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 274.) There is also a very general construct to allow the traditional "goto" control structure, called prog.

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. The general ones are *catch and *throw; there is also return and its variants, used for exiting the iteration constructs do, loop, and prog.

Zetalisp also provides a coroutine capability, explained in the section on stack-groups (chapter 12, page 186), and a multiple-process facility (see chapter 26, page 538). There is also a facility for generic function calling using message passing; see chapter 20, page 321.

4.1 Conditionals

if

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... \(\text{Special Form}\)

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... \(\text{Special Form}\)

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** \(\text{Special Form}\)

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

\[
\text{cond} \quad \text{(antecedent consequent consequent...)} \\
\quad \text{(antecedent)} \\
\quad \text{(antecedent consequent ...)} \\
\quad \ldots 
\]

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

Cond processes its clauses in order from left to right. First, the antecedent 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 consequent forms which are evaluated in order from left to right. After evaluating the consequents, cond returns without inspecting any remaining clauses. The value of the cond special form is the value of the last consequent evaluated, or the value of the antecedent if there were no consequents in the clause. If cond runs out of clauses, that is, if every antecedent evaluates to nil, and thus no case is selected, the value of the cond is nil.

**Example:**

\[
\text{cond} \quad \text{((zerop x))} \quad ;\text{First clause:} \\
\quad \text{(+ y 3))} \quad ;\text{ (zerop x) is the antecedent.} \\
\quad \text{(+ y 3))} \quad ;\text{ (+ y 3) is the consequent.} \\
\quad \text{((null y))} \quad ;\text{A clause with 2 consequents:} \\
\quad \text{(setq y 4))} \quad ;\text{this} \\
\quad \text{(cons x y))} \quad ;\text{and this.} \\
\quad \text{(z))} \quad ;\text{A clause with no consequents: the antecedent is} \\
\quad \text{(t 105))} \quad ;\text{just z. If z is non-nil, it will be returned.} \\
\quad \text{(t 105))} \quad ;\text{is always satisfied.} \\
\quad \text{)} \quad ;\text{This is the end of the cond.}
\]
**cond-every**

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

**and 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. So you can use it as a predicate:

```lisp
(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:

```lisp
(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 unbound.

You can also use and as a simple conditional form:

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

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

**or 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.

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

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

**Selectq** is a conditional which chooses one of its clauses to execute by comparing the value of a form against various constants, which are typically keyword symbols. Its form is as follows:

\[
\text{selectq key-form}
\begin{align*}
&\text{test consequent consequent} \\
&\text{test consequent consequent} \\
&\text{test consequent consequent} \\
\ldots
\end{align*}
\]

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 consequents of this clause are evaluated, and selectq returns the value of the last consequent. If there are no matches, selectq returns nil.

A *test* may be any of:

1) A symbol
   If the *key* is eq to the symbol, it matches.

2) A number
   If the *key* is eq to the number, it matches. Only small numbers (fixnums) will work.

3) A list
   If the *key* is eq to one of the elements of the list, then it matches. The elements of the list should be symbols or fixnums.

4) t or otherwise
   The symbols t and otherwise are special keywords 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.

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

Example:

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

is equivalent to

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

Also see defselect (page 167), a special form for defining a function whose body is like a selectq.
**select**

Special Form

select is the same as 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**

Special Form

selector is the same as 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**

Special Form

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.
The patterns can be arbitrary s-expressions, with variables signified by `#:variable'. When the pattern is matched against the object, the variables are to be bound to their matched values. Different occurrences of the same variable in a given pattern must match to the same thing, so that

\[
\begin{align*}
\text{(select-match '(a b c)} \\
&((#:x b #:x) \ t \ 'lose) \\
&((#:x b #:y) \ t \ 'win) \\
&\text{(otherwise 'lose-big)})
\end{align*}
\]

returns win. The variables mentioned in the patterns need not be bound by the user; they are bound by the expression resulting from the expansion of the select-match.

The expression `#:ignore' or `#:nil matches everything and binds no variable. `#:nil is preferred.

**dispatch**

*Special Form*

(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 116.

