18. A Few FORMAT Recipes

Common Lisp's **FORMAT** function is--along with the extended **LOOP** macro--one of the two Common Lisp features that inspires a strong emotional response in a lot of Common Lisp users. Some love it; others hate it.[1]

FORMAT's fans love it for its great power and concision, while its detractors hate it because of the potential for misuse and its opacity. Complex **FORMAT** control strings sometimes bear a suspicious resemblance to line noise, but **FORMAT** remains popular with Common Lispers who like to be able to generate little bits of human-readable output without having to clutter their code with lots of output-generating code. While **FORMAT**'s control strings can be cryptic, at least a single **FORMAT** expression doesn't clutter things up too badly. For instance, suppose you want to print the values in a list delimited with commas. You could write this:

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

That's not too bad, but anyone reading this code has to mentally parse it just to figure out that all it's doing is printing the contents of `list` to standard output. On the other hand, you can tell at a glance that the following expression is printing `list`, in some form, to standard output:

```
(format t "~{~a~^, ~}" list)
```

If you care exactly what form the output will take, then you'll have to examine the control string, but if all you want is a first-order approximation of what this line of code is doing, that's immediately available.

At any rate, you should have at least a reading knowledge of **FORMAT**, and it's worth getting a sense of what it can do before you affiliate yourself with the pro- or anti-**FORMAT** camp. It's also important to understand at least the basics of **FORMAT** because other standard functions, such as the condition-signaling functions discussed in the next chapter, use **FORMAT**-style control strings to generate output.

To further complicate matters, **FORMAT** supports three quite different kinds of formatting: printing tables of data, *pretty-printing* s-expressions, and generating human-readable messages with interpolated values. Printing tables of data as text is a bit pass� these days; it's one of those reminders that Lisp is nearly as old as FORTRAN. In fact, several of the directives you can use to print floating-point values in fixed-width fields were based quite directly on FORTRAN *edit descriptors*, which are used in FORTRAN to read and print columns of data arranged in fixed-width fields. However, using Common Lisp as a FORTRAN replacement is beyond the scope of this book, so I won't discuss those aspects of **FORMAT**.

Pretty-printing is likewise beyond the scope of this book--not because it's pass� but just because it's too big a topic. Briefly, the Common Lisp pretty printer is a customizable system for printing block-structured data such as--but not limited to--s-expressions while varying indentation and dynamically adding line breaks as needed. It's a great thing when you need it, but it's not often needed in day-to-day programming.[2]

Instead, I'll focus on the parts of **FORMAT** you can use to generate human-readable strings with interpolated values. Even limiting the scope in that way, there's still a fair bit to cover. You shouldn't feel obliged to remember every detail described in this chapter. You can get quite far with just a few **FORMAT** idioms. I'll describe the most important features of **FORMAT** first; it's up to you how much of a **FORMAT** wizard you want to become.

The FORMAT Function

As you've seen in previous chapters, the **FORMAT** function takes two required arguments: a destination for its output and a control string that contains literal text and embedded *directives*. Any additional arguments provide the values used by the directives in the control string that interpolate values into the output. I'll refer to these arguments as *format arguments*.

The first argument to **FORMAT**, the destination for the output, can be **T**, **NIL**, a stream, or a string with a fill pointer. **T** is shorthand for the stream ***STANDARD-OUTPUT***, while **NIL** causes **FORMAT** to generate its output to a string, which it then returns.[3] If the destination is a stream, the output is written to the stream. And if the destination is a string with a fill pointer, the formatted output is added to the end of the string and the fill pointer is adjusted appropriately. Except when the destination is **NIL** and it returns a string, **FORMAT** returns **NIL**.

The second argument, the control string, is, in essence, a program in the **FORMAT** language. The **FORMAT** language isn't Lispy at all--its basic syntax is based on characters, not s-expressions, and it's optimized for compactness rather than easy comprehension. This is why a complex **FORMAT** control string can end up looking like line noise.

