4. Flow of Control

Lisp provides a variety of structures for flow of control.

Function application is the basic method for construction of programs. Operations are written as the application of a function to its arguments. Usually, Lisp programs are written as a large collection of small functions, each of which implements a simple operation. These functions operate by calling one another, and so larger operations are defined in terms of smaller ones.

A function may always call itself in Lisp. The calling of a function by itself is known as *recursion*; it is analogous to mathematical induction.

The performing of an action repeatedly (usually with some changes between repetitions) is called *iteration*, and is provided as a basic control structure in most languages. The *do* statement of PL/I, the *for* statement of ALGOL/60, and so on are examples of iteration primitives. Lisp provides two general iteration facilities: **do** and **loop**, as well as a variety of special-purpose iteration facilities. (**loop** is sufficiently complex that it is explained in its own chapter later in the manual; see page 350.)

A *conditional* construct is one which allows a program to make a decision, and do one thing or another based on some logical condition. Lisp provides the simple one-way conditionals **and** and **or**, the simple two-way conditional **if**, and more general multi-way conditionals such as **cond** and **selectq**. The choice of which form to use in any particular situation is a matter of personal taste and style.

There are some *non-local exit* control structures, analogous to the *leave*, *exit*, and *escape* constructs in many modern languages. Zetalisp provides for both static (lexical) non-local exits with **block** and **return-from** and dynamic non-local exits with **catch** and **throw**. Another kind of non-local exit is the goto, provided by the **tagbody** and **go** constructs.

Zetalisp also provides a coroutine capability, explained in the section on *stack-groups* (chapter 13, page 256), and a multiple-process facility (see chapter 29, page 682). There is also a facility for generic function calling using message passing; see chapter 21, page 401.

4.1 Compound Statements

progn *body...* *Special form*

The *body* forms are evaluated in order from left to right and the value of the last one is returned. **progn** is the primitive control structure construct for "compound statements".
Example:

```
(foo (cdr a)
     (progn (setq b (extract frob))
            (car b))
     (cadr b))
```

Lambda-expressions, cond forms, do forms, and many other control structure forms use progn implicitly, that is, they allow multiple forms in their bodies.

prog1 *first-form body...* *Special form*

prog1 is similar to progn, but it returns the value of its *first* form rather than its last. It is most commonly used to evaluate an expression with side effects, and return a value which must be computed *before* the side effects happen.

Example:

```
(setq x (prog1 y (setq y x)))
```

interchanges the values of the variables *x* and *y*. prog1 never returns multiple values.

prog2 *first-form second-form body...* *Special form*

prog2 is similar to progn and prog1, but it returns its *second* form. It is included largely for compatibility with old programs.

4.2 Conditionals

if *Special form*

if is the simplest conditional form. The "if-then" form looks like:

```
(if predicate-form then-form)
```

predicate-form is evaluated, and if the result is non-nil, the *then-form* is evaluated and its result is returned. Otherwise, nil is returned.

In the "if-then-else" form, it looks like

```
(if predicate-form then-form else-form)
```

predicate-form is evaluated, and if the result is non-nil, the *then-form* is evaluated and its result is returned. Otherwise, the *else-form* is evaluated and its result is returned.

If there are more than three subforms, if assumes you want more than one *else-form*; if the predicate returns nil, they are evaluated sequentially and the result of the last one is returned.

when *condition body...* *Macro*

If *condition* evaluates to something non-nil, the *body* is executed and its value(s) returned. Otherwise, the value of the when is nil and the *body* is not executed.

unless *condition body...* *Macro*

If *condition* evaluates to nil, the *body* is executed and its value(s) returned. Otherwise, the value of the unless is nil and the *body* is not executed.

cond *Special form*

The cond special form consists of the symbol cond followed by several *clauses*. Each clause consists of a predicate form, called the *condition*, followed by zero or more *body* forms.

```
(cond (condition body body...)
      (condition)
      (condition body ...)
      ... )
```

The idea is that each clause represents a case which is selected if its condition is satisfied and the conditions of all preceding clauses were not satisfied. When a clause is selected, its body forms are evaluated.

cond processes its clauses in order from left to right. First, the condition of the current clause is evaluated. If the result is **nil**, **cond** advances to the next clause. Otherwise, the cdr of the clause is treated as a list of body forms which are evaluated in order from left to right. After evaluating the body forms, **cond** returns without inspecting any remaining clauses. The value of the **cond** form is the value of the last body form evaluated, or the value of the condition if there were no body forms in the clause. If **cond** runs out of clauses, that is, if every condition evaluates to **nil**, and thus no case is selected, the value of the **cond** is **nil**.

Example:

```
(cond ((zerop x)      ;First clause:
       (+ y 3))       ; (zerop x) is the condition.
                      ; (+ y 3) is the body.
      ((null y)       ;A clause with 2 body forms:
       (setq y 4)     ; this
       (cons x z))    ; and this.
      (z)             ;A clause with no body forms: the condition is
                      ; just z. If z is non-nil, it is returned.
      (t              ;A condition of t
       105)           ; is always satisfied.
 )                    ;This is the end of the cond.
```

cond-every *Macro*

cond-every has the same syntax as **cond**, but executes every clause whose condition is satisfied, not just the first. If a condition is the symbol **otherwise**, it is satisfied if and only if no preceding condition is satisfied. The value returned is the value of the last body form in the last clause whose condition is satisfied. Multiple values are not returned.

and *form...* *Special form*

and evaluates the *forms* one at a time, from left to right. If any *form* evaluates to **nil**, and immediately returns **nil** without evaluating the remaining *forms*. If all the *forms* evaluate to non-**nil** values, **and** returns the value of the last *form*.

and can be used in two different ways. You can use it as a logical **and** function, because it returns a true value only if all of its arguments are **true**:

```
(if (and socrates-is-a-person
         all-people-are-mortal)
    (setq socrates-is-mortal t))
```

Because the order of evaluation is well-defined, you can do

```
(if (and (boundp 'x)
         (eq x 'foo))
    (setq y 'bar))
```

knowing that the x in the eq form will not be evaluated if x is found to be void.