Example:

\[
\begin{align*}
\text{(princ (dispatch 0202 cat-type)} \\
&\quad (0 "Siamese.") \\
&\quad (1 "Persian.") \\
&\quad (2 "Alley.") \\
&\quad (3 (ferror nil \\
&\quad \quad "~S is not a known cat type." \\
&\quad \quad cat-type)))
\end{align*}
\]

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

**selectq-every**

*Special Form*

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:

\[
\begin{align*}
\text{(selectq-every animal)} \\
&\quad ((\text{cat dog}) \ (setq legs 4)) \\
&\quad ((\text{bird man}) \ (setq legs 2)) \\
&\quad ((\text{cat bird}) \ (\text{put-in-oven animal})) \\
&\quad ((\text{cat dog man}) \ (\text{beware-of animal}))
\end{align*}
\]

**caseq**

*Special Form*

The caseq special form is provided for Maclisp compatibility. It is exactly the same as selectq. This is not perfectly 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 will work correctly so long as they don't use the symbol otherwise as the key.
4.2 Iteration

**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.

The more general, so-called "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 will never be processed.
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 infinite loop can be terminated by use of return or *throw.

If a return special form is evaluated inside the body of a do, then the do immediately stops, unbinds its variables, and returns the values given to return. See page 52 for more details about return and its variants. go special forms (see page 52) and prog-tags can also be used inside the body of a do and they mean the same thing that they do inside prog forms, but we discourage their use since they complicate the control structure in a hard-to-understand way.

The other, so-called "old-style" do looks like:

\[\text{(do var init repeat end-test body...)}\]

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 evaluates 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.

Examples of the older variety of do:

\[
\begin{align*}
\text{(setq n (array-length foo-array))} \\
\text{(do i 0 (1+ i) (= i n))} \\
\text{(aset 0 foo-array i))} \\
\end{align*}
\]

; zeroes out the array foo-array

\[
\begin{align*}
\text{(do zz x (cdr zz) (or (null zz)} \\
\text{(zerop (f (car zz)))))} \\
\text{; this applies f to each element of x} \\
\text{; continuously until f returns zero.} \\
\text{; Note that the do has no body.}
\end{align*}
\]

\text{return forms are often useful to do simple searches:}

\[
\begin{align*}
\text{(do i 0 (1+ i) (= i n))} & \quad ; \text{iterate over the length of foo-array.} \\
\text{(and (= (aref foo-array i) 5))} & \quad ; \text{if we find an element which} \\
\text{; equals 5,} \\
\text{(return i)))} & \quad ; \text{then return its index.}
\end{align*}
\]

Examples of the new form of do:

\[
\begin{align*}
\text{(do ((i 0 (1+ i))} & \quad ; \text{this is just the same as the above example,} \\
\text{(n (array-length foo-array)))} \\
\text{((= i n))} & \quad ; \text{but written as a new-style do.} \\
\text{(aset 0 foo-array i))} & \quad ; \text{Note how the setq is avoided.}
\end{align*}
\]
{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.

In either form of do, the body may contain 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.

   (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 57).

Also see loop (page 274), a general iteration facility based on a keyword syntax rather than a list-
structure syntax.

\[\textbf{do*}\]

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

\[\textbf{Special Form}\]

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 will be the car of x. The same construction with do would
get an error on entry since x would not have an old value yet.
**do-named**

*Special Form*

Sometimes one **do** is contained inside the body of an outer **do**. The return function always returns from the innermost surrounding **do**, but sometimes you want to return from an outer **do** while within an inner **do**. You can do this by giving the outer **do** a name. You use **do-named** instead of **do** for the outer **do**, and use **return-from** (see page 53), specifying that name, to return from the **do-named**.

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

Example:

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

If the symbol **t** is used as the name, then it will be made "invisible" to **returns**; that is, **returns** inside that **do-named** will return to the next outermost level whose name is not **t**. **(return-from t ...)** will return from a **do-named** named **t**. This feature is not intended to be used by user-written code; it is for macros to expand into.

If the symbol **nil** is used as the name, it is as if this were a regular **do**. Not having a name is the same as being named **nil**.

**progs** and **loops** can have names just as **dos** can. Since the same functions are used to return from all of these forms, all of these names are in the same name-space; a **return** returns from the innermost enclosing iteration form, no matter which of these it is, and so you need to use names if you nest any of them within any other and want to return to an outer one from inside an inner one.

**do*-named**

*Special Form*

This special form offers a combination of the features of **do** and those of **do-named**.

