- must-be-number, 771
- MYCIN, xii, 461, 531, 532, 535, 541, 542,
552, 553, 557–559, 903
- mycin, 533, 552

- N, 660
- N, 693
- Naish, Lee, 383
- nalist, 498
- nalist-push, 499
- Name, 660
- Name, 694, 701
- name, 716, 731
- name clashes, 279
- name!, 786
- name-of, 601, 602
- named, 852
- names, 737
- nconc, 80, 848
- nconcing, 849
- negate-node, 623
- negate-value, 632
- negated predicates, 496
- negation, 485
- negative?, 756
- neighbors, 198, 602, 621
- neural nets, 651
- never, 847
- New Flavors, 458
- new-account, 437
- new-fn, 795
- new-instance, 533, 543
- new-parm, 541
- new-states, 207
- new-symbol, 302, 391
- new-tree, 658, 666, 671
- Newell, Alan, 109, 147, 596
- newer-file-p, 894
- newline, 804
- next-instr, 795, 819
- next-to-play, 602, 606
- nextto, 374
- NIL, 821
- nil, 10
- Nilsson, Nils, 147, 214, 503
- nim, 311
- nintersection, 80
- n1/0, 413
- nlist, 475
- nlist-list, 476
- nlist-n, 476
- nlist-push, 476
- no-bindings, 157
- no-states-p, 449
- no-unknown, 228
- node, 623, 631
- noise-word-p, 225
- nominative case, 717
- non-Horn clauses, 504
- NONLIN, 147
- nonlocal exit, 768
- nonrestrictive clauses, 750
- normalize, 518
- normalize-poly, 513, 518
- Norvig, Peter, 384
- not, 415, 424
- not-numberp, 246
- not/1, 415
- notation
 - $O(f(n))$, 274
 - dotted pair, 69
 - infix, 240

- package prefix, 835
- prefix, 228, 240
- Noun, 36, 695, 698, 701
- noun, 716, 731, 742
- noun-phrase, 36, 38
- NP, 660
- NP, 687, 688, 692, 694, 698, 701, 703, 716, 717
- NP-hard, 146, 461
- NP2, 716, 718
- nreverse, 80
- nset-difference, 80
- nsubst, 80
- nth, 69, 73
- NU-Prolog, 383
- null, 69
- number-and-negation, 20
- number-of-labelings, 569, 570
- numberp/1, 745
- numbers-and-negations, 20
- nunion, 80
- Nygaard, Krysten, 456

- O'Keefe, Richard, 383, 423
- object, 3, 436
- object-oriented
 - programming, 434
- objective case, 717
- occurs check, 356, 471
- occurs-check, 356, 361
- omniscience, 464
- once-only, 854
- one-of, 36, 275
- one-unknown, 229
- op, 114, 126, 127
- op?, 302
- opcode, 795, 812
- open, 83
- opening book, 649
- operator precedence, 240
- operators-and-inverses, 228
- opponent, 601, 602

- OPS5, 266
- opt-rel-pronoun, 721
- opt-word, 729
- optimize, 795, 818
- optimize-1, 818
- optimizing arithmetic operations, 793
- or, 53, 415, 429, 764
- ORBIT, 825
- orderings, 139
- ordinal, 716, 732
- Othello, 597
- othello
 - bracketing piece, 605
 - cheat, 606
 - corner squares, 608
 - current mobility, 637
 - edge squares, 608
 - edge stability, 637
 - end game, 649
 - legal move, 604
 - mobility, 637
 - plausible move generator, 647
 - potential mobility, 637
 - stable, 643
 - unstable, 643
 - valid move, 604
- othello, 605, 624
- semistable, 643
- othello-series, 623, 626, 628
- outer, 601, 602

- P, 660
- p-add-into!, 525
- p-lists, 74
- package, 754, 834, 889–890
- package prefix notation, 835
- pair?, 756
- parameter
 - keyword, 98
 - optional, 98
 - order, 889
 - rest, 778