You can also use and as a simple conditional form:

```
(and (setq temp (assq x y))
     (rplacd temp z))
(and bright-day
     glorious-day
     (princ "It is a bright and glorious day."))
```

but when is usually preferable.

Note: (and) => t, which is the identity for the and operation.

or *form...* *Special form*

or evaluates the *forms* one by one from left to right. If a *form* evaluates to nil, or proceeds to evaluate the next *form*. If there are no more *forms*, or returns nil. But if a *form* evaluates to a non-nil value, or immediately returns that value without evaluating any remaining *forms*.

As with and, or can be used either as a logical or function, or as a conditional:

```
(or it-is-fish it-is-fowl)

(or it-is-fish it-is-fowl
    (print "It is neither fish nor fowl."))
```

but it is often possible and cleaner to use unless in the latter case.

Note: (or) => nil, the identity for this operation.

selectq *Macro*
case *Macro*
caseq *Macro*

selectq is a conditional which chooses one of its clauses to execute by comparing the value of a form against various constants using eql. Its form is as follows:

```
(selectq key-form
  (test body...)
  (test body...)
  (test body...)
  ...)
```

The first thing selectq does is to evaluate *key-form*; call the resulting value *key*. Then selectq considers each of the clauses in turn. If *key* matches the clause's *test*, the body of the clause is evaluated, and selectq returns the value of the last body form. If there are no matches, selectq returns nil.

A *test* may be any of:

1) A symbol	If the *key* is eql to the symbol, it matches.
2) A number	If the *key* is eql to the number, it matches. *key* must have the same type and the same value as the number.
3) A list	If the *key* is eql to one of the elements of the list, then it matches. The elements of the list should be symbols or numbers.
4) t or otherwise	The symbols t and otherwise are special tests which match anything. Either symbol may be used, it makes no difference; t is mainly for compatibility with Maclisp's caseq construct. To be useful, this should be the last clause in the selectq.

Example:

```
(selectq x
  (foo (do-this))
  (bar (do-that))
  ((baz quux mum) (do-the-other-thing))
  (otherwise (ferror nil "Never heard of ~S" x)))
```

is equivalent to

```
(cond ((eq x 'foo) (do-this))
      ((eq x 'bar) (do-that))
      ((memq x '(baz quux mum)) (do-the-other-thing))
      (t (ferror nil "Never heard of ~S" x)))
```

Note that the *tests* are *not* evaluated; if you want them to be evaluated use select rather than selectq.

case is the Common Lisp name for this construct. caseq is the Maclisp name; it identical to selectq, which is not totally compatible with Maclisp, because selectq accepts otherwise as well as t where caseq would not accept otherwise, and because Maclisp does some error-checking that selectq does not. Maclisp programs that use caseq work correctly so long as they don't use the symbol otherwise as a key.

ecase *key-form clauses...* *Macro*

Like case except that an uncorrectable error is signaled if every clause fails. t or otherwise clauses are not allowed.

ccase *place clauses...* *Macro*

Like ecase except that the error is correctable. The first argument is called *place* because it must be setf'able. If the user proceeds from the error, a new value is read and stored into *place*; then the clauses are tested again using the new value. Errors repeat until a value is specified which makes some clause succeed.

Also see defselect (page 236), a special form for defining a function whose body is like a selectq.

select *Macro*

select is like selectq, except that the elements of the *tests* are evaluated before they are used.

This creates a syntactic ambiguity: if (bar baz) is seen the first element of a clause, is it a list of two forms, or is it one form? select interprets it as a list of two forms. If you want to have a clause whose test is a single form, and that form is a list, you have to write it as a list of one form.

Example:

```
(select (frob x)
   (foo 1)
   ((bar baz) 2)
   (((current-frob)) 4)
   (otherwise 3))
```

is equivalent to

```
(let ((var (frob x)))
  (cond ((eq var foo) 1)
        ((or (eq var bar) (eq var baz)) 2)
        ((eq var (current-frob)) 4)
        (t 3)))
```

selector *Macro*

selector is like select, except that you get to specify the function used for the comparison instead of eq. For example,

```
(selector (frob x) equal
   (('(one . two)) (frob-one x))
   (('(three . four)) (frob-three x))
   (otherwise (frob-any x)))
```

is equivalent to

```
(let ((var (frob x)))
  (cond ((equal var '(one . two)) (frob-one x))
        ((equal var '(three . four)) (frob-three x))
        (t (frob-any x))))
```

select-match *Macro*

select-match is like select but each clause can specify a pattern to match the key against. The general form of use looks like

```
(select-match key-form
  (pattern condition body...)
  (pattern condition body...)
  ...
  (otherwise body...))
```

The value of *key-form* is matched against the *patterns* one at a time until a match succeeds and the accompanying *condition* evaluates to something non-nil. At this point the *body* of that clause is executed and its value(s) returned. If all the patterns/conditions fail, the *body* of the otherwise clause (if any) is executed. A pattern can test the shape of the key object, and set variables which the *condition* form can refer to. All the variables set by the patterns are bound locally to the select-match form.

The patterns are matched using list-match-p (page 92).