Most of **FORMAT**'s directives simply interpolate an argument into the output in one form or another. Some directives, such as `~%`, which causes **FORMAT** to emit a newline, don't consume any arguments. And others, as you'll see, can consume more than one argument. One directive even allows you to jump around in the list of arguments in order to process the same argument more than once or to skip certain arguments in certain situations. But before I discuss specific directives, let's look at the general syntax of a directive.

FORMAT Directives

All directives start with a tilde (~) and end with a single character that identifies the directive. You can write the character in either upper- or lowercase. Some directives take *prefix parameters*, which are written immediately following the tilde, separated by commas, and used to control things such as how many digits to print after the decimal point when printing a floating-point number. For example, the `~$` directive, one of the directives used to print floating-point values, by default prints two digits following the decimal point.

```
CL-USER> (format t "~$" pi)
3.14
NIL
```

However, with a prefix parameter, you can specify that it should print its argument to, say, five decimal places like this:

```
CL-USER> (format t "~5$" pi)
3.14159
NIL
```

The values of prefix parameters are either numbers, written in decimal, or characters, written as a single quote followed by the desired character. The value of a prefix parameter can also be derived from the format arguments in two ways: A prefix parameter of `v` causes **`FORMAT`** to consume one format argument and use its value for the prefix parameter. And a prefix parameter of # will be evaluated as the number of remaining format arguments. For example:

```
CL-USER> (format t "~v$" 3 pi)
3.142
NIL
CL-USER> (format t "~#$" pi)
3.1
NIL
```

I'll give some more realistic examples of how you can use the `#` argument in the section "Conditional Formatting."

You can also omit prefix parameters altogether. However, if you want to specify one parameter but not the ones before it, you must include a comma for each unspecified parameter. For instance, the `~F` directive, another directive for printing floating-point values, also takes a parameter to control the number of decimal places to print, but it's the second parameter rather than the first. If you want to use `~F` to print a number to five decimal places, you can write this:

```
CL-USER> (format t "~,5f" pi)
3.14159
NIL
```

You can also modify the behavior of some directives with colon and at-sign *modifiers*, which are placed after any prefix parameters and before the directive's identifying character. These modifiers change the behavior of the directive in small ways. For instance, with a colon modifier, the `~D` directive used to output integers in decimal emits the number with commas separating every three digits, while the at-sign modifier causes `~D` to include a plus sign when the number is positive.

```
CL-USER> (format t "~d" 1000000)
1000000
NIL
CL-USER> (format t "~:d" 1000000)
1,000,000
NIL
CL-USER> (format t "~@d" 1000000)
+1000000
NIL
```

When it makes sense, you can combine the colon and at-sign modifiers to get both modifications.

```
CL-USER> (format t "~:@d" 1000000)
+1,000,000
NIL
```

In directives where the two modified behaviors can't be meaningfully combined, using both modifiers is either undefined or given a third meaning.

Basic Formatting

Now you're ready to look at specific directives. I'll start with several of the most commonly used directives, including some you've seen in previous chapters.

The most general-purpose directive is `~A`, which consumes one format argument of any type and outputs it in *aesthetic* (human-readable) form. For example, strings are output without quotation marks or escape characters, and numbers are output in a natural way for the type of number. If you just want to emit a value for human consumption, this directive is your best bet.

```
(format nil "The value is: ~a" 10)           ==> "The value is: 10"
(format nil "The value is: ~a" "foo")        ==> "The value is: foo"
(format nil "The value is: ~a" (list 1 2 3)) ==> "The value is: (1 2 3)"
```

A closely related directive, `~S`, likewise consumes one format argument of any type and outputs it. However, `~S` tries to generate output that can be read back in with **`READ`**. Thus, strings will be enclosed in quotation marks, symbols will be package-qualified when necessary, and so on. Objects that don't have a **`READ`**able representation are printed with the unreadable object syntax, #<>. With a colon modifier, both the `~A` and `~S` directives emit **`NIL`** as `()` rather than **`NIL`**. Both the `~A` and `~S` directives also take up to four prefix parameters, which can be used to control whether padding is added after (or before with the at-sign modifier) the value, but those parameters are only really useful for generating tabular data.

