22. LOOP for Black Belts

In Chapter 7 I briefly discussed the extended **`LOOP`** macro. As I mentioned then, **`LOOP`** provides what is essentially a special-purpose language just for writing iteration constructs.

This might seem like a lot of bother--inventing a whole language just for writing loops. But if you think about the ways loops are used in programs, it actually makes a fair bit of sense. Any program of any size at all will contain quite a number of loops. And while they won't all be the same, they won't all be unique either; patterns will emerge, particularly if you include the code immediately preceding and following the loops--patterns of how things are set up for the loop, patterns in what gets done in the loop proper, and patterns in what gets done after the loop. The **`LOOP`** language captures these patterns so you can express them directly.

The **`LOOP`** macro has a lot of parts--one of the main complaints of **`LOOP`**'s detractors is that it's too complex. In this chapter, I'll tackle **`LOOP`** head on, giving you a systematic tour of the various parts and how they fit together.

The Parts of a LOOP

You can do the following in a **`LOOP`**:

- Step variables numerically and over various data structures
- Collect, count, sum, minimize, and maximize values seen while looping
- Execute arbitrary Lisp expressions
- Decide when to terminate the loop
- Conditionally do any of these

Additionally, **`LOOP`** provides syntax for the following:

- Creating local variables for use within the loop
- Specifying arbitrary Lisp expressions to run before and after the loop proper

The basic structure of a **`LOOP`** is a set of clauses, each of which begins with a *loop keyword*.[1] How each clause is parsed by the **`LOOP`** macro depends on the keyword. Some of the main keywords, which you saw in Chapter 7, are `for`, `collecting`, `summing`, `counting`, `do`, and `finally`.

Iteration Control

Most of the so-called iteration control clauses start with the loop keyword `for`, or its synonym `as`,[2] followed by the name of a variable. What follows after the variable name depends on the type of `for` clause.

The subclauses of a `for` clause can iterate over the following:

- Ranges of numbers, up or down, by specified intervals
- The individual items of a list
- The cons cells that make up a list
- The elements of a vector, including subtypes such as strings and bit vectors
- The pairs of a hash table
- The symbols in a package
- The results of repeatedly evaluating a given form

A single loop can have multiple `for` clauses with each clause naming its own variable. When a loop has multiple `for` clauses, the loop terminates as soon as any `for` clause reaches its end condition. For instance, the following loop:

```
(loop
  for item in list
  for i from 1 to 10
  do (something))
```

will iterate at most ten times but may stop sooner if `list` contains fewer than ten items.

Counting Loops

Arithmetic iteration clauses control the number of times the loop body will be executed by stepping a variable over a range of numbers, executing the body once per step. These clauses consist of from one to three of the following *prepositional phrases* after the `for` (or `as`): the *from where* phrase, the *to where* phrase, and the *by how much* phrase.

The *from where* phrase specifies the initial value of the clause's variable. It consists of one of the prepositions `from`, `downfrom`, or `upfrom` followed by a form, which supplies the initial value (a number).

The *to where* phrase specifies a stopping point for the loop and consists of one of the prepositions `to`, `upto`, `below`, `downto`, or `above` followed by a form, which supplies the stopping point. With `upto` and `downto`, the loop body will be terminated (without executing the body again) when the variable passes the stopping point; with `below` and `above`, it stops one iteration earlier.The *by how much* phrase consists of the prepositions `by` and a form, which

must evaluate to a positive number. The variable will be stepped (up or down, as determined by the other phrases) by this amount on each iteration or by one if it's omitted.

You must specify at least one of these prepositional phrases. The defaults are to start at zero, increment the variable by one at each iteration, and go forever or, more likely, until some other clause terminates the loop. You can modify any or all of these defaults by adding the appropriate prepositional phrases. The only wrinkle is that if you want decremental stepping, there's no default *from where* value, so you must specify one with either `from` or `downfrom`. So, the following:

```
(loop for i upto 10 collect i)
```

collects the first eleven integers (zero to ten), but the behavior of this:

```
(loop for i downto -10 collect i)         ; wrong
```

is undefined. Instead, you need to write this:

```
(loop for i from 0 downto -10 collect i)
```

