3. Practical: A Simple Database

Obviously, before you can start building real software in Lisp, you'll have to learn the language. But let's face it--you may be thinking, "'Practical Common Lisp,' isn't that an oxymoron? Why should you be expected to bother learning all the details of a language unless it's actually good for something you care about?" So I'll start by giving you a small example of what you can do with Common Lisp. In this chapter you'll write a simple database for keeping track of CDs. You'll use similar techniques in Chapter 27 when you build a database of MP3s for our streaming MP3 server. In fact, you could think of this as part of the MP3 software project--after all, in order to have a bunch of MP3s to listen to, it might be helpful to be able to keep track of which CDs you have and which ones you need to rip.

In this chapter, I'll cover just enough Lisp as we go along for you to understand how the code works. But I'll gloss over quite a few details. For now you needn't sweat the small stuff--the next several chapters will cover all the Common Lisp constructs used here, and more, in a much more systematic way.

One terminology note: I'll discuss a handful of Lisp operators in this chapter. In Chapter 4, you'll learn that Common Lisp provides three distinct kinds of operators: *functions*, *macros*, and *special operators*. For the purposes of this chapter, you don't really need to know the difference. I will, however, refer to different operators as functions or macros or special operators as appropriate, rather than trying to hide the details behind the word *operator*. For now you can treat *function*, *macro*, and *special operator* as all more or less equivalent.[1]

Also, keep in mind that I won't bust out all the most sophisticated Common Lisp techniques for your very first post-"hello, world" program. The point of this chapter isn't that this is how you would write a database in Lisp; rather, the point is for you to get an idea of what programming in Lisp is like and to see how even a relatively simple Lisp program can be quite featureful.

CDs and Records

To keep track of CDs that need to be ripped to MP3s and which CDs should be ripped first, each record in the database will contain the title and artist of the CD, a rating of how much the user likes it, and a flag saying whether it has been ripped. So, to start with, you'll need a way to represent a single database record (in other words, one CD). Common Lisp gives you lots of choices of data structures from a simple four-item list to a user-defined class, using the Common Lisp Object System (CLOS).

For now you can stay at the simple end of the spectrum and use a list. You can make a list with the **`LIST`** function, which, appropriately enough, returns a list of its arguments.

```
CL-USER> (list 1 2 3)
(1 2 3)
```

You could use a four-item list, mapping a given position in the list to a given field in the record. However, another flavor of list--called a *property list*, or *plist* for short--is even more convenient. A plist is a list where every other element, starting with the first, is a *symbol* that describes what the next element in the list is. I won't get into all the details of exactly what a symbol is right now; basically it's a name. For the symbols that name the fields in the CD database, you can use a particular kind of symbol, called a *keyword* symbol. A keyword is any name that starts with a colon (`:`), for instance, `:foo`. Here's an example of a plist using the keyword symbols `:a`, `:b`, and `:c` as property names:

```
CL-USER> (list :a 1 :b 2 :c 3)
(:A 1 :B 2 :C 3)
```

Note that you can create a property list with the same **`LIST`** function as you use to create other lists; it's the contents that make it a plist.

The thing that makes plists a convenient way to represent the records in a database is the function **`GETF`**, which takes a plist and a symbol and returns the value in the plist following the symbol, making a plist a sort of poor man's hash table. Lisp has real hash tables too, but plists are sufficient for your needs here and can more easily be saved to a file, which will come in handy later.

```
CL-USER> (getf (list :a 1 :b 2 :c 3) :a)
1
CL-USER> (getf (list :a 1 :b 2 :c 3) :c)
3
```

Given all that, you can easily enough write a function `make-cd` that will take the four fields as arguments and return a plist representing that CD.

```
(defun make-cd (title artist rating ripped)
  (list :title title :artist artist :rating rating :ripped ripped))
```

The word **`DEFUN`** tells us that this form is defining a new function. The name of the function is `make-cd`. After the name comes the parameter list. This function has four parameters: `title`, `artist`, `rating`, and `ripped`. Everything after the parameter list is the body of the function. In this case the body is just one form, a call to **`LIST`**. When `make-cd` is called, the arguments passed to the call will be bound to the variables in the parameter list. For instance, to make a record for the CD *Roses* by Kathy Mattea, you might call `make-cd` like this:

```
CL-USER> (make-cd "Roses" "Kathy Mattea" 7 t)
(:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T)
```