Example:

```
(select-match '(a b c)
  ('(,x b ,x) t (vector x))
  ('((,x ,y) b . ,ignore) t (list x y))
  ('(,x b ,y) (symbolp x) (cons x y))
  (otherwise 'lose-big))
```

returns (a . c), having checked (symbolp 'a). The first clause matches only if the there are three elements, the first and third elements are equal and the second element is b. The second matches only if the first element is a list of length two and the second element is b. The third clause accepts any list of length three whose second element is b. The fourth clause accepts anything that did not match the previous clauses.

select-match generates highly optimized code using special instructions.

dispatch *Macro*

(dispatch *byte-specifier number clauses...*) is the same as select (not selectq), but the key is obtained by evaluating (ldb *byte-specifier number*). *byte-specifier* and *number* are both evaluated. Byte specifiers and ldb are explained on page 155.

Example:

```
(princ (dispatch (byte 2 13) cat-type
          (0 "Siamese.")
          (1 "Persian.")
          (2 "Alley.")
          (3 (ferror nil
                     "~S is not a known cat type."
                     cat-type))))
```

It is not necessary to include all possible values of the byte which is dispatched on.

selectq-every *Macro*

selectq-every has the same syntax as selectq, but, like cond-every, executes every selected clause instead of just the first one. If an otherwise clause is present, it is selected if and only if no preceding clause is selected. The value returned is the value of the last form in the last selected clause. Multiple values are not returned. Example:

```
(selectq-every animal
  ((cat dog) (setq legs 4))
  ((bird man) (setq legs 2))
  ((cat bird) (put-in-oven animal))
  ((cat dog man) (beware-of animal)))
```

4.2.1 Comparison Predicates

eq *x y*

(eq *x y*) => t if and only if *x* and *y* are the same object. It should be noted that things that print the same are not necessarily eq to each other. In particular, numbers with the same value need not be eq, and two similar lists are usually not eq.

Examples:

```
(eq 'a 'b) => nil
(eq 'a 'a) => t
(eq (cons 'a 'b) (cons 'a 'b)) => nil
(setq x (cons 'a 'b)) (eq x x) => t
```

Note that in Zetalisp equal fixnums are eq; this is not true in Maclisp. Equality does not imply eq-ness for other types of numbers. To compare numbers, use =; see page 139.

neq *x y*

(neq *x y*) = (not (eq *x y*)). This is provided simply as an abbreviation for typing convenience.

eql *x y*

eql is the same as eq except that if *x* and *y* are numbers of the same type they are eql if they are =.

equal *x y*

The equal predicate returns t if its arguments are similar (isomorphic) objects. Two numbers are equal if they have the same value and type (for example, a float is never equal to a fixnum, even if = is true of them). For conses, equal is defined recursively as the two cars being equal and the two cdrs being equal. Two strings are equal if they have the same length, and the characters composing them are the same; see string=, page 214. Alphabetic case is significant. All other objects are equal if and only if they are eq. Thus equal could have been defined by:

```
(defun equal (x y)
  (cond ((eq x y) t)
        ((and (numberp x) (numberp y))
         (= x y))
        ((and (stringp x) (stringp y))
         (string-equal x y))
        ((and (consp x) (consp y))
         (and (equal (car x) (car y))
              (equal (cdr x) (cdr y))))))
```

As a consequence of the above definition, it can be seen that equal may compute forever when applied to looped list structure. In addition, eq always implies equal; that is, if (eq a b) then (equal a b). An intuitive definition of equal (which is not quite correct) is that two objects are equal if they look the same when printed out. For example:

```
(setq a '(1 2 3))
(setq b '(1 2 3))
(eq a b) => nil
(equal a b) => t
(equal "Foo" "foo") => nil
```

equalp *x y*

equalp is a broader kind of equality than equal. Two objects that are equal are always equalp. In addition, numbers of different types are equalp if they are =. Two character objects are equalp if they are char-equal (that is, they are compared ignoring font, case and meta bits).

Two arrays of any sort are equalp if they have the same dimensions and corresponding elements are equalp. In particular, this means that two strings are equalp if they match ignoring case and font information.

```
(equalp "Foo" "foo") => t
(equalp '1 '1.0) => t
(equalp '(1 "Foo") '(1.0 "foo")) => t
```

Because equalp is a Common Lisp function, it regards a string as having character objects as its elements:

```
(equalp "Foo" #(#≠/F #≠/o #≠/o)) => t
(equalp "Foo" #(#/F #/o #/o)) => nil
```

not *x*
null *x*

not returns t if *x* is nil, else nil. null is the same as not; both functions are included for the sake of clarity. Use null to check whether something is nil; use not to invert the sense of a logical value. Some people prefer to distinguish between nil as falsehood and nil as the empty list by writing:

```
(cond ((not (null lst)) ... )
      ( ... ))
```

rather than

```
(cond (lst ... )
      ( ... ))
```

There is no loss of efficiency, since these compile into exactly the same instructions.

4.3 Iteration

do *Special form*

The do special form provides a simple generalized iteration facility, with an arbitrary number of "index variables" whose values are saved when the do is entered and restored when it is left, i.e. they are bound by the do. The index variables are used in the iteration performed by do. At the beginning, they are initialized to specified values, and then at the end of each trip around the loop the values of the index variables are changed according to specified rules. do allows the programmer to specify a predicate which determines when the iteration will terminate. The value to be returned as the result

of the form may, optionally, be specified.

do comes in two varieties, new-style and old-style. The old-style do is obsolete and exists for Maclisp compatibility only. The more general, "new-style" do looks like:

```
(do ((var init repeat) ...)
    (end-test exit-form ...)
    body...)
```

The first item in the form is a list of zero or more index variable specifiers. Each index variable specifier is a list of the name of a variable *var*, an initial value form *init*, which defaults to nil if it is omitted, and a repeat value form *repeat*. If *repeat* is omitted, the *var* is not changed between repetitions. If *init* is omitted, the *var* is initialized to nil.