The other two most frequently used directives are `~%`, which emits a newline, and `~&`, which emits a *fresh line*. The difference between the two is that `~%` always emits a newline, while `~&` emits one only if it's not already at the beginning of a line. This is handy when writing loosely coupled functions that each generate a piece of output and that need to be combined in different ways. For instance, if one function generates output that ends with a newline (`~%`) and another function generates some output that starts with a fresh line (`~&`), you don't have to worry about getting an extra blank line if you call them one after the other. Both of these directives can take a single prefix parameter that specifies the number of newlines to emit. The `~%` directive will simply emit that many newline characters, while the `~&` directive will emit either *n* - 1 or *n* newlines, depending on whether it starts at the beginning of a line.

Less frequently used is the related `~~` directive, which causes **`FORMAT`** to emit a literal tilde. Like the `~%` and `~&` directives, it can be parameterized with a number that controls how many tildes to emit.

Character and Integer Directives

In addition to the general-purpose directives, `~A` and `~S`, **`FORMAT`** supports several directives that can be used to emit values of specific types in particular ways. One of the simplest of these is the `~C` directive, which is used to emit characters. It takes no prefix arguments but can be modified with the colon and at-sign modifiers. Unmodified, its behavior is no different from `~A` except that it works only with characters. The modified versions are more useful. With a colon modifier, `~:C` outputs *nonprinting* characters such as space, tab, and newline by name. This is useful if you want to emit a message to the user about some character. For instance, the following:

```
(format t "Syntax error. Unexpected character: ~:c" char)
```

can emit messages like this:

```
Syntax error. Unexpected character: a
```

but also like the following:

```
Syntax error. Unexpected character: Space
```

With the at-sign modifier, `~@C` will emit the character in Lisp's literal character syntax.

```
CL-USER> (format t "~@c~%" #\a)
#\a
NIL
```

With both the colon and at-sign modifiers, the `~C` directive can print extra information about how to enter the character at the keyboard if it requires special key combinations. For instance, on the Macintosh, in certain applications you can enter a null character (character code 0 in ASCII or in any ASCII superset such as ISO-8859-1 or Unicode) by pressing the Control key and typing @. In OpenMCL, if you print the null character with the `~:C` directive, it tells you this:

```
(format nil "~:@c" (code-char 0)) ==> "^@ (Control @)"
```

However, not all Lisps implement this aspect of the `~C` directive. And even if they do, it may or may not be accurate--for instance, if you're running OpenMCL in SLIME, the `C-@` key chord is intercepted by Emacs, invoking `set-mark-command`.[4]

Format directives dedicated to emitting numbers are another important category. While you can use the `~A` and `~S` directives to emit numbers, if you want fine control over how they're printed, you need to use one of the number-specific directives. The numeric directives can be divided into two subcategories: directives for formatting integer values and directives for formatting floating-point values.

Five closely related directives format integer values: `~D`, `~X`, `~O`, `~B`, and `~R`. The most frequently used is the `~D` directive, which outputs integers in base 10.

```
(format nil "~d" 1000000) ==> "1000000"
```

As I mentioned previously, with a colon modifier it adds commas.

```
(format nil "~:d" 1000000) ==> "1,000,000"
```

And with an at-sign modifier, it always prints a sign.

```
(format nil "~@d" 1000000) ==> "+1000000"
```

And the two modifiers can be combined.

```
(format nil "~:@d" 1000000) ==> "+1,000,000"
```

The first prefix parameter can specify a minimum width for the output, and the second parameter can specify a padding character to use. The default padding character is space, and padding is always inserted before the number itself.

```
(format nil "~12d" 1000000)     ==> "     1000000"
(format nil "~12,'0d" 1000000) ==> "000001000000"
```