Filing CDs

A single record, however, does not a database make. You need some larger construct to hold the records. Again, for simplicity's sake, a list seems like a good choice. Also for simplicity you can use a global variable, `*db*`, which you can define with the **`DEFVAR`** macro. The asterisks (*) in the name are a Lisp naming convention for global variables.[2]

```
(defvar *db* nil)
```

You can use the **`PUSH`** macro to add items to `*db*`. But it's probably a good idea to abstract things a tiny bit, so you should define a function `add-record` that adds a record to the database.

```
(defun add-record (cd) (push cd *db*))
```

Now you can use `add-record` and `make-cd` together to add CDs to the database.

```
CL-USER> (add-record (make-cd "Roses" "Kathy Mattea" 7 t))
((:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
CL-USER> (add-record (make-cd "Fly" "Dixie Chicks" 8 t))
((:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
CL-USER> (add-record (make-cd "Home" "Dixie Chicks" 9 t))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
```

The stuff printed by the REPL after each call to `add-record` is the return value, which is the value returned by the last expression in the function body, the **`PUSH`**. And **`PUSH`** returns the new value of the variable it's modifying. So what you're actually seeing is the value of the database after the record has been added.

Looking at the Database Contents

You can also see the current value of `*db*` whenever you want by typing `*db*` at the REPL.

```
CL-USER> *db*
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
```

However, that's not a very satisfying way of looking at the output. You can write a `dump-db` function that dumps out the database in a more human-readable format, like this:

```
TITLE:    Home
ARTIST:   Dixie Chicks
RATING:   9
RIPPED:   T

TITLE:    Fly
ARTIST:   Dixie Chicks
RATING:   8
RIPPED:   T

TITLE:    Roses
ARTIST:   Kathy Mattea
RATING:   7
RIPPED:   T
```

The function looks like this:

```
(defun dump-db ()
  (dolist (cd *db*)
    (format t "~{~a:~10t~a~%~}~%" cd)))
```

This function works by looping over all the elements of `*db*` with the **`DOLIST`** macro, binding each element to the variable `cd` in turn. For each value of `cd`, you use the **`FORMAT`** function to print it.

Admittedly, the **`FORMAT`** call is a little cryptic. However, **`FORMAT`** isn't particularly more complicated than C or Perl's `printf` function or Python's string-`%` operator. In Chapter 18 I'll discuss **`FORMAT`** in greater detail. For now we can take this call bit by bit. As you saw in Chapter 2, **`FORMAT`** takes at least two arguments, the first being the stream where it sends its output; `t` is shorthand for the stream `*standard-output*`.

The second argument to **`FORMAT`** is a format string that can contain both literal text and directives telling **`FORMAT`** things such as how to interpolate the rest of its arguments. Format directives start with ~ (much the way `printf`'s directives start with `%`). **`FORMAT`** understands dozens of directives, each with their own set of options.[3] However, for now I'll just focus on the ones you need to write `dump-db`.

The `~a` directive is the *aesthetic* directive; it means to consume one argument and output it in a human-readable form. This will render keywords without the leading : and strings without quotation marks. For instance:

```
CL-USER> (format t "~a" "Dixie Chicks")
Dixie Chicks
NIL
```

or:

```
CL-USER> (format t "~a" :title)
TITLE
NIL
```

The `~t` directive is for tabulating. The `~10t` tells **`FORMAT`** to emit enough spaces to move to the tenth column before processing the next `~a`. A `~t` doesn't consume any arguments.

```
CL-USER> (format t "~a:~10t~a" :artist "Dixie Chicks")
ARTIST:   Dixie Chicks
NIL
```

Now things get slightly more complicated. When **`FORMAT`** sees `~{` the next argument to be consumed must be a list. **`FORMAT`** loops over that list, processing the directives between the `~{` and `~}`, consuming as many elements of the list as needed each time through the list. In `dump-db`, the **`FORMAT`** loop will consume one keyword and one value from the list each time through the loop. The `~%` directive doesn't consume any arguments but tells **`FORMAT`** to emit a

newline. Then after the ~} ends the loop, the last ~% tells **FORMAT** to emit one more newline to put a blank line between each CD.