Also note that because **`LOOP`** is a macro, which runs at compile time, it has to be able to determine the direction to step the variable based solely on the prepositions--not the values of the forms, which may not be known until runtime. So, the following:

```
(loop for i from 10 to 20 ...)
```

works fine since the default is incremental stepping. But this:

```
(loop for i from 20 to 10 ...)
```

won't know to count down from twenty to ten. Worse yet, it won't give you an error--it will just not execute the loop since `i` is already greater than ten. Instead, you must write this:

```
(loop for i from 20 downto 10 ...)
```

or this:

```
(loop for i downfrom 20 to 10 ...)
```

Finally, if you just want a loop that repeats a certain number of times, you can replace a clause of the following form:

```
for i from 1 to number-form
```

with a `repeat` clause like this:

```
repeat number-form
```

These clauses are identical in effect except the `repeat` clause doesn't create an explicit loop variable.

Looping Over Collections and Packages

The `for` clauses for iterating over lists are much simpler than the arithmetic clauses. They support only two prepositional phrases, `in` and `on`.

A phrase of this form:

```
for var in list-form
```

steps *var* over all the elements of the list produced by evaluating *list-form*.

```
(loop for i in (list 10 20 30 40) collect i) ==> (10 20 30 40)
```

Occasionally this clause is supplemented with a `by` phrase, which specifies a function to use to move down the list. The default is **CDR** but can be any function that takes a list and returns a sublist. For instance, you could collect every other element of a list with a loop like this:

```
(loop for i in (list 10 20 30 40) by #'cddr collect i) ==> (10 30)
```

An `on` prepositional phrase is used to step *var* over the cons cells that make up a list.

```
(loop for x on (list 10 20 30) collect x) ==> ((10 20 30) (20 30) (30))
```

This phrase too can take a `by` preposition:

```
(loop for x on (list 10 20 30 40) by #'cddr collect x) ==> ((10 20 30 40) (30 40))
```

Looping over the elements of a vector (which includes strings and bit vectors) is similar to looping over the elements of a list except the preposition `across` is used instead of `in`.[3] For instance:

```
(loop for x across "abcd" collect x) ==> (#\a #\b #\c #\d)
```

Iterating over a hash table or package is slightly more complicated because hash tables and packages have different sets of values you might want to iterate over--the keys or values in a hash table and the different kinds of symbols in a package. Both kinds of iteration follow the same pattern. The basic pattern looks like this:

```
(loop for var being the things in hash-or-package ...)
```

For hash tables, the possible values for *things* are `hash-keys` and `hash-values`, which cause `var` to be bound to successive values of either the keys or the values of the hash table. The *hash-or-package* form is evaluated once to produce a value, which must be a hash table.

To iterate over a package, *things* can be `symbols`, `present-symbols`, and `external-symbols`, which cause *var* to be bound to each of the symbols accessible in a package, each of the symbols present in a package (in other words, interned or imported into that package), or each of the symbols that have been exported from the package. The *hash-or-package* form is evaluated to produce the name of a package, which is looked up as if by

FIND-PACKAGE or a package object. Synonyms are also available for parts of the `for` clause. In place of `the`, you can use `each`; you can use `of` instead of `in`; and you can write the *things* in the singular (for example, `hash-key` or `symbol`).

Finally, since you'll often want both the keys and the values when iterating over a hash table, the hash table clauses support a `using` subclause at the end of the hash table clause.

```
(loop for k being the hash-keys in h using (hash-value v) ...)
(loop for v being the hash-values in h using (hash-key k) ...)
```

Both of these loops will bind `k` to each key in the hash table and `v` to the corresponding value. Note that the first element of the `using` subclause must be in the singular form.[4]

Equals-Then Iteration

If none of the other `for` clauses supports exactly the form of variable stepping you need, you can take complete control over stepping with an *equals-then* clause. This clause is similar to the binding clauses in a **DO** loop but cast in a more Algolish syntax. The template is as follows:

```
(loop for var = initial-value-form [ then step-form ] ...)
```

As usual, *var* is the name of the variable to be stepped. Its initial value is obtained by evaluating *initial-value-form* once before the first iteration. In each subsequent iteration, *step-form* is evaluated, and its value becomes the new value of *var*. With no `then` part to the clause, the *initial-value-form* is reevaluated on each iteration to provide the new value. Note that this is different from a **DO** binding clause with no step form.