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

*Special Form*

**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:

```lisp
(dotimes (i (// m n))
  (frob i))
```

is equivalent to:
(do ((i 0 (+ i)))
    (count (/= 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 (/= m n) rather than after.
You can use return and go and prog-tags inside the body, as with do. dotimes forms return nil or the value of value-expression 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...  Special Form
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 prog-tags inside the body, as with do.

do-forever body...  Special Form
Executes the forms in the body over and over, or until one exits with return.

keyword-extract  Special Form
keyword-extract is a semi-obsolete method of decoding keyword arguments, used before &key (see page 23) was implemented. The form

(keyword-extract key-list iteration-var
   keywords flags other-clauses...)
will parse the keywords out into local variables of the function. key-list is a form which evaluates to the list of keyword arguments; it is generally the function’s &rest argument. iteration-var is a variable used to iterate over the list; sometimes other-clauses will use the form

(car (setq iteration-var (cdr iteration-var)))
to extract the next element of the list. (Note that this is not the same as pop, because it does the car after the cdr, not before.)
keywords defines the symbols which are keywords to be followed by an argument. Each element of keywords is either the name of a local variable which receives the argument and is also the keyword, or a list of the keyword and the variable, for use when they are different or the keyword is not to go in the keyword package. Thus if keywords is (foo (ugh bletch) bar) then the keywords recognized will be :foo, ugh, and :bar. If :foo is specified its argument will be stored into foo. If :bar is specified its argument will be stored into bar. If ugh is specified its argument will be stored into bletch.

Note that keyword-extract does not bind these local variables; it assumes you will have done that somewhere else in the code that contains the keyword-extract form.

flags defines the symbols which are keywords not followed by an argument. If a flag is seen its corresponding variable is set to t. (You are assumed to have initialized it to nil when you bound it with let or &aux.) As in keywords, an element of flags may be either a variable from which the keyword is deduced, or a list of the keyword and the variable. Note: this style of calling convention is now considered undesirable. The gain in uniformity from requiring an explicit value with each keyword greatly outweighs the convenience of not having to say t.

If there are any other-clauses, they are selects clauses selecting on the keyword being processed. These clauses are for handling any keywords that are not handled by the keywords and flags elements. These can be used to do special processing of certain keywords for which simply storing the argument into a variable is not good enough. After the other-clauses there will be an otherwise clause to complain about any undefined keywords found in key-list.

prog

prog is a special form which provides temporary variables, sequential evaluation of forms, and a "goto" facility. A typical prog looks like:

\[
\begin{align*}
\text{(prog (var1 var2 (var3 init3) var4 (var5 init5))} \\
\text{tag1} \\
\text{statement1} \\
\text{statement2} \\
\text{tag2} \\
\text{statement3} \\
\vdots
\end{align*}
\]

The first subform of a prog is a list of variables, each of which may optionally have an initialization form. The first thing evaluation of a prog form does is to evaluate all of the init forms. Then each variable that had an init form is bound to its value, and the variables that did not have an init form are bound to nil.

Example:

\[
\begin{align*}
\text{(prog ((a t) b (c 5) (d (car '(zz . pp)))}) \\
\text{body...}
\end{align*}
\]

The initial value of a is t, that of b is nil, that of c is the fixnum 5, and that of d is the symbol zz. The binding and initialization of the variables is done in parallel; that is, all the initial values are computed before any of the variables are changed. prog* (see page 52) is the same as prog except that this initialization is sequential rather than
The part of a `prog` after the variable list is called the *body*. Each element of the body is either a symbol, in which case it is called a *tag*, or anything else (almost always a list), in which case it is called a *statement*.

After `prog` binds the variables, it processes each form in its body sequentially. *tags* are skipped over. *statements* are evaluated, and their returned values discarded. If the end of the body is reached, the `prog` returns `nil`. However, two special forms may be used in `prog` bodies to alter the flow of control. If `(return x)` is evaluated, `prog` stops processing its body, evaluates `x`, and returns the result. If `(go tag)` is evaluated, `prog` jumps to the part of the body labeled with the `tag`, where processing of the body is continued. `tag` is not evaluated. `return` and `go` and their variants are explained fully below.

The compiler requires that `go` and `return` forms be *lexically* within the scope of the `prog`; it is not possible for a function called from inside a `prog` body to `return` to the `prog`. That is, the `return` or `go` must be inside the `prog` itself, not inside a function called by the `prog`. (This restriction happens not to be enforced in the interpreter, but since all programs are eventually compiled, the convention should be adhered to. The restriction will be imposed in future implementations of the interpreter.)

See also the `do` special form, which uses a body similar to `prog`. The `do`, `*catch`, and `*throw` special forms are included in Zetalisp as an attempt to encourage goto-less programming style, which often leads to more readable, more easily maintained code. The programmer is recommended to use these forms instead of `prog` wherever reasonable.