Technically, you could have also used **FORMAT** to loop over the database itself, turning our dump-db function into a one-liner.

```
(defun dump-db ()
  (format t "~{~{~a:~10t~a~%~}~%~}" *db*))
```

That's either very cool or very scary depending on your point of view.

Improving the User Interaction

While our add-record function works fine for adding records, it's a bit Lispy for the casual user. And if they want to add a bunch of records, it's not very convenient. So you may want to write a function to prompt the user for information about a set of CDs. Right away you know you'll need some way to prompt the user for a piece of information and read it. So let's write that.

```
(defun prompt-read (prompt)
  (format *query-io* "~a: " prompt)
  (force-output *query-io*)
  (read-line *query-io*))
```

You use your old friend **FORMAT** to emit a prompt. Note that there's no ~% in the format string, so the cursor will stay on the same line. The call to **FORCE-OUTPUT** is necessary in some implementations to ensure that Lisp doesn't wait for a newline before it prints the prompt.

Then you can read a single line of text with the aptly named **READ-LINE** function. The variable *query-io* is a global variable (which you can tell because of the * naming convention for global variables) that contains the input stream connected to the terminal. The return value of prompt-read will be the value of the last form, the call to **READ-LINE**, which returns the string it read (without the trailing newline.)

You can combine your existing make-cd function with prompt-read to build a function that makes a new CD record from data it gets by prompting for each value in turn.

```
(defun prompt-for-cd ()
  (make-cd
   (prompt-read "Title")
   (prompt-read "Artist")
   (prompt-read "Rating")
   (prompt-read "Ripped [y/n]")))
```

That's almost right. Except prompt-read returns a string, which, while fine for the Title and Artist fields, isn't so great for the Rating and Ripped fields, which should be a number and a boolean. Depending on how sophisticated a user interface you want, you can go to arbitrary lengths to validate the data the user enters. For now let's lean toward the quick and dirty: you can wrap the prompt-read for the rating in a call to Lisp's **PARSE-INTEGER** function, like this:

```
(parse-integer (prompt-read "Rating"))
```

Unfortunately, the default behavior of **PARSE-INTEGER** is to signal an error if it can't parse an integer out of the string or if there's any non-numeric junk in the string. However, it takes an optional keyword argument `:junk-allowed`, which tells it to relax a bit.

```
(parse-integer (prompt-read "Rating") :junk-allowed t)
```

But there's still one problem: if it can't find an integer amidst all the junk, **PARSE-INTEGER** will return `NIL` rather than a number. In keeping with the quick-and-dirty approach, you may just want to call that 0 and continue. Lisp's **OR** macro is just the thing you need here. It's similar to the "short-circuiting" `||` in Perl, Python, Java, and C; it takes a series of expressions, evaluates them one at a time, and returns the first non-nil value (or **NIL** if they're all **NIL**). So you can use the following:

```
(or (parse-integer (prompt-read "Rating") :junk-allowed t) 0)
```

to get a default value of 0.

Fixing the code to prompt for Ripped is quite a bit simpler. You can just use the Common Lisp function **Y-OR-N-P**.

```
(y-or-n-p "Ripped [y/n]: ")
```

In fact, this will be the most robust part of `prompt-for-cd`, as **Y-OR-N-P** will reprompt the user if they enter something that doesn't start with *y*, *Y*, *n*, or *N*.

Putting those pieces together you get a reasonably robust `prompt-for-cd` function.

```
(defun prompt-for-cd ()
  (make-cd
   (prompt-read "Title")
   (prompt-read "Artist")
   (or (parse-integer (prompt-read "Rating") :junk-allowed t) 0)
   (y-or-n-p "Ripped [y/n]: ")))
```

Finally, you can finish the "add a bunch of CDs" interface by wrapping `prompt-for-cd` in a function that loops until the user is done. You can use the simple form of the **LOOP** macro, which repeatedly executes a body of expressions until it's exited by a call to **RETURN**. For example:

```
(defun add-cds ()
  (loop (add-record (prompt-for-cd))
      (if (not (y-or-n-p "Another? [y/n]: ")) (return))))
```