The *step-form* can refer to other loop variables, including variables created by other `for` clauses later in the loop. For instance:

```
(loop repeat 5
      for x = 0 then y
      for y = 1 then (+ x y)
      collect y) ==> (1 2 4 8 16)
```

However, note that each `for` clause is evaluated separately in the order it appears. So in the previous loop, on the second iteration `x` is set to the value of `y` before `y` changes (in other words, `1`). But `y` is then set to the sum of its old value (still `1`) and the new value of `x`. If the order of the `for` clauses is reversed, the results change.

```
(loop repeat 5
      for y = 1 then (+ x y)
      for x = 0 then y
      collect y) ==> (1 1 2 4 8)
```

Often, however, you'll want the step forms for multiple variables to be evaluated before any of the variables is given its new value (similar to how **DO** steps its variables). In that case, you can join multiple `for` clauses by replacing all but the first `for` with `and`. You saw this formulation

already in the **LOOP** version of the Fibonacci computation in Chapter 7. Here's another variant, based on the two previous examples:

```
(loop repeat 5
      for x = 0 then y
      and y = 1 then (+ x y)
      collect y) ==> (1 1 2 3 5)
```

Local Variables

While the main variables needed within a loop are usually declared implicitly in `for` clauses, sometimes you'll need auxiliary variables, which you can declare with `with` clauses.

```
with var [ = value-form ]
```

The name *var* becomes the name of a local variable that will cease to exist when the loop finishes. If the `with` clause contains an = `value-form` part, the variable will be initialized, before the first iteration of the loop, to the value of *value-form*.

Multiple `with` clauses can appear in a loop; each clause is evaluated independently in the order it appears and the value is assigned before proceeding to the next clause, allowing later variables to depend on the value of already declared variables. Mutually independent variables can be declared in one `with` clause with an `and` between each declaration.

Destructuring Variables

One handy feature of **LOOP** I haven't mentioned yet is the ability to destructure list values assigned to loop variables. This lets you take apart the value of lists that would otherwise be assigned to a loop variable, similar to the way **DESTRUCTURING-BIND** works but a bit less elaborate. Basically, you can replace any loop variable in a `for` or `with` clause with a tree of symbols, and the list value that would have been assigned to the simple variable will instead be destructured into variables named by the symbols in the tree. A simple example looks like this:

```
CL-USER> (loop for (a b) in '((1 2) (3 4) (5 6))
               do (format t "a: ~a; b: ~a~%" a b))
a: 1; b: 2
a: 3; b: 4
a: 5; b: 6
NIL
```

The tree can also include dotted lists, in which case the name after the dot acts like a **&rest** parameter, being bound to a list containing any remaining elements of the list. This is particularly handy with `for`/`on` loops since the value is always a list. For instance, this **LOOP** (which I used in Chapter 18 to emit a comma-delimited list):

```
(loop for cons on list
    do (format t "~a" (car cons))
    when (cdr cons) do (format t ", "))
```

could also be written like this:

```
(loop for (item . rest) on list
    do (format t "~a" item)
    when rest do (format t ", "))
```

If you want to ignore a value in the destructured list, you can use **NIL** in place of a variable name.

```
(loop for (a nil) in '((1 2) (3 4) (5 6)) collect a) ==> (1 3 5)
```

If the destructuring list contains more variables than there are values in the list, the extra variables are set to **NIL**, making all the variables essentially like **&optional** parameters. There isn't, however, any equivalent to **&key** parameters.

Value Accumulation

The value accumulation clauses are perhaps the most powerful part of **LOOP**. While the iteration control clauses provide a concise syntax for expressing the basic mechanics of looping, they aren't dramatically different from the equivalent mechanisms provided by **DO**, **DOLIST**, and **DOTIMES**.

The value accumulation clauses, on the other hand, provide a concise notation for a handful of common loop idioms having to do with accumulating values while looping. Each accumulation clause starts with a verb and follows this pattern:

```
verb form [ into var ]
```

Each time through the loop, an accumulation clause evaluates *form* and saves the value in a manner determined by the *verb*. With an `into` subclause, the value is saved into the variable named by *var*. The variable is local to the loop, as if it'd been declared in a `with` clause. With no `into` subclause, the accumulation clause instead accumulates a default value for the whole loop expression.