An index variable specifier can also be just the name of a variable, rather than a list. In this case, the variable has an initial value of nil, and is not changed between repetitions.

All assignment to the index variables is done in parallel. At the beginning of the first iteration, all the *init* forms are evaluated, then the *vars* are bound to the values of the *init* forms, their old values being saved in the usual way. Note that the *init* forms are evaluated *before* the *vars* are bound, i.e. lexically *outside* of the do. At the beginning of each succeeding iteration those *vars* that have *repeat* forms get set to the values of their respective *repeat* forms. Note that all the *repeat* forms are evaluated before any of the *vars* is set.

The second element of the do-form is a list of an end-testing predicate form *end-test*, and zero or more forms, called the *exit-forms*. This resembles a cond clause. At the beginning of each iteration, after processing of the variable specifiers, the *end-test* is evaluated. If the result is nil, execution proceeds with the body of the do. If the result is not nil, the *exit-forms* are evaluated from left to right and then do returns. The value of the do is the value of the last *exit-form*, or nil if there were no *exit-forms* (*not* the value of the *end-test*, as you might expect by analogy with cond).

Note that the *end-test* gets evaluated before the first time the body is evaluated. do first initializes the variables from the *init* forms, then it checks the *end-test*, then it processes the body, then it deals with the *repeat* forms, then it tests the *end-test* again, and so on. If the end-test returns a non-nil value the first time, then the body is not executed.

If the second element of the form is nil, there is no *end-test* nor *exit-forms*, and the *body* of the do is executed only once. In this type of do it is an error to have *repeats*. This type of do is no more powerful than let; it is obsolete and provided only for Maclisp compatibility.

If the second element of the form is (nil), the *end-test* is never true and there are no *exit-forms*. The *body* of the do is executed over and over. The resulting infinite loop can be terminated by use of return or throw.

do implicitly creates a block with name nil, so return can be used lexically within a do to exit it immediately. This unbinds the do variables and the do form returns whatever values were specified in the return form. See section 4.4, page 75 for more information

on these matters. The *body* of the do is actually treated as a tagbody, so that it may contain go tags (see section 4.5, page 78), but this usage is discouraged as it is often unclear.

Examples of the new form of do:

```
(do ((i 0 (1+ i))          ; This is just the same as the above example,
     (n (array-length foo-array)))
    ((= i n))              ; but written as a new-style do.
  (aset 0 foo-array i)) ; Note how the setq is avoided.

(do ((z list (cdr z)) ; z starts as list and is cdr'd each time.
     (y other-list)   ; y starts as other-list, and is unchanged by the do.
     (x)              ; x starts as nil and is not changed by the do.
     w)               ; w starts as nil and is not changed by the do.
    (nil)             ; The end-test is nil, so this is an infinite loop.
  body)       ; Presumably the body uses return somewhere.
```

The construction

```
(do ((x e (cdr x))
     (oldx x x))
    ((null x))
  body)
```

exploits parallel assignment to index variables. On the first iteration, the value of oldx is whatever value x had before the do was entered. On succeeding iterations, oldx contains the value that x had on the previous iteration.

The *body* of a do may contains no forms at all. Very often an iterative algorithm can be most clearly expressed entirely in the *repeats* and *exit-forms* of a new-style do, and the *body* is empty. For example,

```
(do ((x x (cdr x))
     (y y (cdr y))
     (z nil (cons (f x y) z)))  ;exploits parallel assignment.
    ((or (null x) (null y))
     (nreverse z))              ;typical use of nreverse.
    )                           ;no do-body required.
```

is like (maplist 'f x y) (see page 84).

The old-style do exists only for Maclisp compatibility. It looks like:

(do *var init repeat end-test body...*)

The first time through the loop *var* gets the value of the *init* form; the remaining times through the loop it gets the value of the *repeat* form, which is re-evaluated each time. Note that the *init* form is evaluated before *var* is bound, i.e. lexically *outside* of the do. Each time around the loop, after *var* is set, *end-test* is evaluated. If it is non-nil, the do finishes and returns nil. If the *end-test* evaluated to nil, the *body* of the loop is executed. As with the new-style do, return and go may be used in the body, and they have the same meaning.

Also see loop (page 350), a general iteration facility based on a keyword syntax rather than a list-structure syntax.

do* *Special form*

In a word, do* is to do as let* is to let.

do* works like do but binds and steps the variables sequentially instead of in parallel. This means that the *init* form for one variable can use the values of previous variables. The *repeat* forms refer to the new values of previous variables instead of their old values. Here is an example:

```
(do* ((x xlist (cdr x))
      (y (car x) (car x)))
  (print (list x y)))
```

On each iteration, *y*'s value is the car of *x*. The same construction with do might get an error on entry since *x* might not be bound yet.

do-named *Special form*

do-named is like do but defines a block with a name explicitly specified by the programmer in addition to the block named nil which every do defines. This makes it possible to use return-from to return from this do-named even from within an inner do. An ordinary return there would return from the inner do instead. do-named is obsolete now that block, which is more general and more coherent, exists. See section 4.4, page 75 for more information on block and return-from.