- parameter list, 12
- parm, 533, 541
- parm-type, 533, 541
- PARRY, 153, 154, 167
- parse, 658, 659, 668, 671, 680, 681
- parse-condition, 547
- parse-lhs, 658
- parse-loop-body, 844
- parse-namestring, 877
- parse-reply, 533, 540
- parser, 658, 662, 680, 681
- partial evaluation, 267
- partition-if, 256
- Pascal, ix, 26–29, 51, 55, 57, 66, 98, 176, 266, 434, 623
- passivize-sense, 743
- passivize-subcat, 743
- password-account, 441
- past participles, 720
- past tense, 722
- pat-match, 155, 156, 158, 160, 181
- pat-match-1, 332
- pat-match-abbrev, 187
- path, 200
- path-states, 210
- Patil, Ramesh, 663
- pattern matcher, 509
- pattern matching and unification, 352
- Pearl, Judea, 558, 559, 648, 652
- peephole optimizer, 805, 818
- Pereira, Fernando, 383, 426, 711, 748
- Pereira, Luis, 426
- Perlis, Alan, 3, 265, 348, 866
- Perlmutter, Barak A., 458
- permutations, 150
- permute, 675, 680, 682
- permute-vector!, 682
- Peters, Stanley, 711
- piece, 601, 602
- piece-stability, 644
- pipe, 281
- pipe-elt, 282
- pipes, 840
- place, 55
- Plaisted, David A., 504
- PLANNER, 382
- Plauger, J., viii
- play-game, 313
- play-games, 312
- poiuyt, 582
- poly, 513, 514
- poly*poly, 513, 517
- poly*same, 513, 517
- poly+, 513, 515
- poly+poly, 513, 516
- poly+same, 513, 516
- poly-, 513, 515
- poly/poly, 529
- poly², 523
- polyⁿ, 513, 518, 523, 524
- polyhedra, 565
- polynomial, 512, 513
- polynomials, 510
- polysemous, 730
- POP, 785, 814
- pop, 56
- pop-end, 881, 882
- pop-state, 449
- position, 62
- position-if, 60
- possible worlds, 485, 496, 497
- possible-edge-move, 641
- possible-edge-moves-value, 641
- possible-labelings, 569
- postdeterminers, 721
- PP, 660
- PP, 38, 716, 720
- PP*, 38
- pprint, 839
- Prade, Henri, 558
- preconditions, 112
- precycling, 634
- predeterminers, 721
- predicate, 888

- calculus, 463
- equality, 70
- recognizer, 81
- predicate, 360, 361
- predicative adjectives, 749
- prefer-disjoint, 674
- prefer-not-singleton, 674
- prefer-subset, 674
- prefer<, 674
- preferences, 670
- prefix notation, 4, 228, 240
- prefix->canon, 513, 515
- prefix->infix, 229, 242, 519, 520
- Prep, 38
- prep, 716, 732
- prepend, 192
- prepositional phrases, 720
- prepositions, 739
- prerequisite clobbers siblinggoal, 120, 139
- present participles, 720
- present tense, 722
- pretty printing, 839
- price-is-right, 195
- PRIM, 815
- prim, 795, 804
- primitive operation, 803
- primitive-p, 795, 804
- prin1, 83
- princ, 83
- print-board, 602, 603, 625
- print-condition, 533, 551
- print-conditions, 533, 551
- print-equations, 228, 236
- print-fn, 786, 790
- print-labelings, 569, 571
- print-path, 203
- print-proc, 757, 768
- print-rule, 533, 545, 551
- print-sqrt-abs, 771
- print-table, 771
- print-variable, 340
- print-vertex, 569, 573
- print-why, 533, 552
- print-world, 501
- priority queue, 459
- Pro, 660
- probability theory, 557
- problem
 - (find item list) failed, 874
 - change to function ignored, 869
 - closures don't work, 871
 - deletion didn't take effect, 873
 - leaping before you look, 121
 - line-diagram labling, 565
 - multiple values lost, 874
 - no response, 867
 - prerequisite clobbers siblinggoal, 120, 139
 - recursive subgoal, 123
- problem, 449
- problem-combiner, 450, 452
- problem-combiner :around, 452
- problem-successors, 451, 453
- proc, 757
- procedural attachment, 463
- procedure?, 756
- profile, 290
- profile-count, 289
- profile-enter, 293
- profile-exit, 293
- profile-report, 289, 294
- profile-time, 294
- profile1, 289, 291
- profiled-fn, 289, 293
- profiling, 288
- prog, 767
- progn, 64
- programming
 - data-driven, 182
 - functional style, 435, 839
 - idioms, 60
 - imperative style, 434
 - in the large, 890