Now you can use `add-cds` to add some more CDs to the database.

```
CL-USER> (add-cds)
Title: Rockin' the Suburbs
Artist: Ben Folds
Rating: 6
Ripped  [y/n]: y
Another?  [y/n]: y
```

```
Title: Give Us a Break
Artist: Limpopo
Rating: 10
Ripped  [y/n]: y
Another?  [y/n]: y
Title: Lyle Lovett
Artist: Lyle Lovett
Rating: 9
Ripped  [y/n]: y
Another?  [y/n]: n
NIL
```

Saving and Loading the Database

Having a convenient way to add records to the database is nice. But it's not so nice that the user is going to be very happy if they have to reenter all the records every time they quit and restart Lisp. Luckily, with the data structures you're using to represent the data, it's trivially easy to save the data to a file and reload it later. Here's a `save-db` function that takes a filename as an argument and saves the current state of the database:

```
(defun save-db (filename)
  (with-open-file (out filename
                   :direction :output
                   :if-exists :supersede)
    (with-standard-io-syntax
      (print *db* out))))
```

The **`WITH-OPEN-FILE`** macro opens a file, binds the stream to a variable, executes a set of expressions, and then closes the file. It also makes sure the file is closed even if something goes wrong while evaluating the body. The list directly after **`WITH-OPEN-FILE`** isn't a function call but rather part of the syntax defined by **`WITH-OPEN-FILE`**. It contains the name of the variable that will hold the file stream to which you'll write within the body of **`WITH-OPEN-FILE`**, a value that must be a file name, and then some options that control how the file is opened. Here you specify that you're opening the file for writing with `:direction :output` and that you want to overwrite an existing file of the same name if it exists with `:if-exists :supersede`.

Once you have the file open, all you have to do is print the contents of the database with `(print *db* out)`. Unlike **`FORMAT`**, **`PRINT`** prints Lisp objects in a form that can be read back in by the Lisp reader. The macro **`WITH-STANDARD-IO-SYNTAX`** ensures that certain variables that affect the behavior of **`PRINT`** are set to their standard values. You'll use the same macro when you read the data back in to make sure the Lisp reader and printer are operating compatibly.

The argument to `save-db` should be a string containing the name of the file where the user wants to save the database. The exact form of the string will depend on what operating system they're using. For instance, on a Unix box they should be able to call `save-db` like this:

```
CL-USER> (save-db "~/my-cds.db")
((:TITLE "Lyle Lovett" :ARTIST "Lyle Lovett" :RATING 9 :RIPPED T)
 (:TITLE "Give Us a Break" :ARTIST "Limpopo" :RATING 10 :RIPPED T)
```

```
(:TITLE "Rockin' the Suburbs" :ARTIST "Ben Folds" :RATING 6 :RIPPED
 T)
(:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
(:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
(:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 9 :RIPPED T))
```

On Windows, the filename might be something like "`c:/my-cds.db`" or "`c:\\my-cds.db`."[4]

You can open this file in any text editor to see what it looks like. You should see something a lot like what the REPL prints if you type `*db*`.

The function to load the database back in is similar.

```
(defun load-db (filename)
  (with-open-file (in filename)
    (with-standard-io-syntax
      (setf *db* (read in)))))
```

This time you don't need to specify `:direction` in the options to **`WITH-OPEN-FILE`**, since you want the default of `:input`. And instead of printing, you use the function **`READ`** to read from the stream `in`. This is the same reader used by the REPL and can read any Lisp expression you could type at the REPL prompt. However, in this case, you're just reading and saving the expression, not evaluating it. Again, the **`WITH-STANDARD-IO-SYNTAX`** macro ensures that **`READ`** is using the same basic syntax that `save-db` did when it **`PRINT`**ed the data.

The **`SETF`** macro is Common Lisp's main assignment operator. It sets its first argument to the result of evaluating its second argument. So in `load-db` the `*db*` variable will contain the object read from the file, namely, the list of lists written by `save-db`. You do need to be careful about one thing--`load-db` clobbers whatever was in `*db*` before the call. So if you've added records with `add-record` or `add-cds` that haven't been saved with `save-db`, you'll lose them.