If the first subform of a `prog` is a non-nil symbol (rather than a variable list), it is the name of the `prog`, and `return-from` (see page 53) can be used to return from it. See `do-named`, page 48.

Example:

```lisp
(prog (x y z) ; x, y, z are prog variables - temporaries.
  (setq y (car w) z (cdr w)) ; w is a free variable.
  loop
    (cond ((null y) (return x))
          ((null z) (go err)))
  rejoin
    (setq x (cons (cons (car y) (car z))
                  x))
    (setq y (cdr y)
            z (cdr z))
    (go loop)
err
  (break are-you-sure? t)
  (setq z y)
  (go rejoin))
```
progs*

The progs* 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. For example,

\[
\text{(progs* ((y z) (x (car y)))}
\text{(return x))}
\]

returns the car of the value of z.

\text{go tag}

The go special form is used to do a "go-to" within the body of a do or a prog. The tag must be a symbol. It is not evaluated. go transfers control to the point in the body labelled by a tag eq to the one given. If there is no such tag in the body, the bodies of lexically containing progs and dos (if any) are examined as well. If no tag is found, an error is signalled.

Example:

\[
\text{(prog (x y z)}
\text{(setq x some-frob)}
\text{loop}
\text{do something}
\text{(if some-predicate (go endtag))}
\text{do something more}
\text{(if (minusp x) (go loop))}
\text{endtag}
\text{(return z))}
\]

\text{return value...}

\text{return} is used to exit from a prog-like special form (prog, progs*, do, do-named, dotimes, dolist, loop, etc.) The value forms are evaluated, and the resulting values are returned by the prog as its values.

In addition, break (see page 644) recognizes the typed-in form (return value) specially. If this form is typed at a break, value will be evaluated and returned as the value of break. If not specially recognized by break, and not inside a prog-like form, return will cause an error.

Example:

\[
\text{(do ((x x (cdr x)))}
\text{(n 0 (+ n 2)))}
\text{((null x) n)}
\text{(cond ((atom (car x)})
\text{ (setq n (1+ n))}
\text{ ((memq (caar x) '(sys boom bleah))}
\text{ (return n))))}
\]

Note that the \text{return} form is very unusual: it does not ever return a value itself, in the conventional sense. It isn't useful to write \text{(setq a (return 3))}, because when the return form is evaluated, the containing do or prog is immediately exited, and the setq never happens.
A return form may appear as or inside an argument to a regular function, but if the return is executed then the function will never actually be called. The same is true of go. For example,

```
(prog ()
  (foo (if a (return t) nil)))
```

foo will actually be called only if a's value is nil. However, this style of coding is not recommended.

return can also be used with multiple arguments, to return multiple values from a prog or do. For example,

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

This function is like assq, but it returns an additional value which is the index in the table of the entry it found.

However, if you use return with only one subform, then the prog or do will return all of the values returned by that subform. That is, if you do

```
(prog ()
  ... (return (foo 2)))
```

and the function foo returns many values, then the prog will return all of those values. In fact, this means that

```
(return (values form1 form2 form3))
```

is the same as

```
(return form1 form2 form3)
```

To return precisely one value, use (return (values form)).

It is legal to write simply (return), which will return from the prog without returning any values.

See section 3.5, page 33 for more information.

**return-from name value...**

*Special Form*

The value forms are evaluated, and then are returned from the innermost containing prog-like special form whose name is name. See the description of do-named (page 48) in which named dos and progs are explained.

**return-list list**

This function is like return except that the prog returns all of the elements of list; if list has more than one element, the prog does a multiple-value return.

This is semi-obsolete now, since (return (values-list list)) does the same thing.
To direct the returned values to a prog or do-named of a specific name, use
\( \text{return-from name (values-list list)} \).

Also see defunp (page 159), a variant of defun that incorporates a prog into the function body.

4.3 Non-Local Exits

\texttt{*catch tag body...} \hfill \textbf{Special Form}

\texttt{*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 value of the last form is 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 value that was 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) will catch 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).

The values t and nil for tag are special: a *catch whose tag is one of these values will catch throws to any 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.