- logic, 435
 - object-oriented style, 434
 - procedural style, 434
 - rule-based, 435
- Project MAC, 239
- Prolog, ix, xii, xv, 63, 144, 155, 287, 348–351, 355, 356, 358–360, 364, 366–368, 371–374, 376, 378, 380–382, 384–386, 388, 389, 391, 407, 408, 411–413, 415–421, 423–428, 431, 435, 455, 462, 464–472, 480–482, 489, 497, 504, 505, 531, 532, 536, 538, 541–544, 684, 685, 690, 691, 693, 697, 708, 711–713, 732, 745
 - Prolog II, 355
 - Prolog III, 383
 - prolog-compile, 390, 391
 - prolog-compile-symbols, 391, 409
 - prolog-compiler-macro, 391, 395
 - Prolog-In-Lisp, 360, 424
 - prompt, 4
 - prompt-and-read, 867
 - prompt-and-read-vals, 533, 539
 - prompt-generator, 178
 - pronoun, 716, 731
 - pronouns, 736
 - propagate-constraints, 569, 571, 590
 - proper-listp, 391, 396
 - property lists, 74
 - prototypes, 469
 - prove, 361, 362, 367, 368, 380, 483
 - prove-all, 361, 362, 367, 380, 483
 - punctuation-p, 709
 - push, 56
 - put-db, 533, 537
 - put-diagram, 576
 - put-first, 623, 636
 - put-optimizer, 819
 - put-rule, 533, 545
 - put-trie, 344
 - Pygmalion, 152
 - quasi-q, 795, 824
 - quasiquote, 822
 - Quayle, Dan, 735
 - query-bind, 482
 - query-user, 677
 - questions, 725, 726
 - queue, 341
 - queue-contents, 342
 - queue-nconc, 343
 - Quillian, M. Ross, 503
 - Quirk, Randolph, 748
 - quote, 427, 754
 - quote mark ('), 6
 - r15-test, 522
 - Rabbit, 825
 - Ramsey, Allan, xv, 594, 748
 - random-choice, 773
 - random-elt, 36, 166, 276, 322, 602
 - random-mem, 322
 - random-ordering strategy, 630
 - random-othello-series, 623, 627
 - random-strategy, 602, 607
 - rapid-prototyping, 265
 - rat*rat, 513, 529
 - rat+rat, 513, 529
 - rat-denominator, 513, 527
 - rat-numerator, 513, 527
 - rat/rat, 513, 529
 - rational number, 526
 - read, 83
 - read-char, 83, 895
 - read-eval-print loop, 176, 821
 - read-from-string, 876
 - read-line, 83
 - read-time-case, 313
 - read/1, 413
 - reading, 24
 - readtable, 712, 821
 - reasoning with uncertainty, 531
 - recursion, 62

- recursive, 17
- recursive subgoal, 123
- REDUCE, 259
- reduce, 62, 860
- reduce*, 860
- reduce-list, 862
- reduce-vect, 860
- referential transparency, 423, 856
- regression testing, 90
- reject-premise, 533, 547
- rel, 485, 491
- rel-clause, 698, 701, 703, 716, 720
- rel-pro, 716
- relation-arity, 390, 391
- relations, 485
- relative clauses, 720
- remhash, 74
- remove, 61, 62
- remove-if, 61
- remove-if-not, 61, 100
- remove-punctuation, 709
- remq, 334
- rename-variables, 361, 363
- repeat, 423, 674, 845
- repeat/0, 423
- repeat/fail loop, 423
- replace, 624, 634
- replace-?-vars, 373, 496, 870, 871
- report-findings, 533, 550
- representation
 - boxed, 317
 - knowledge, 461
 - printed, 52
 - unboxed, 317
- reset, 773
- resource, 336
- rest, 10, 69
- restrictive clauses, 750
- ret-addr, 795, 813
- retrieve, 480, 488
- retrieve-bagof, 489
- retrieve-bagof-in-world, 501
- retrieve-conjunction, 487
- retrieve-fact, 487
- retrieve-in-world, 501
- retrieve-matches, 480
- retrieve-setof, 489
- RETURN, 785, 798, 814, 816, 820
- return, 65, 754, 852
- return-from, 837
- return-if, 847
- reuse-cons, 333, 361
- rev, 411
- rev-funcall, 674
- rev-scorer, 674
- reverse, 69, 411
- reverse-label, 569, 573
- Reversi, 597
- Rich, Elaine, 594
- Riesbeck, Christopher, xv
- RISC, 811
- Risch, R., 239, 528, 260
- Robinson, J. A., 382
- Robinson, Peter J., 504
- robotics, 564
- Rose, Brian, 637
- Rosenbloom, Paul, 637, 645, 652
- round-robin, 623, 628
- Roussel, Jacqueline, 382
- Ruf, Erik, 777
- rule, 242, 533, 545, 658, 666, 671, 690
- rule-based programming, 435
- rule-based translation, 509
- rule-based translator, 224
- rule-based-translator, 189
- rule-lhs, 275
- rule-pattern, 163
- rule-responses, 163
- rule-rhs, 275
- rules, 350
 - examples, 705
 - left-recursive, 705
- rules-for, 298, 682
- rules-starting-with, 658