Querying the Database

Now that you have a way to save and reload the database to go along with a convenient user interface for adding new records, you soon may have enough records that you won't want to be dumping out the whole database just to look at what's in it. What you need is a way to query the database. You might like, for instance, to be able to write something like this:

```
(select :artist "Dixie Chicks")
```

and get a list of all the records where the artist is the Dixie Chicks. Again, it turns out that the choice of saving the records in a list will pay off.

The function **`REMOVE-IF-NOT`** takes a predicate and a list and returns a list containing only the elements of the original list that match the predicate. In other words, it has removed all the elements that don't match the predicate. However, **`REMOVE-IF-NOT`** doesn't really remove

anything--it creates a new list, leaving the original list untouched. It's like running grep over a file. The predicate argument can be any function that accepts a single argument and returns a boolean value--**NIL** for false and anything else for true.

For instance, if you wanted to extract all the even elements from a list of numbers, you could use **REMOVE-IF-NOT** as follows:

```
CL-USER> (remove-if-not #'evenp '(1 2 3 4 5 6 7 8 9 10))
(2 4 6 8 10)
```

In this case, the predicate is the function **EVENP**, which returns true if its argument is an even number. The funny notation `#'` is shorthand for "Get me the function with the following name." Without the `#'`, Lisp would treat `evenp` as the name of a variable and look up the value of the variable, not the function.

You can also pass **REMOVE-IF-NOT** an anonymous function. For instance, if **EVENP** didn't exist, you could write the previous expression as the following:

```
CL-USER> (remove-if-not #'(lambda (x) (= 0 (mod x 2))) '(1 2 3 4 5 6 7 8 9 10))
(2 4 6 8 10)
```

In this case, the predicate is this anonymous function:

```
(lambda (x) (= 0 (mod x 2)))
```

which checks that its argument is equal to 0 modulus 2 (in other words, is even). If you wanted to extract only the odd numbers using an anonymous function, you'd write this:

```
CL-USER> (remove-if-not #'(lambda (x) (= 1 (mod x 2))) '(1 2 3 4 5 6 7 8 9 10))
(1 3 5 7 9)
```

Note that `lambda` isn't the name of the function--it's the indicator you're defining an anonymous function.[5] Other than the lack of a name, however, a **LAMBDA** expression looks a lot like a **DEFUN**: the word `lambda` is followed by a parameter list, which is followed by the body of the function.

To select all the Dixie Chicks' albums in the database using **REMOVE-IF-NOT**, you need a function that returns true when the artist field of a record is `"Dixie Chicks"`. Remember that we chose the plist representation for the database records because the function **GETF** can extract named fields from a plist. So assuming `cd` is the name of a variable holding a single database record, you can use the expression `(getf cd :artist)` to extract the name of the artist. The function **EQUAL**, when given string arguments, compares them character by character. So `(equal (getf cd :artist) "Dixie Chicks")` will test whether the artist field of a given CD is equal to `"Dixie Chicks"`. All you need to do is wrap that expression in a **LAMBDA** form to make an anonymous function and pass it to **REMOVE-IF-NOT**.

```
CL-USER> (remove-if-not
  #'(lambda (cd) (equal (getf cd :artist) "Dixie Chicks")) *db*)
```

```
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

Now suppose you want to wrap that whole expression in a function that takes the name of the artist as an argument. You can write that like this:

```
(defun select-by-artist (artist)
  (remove-if-not
   #'(lambda (cd) (equal (getf cd :artist) artist))
   *db*))
```

Note how the anonymous function, which contains code that won't run until it's invoked in **`REMOVE-IF-NOT`**, can nonetheless refer to the variable `artist`. In this case the anonymous function doesn't just save you from having to write a regular function--it lets you write a function that derives part of its meaning--the value of `artist`--from the context in which it's embedded.

So that's `select-by-artist`. However, selecting by artist is only one of the kinds of queries you might like to support. You *could* write several more functions, such as `select-by-title`, `select-by-rating`, `select-by-title-and-artist`, and so on. But they'd all be about the same except for the contents of the anonymous function. You can instead make a more general `select` function that takes a function as an argument.

```
(defun select (selector-fn)
  (remove-if-not selector-fn *db*))
```

So what happened to the `#'`? Well, in this case you don't want **`REMOVE-IF-NOT`** to use the function named `selector-fn`. You want it to use the anonymous function that was passed as an argument to `select` in the *variable* `selector-fn`. Though, the `#'` comes back in the *call* to `select`.