The possible verbs are `collect`, `append`, `nconc`, `count`, `sum`, `maximize`, and `minimize`. Also available as synonyms are the present participle forms: `collecting`, `appending`, `nconcing`, `counting`, `summing`, `maximizing`, and `minimizing`.

A `collect` clause builds up a list containing all the values of *form* in the order they're seen. This is a particularly useful construct because the code you'd have to write to collect a list in order as efficiently as **LOOP** does is more painful than you'd normally write by hand.[5] Related to `collect` are the verbs `append` and `nconc`. These verbs also accumulate values into a list, but they join the values, which must be lists, into a single list as if by the functions **APPEND** or **NCONC**. [6]

The remaining accumulation clauses are used to accumulate numeric values. The verb `count` counts the number of times *form* is true, `sum` collects a running total of the values of *form*, `maximize` collects the largest value seen for *form*, and `minimize` collects the smallest. For instance, suppose you define a variable `*random*` that contains a list of random numbers.

```
(defparameter *random* (loop repeat 100 collect (random 10000)))
```

Then the following loop will return a list containing various summary information about the numbers:

```
(loop for i in *random*
   counting (evenp i) into evens
   counting (oddp i) into odds
   summing i into total
   maximizing i into max
   minimizing i into min
   finally (return (list min max total evens odds)))
```

Unconditional Execution

As useful as the value accumulation constructs are, **`LOOP`** wouldn't be a very good general-purpose iteration facility if there wasn't a way to execute arbitrary Lisp code in the loop body.

The simplest way to execute arbitrary code within a loop body is with a `do` clause. Compared to the clauses I've described so far, with their prepositions and subclauses, `do` is a model of Yodaesque simplicity.[7] A `do` clause consists of the word `do` (or `doing`) followed by one or more Lisp forms that are all evaluated when the `do` clause is. The `do` clause ends at the closing parenthesis of the loop or the next loop keyword.

For instance, to print the numbers from one to ten, you could write this:

```
(loop for i from 1 to 10 do (print i))
```

Another, more dramatic, form of immediate execution is a `return` clause. This clause consists of the word `return` followed by a single Lisp form, which is evaluated, with the resulting value immediately returned as the value of the loop.

You can also break out of a loop in a `do` clause using any of Lisp's normal control flow operators, such as **`RETURN`** and **`RETURN-FROM`**. Note that a `return` clause always returns from the immediately enclosing **`LOOP`** expression, while a **`RETURN`** or **`RETURN-FROM`** in a `do` clause can return from any enclosing expression. For instance, compare the following:

```
(block outer
  (loop for i from 0 return 100) ; 100 returned from LOOP
  (print "This will print")
  200) ==> 200
```

to this:

```
(block outer
  (loop for i from 0 do (return-from outer 100)) ; 100 returned from BLOCK
  (print "This won't print")
  200) ==> 100
```

The `do` and `return` clauses are collectively called the *unconditional execution* clauses.

Conditional Execution

Because a `do` clause can contain arbitrary Lisp forms, you can use any Lisp expressions you want, including control constructs such as **`IF`** and **`WHEN`**. So, the following is one way to write a loop that prints only the even numbers between one and ten:

```
(loop for i from 1 to 10 do (when (evenp i) (print i)))
```

However, sometimes you'll want conditional control at the level of loop clauses. For instance, suppose you wanted to sum only the even numbers between one and ten using a `summing` clause. You couldn't write such a loop with a `do` clause because there'd be no way to "call" the `sum i` in the middle of a regular Lisp form. In cases like this, you need to use one of **`LOOP`**'s own conditional expressions like this:

```
(loop for i from 1 to 10 when (evenp i) sum i) ==> 30
```

`LOOP` provides three conditional constructs, and they all follow this basic pattern:

```
conditional test-form loop-clause
```

The *conditional* can be `if`, `when`, or `unless`. The *test-form* is any regular Lisp form, and *loop-clause* can be a value accumulation clause (`count`, `collect`, *and so on), an unconditional execution* clause, or another conditional execution clause. Multiple loop clauses can be attached to a single conditional by joining them with `and`.