- run-attached-fn, 490
- run-examples, 709
- run-prolog, 391, 409
- Russell, Bertrand, 20
- Russell, Steve, 777
- Russell, Stuart, 504, 650

- S, 660
- S, 686–688, 692, 699, 701, 703, 716, 725, 726
- Sacerdoti, Earl, 147
- Sager, Naomi, 749
- SAINT, 239, 259
- same-shape-tree, 76
- Samuel, A. L., 651
- Sangal, Rajeev, xiv
- satisficing, 146
- satisfy-premises, 533, 546
- SAVE, 814
- sbit, 73
- Schank, Roger C., 655
- Scheme, ix, xii, 63, 91, 280, 372, 425, 753, 755–757, 759–760, 763–764, 766–768, 770–774, 776–780, 790, 795, 799, 810–811, 816, 821–830, 833, 836, 856, 879, 882, 888
 - spelling conventions, 755
 - T dialect, 825
- scheme, 757, 760, 774, 795
- scheme-macro, 757, 763
- scheme-macro-expand, 757, 763
- scheme-read, 822
- scheme-top-level, 816
- Schmolze, J. G., 503
- scope, 93
- search, 140, 572
 - A*, 208, 459
 - ahead, 610
 - aspiration, 648
 - beam, 195
 - best-first, 194
 - breadth-first, 192, 544
 - brute-force, 620
 - degrades gracefully, 647
 - depth-first, 191, 544
 - heuristic, 204
 - hill-climbing, 197, 651
 - ply, 610
 - tools, 448
 - zero-window, 648
- search-all, 211
- search-gps, 212
- search-n, 218
- search-solutions, 569, 572
- searcher, 449
- searcher :before, 450
- second, 10, 69
- segment-match, 161, 162, 185
- segment-match+, 186
- segment-match-fn, 183
- segment-match?, 186
- segment-matcher, 183
- segment-pattern-p, 183
- self-and-double, 19
- sem, 674
- semantics, 656
- semipredicates, 127
- semi-stable, 643
- send, 438
- sentence, 36
- seq, 786, 789
- series facility, 840
- set, 346
- set, 95
- set!, 754, 756
- set-binding!, 378, 379
- set-car!, 756
- SET-CC, 810, 815
- set-diff, 670
- set-difference, 895
- set-global-var!, 757, 759
- set-macro-character, 714
- set-simp-fn, 252

- set-var!, 757, 759
- set-world-current, 500
- setf, 8, 55, 514
 - methods, 514
- setof/3, 417
- seven name spaces, 836
- shadow, 825, 836
- Shafer, Glenn, 557, 559
- Shakespeare, William, 597
- Shannon, Claude E., 652
- Shapiro, Ehud, 382
- Shapiro, Stuart, 213
- Shaw, George Bernard, 315
- Shaw, J. C., 596
- Shieber, Stuart, 383, 711, 748
- Shortliffe, Edward H., 531, 553,
557, 558
- show-city-path, 203
- show-diagram, 569, 574
- show-fn, 786, 791, 813
- show-prolog-solutions, 361, 365
- show-prolog-vars, 361, 365, 369, 484
- show-prolog-vars/2, 410
- show-vertex, 569, 573
- side effect, 802, 886
- side-effect-free-p, 855
- sieve, 285
- Simon, Herbert, 109, 146, 147, 596
- simp, 244
- simp-fn, 252
- simp-rule, 246
- simple-array, 321, 876
- simple-equal, 155
- simple-vector, 321, 876
- simplifier, 244
- simplify, 244
- simplify-by-fn, 252
- simplify-exp, 244, 252, 297, 306
- Simula, 456
- SIN, 239, 259
- single-float, 320
- single-matcher, 183
- single-pattern-p, 183
- Skolem constant, 467, 485, 493
- Skolem function, 467
- Skolem, Thoralf, 467
- Slagle, James, 239, 259
- slot-constituent, 728
- slot-number, 743
- Smalltalk, 457
- Software Tools in Pascal*, viii
- solve, 226
- solve-arithmetic, 229
- some, 62
- sort, 69
- sort*, 312
- sort-vector, 826
- sorter, 194, 217
- span-length, 674
- sparse polynomials, 511
- special, 837
- special form, 9
 - expression, 9, 22
 - operator, 9
- spelling corrector, 560
- sqrt, 885
- stack, 56
- STANDARD LISP, ix
- Staples, John, 504
- starts-with, 126, 317
- state space, 190
- statements, 5
- static-edge-stability, 644
- static-ordering strategy, 631
- Steele, Guy L., Jr., xiii, 48, 457, 777,
781, 810, 825
- Stefik, Mark, 458
- step, 85
- step-daughter, 385
- Stepanov, Alexander A., 27
- Sterling, Leon, 234, 382
- Steve's Ice Cream, 457
- Stickel, Mark, 426, 504
- storage management, 25