The syntax of do-named is like do except that the symbol do-named is immediately followed by the block name, which should be a symbol.
Example:

```
(do-named george ((a 1 (1+ a))
                  (d 'foo))
                 ((> a 4) 7)
  (do ((c b (cdr c)))
      ((null c))
    ...
    (return-from george (cons b d))
    ...))
```

is equivalent to

```
(block george
  (do ((a 1 (1+ a))
       (d 'foo))
      ((> a 4) 7)
    (do ((c b (cdr c)))
        ((null c))
      ...
      (return-from george (cons b d))
      ...)))
```

t as the name of a do-named behaves somewhat peculiarly, and therefore should be avoided.

do*-named *Special form*

This special form offers a combination of the features of do* and those of do-named. It is obsolete, as is do-named, since it is cleaner to use block.

dotimes (*index count* [*value-expression*]) *body...* *Macro*

dotimes is a convenient abbreviation for the most common integer iteration. dotimes performs *body* the number of times given by the value of *count*, with *index* bound to 0, 1, etc. on successive iterations. When the *count* is exhausted, the value of *value-expression* is returned; or nil, if *value-expression* is missing.

Example:

```
(dotimes (i (truncate m n))
  (frob i))
```

is equivalent to:

```
(do ((i 0 (1+ i))
     (count (truncate m n)))
    ((≥ i count))
  (frob i))
```

except that the name count is not used. Note that i takes on values starting at zero rather than one, and that it stops before taking the value (truncate m n) rather than after. You can use return and go and tagbody-tags inside the body, as with do. dotimes forms return the value of *value-expression*, or nil, unless returned from explicitly with return. For example:

```
(dotimes (i 5)
  (if (eq (aref a i) 'foo)
      (return i)))
```

This form searches the array that is the value of a, looking for the symbol foo. It returns the fixnum index of the first element of a that is foo, or else nil if none of the elements are foo.

dolist (*item list* [*value-expression*]) *body...* *Macro*

dolist is a convenient abbreviation for the most common list iteration. dolist performs *body* once for each element in the list which is the value of *list*, with *item* bound to the successive elements. If the list is exhausted, the value of *value-expression* is returned; or nil, if *value-expression* is missing.

Example:

```
(dolist (item (frobs foo))
  (mung item))
```

is equivalent to:

```
(do ((lst (frobs foo) (cdr lst))
     (item))
    ((null lst))
  (setq item (car lst))
  (mung item))
```

except that the name lst is not used. You can use return and go and tagbody-tags inside the body, as with do.

do-forever *body...* *Macro*

Executes the forms in the body over and over, or until a non-local exit (such as **return**).

4.4 Static Non-Local Exits

The static non-local exit allows code deep within a construct to jump to the end of that construct instantly, not executing anything except **unwind-protect**'s on the way. The construct which defines a static level that can be exited non-locally is called **block** and the construct which exits it is called **return-from**. The **block** being exited must be lexically visible from the **return-from** which says to exit it; this is what 'static' means. By contrast, **catch** and **throw** provide for dynamic non-local exits; refer to the following section. Here is an example of using a static non-local exit:

```
(block top
  (let ((v1 (do-1)))
    (when (all-done v1) (return-from top v1))
    (do-2))
  (do-3)
  ...
  (do-last))
```

If **(all-done v1)** returns non-nil, the entire **block** immediately returns the value of **v1**. Otherwise, the rest of the body of the block is executed sequentially, and ultimately the value or values of **(do-last)** are returned.

Note that the **return-from** form is very unusual: it does not ever return a value itself, in the conventional sense. It isn't useful to write **(setq a (return-from foo 3))**, because when the **return-from** form is evaluated, the containing **block** is immediately exited, and the **setq** never happens.

The fact that **block**'s and **return-from**'s are matched up lexically means you cannot do this:

```
(defun foo (a)
  (block foo1
    (bar a)))

(defun bar (x)
  (return-from foo1 x))
```

The **(return-from foo1 x)** gets an error because there is no lexically visible **block** named **foo1**. The suitable **block** in the caller, **foo**, is not even noticed.

Static handling allows the compiler to produce good code for **return-from**. It is also useful with functional arguments:

```
(defun first-symbol (list)
  (block done
    (mapc #'(lambda (elt)
              (if (symbolp elt) (return-from done elt)))
          list)))
```

The return-from done sees the block done lexically. Even if mapc had a block in it named done it would have no effect on the execution of first-symbol.

When a function is defined with defun with a name which is a symbol, a block whose name is the function name is automatically placed around the body of the function definition. For example,

```
(defun foo (a)
  (if (evenp a)
      (return-from foo (list a)))
  (1+ a))

(foo 4) => (4)
(foo 5) => 6
```

A function written explicitly with lambda does not have a block unless you write one yourself.

A named prog, or a do-named, implicitly defines a block with the specified name. So you can exit those constructs with return-from. In fact, the ability to name prog's was the original way to define a place for return-from to exit, before block was invented.

Every prog, do or loop, whether named or not, implicitly defines a block named nil. Thus, named prog's define *two* block's, one named nil and one named whatever name you specify. As a result, you can use return (an abbreviation for return-from nil) to return from the innermost lexically containing prog, do or loop (or from a block nil if you happen to write one). This function is like assq, but it returns an additional value which is the index in the table of the entry it found. For example,

```
(defun assqn (x table)
  (do ((l table (cdr l))
       (n 0 (1+ n)))
      ((null l) nil)
    (if (eq (caar l) x)
        (return (values (car l) n)))))
```

There is one exception to this: a prog, do or loop with name t defines only the block named t, no block named nil. The compiler used to make use of this feature in expanding certain built-in constructs into others.

block *name body...* *Special form*

Executes *body*, returning the values of the last form in *body*, but permitting non-local exit using **return-from** forms present lexically within *body*. *name* is not evaluated, and is used to match up **return-from** forms with their **block**'ss.

```
(block foo
  (return-from foo 24) t) => 24
(block foo t) => t
```

return-from *name values* *Special form*

Performs a non-local exit from the innermost lexically containing **block** whose name is *name*. *name* is not evaluated. When the compiler is used, **return-from**'s are matched up with **block**'s at compile time.

values is evaluated and its values become the values of the exited **block** form.