As an extra bit of syntactic sugar, within the first loop clause, after the test form, you can use the variable `it` to refer to the value returned by the test form. For instance, the following loop collects the non-**`NIL`** values found in `some-hash` when looking up the keys in `some-list`:

```
(loop for key in some-list when (gethash key some-hash) collect it)
```

A conditional clause is executed each time through the loop. An `if` or `when` clause executes its *loop-clause* if *test-form* evaluates to true. An `unless` reverses the test, executing *loop-clause* only when *test-form* is **`NIL`**. Unlike their Common Lisp namesakes, **`LOOP`**'s `if` and `when` are merely synonyms--there's no difference in their behavior.

All three conditional clauses can also take an `else` branch, which is followed by another loop clause or multiple clauses joined by `and`. When conditional clauses are nested, the set of clauses connected to an inner conditional clause can be closed with the word `end`. The `end` is optional

when not needed to disambiguate a nested conditional--the end of a conditional clause will be inferred from the end of the loop or the start of another clause not joined by `and`.

The following rather silly loop demonstrates the various forms of **`LOOP`** conditionals. The `update-analysis` function will be called each time through the loop with the latest values of the various variables accumulated by the clauses within the conditionals.

```
(loop for i from 1 to 100
      if (evenp i)
        minimize i into min-even and
        maximize i into max-even and
        unless (zerop (mod i 4))
          sum i into even-not-fours-total
        end
        and sum i into even-total
      else
        minimize i into min-odd and
        maximize i into max-odd and
        when (zerop (mod i 5))
          sum i into fives-total
        end
        and sum i into odd-total
      do (update-analysis min-even
                          max-even
                          min-odd
                          max-odd
                          even-total
                          odd-total
                          fives-total
                          even-not-fours-total))
```

Setting Up and Tearing Down

One of the key insights the designers of the **`LOOP`** language had about actual loops "in the wild" is that the loop proper is often preceded by a bit of code to set things up and then followed by some more code that does something with the values computed by the loop. A trivial example, in Perl,[8] might look like this:

```
my $evens_sum = 0;
my $odds_sum  = 0;
foreach my $i (@list_of_numbers) {
  if ($i % 2) {
    $odds_sum += $i;
  } else {
    $evens_sum += $i;
  }
}
if ($evens_sum > $odds_sum) {
  print "Sum of evens greater\n";
} else {
  print "Sum of odds greater\n";
}
```

The loop proper in this code is the `foreach` statement. But the `foreach` loop doesn't stand on its own: the code in the loop body refers to variables declared in the two lines before the loop.[9] And the work the loop does is all for naught without the `if` statement after the loop that actually reports the results. In Common Lisp, of course, the **`LOOP`** construct is an expression that returns

a value, so there's even more often a need to do something after the loop proper, namely, generate the return value.

So, said the **LOOP** designers, let's give a way to include the code that's really part of the loop in the loop itself. Thus, **LOOP** provides two keywords, `initially` and `finally`, that introduce code to be run outside the loop's main body.

After the `initially` or `finally`, these clauses consist of all the Lisp forms up to the start of the next loop clause or the end of the loop. All the `initially` forms are combined into a single *prologue*, which runs once, immediately after all the local loop variables are initialized and before the body of the loop. The `finally` forms are similarly combined into a *epilogue* to be run after the last iteration of the loop body. Both the prologue and epilogue code can refer to local loop variables.

The prologue is always run, even if the loop body iterates zero times. The loop can return without running the epilogue if any of the following happens:

- A `return` clause executes.
- **RETURN** , **RETURN-FROM**, or another transfer of control construct is called from within a Lisp form within the body.[10]
- The loop is terminated by an `always`, `never`, or `thereis` clause, as I'll discuss in the next section.

Within the epilogue code, **RETURN** or **RETURN-FROM** can be used to explicitly provide a return value for the loop. Such an explicit return value will take precedence over any value that might otherwise be provided by an accumulation or termination test clause.