These parameters are handy for formatting things such as dates in a fixed-width format.

```
(format nil "~4,'0d-~2,'0d-~2,'0d" 2005 6 10) ==> "2005-06-10"
```

The third and fourth parameters are used in conjunction with the colon modifier: the third parameter specifies the character to use as the separator between groups and digits, and the fourth parameter specifies the number of digits per group. These parameters default to a comma and the number 3. Thus, you can use the directive `~:D` without parameters to output large integers in standard format for the United States but can change the comma to a period and the grouping from 3 to 4 with `~,,'.,4D`.

```
(format nil "~:d" 100000000)       ==> "100,000,000"
(format nil "~,,'.,4:d" 100000000) ==> "1.0000.0000"
```

Note that you must use commas to hold the places of the unspecified width and padding character parameters, allowing them to keep their default values.

The `~X`, `~O`, and `~B` directives work just like the `~D` directive except they emit numbers in hexadecimal (base 16), octal (base 8), and binary (base 2).

```
(format nil "~x" 1000000) ==> "f4240"
(format nil "~o" 1000000) ==> "3641100"
(format nil "~b" 1000000) ==> "11110100001001000000"
```

Finally, the `~R` directive is the general *radix* directive. Its first parameter is a number between 2 and 36 (inclusive) that indicates what base to use. The remaining parameters are the same as the four parameters accepted by the `~D`, `~X`, `~O`, and `~B` directives, and the colon and at-sign modifiers modify its behavior in the same way. The `~R` directive also has some special behavior when used with no prefix parameters, which I'll discuss in the section "English-Language Directives."

Floating-Point Directives

Four directives format floating-point values: `~F`, `~E`, `~G`, and `~$`. The first three of these are the directives based on FORTRAN's edit descriptors. I'll skip most of the details of those directives since they mostly have to do with formatting floating-point values for use in tabular form. However, you can use the `~F`, `~E`, and `~$` directives to interpolate floating-point values into text. The `~G`, or *general,* floating-point directive, on the other hand, combines aspects of the `~F` and `~E` directives in a way that only really makes sense for generating tabular output.

The `~F` directive emits its argument, which should be a number,[5] in decimal format, possibly controlling the number of digits after the decimal point. The `~F` directive is, however, allowed to use computerized scientific notation if the number is sufficiently large or small. The `~E` directive, on the other hand, always emits numbers in computerized scientific notation. Both of these directives take a number of prefix parameters, but you need to worry only about the second, which controls the number of digits to print after the decimal point.

```
(format nil "~f" pi)   ==> "3.141592653589793d0"
(format nil "~,4f" pi) ==> "3.1416"
(format nil "~e" pi)   ==> "3.141592653589793d+0"
(format nil "~,4e" pi) ==> "3.1416d+0"
```

The `~$`, or monetary, directive is similar to `~F` but a bit simpler. As its name suggests, it's intended for emitting monetary units. With no parameters, it's basically equivalent to `~,2F`. To modify the number of digits printed after the decimal point, you use the *first* parameter, while the second parameter controls the minimum number of digits to print before the decimal point.

```
(format nil "~$" pi)    ==> "3.14"
(format nil "~2,4$" pi) ==> "0003.14"
```

All three directives, `~F`, `~E`, and `~$`, can be made to always print a sign, plus or minus, with the at-sign modifier.[6]

English-Language Directives

Some of the handiest **FORMAT** directives for generating human-readable messages are the ones for emitting English text. These directives allow you to emit numbers as English words, to emit plural markers based on the value of a format argument, and to apply case conversions to sections of **FORMAT**'s output.