A **return-from** form may appear as or inside an argument to a regular function, but if the **return-from** is executed then the function will never actually be called. For example,

```
(block done
  (foo (if a (return-from done t) nil)))
```

foo is actually called only if **a**'s value is **nil**. This style of coding is not recommended when **foo** is actually a function.

return-from can also be used with zero value forms, or with several value forms. Then *one* value is returned from each value form. Originally **return-from** always returned only one value from each value form, even when there was only one value form. Passing back all the values when there is a single *values* form is a later change, which is also the Common Lisp standard. In fact, the single value form case is much more powerful and subsumes all the others. For example,

```
(return-from foo 1 2)
```

is equivalent to

```
(return-from foo (values 1 2))
```

and

```
(return-from foo)
```

is equivalent to

```
(return-from foo (values))
```

It is unfortunate that the case of one value form is treated differently from all other cases, but the power of being able to propagate any number of values from a single form is worth it.

To return precisely one value, use (**return-from foo** (**values** *form*)). It is legal to write simply (**return-from foo**), which returns no values from the **block**. See section 3.7, page 55 for more information.

return *values* *Special form*

Is equivalent to (**return-from nil** *values*). It returns from a **block** whose name is **nil**.

In addition, **break** (see page 795) recognizes the typed-in form (**return** *value*) specially. **break** evaluates *value* and returns it.

return-list *list* *Special form*

This function is like return except that each element of *list* is returned as a separate value from the block that is exited.

return-list is obsolete, since (return (values-list *list*)) does the same thing.

4.5 Tags and Gotos

Jumping to a label or *tag* is another kind of static non-local exit. Compared with return-from, it allows more flexibility in choosing where to send control to, but does not allow values to be sent along. This is because the tag does not have any way of saying what to *do* with any values.

To define a tag, the tagbody special form is used. In the body of a tagbody, all lists are treated as forms to be evaluated (called *statements* when they occur in this context). If no goto happens, all the forms are evaluated in sequence and then the tagbody form returns nil. Thus, the statements are evaluated only for effect.

An element of the tagbody's body which is a symbol is not a statement but a tag instead. It identifies a place in the sequence of statements which you can go to. Going to a tag is accomplished by the form (go *tag*), executed at any point lexically within the tagbody.

go transfers control immediately to the first statement following *tag* in its tagbody, pausing only to deal with any unwind-protects that are being exited as a result. If there are no more statements after *tag* in its tagbody, then that tagbody returns nil immediately.

All lexically containing tagbody's are eligible to contain the specified tag, with the innermost tagbody taking priority. If no suitable tag is found, an error is signaled. The compiler matches go's with tags at compile time and issues a compiler warning if no tag is found. Example:

```
(block nil
  (tagbody
    (setq x somefrob)
  loop
    do something
    (if some predicate (go endtag))
    do something more
    (if (minusp x) (go loop))
  endtag
    (return z)))
```

is a kind of iteration made out of go-to's. This tagbody can never exit normally because the return in the last statement takes control away from it. This use of a return and block is how one encapsulates a tagbody to produce a non-nil value.

It works to go from an internal lambda function to a tag in a lexically containing function, as in

```
(defun foo (a)
  (tagbody
   t1
    (bar #'(lambda () (go t1)))))
```

If bar ever invokes its argument, control goes to t1 and bar is invoked anew. Not very useful, but it illustrates the technique.

tagbody *statements-and-tags...* *Special form*

Executes all the elements of *statements-and-tags* which are lists (the statements), and then returns nil. But meanwhile, all elements of *statements-and-tags* which are symbols (the tags) are available for use with go in any of the statements. Atoms other than symbols are meaningless in a tagbody.

The reason that tagbody returns nil rather than the value of the last statement is that the designers of Common Lisp decided that one could not reliably return a value from the tagbody by writing it as the last statement since some of the time the expression for the desired value would be a symbol rather than a list, and then it would be taken as a tag rather than the last statement and it would not work.

go *tag* *Special form*

The go special form is used to "go-to" a tag defined in a lexically containing tagbody form (or other form which implicitly expands into a tagbody, such as prog, do or loop). *tag* must be a symbol. It is not evaluated.

prog *Special form*

prog is an archaic special form which provides temporary variables, static non-local exits, and tags for go. These aspects of prog were individually abstracted out to inspire let, block and tagbody. Now prog is obsolete, as it is much cleaner to use let, block, tagbody or all three of them, or do or loop. But prog appears in so many programs that it cannot be eliminated.

A typical prog looks like (prog (*variables...*) *body...*), which is equivalent to

```
(block nil
  (let (variables...)
    (tagbody body...)))
```

If the first subform of a prog is a non-nil symbol (rather than a list of variables), it is the name of the prog, and return-from (see page 77) can be used to return from it. A *named prog* looks like

```
(prog name (variables...) body...)
```

and is equivalent to

```
(block name
  (block nil
    (let (variables...)
      (tagbody body...))))
```

prog* *Special form*

The prog* special form is almost the same as prog. The only difference is that the binding and initialization of the temporary variables is done *sequentially*, so each one can depend on the previous ones. Thus, the equivalent code would use let* rather than let.