To allow **RETURN-FROM** to be used to return from a specific loop (useful when nesting **LOOP** expressions), you can name a **LOOP** with the loop keyword `named`. If a `named` clause appears in a loop, it must be the first clause. For a simple example, assume `lists` is a list of lists and you want to find an item that matches some criteria in one of those nested lists. You could find it with a pair of nested loops like this:

```
(loop named outer for list in lists do
     (loop for item in list do
          (if (what-i-am-looking-for-p item)
            (return-from outer item))))
```

Termination Tests

While the `for` and `repeat` clauses provide the basic infrastructure for controlling the number of iterations, sometimes you'll need to break out of a loop early. You've already seen how a `return` clause or a **RETURN** or **RETURN-FROM** within a `do` clause can immediately terminate the loop; but just as there are common patterns for accumulating values, there are also common patterns for deciding when it's time to bail on a loop. These patterns are supported in **LOOP** by

the termination clauses, `while`, `until`, `always`, `never`, and `thereis`. They all follow the same pattern.

```
loop-keyword test-form
```

All five evaluate *test-form* each time through the iteration and decide, based on the resulting value, whether to terminate the loop. They differ in what happens after they terminate the loop--if they do--and how they decide.

The loop keywords `while` and `until` introduce the "mild" termination clauses. When they decide to terminate the loop, control passes to the epilogue, skipping the rest of the loop body. The epilogue can then return a value or do whatever it wants to finish the loop. A `while` clause terminates the loop the first time the test form is false; `until`, conversely, stops it the first time the test form is true.

Another form of mild termination is provided by the **`LOOP-FINISH`** macro. This is a regular Lisp form, not a loop clause, so it can be used anywhere within the Lisp forms of a `do` clause. It also causes an immediate jump to the loop epilogue. It can be useful when the decision to break out of the loop can't be easily condensed into a single form that can be used with a `while` or `until` clause.

The other three clauses--`always`, `never`, and `thereis`--terminate the loop with extreme prejudice; they immediately return from the loop, skipping not only any subsequent loop clauses but also the epilogue. They also provide a default value for the loop even when they don't cause the loop to terminate. However, if the loop is *not* terminated by one of these termination tests, the epilogue is run and can return a value other than the default provided by the termination clauses.

Because these clauses provide their own return values, they can't be combined with accumulation clauses unless the accumulation clause has an `into` subclause. The compiler (or interpreter) should signal an error at compile time if they are.The `always` and `never` clauses return only boolean values, so they're most useful when you need to use a loop expression as part of a predicate. You can use `always` to check that the test form is true on every iteration of the loop. Conversely, `never` tests that the test form evaluates to **`NIL`** on every iteration. If the test form fails (returning **`NIL`** in an `always` clause or non-**`NIL`** in a `never` clause), the loop is immediately terminated, returning **`NIL`**. If the loop runs to completion, the default value of **`T`** is provided.

For instance, if you want to test that all the numbers in a list, `numbers`, are even, you can write this:

```
(if (loop for n in numbers always (evenp n))
    (print "All numbers even."))
```

Equivalently you could write the following:

```
(if (loop for n in numbers never (oddp n))
    (print "All numbers even."))
```

A `thereis` clause is used to test whether the test form is *ever* true. As soon as the test form returns a non-**`NIL`** value, the loop is terminated, returning that value. If the loop runs to completion, the `thereis` clause provides a default return value of **`NIL`**.

```
(loop for char across "abc123" thereis (digit-char-p char)) ==> 1

(loop for char across "abcdef" thereis (digit-char-p char)) ==> NIL
```

Putting It All Together

Now you've seen all the main features of the **`LOOP`** facility. You can combine any of the clauses I've discussed as long as you abide by the following rules:

- The `named` clause, if any, must be the first clause.
- After the `named` clause come all the `initially`, `with`, `for`, and `repeat` clauses.
- Then comes the body clauses: conditional and unconditional execution, accumulation, and termination test.[11]
- End with any `finally` clauses.

The **`LOOP`** macro will expand into code that performs the following actions:

- Initializes all local loop variables as declared with `with` or `for` clauses as well as those implicitly created by accumulation clauses. The initial value forms are evaluated in the order the clauses appear in the loop.
- Execute the forms provided by any `initially` clauses--the prologue--in the order they appear in the loop.
- Iterate, executing the body of the loop as described in the next paragraph.
- Execute the forms provided by any `finally` clauses--the epilogue--in the order they appear in the loop.