*catch returns up to four values; trailing null values are not returned for reasons of microcode simplicity, but the values not returned will default to nil if they are received with the multiple-value or multiple-value-bind special forms. If the catch completes normally, the first value is the value of form and the second is nil. If a *throw occurs, the first value is the second argument to *throw, and the second value is the first argument to *throw, the tag thrown to. The third and fourth values are the third and fourth arguments to *unwind-stack (see page 56) if that was used in place of *throw; otherwise these values are nil. To summarize, the four values returned by *catch are the value, the tag, the active-frame-count, and the action.

Example

\begin{verbatim}
(*catch 'negative
  (mapcar (function (lambda (x)
    (cond ((minusp x)
      (*throw 'negative x))
    (t (f x)))))
  y))
\end{verbatim}

which returns a list of f of each element of y if they are all positive, otherwise the first negative member of y.
Note that *catch returns its own extra values, and so it does not propagate multiple values back from the last form.

```
catch-continuation tag throw-cont non-throw-cont body...     Special Form
catch-continuation-if cond-form tag throw-cont
    non-throw-cont body...                                 Special Form
```

The catch-continuation special form makes it convenient to pass back multiple values from the body, but still discriminate based on 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 returned by *catch are passed to the function throw-cont, and its values are returned.

If either of the continuations 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 will be always be called.

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

```
*throw tag value
```

*throw is used with *catch as a structured non-local exit mechanism.

(*throw tag x) throws the value of x back to the most recent *catch labelled with tag or t or nil. Other *catches are skipped over. Both x and tag are evaluated, unlike the Maclisp throw function.

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

See the description of *catch for further details.

```
catch form tag
throw form tag
```

Macro

Macro

catch and throw are provided only for Maclisp compatibility. (catch form tag) is the same as (*catch 'tag form), and (throw form tag) is the same as (*throw 'tag form). The forms of catch and throw without tags are not supported.
unwind-stack tag value active-frame-count action

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

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-protects receive control, but catch-alls 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 return 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 facility of Lisp creates a situation in which the above code won't work, however: if hairy-function should do a *throw to a *catch which is 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 error-handler 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 does a *throw which attempts to quit out of the evaluation of the unwind-protect, the (turn-off-water-faucet) form will be evaluated in between the time of the *throw and the time at which the *catch returns. 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

\[
\begin{align*}
\text{(unwind-protect protected-form} \\
\text{cleanup-form1} \\
\text{cleanup-form2} \\
\ldots 
\end{align*}
\]

protected-form is evaluated, and when it returns or when it attempts to quit 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.

\text{catch-all body...} \quad \text{Macro}

catch-all form is like (*catch some-tag form) except that it will catch a *throw to any tag at all. Since the tag thrown to is the second returned value, the caller of catch-all may continue throwing to that tag if he wants. The one thing that catch-all will 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.

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

### 4.4 Mapping

- map \( fcn \) &rest \( lists \)
- mapc \( fcn \) &rest \( lists \)
- maplist \( fcn \) &rest \( lists \)
- mapcar \( fcn \) &rest \( lists \)
- mapcons \( fcn \) &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 cdr, 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. \texttt{mapcar} will proceed down the lists \( x1, x2, \ldots, xn \) in parallel. The first argument to \( f \) will come 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 the function will be called repeatedly over and over. 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 \texttt{mapcar}. The function \texttt{circular-list} is useful for creating such lists; see page 66.

There are five other mapping functions besides \texttt{mapcar}. \texttt{maplist} is like \texttt{mapcar} except that the function is applied to the list and successive \texttt{cdr}'s of that list rather than to successive elements of the list. \texttt{map} and \texttt{mapc} are like \texttt{maplist} and \texttt{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. \texttt{mapcan} and \texttt{mapconcat} are like \texttt{mapcar} and \texttt{maplist} respectively, except that they combine the results of the function using \texttt{nconc} instead of \texttt{list}. That is, \texttt{mapconcat} could have been defined by

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

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

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

Often \( f \) will be 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.

<table>
<thead>
<tr>
<th>applies function to</th>
<th>successive sublists</th>
<th>successive elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>its own second argument</td>
<td>map</td>
<td>mapc</td>
</tr>
<tr>
<td>returns list of the function results</td>
<td>maplist</td>
<td>mapcar</td>
</tr>
<tr>
<td>nconc of the function results</td>
<td>mapcon</td>
<td>mapcan</td>
</tr>
</tbody>
</table>

There are also functions (mapatoms and mapatoms-all) for mapping over all symbols in certain packages. See the explanation of packages (chapter 24, page 506).

You can also do what the mapping functions do in a different way by using loop. See page 274.