4.6 Dynamic Non-Local Exits

catch *tag body...* *Special form*

catch is a special form used with the throw function to do non-local exits. First *tag* is evaluated; the result is called the *tag* of the catch. Then the *body* forms are evaluated sequentially, and the values of the last form are returned. However, if, during the evaluation of the body, the function throw is called with the same tag as the tag of the catch, then the evaluation of the body is aborted, and the catch form immediately returns the values of the second argument to throw without further evaluating the current *body* form or the rest of the body.

The *tag*'s are used to match up throw's with catch's. (catch 'foo *form*) catches a (throw 'foo *form*) but not a (throw 'bar *form*). It is an error if throw is done when there is no suitable catch (or catch-all; see below).

Any Lisp object may be used as a catch tag. The values t and nil for *tag* are special: a catch whose tag is one of these values catches throws regardless of tag. These are only for internal use by unwind-protect and catch-all respectively. The only difference between t and nil is in the error checking; t implies that after a "cleanup handler" is executed control will be thrown again to the same tag, therefore it is an error if a specific catch for this tag does not exist higher up in the stack. With nil, the error check isn't done. Example:

```
(catch 'negative
   (values
     (mapcar #'(lambda (x)
                 (cond ((minusp x)
                        (throw 'negative
                          (values x :negative)))
                       (t (f x)) )))
             y)
     :positive))
```

returns a list of f of each element of y, and :positive, if they are all positive, otherwise the first negative member of y, and :negative.

catch-continuation *tag throw-cont non-throw-cont body...* *Macro*
catch-continuation-if *cond-form tag throw-cont non-throw-cont body...* *Macro*

The catch-continuation special form makes it convenient to discriminate whether exit is normal or due to a throw.

The *body* is executed inside a catch on *tag* (which is evaluated). If *body* returns normally, the function *non-throw-cont* is called, passing all the values returned by the last form in *body* as arguments. This function's values are returned from the catch-continuation.

If on the other hand a throw to *tag* occurs, the values thrown are passed to the function *throw-cont*, and its values are returned.

If a continuation is explicitly written as nil, it is not called at all. The arguments that would have been passed to it are returned instead. This is equivalent to using values as the function; but explicit nil is optimized, so use that.

catch-continuation-if differs only in that the catch is not done if the value of the *cond-form* is nil. In this case, the non-throw continuation if any is always called.

In the general case, consing is necessary to record the multiple values, but if a continuation is an explicit #'(lambda ...) with a fixed number of arguments, or if a continuation is nil, it is open coded and the consing is avoided.

throw *tag values-form* *Special form*

throw is the primitive for exiting from a surrounding catch. *tag* is evaluated, and the result is matched (with eq) against the tags of all active catch'es; the innermost matching one is exited. If no matching catch is dynamically active, an error is signaled.

All the values of *values-form* are returned from the exited catch.

catch'es with tag nil always match any throw. They are really catch-all's. So do catch'es with tag t, which are unwind-protect's, but if the only matching catch'es are these then an error is signaled anyway. This is because an unwind-protect always throws again after its cleanup forms are finished; if there is nothing to catch after the last unwind-protect, an error will happen then, and it is better to detect the error sooner.

The values t, nil, and 0 for *tag* are reserved and used for internal purposes. nil may not be used, because it would cause confusion in handling of unwind-protect's. t may only be used with *unwind-stack. 0 and nil are used internally when returning out of an unwind-protect.

***catch** *form tag* *Macro*
***throw** *form tag* *Macro*

Old, obsolete names for catch and throw.

sys:throw-tag-not-seen (error) *Condition*

This is signaled when throw (or *unwind-stack) is used and there is no catch for the specified tag. The condition instance supports these extra operations:

:tag — The tag being thrown to.

:value — The value being thrown (the second argument to throw).

:count
:action — The additional two arguments given to *unwind-stack, if that was used.

The error occurs in the environment of the throw; no unwinding has yet taken place.

The proceed type :new-tag expects one argument, a tag to throw to instead.

***unwind-stack** *tag value active-frame-count action*

This is a generalization of **throw** provided for program-manipulating programs such as the debugger.

tag and *value* are the same as the corresponding arguments to **throw**.

A *tag* of **t** invokes a special feature whereby the entire stack is unwound, and then the function *action* is called (see below). During this process **unwind-protect**'s receive control, but **catch-all**'s do not. This feature is provided for the benefit of system programs which want to unwind a stack completely.

active-frame-count, if non-**nil**, is the number of frames to be unwound. The definition of a frame is implementation-dependent. If this counts down to zero before a suitable **catch** is found, the ***unwind-stack** terminates and *that frame* returns *value* to whoever called it. This is similar to Maclisp's **freturn** function.

If *action* is non-**nil**, whenever the ***unwind-stack** would be ready to terminate (either due to *active-frame-count* or due to *tag* being caught as in **throw**), instead *action* is called with one argument, *value*. If *tag* is **t**, meaning throw out the whole way, then the function *action* is not allowed to return. Otherwise the function *action* may return and its value will be returned instead of *value* from the **catch**—or from an arbitrary function if *active-frame-count* is in use. In this case the **catch** does not return multiple values as it normally does when thrown to. Note that it is often useful for *action* to be a stack-group.

Note that if both *active-frame-count* and *action* are **nil**, ***unwind-stack** is identical to **throw**.

unwind-protect *protected-form cleanup-form...* *Special form*

Sometimes it is necessary to evaluate a form and make sure that certain side-effects take place after the form is evaluated; a typical example is:

```
(progn
   (turn-on-water-faucet)
   (hairy-function 3 nil 'foo)
   (turn-off-water-faucet))
```

The non-local exit facilities of Lisp create situations in which the above code won't work, however: if **hairy-function** should use **throw**, **return** or **go** to transfer control outside of the **progn** form, then **(turn-off-water-faucet)** will never be evaluated (and the faucet will presumably be left running). This is particularly likely if **hairy-function** gets an error and the user tells the debugger to give up and flush the computation.