While the loop is iterating, the body is executed by first stepping any iteration control variables and then executing any conditional or unconditional execution, accumulation, or termination test clauses in the order they appear in the loop code. If any of the clauses in the loop body terminate the loop, the rest of the body is skipped and the loop returns, possibly after running the epilogue.

And that's pretty much all there is to it.[12] You'll use **`LOOP`** fairly often in the code later in this book, so it's worth having some knowledge of it. Beyond that, it's up to you how much you use it.

And with that, you're ready to dive into the practical chapters that make up the rest of the book--up first, writing a spam filter.

---

[1]The term *loop keyword* is a bit unfortunate, as loop keywords aren't keywords in the normal sense of being symbols in the `KEYWORD` package. In fact, any symbol, from any package, with the appropriate name will do; the **`LOOP`** macro cares only about their names. Typically, though, they're written with no package qualifier and are thus read (and interned as necessary) in the current package.

[2]Because one of the goals of **`LOOP`** is to allow loop expressions to be written with a quasi-English syntax, many of the keywords have synonyms that are treated the same by **`LOOP`** but allow some freedom to express things in slightly more idiomatic English for different contexts.

[3]You may wonder why **`LOOP`** can't figure out whether it's looping over a list or a vector without needing different prepositions. This is another consequence of **`LOOP`** being a macro: the value of the list or vector won't be known until runtime, but **`LOOP`**, as a macro, has to generate code at compile time. And **`LOOP`**'s designers wanted it to generate extremely efficient code. To be able to generate efficient code for looping across, say, a vector, it needs to know at compile time that the value will be a vector at runtime--thus, the different prepositions are needed.

[4]Don't ask me why **`LOOP`**'s authors chickened out on the no-parentheses style for the **`using`** subclause.

[5]The trick is to keep ahold of the tail of the list and add new cons cells by **`SETF`**ing the **`CDR`** of the tail. A handwritten equivalent of the code generated by `(loop for i upto 10 collect i)` would look like this:

```
(do ((list nil) (tail nil) (i 0 (1+ i)))
    ((> i 10) list)
  (let ((new (cons i nil)))
    (if (null list)
        (setf list new)
        (setf (cdr tail) new))
    (setf tail new)))
```

Of course you'll rarely, if ever, write code like that. You'll use either **`LOOP`** or (if, for some reason, you don't want to use **`LOOP`**) the standard **`PUSH`**/**`NREVERSE`** idiom for collecting values.

[6]Recall that **`NCONC`** is the destructive version of **`APPEND`**--it's safe to use an `nconc` clause only if the values you're collecting are fresh lists that don't share any structure with other lists. For instance, this is safe:

```
(loop for i upto 3 nconc (list i i)) ==> (0 0 1 1 2 2 3 3)
```

But this will get you into trouble:

```
(loop for i on (list 1 2 3) nconc i) ==> undefined
```

The later will most likely get into an infinite loop as the various parts of the list produced by (list 1 2 3) are destructively modified to point to each other. But even that's not guaranteed--the behavior is simply undefined.

[7]"No! Try not. Do . . . or do not. There is no try." -- Yoda, *The Empire Strikes Back*

[8]I'm not picking on Perl here--this example would look pretty much the same in any language that bases its syntax on C's.

[9]Perl would let you get away with not declaring those variables if your program didn't `use strict`. But you should *always* `use strict` in Perl. The equivalent code in Python, Java, or C would always require the variables to be declared.

[10]You can cause a loop to finish normally, running the epilogue, from Lisp code executed as part of the loop body with the local macro **`LOOP-FINISH`**.

[11]Some Common Lisp implementations will let you get away with mixing body clauses and `for` clauses, but that's strictly undefined, and some implementations will reject such loops.

[12]The one aspect of **`LOOP`** I haven't touched on at all is the syntax for declaring the types of loop variables. Of course, I haven't discussed type declarations outside of **`LOOP`** either. I'll cover the general topic a bit in Chapter 32. For information on how they work with **`LOOP`**, consult your favorite Common Lisp reference.