The `~R` directive, which I discussed in "Character and Integer Directives," when used with no base specified, prints numbers as English words or Roman numerals. When used with no prefix parameter and no modifiers, it emits the number in words as a cardinal number.

```
(format nil "~r" 1234) ==> "one thousand two hundred thirty-four"
```

With the colon modifier, it emits the number as an ordinal.

```
(format nil "~:r" 1234) ==> "one thousand two hundred thirty-fourth"
```

And with an at-sign modifier, it emits the number as a Roman numeral; with both an at-sign and a colon, it emits "old-style" Roman numerals in which fours and nines are written as IIII and VIIII instead of IV and IX.

```
(format nil "~@r" 1234)  ==> "MCCXXXIV"
(format nil "~:@r" 1234) ==> "MCCXXXIIII"
```

For numbers too large to be represented in the given form, `~R` behaves like `~D`.

To help you generate messages with words properly pluralized, **FORMAT** provides the `~P` directive, which simply emits an *s* unless the corresponding argument is `1`.

```
(format nil "file~p" 1)  ==> "file"
(format nil "file~p" 10) ==> "files"
(format nil "file~p" 0)  ==> "files"
```

Typically, however, you'll use `~P` with the colon modifier, which causes it to reprocess the previous format argument.

```
(format nil "~r file~:p" 1)  ==> "one file"
(format nil "~r file~:p" 10) ==> "ten files"
(format nil "~r file~:p" 0)  ==> "zero files"
```

With the at-sign modifier, which can be combined with the colon modifier, `~P` emits either *y* or *ies*.

```
(format nil "~r famil~:@p" 1)  ==> "one family"
(format nil "~r famil~:@p" 10) ==> "ten families"
(format nil "~r famil~:@p" 0)  ==> "zero families"
```

Obviously, `~P` can't solve all pluralization problems and is no help for generating messages in other languages, but it's handy for the cases it does handle. And the `~[` directive, which I'll discuss in a moment, gives you a more flexible way to conditionalize parts of **FORMAT**'s output.

The last directive for dealing with emitting English text is `~(`, which allows you to control the case of text in the output. Each `~(` is paired with a `~)`, and all the output generated by the portion of the control string between the two markers will be converted to all lowercase.

```
(format nil "~(~a~)" "FOO") ==> "foo"
(format nil "~(~@r~)" 124)  ==> "cxxiv"
```

You can modify `~(` with an at sign to make it capitalize the first word in a section of text, with a colon to make it to capitalize all words, and with both modifiers to convert all text to uppercase. (A *word* for the purpose of this directive is a sequence of alphanumeric characters delimited by nonalphanumeric characters or the ends of the text.)

```
(format nil "~(~a~)" "tHe Quick BROWN foX")   ==> "the quick brown fox"
(format nil "~@(~a~)" "tHe Quick BROWN foX")  ==> "The quick brown fox"
(format nil "~:(~a~)" "tHe Quick BROWN foX")  ==> "The Quick Brown Fox"
(format nil "~:@(~a~)" "tHe Quick BROWN foX") ==> "THE QUICK BROWN FOX"
```

Conditional Formatting

In addition to directives that interpolate arguments and modify other output, **`FORMAT`** provides several directives that implement simple control constructs within the control string. One of these, which you used in Chapter 9, is the *conditional* directive `~[`**.** This directive is closed by a corresponding `~]`, and in between are a number of clauses separated by `~;`. The job of the `~[` directive is to pick one of the clauses, which is then processed by **`FORMAT`**. With no modifiers or parameters, the clause is selected by numeric index; the `~[` directive consumes a format argument, which should be a number, and takes the *nth* (zero-based) clause where *N* is the value of the argument.

```
(format nil "~[cero~;uno~;dos~]" 0) ==> "cero"
(format nil "~[cero~;uno~;dos~]" 1) ==> "uno"
(format nil "~[cero~;uno~;dos~]" 2) ==> "dos"
```

If the value of the argument is greater than the number of clauses, nothing is printed.

```
(format nil "~[cero~;uno~;dos~]" 3) ==> ""
```

However, if the last clause separator is `~:;` instead of `~;`, then the last clause serves as a default clause.

```
(format nil "~[cero~;uno~;dos~:;mucho~]" 3)   ==> "mucho"
(format nil "~[cero~;uno~;dos~:;mucho~]" 100) ==> "mucho"
```