- strategy
 - generate-and-test, 376
 - random-ordering, 630
 - static-ordering, 631
- stream, 281
- string->list, 709
- string-set!, 756
- strings, 22
- strip-vowel, 743
- STRIPS, 112
- Stroustrup, Bjarne, 459
- structure sharing, 425
- Stubblefield, William A., 558
- student, 224
- sub, 485, 491
- subcategories, 693
- subject, 716, 724
- subject-predicate agreement, 724
- subjective case, 717
- sublis, 156, 356
- subseq, 69, 895
- subst, 68, 76, 333
- subst-bindings, 357, 361
- substances, 469
- substitute, 62
- subtypep, 82
- success continuation, 389, 425
- successors, 203
- sum, 674, 849
- sum-squares, 316
- summing, 849
- Sussman Anomaly, 142
- Sussman, Gerald, 142, 213, 307, 367, 383, 511, 777, 781, 810, 825
- Svartik, Jan, 748
- svref, 73, 512
- switch-strategies, 602, 623, 627
- switch-viewpoint, 165
- symbol, 23
 - external, 835
 - internal, 835
 - uninterned, 855
- symbol, 302, 391, 623
- symbol-plist, 75
- symbol-value, 94
- symbolic differentiation, 249
- Symbolics, 458
- syntax, 656
 - frame-based, 485
 - uniform, 28
- sys-action, 893
- T, 820
- table, 343
- tag bits, 811
- tagbody, 767, 837
- tail, 282, 284
- tail-recursive, 766
- tail-recursive call, 323
- Tanimoto, Steven, 259, 594
- target, 795, 819
- Tatar, Deborah G., xiv
- Tate, Austin, 147, 148
- tax-bracket, 54
- tconc, 341, 342
- Teiresias, 549
- tense, 722
- tense-sem, 731
- terminal-tree-p, 668
- terpri, 84
- test-bears, 492
- test-ex, 90
- test-index, 477
- test-it, 295
- test-unknown-word, 745
- the, 512
- thematic fronting, 725
- theorem proving, 460
- thereis, 847
- think-ahead, 649
- third, 10, 69
- Thomason, Rich, 711
- Thoreau, Henry David, 238
- throw, 624, 754, 769, 837

- thunks, 307
- TI Explorer Lisp Machine, 292, 321
- TI Lisp Machine, 858
- time, 90
- time-string, 623, 626
- TJUMP, 785, 814, 820
- tone language, 755
- top, 813
- top-down parser, 679
- top-edge, 640
- top-level-prove, 361, 364, 368, 391, 409, 484
- Touretzky, David, x, xiv
- tower, 584
- trace, 16
- tracing, 427
- tracing, 16
- tractability, 464
- transitive verbs, 693
- translate-exp, 495
- translate-pair, 224
- translate-to-expression, 224
- tree, 76, 649
- tree, 666, 671
- tree-equal, 76
- tree-lhs, 658
- tree-rhs, 658
- tree-score-or-0, 673
- tree-search, 191, 217
- trie, 343, 472
- trie, 344
- trie-deleted, 344
- triangular vertices, 565
- trip, 199, 200
- trip-problem, 453
- true, 76, 430, 532, 533
- true-p, 533, 536
- truly amazing, wonderful thing, 771
- truth maintenance system (TMS), 497, 504
 - assumption-based (ATMS), 504
- try, 744
- try-dcg, 744
- Tversky, Amos, 558
- TWEAK, 148
- Twenty Questions, 80
- two-dimensional array, 599
- type declaration, 876
- type-checking, 316
- type-of, 82
- typep, 82
- uappend, 335
- ucons, 334
- ulist, 335
- Ullman, J. D., 307
- unbound, 377
- unbound-var-p, 418
- uncertainty, 464
- undebug, 124
- undo-bindings!, 379, 397
- unfactorize, 254
- unification, 349, 352, 684
 - and pattern matching, 352
- unifier, 357
- unify, 354, 356, 361
- unify!, 378, 391, 397
- unify-variable, 354-356, 361
- union*, 670
- unique, 335, 345
- unique name assumption, 467
- unique-cons, 335
- unique-find-anywhere-if, 361, 363
- unity path, 560
- unknown, 532, 533
- unknown words, 664
- unknown-p, 228
- unless, 53, 851
- unprofile, 290
- unprofile1, 289, 291
- untrace, 16
- unwind-protect, 294
- update-cf, 533, 537
- use, 130, 663, 671