In order to allow the above program to work, it can be rewritten using **unwind-protect** as follows:

```
(unwind-protect
    (progn (turn-on-water-faucet)
           (hairy-function 3 nil 'foo))
  (turn-off-water-faucet))
```

If hairy-function transfers control out of the evaluation of the unwind-protect, the (turn-off-water-faucet) form is evaluated during the transfer of control, before control arrives at the catch, block or go tag to which it is being transferred.

If the progn returns normally, then the (turn-off-water-faucet) is evaluated, and the unwind-protect returns the result of the progn.

The general form of unwind-protect looks like

```
(unwind-protect
    protected-form
  cleanup-form1
  cleanup-form2
  ...)
```

protected-form is evaluated, and when it returns or when it attempts to transfer control out of the unwind-protect, the *cleanup-forms* are evaluated. The value of the unwind-protect is the value of *protected-form*. Multiple values returned by the *protected-form* are propagated back through the unwind-protect.

The cleanup forms are run in the variable-binding environment that you would expect: that is, variables bound outside the scope of the unwind-protect special form can be accessed, but variables bound inside the *protected-form* can't be. In other words, the stack is unwound to the point just outside the *protected-form*, then the cleanup handler is run, and then the stack is unwound some more.

catch-all *body...* *Macro*

(catch-all *form*) is like (catch *some-tag form*) except that it catches a throw to any tag at all. Since the tag thrown to is one of the returned values, the caller of catch-all may continue throwing to that tag if he wants. The one thing that catch-all does not catch is a *unwind-stack with a tag of t. catch-all is a macro which expands into catch with a *tag* of nil.

catch-all returns all the values thrown to it, or returned by the body, plus three additional values: the tag thrown to, the active-frame-count, and the action. The tag value is nil if the body returned normally. The last two values are the third and fourth arguments to *unwind-stack (see page 82) if that was used, or nil if an ordinary throw was done or if the body returned normally.

If you think you want this, most likely you are mistaken and you really want unwind-protect.

4.7 Mapping

map *fcn* &rest *lists*
mapl *fcn* &rest *lists*
mapc *fcn* &rest *lists*
maplist *fcn* &rest *lists*
mapcar *fcn* &rest *lists*
mapcon *fcn* &rest *lists*
mapcan *fcn* &rest *lists*

Mapping is a type of iteration in which a function is successively applied to pieces of a list. There are several options for the way in which the pieces of the list are chosen and for what is done with the results returned by the applications of the function.

For example, **mapcar** operates on successive *elements* of the list. As it goes down the list, it calls the function giving it an element of the list as its one argument: first the car, then the cadr, then the caddr, etc., continuing until the end of the list is reached. The value returned by **mapcar** is a list of the results of the successive calls to the function. An example of the use of **mapcar** would be **mapcar**'ing the function **abs** over the list **(1 -2 -4.5 6.0e15 -4.2)**, which would be written as **(mapcar (function abs) '(1 -2 -4.5 6.0e15 -4.2))**. The result is **(1 2 4.5 6.0e15 4.2)**.

In general, the mapping functions take any number of arguments. For example,

```
(mapcar f x1 x2 ... xn)
```

In this case *f* must be a function of *n* arguments. **mapcar** proceeds down the lists *x1*, *x2*, ..., *xn* in parallel. The first argument to *f* comes from *x1*, the second from *x2*, etc. The iteration stops as soon as any of the lists is exhausted. (If there are no lists at all, then there are no lists to be exhausted, so *f* is called repeatedly without end. This is an obscure way to write an infinite loop. It is supported for consistency.) If you want to call a function of many arguments where one of the arguments successively takes on the values of the elements of a list and the other arguments are constant, you can use a circular list for the other arguments to **mapcar**. The function **circular-list** is useful for creating such lists; see page 93.

There are five other mapping functions besides **mapcar**. **maplist** is like **mapcar** except that the function is applied to the list and successive cdrs of that list rather than to successive elements of the list. **map** (or **mapl**) and **mapc** are like **maplist** and **mapcar** respectively, except that they don't return any useful value. These functions are used when the function is being called merely for its side-effects, rather than its returned values. **mapcan** and **mapcon** are like **mapcar** and **maplist** respectively, except that they combine the results of the function using **nconc** instead of **list**. That is, **mapcon** could have been defined by

```
(defun mapcon (f x y)
    (apply 'nconc (maplist f x y)))
```

Of course, this definition is less general than the real one.

Sometimes a **do** or a straightforward recursion is preferable to a map; however, the mapping functions should be used wherever they naturally apply because this increases the clarity of the code.

Often *f* is a lambda-expression, rather than a symbol; for example,

```
(mapcar (function (lambda (x) (cons x something)))
        some-list)
```

The functional argument to a mapping function must be a function, acceptable to apply—it cannot be a macro or the name of a special form.

Here is a table showing the relations between the six map functions.

		applies function to	
		successive sublists	successive elements
returns	its own second argument	map(l)	mapc
	list of the function results	maplist	mapcar
	nconc of the function results	mapcon	mapcan

Note that map and mapl are synonymous. map is the traditional name of this function. mapl is the Common Lisp name. In Common Lisp, the function map does something different and incompatible; see cli:map, page 191. mapl works the same in traditional Zetalisp and Common Lisp.

There are also functions (mapatoms and mapatoms-all) for mapping over all symbols in certain packages. See the explanation of packages (chapter 27, page 636).

You can also do what the mapping functions do in a different way by using loop. See page 350.