It's also possible to specify the clause to be selected using a prefix parameter. While it'd be silly to use a literal value in the control string, recall that # used as a prefix parameter means the number of arguments remaining to be processed. Thus, you can define a format string such as the following:

```
(defparameter *list-etc*
  "~#[NONE~;~a~;~a and ~a~:;~a, ~a~]~#[~; and ~a~:;, ~a, etc~].")
```

and then use it like this:

```
(format nil *list-etc*)                  ==> "NONE."
(format nil *list-etc* 'a)               ==> "A."
(format nil *list-etc* 'a 'b)            ==> "A and B."
(format nil *list-etc* 'a 'b 'c)         ==> "A, B and C."
(format nil *list-etc* 'a 'b 'c 'd)      ==> "A, B, C, etc."
(format nil *list-etc* 'a 'b 'c 'd 'e)   ==> "A, B, C, etc."
```

Note that the control string actually contains two `~[~]` directives--both of which use `#` to select the clause to use. The first consumes between zero and two arguments, while the second consumes one more, if available. **`FORMAT`** will silently ignore any arguments not consumed while processing the control string.

With a colon modifier, the `~[` can contain only two clauses; the directive consumes a single argument and processes the first clause if the argument is **`NIL`** and the second clause is otherwise. You used this variant of `~[` in Chapter 9 to generate pass/fail messages, like this:

```
(format t "~:[FAIL~;pass~]" test-result)
```

Note that either clause can be empty, but the directive must contain a `~;`.

Finally, with an at-sign modifier, the `~[` directive can have only one clause. The directive consumes one argument and, if it's non-**`NIL`**, processes the clause after backing up to make the argument available to be consumed again.

```
(format nil "~@[x = ~a ~]~@[y = ~a~]" 10 20)    ==> "x = 10 y = 20"
(format nil "~@[x = ~a ~]~@[y = ~a~]" 10 nil)   ==> "x = 10 "
(format nil "~@[x = ~a ~]~@[y = ~a~]" nil 20)   ==> "y = 20"
(format nil "~@[x = ~a ~]~@[y = ~a~]" nil nil)  ==> ""
```

Iteration

Another **`FORMAT`** directive that you've seen already, in passing, is the iteration directive `~{`. This directive tells **`FORMAT`** to iterate over the elements of a list or over the implicit list of the format arguments.

With no modifiers, `~{` consumes one format argument, which must be a list. Like the `~[` directive, which is always paired with a `~]` directive, the `~{` directive is always paired with a closing `~}`. The text between the two markers is processed as a control string, which draws its arguments from the list consumed by the `~{` directive. **`FORMAT`** will repeatedly process this control string for as long as the list being iterated over has elements left. In the following example, the `~{` consumes the single format argument, the list `(1 2 3)`, and then processes the control string `"~a, "`, repeating until all the elements of the list have been consumed.

```
(format nil "~{~a, ~}" (list 1 2 3)) ==> "1, 2, 3, "
```

However, it's annoying that in the output the last element of the list is followed by a comma and a space. You can fix that with the `~^` directive; within the body of a `~{` directive, the `~^` causes the iteration to stop immediately, without processing the rest of the control string, when no elements remain in the list. Thus, to avoid printing the comma and space after the last element of a list, you can precede them with a `~^`.

```
(format nil "~{~a~^, ~}" (list 1 2 3)) ==> "1, 2, 3"
```

The first two times through the iteration, there are still unprocessed elements in the list when the `~^` is processed. The third time through, however, after the `~a` directive consumes the `3`, the `~^` will cause **FORMAT** to break out of the iteration without printing the comma and space.