- use-eliza-rules, 164, 188, 189
- use-new-world, 500
- use-package, 836
- use-rule, 533, 546
- use-rules, 533, 545
- use-world, 500

- V, 660
- v-d-neighbors, 576
- val, 485, 491
- valid-p, 602, 604
- values, 364
- van Emden, Maarten H., 504
- var, 377–379, 391
- var/1, 418
- var=, 513, 515
- var>, 513, 515
- variable
 - anonymous, 372
 - class, 436
 - generalized, 55
 - instance, 436
 - lexical, 93, 888
 - logic, 349, 352, 541
 - meta-, 697
 - renaming, 481
 - segment, 159
 - special, 93, 888, 889
- variable, 340, 361
- variable-p, 156, 241, 339, 340, 361
- variables-in, 361, 363
- vector-set!, 756
- Verb, 36
- verb, 716, 730, 734, 742
- verb-phrase, 36
- Verb/intr, 694, 699, 701
- Verb/tr, 694, 699, 701
- vertex, 569
- vividness, 504
- vowel-p, 743
- VP, 660
- VP, 687–689, 693, 694, 699, 701, 703, 716, 722, 723
- Vygotsky, Lev Semenovich, 655

- Waibel, Alex, 637
- Waldinger, Richard, 142
- walk, 400
- Walker, Adrian, 711, 748
- Wall, Robert E., 711
- Waltz filtering, 594
- Waltz, David, 594
- WARPLAN, 144, 147, 148
- Warren Abstract Machine (WAM), 407, 426
- Warren, David, 144, 147, 383, 425–426, 711
- Waterman, David A., 558
- Wefald, Eric, 637, 650
- Wegner, Peter, 435
- weighted-squares, 602, 609
- Weinreb, Daniel, 457
- Weise, Daniel, 267, 777
- Weiss, Sholom M., 558
- Weissman, Clark, 259
- Weizenbaum, Joseph, 152, 167
- when, 53, 851
- while, 67, 68, 847
- white, 601, 602
- Whitehead, Alfred, 20
- Whorf, Benjamin Lee, 655
- Wilcox, Bruce, 30
- Wilensky, Robert, xiii, xiv, 213, 383
- win, 311, 312
- winning-value, 602, 613
- Winograd, Terry, 679
- Winston, Patrick, xiv, 213, 214, 367, 383, 564, 594, 777
- Wirth, N., 385
- with, 850
- with-collection, 863
- with-compilation-unit, 411, 429, 893
- with-open-stream, 83

with-profiling, 294
with-resource, 338
with-undo-bindings, 415
withdraw, 439, 442, 446
withdraw :after, 447
withdraw :before, 447
Wogrin, Nancy, 266
Wong, Douglas, 234
Woods, William A., 503
word/n, 741
world, 499
would-flip?, 602, 605
write/l, 413
writing an interpreter, 42

x-square-for, 642
x-square-p, 642
Xlisp, ix
XP, 727
XP, 716, 725, 729
xyz-coords, 201

yes/no, 533, 541
Yukawa, Keitaro, 504

Zabih, Ramin, 777
Zadeh, Lofti, 558
zebra, 375
zebra puzzle, 373, 411, 502
zero-array, 875
ZetaLisp, 811, 840
ZETALISP, ix
Zimmerman, Scott, 769
Zucker, S. W., 594