With an at-sign modifier, `~{` processes the remaining format arguments as a list.

```
(format nil "~@{~a~^, ~}" 1 2 3) ==> "1, 2, 3"
```

Within the body of a `~{...~}`, the special prefix parameter `#` refers to the number of items remaining to be processed in the list rather than the number of remaining format arguments. You can use that, along with the `~[` directive, to print a comma-separated list with an "and" before the last item like this:

```
(format nil "~{~a~#[~;, and ~:;, ~]~}" (list 1 2 3)) ==> "1, 2, and 3"
```

However, that doesn't really work right if the list is two items long because it adds an extra comma.

```
(format nil "~{~a~#[~;, and ~:;, ~]~}" (list 1 2)) ==> "1, and 2"
```

You could fix that in a bunch of ways. The following takes advantage of the behavior of `~@{` when nested inside another `~{` or `~@{` directive--it iterates over whatever items remain in the list being iterated over by the outer `~{`. You can combine that with a `~#[` directive to make the following control string for formatting lists according to English grammar:

```
(defparameter *english-list*
  "~{~#[~;~a~;~a and ~a~:;~@{~a~#[~;, and ~:;, ~]~}~]~}")

(format nil *english-list* '())        ==> ""
(format nil *english-list* '(1))       ==> "1"
(format nil *english-list* '(1 2))     ==> "1 and 2"
(format nil *english-list* '(1 2 3))   ==> "1, 2, and 3"
(format nil *english-list* '(1 2 3 4)) ==> "1, 2, 3, and 4"
```

While that control string verges on being "write-only" code, it's not too hard to understand if you take it a bit at a time. The outer `~{...~}` will consume and iterate over a list. The whole body of the iteration then consists of a `~#[...~]`; the output generated each time through the iteration will thus depend on the number of items left to be processed from the list. Splitting apart the `~#[...~]` directive on the `~;` clause separators, you can see that it's made up of four clauses, the last of which is a default clause because it's preceded by a `~:;` rather than a plain `~;`. The first clause, for when there are zero elements to be processed, is empty, which makes sense--if there are no more elements to be processed, the iteration would've stopped already. The second clause handles the case of one element with a simple `~a` directive. Two elements are handled with `"~a and ~a"`. And the default clause, which handles three or more elements, consists of another iteration directive, this time using `~@{` to iterate over the remaining elements of the list being processed by the outer `~{`. And the body of that iteration is the control string that can handle a list of three or more elements correctly, which is fine in this context. Because the `~@{` loop consumes all the remaining list items, the outer loop iterates only once.

If you wanted to print something special such as "<empty>" when the list was empty, you have a couple ways to do it. Perhaps the easiest is to put the text you want into the first (zeroth) clause of the outer `~#[` and then add a colon modifier to the closing `~}` of the outer iteration--the colon forces the iteration to be run at least once, even if the list is empty, at which point **FORMAT** processes the zeroth clause of the conditional directive.

```
(defparameter *english-list*
  "~{~#[<empty>~;~a~;~a and ~a~:;~@{~a~#[~;, and ~:;, ~]~}~]~:}")

(format nil *english-list* '()) ==> "<empty>"
```

Amazingly, the `~{` directive provides even more variations with different combinations of prefix parameters and modifiers. I won't discuss them other than to say you can use an integer prefix parameter to limit the maximum number of iterations and that, with a colon modifier, each element of the list (either an actual list or the list constructed by the `~@{` directive) must itself be a list whose elements will then be used as arguments to the control string in the `~:{...~}` directive.

Hop, Skip, Jump

A much simpler directive is the `~*` directive, which allows you to jump around in the list of format arguments. In its basic form, without modifiers, it simply skips the next argument, consuming it without emitting anything. More often, however, it's used with a colon modifier, which causes it to move backward, allowing the same argument to be used a second time. For instance, you can use `~:*` to print a numeric argument once as a word and once in numerals like this:

```
(format nil "~r ~:*(~d)" 1) ==> "one (1)"
```

Or you could implement a directive similar to `~:P` for an irregular plural by combing `~:*` with `~[`.

```
(format nil "I saw ~r el~:*~[ves~;f~:;ves~]." 0) ==> "I saw zero elves."
(format nil "I saw ~r el~:*~[ves~;f~:;ves~]." 1) ==> "I saw one elf."
(format nil "I saw ~r el~:*~[ves~;f~:;ves~]." 2) ==> "I saw two elves."
```

In this control string, the `~R` prints the format argument as a cardinal number. Then the `~:*` directive backs up so the number is also used as the argument to the `~[` directive, selecting between the clauses for when the number is zero, one, or anything else.[7]

Within an `~{` directive, `~*` skips or backs up over the items in the list. For instance, you could print only the keys of a plist like this:

```
(format nil "~{~s~*~^ ~}" '(:a 10 :b 20)) ==> ":A :B"
```

The `~*` directive can also be given a prefix parameter. With no modifiers or with the colon modifier, this parameter specifies the number of arguments to move forward or backward and defaults to one. With an at-sign modifier, the prefix parameter specifies an absolute, zero-based index of the argument to jump to, defaulting to zero. The at-sign variant of `~*` can be useful if you want to use different control strings to generate different messages for the same arguments and if different messages need to use the arguments in different orders.[8]

And More . . .

And there's more--I haven't mentioned the `~?` directive, which can take snippets of control strings from the format arguments or the `~/` directive, which allows you to call an arbitrary function to handle the next format argument. And then there are all the directives for generating tabular and pretty-printed output. But the directives discussed in this chapter should be plenty for the time being.

In the next chapter, you'll move onto Common Lisp's condition system, the Common Lisp analog to other languages' exception and error handling systems.

---

[1]Of course, most folks realize it's not worth getting that worked up over *anything* in a programming language and use it or not without a lot of angst. On the other hand, it's interesting that these two features are the two features in Common Lisp that implement what are essentially domain-specific languages using a syntax not based on s-expressions. The syntax of **`FORMAT`**'s control strings is character based, while the extended **`LOOP`** macro can be understood only in terms of the grammar of the **`LOOP`** keywords. That one of the common knocks on both **`FORMAT`** and **`LOOP`** is that they "aren't Lispy enough" is evidence that Lispers really do like the s-expression syntax.

[2]Readers interested in the pretty printer may want to read the paper "XP: A Common Lisp Pretty Printing System" by Richard Waters. It's a description of the pretty printer that was eventually incorporated into Common Lisp. You can download it from `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-1102a.pdf`.

[3]To slightly confuse matters, most other I/O functions also accept **`T`** and **`NIL`** as *stream designators* but with a different meaning: as a stream designator, **`T`** designates the bidirectional stream **`*TERMINAL-IO*`**, while **`NIL`** designates **`*STANDARD-OUTPUT*`** as an output stream and **`*STANDARD-INPUT*`** as an input stream.

[4]This variant on the `~C` directive makes more sense on platforms like the Lisp Machines where key press events were represented by Lisp characters.

[5]Technically, if the argument isn't a real number, `~F` is supposed to format it as if by the `~D` directive, which in turn behaves like the `~A` directive if the argument isn't a number, but not all implementations get this right.

[6]Well, that's what the language standard says. For some reason, perhaps rooted in a common ancestral code base, several Common Lisp implementations don't implement this aspect of the `~F` directive correctly.

[7]If you find "I saw zero elves" to be a bit clunky, you could use a slightly more elaborate format string that makes another use of `~:*` like this:

```
(format nil "I saw ~[no~:;~:*~r~] el~:*~[ves~;f~:;ves~]." 0) ==> "I saw no elves."
(format nil "I saw ~[no~:;~:*~r~] el~:*~[ves~;f~:;ves~]." 1) ==> "I saw one elf."
(format nil "I saw ~[no~:;~:*~r~] el~:*~[ves~;f~:;ves~]." 2) ==> "I saw two elves."
```

[8]This kind of problem can arise when trying to localize an application and translate human-readable messages into different languages. **`FORMAT`** can help with some of these problems but is by no means a full-blown localization system.