```
CL-USER> (select #'(lambda (cd) (equal (getf cd :artist) "Dixie Chicks")))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

But that's really quite gross-looking. Luckily, you can wrap up the creation of the anonymous function.

```
(defun artist-selector (artist)
  #'(lambda (cd) (equal (getf cd :artist) artist)))
```

This is a function that returns a function and one that references a variable that--it seems--won't exist after `artist-selector` returns.[6] It may seem odd now, but it actually works just the way you'd want--if you call `artist-selector` with an argument of `"Dixie Chicks"`, you get an anonymous function that matches CDs whose `:artist` field is `"Dixie Chicks"`, and if you call it with `"Lyle Lovett"`, you get a different function that will match against an `:artist` field of `"Lyle Lovett"`. So now you can rewrite the call to `select` like this:

```
CL-USER> (select (artist-selector "Dixie Chicks"))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

Now you just need some more functions to generate selectors. But just as you don't want to have to write `select-by-title`, `select-by-rating`, and so on, because they would all be quite similar, you're not going to want to write a bunch of nearly identical selector-function generators, one for each field. Why not write one general-purpose selector-function generator, a function that, depending on what arguments you pass it, will generate a selector function for different fields or maybe even a combination of fields? You can write such a function, but first you need a crash course in a feature called *keyword parameters*.

In the functions you've written so far, you've specified a simple list of parameters, which are bound to the corresponding arguments in the call to the function. For instance, the following function:

```
(defun foo (a b c) (list a b c))
```

has three parameters, `a`, `b`, and `c`, and must be called with three arguments. But sometimes you may want to write a function that can be called with varying numbers of arguments. Keyword parameters are one way to achieve this. A version of `foo` that uses keyword parameters might look like this:

```
(defun foo (&key a b c) (list a b c))
```

The only difference is the `&key` at the beginning of the argument list. However, the calls to this new `foo` will look quite different. These are all legal calls with the result to the right of the ==>:

```
(foo :a 1 :b 2 :c 3)  ==> (1 2 3)
(foo :c 3 :b 2 :a 1)  ==> (1 2 3)
(foo :a 1 :c 3)       ==> (1 NIL 3)
(foo)                 ==> (NIL NIL NIL)
```

As these examples show, the value of the variables `a`, `b`, and `c` are bound to the values that follow the corresponding keyword. And if a particular keyword isn't present in the call, the corresponding variable is set to **`NIL`**. I'm glossing over a bunch of details of how keyword parameters are specified and how they relate to other kinds of parameters, but you need to know one more detail.

Normally if a function is called with no argument for a particular keyword parameter, the parameter will have the value **`NIL`**. However, sometimes you'll want to be able to distinguish between a **`NIL`** that was explicitly passed as the argument to a keyword parameter and the default value **`NIL`**. To allow this, when you specify a keyword parameter you can replace the simple name with a list consisting of the name of the parameter, a default value, and another parameter name, called a *supplied-p* parameter. The supplied-p parameter will be set to true or false depending on whether an argument was actually passed for that keyword parameter in a particular call to the function. Here's a version of `foo` that uses this feature:

```
(defun foo (&key a (b 20) (c 30 c-p)) (list a b c c-p))
```

Now the same calls from earlier yield these results:

```
(foo :a 1 :b 2 :c 3)  ==> (1 2 3 T)
(foo :c 3 :b 2 :a 1)  ==> (1 2 3 T)
(foo :a 1 :c 3)       ==> (1 20 3 T)
(foo)                 ==> (NIL 20 30 NIL)
```

The general selector-function generator, which you can call `where` for reasons that will soon become apparent if you're familiar with SQL databases, is a function that takes four keyword parameters corresponding to the fields in our CD records and generates a selector function that selects any CDs that match all the values given to `where`. For instance, it will let you say things like this:

```
(select (where :artist "Dixie Chicks"))
```

or this:

```
(select (where :rating 10 :ripped nil))
```

The function looks like this:

```
(defun where (&key title artist rating (ripped nil ripped-p))
  #'(lambda (cd)
      (and
       (if title    (equal (getf cd :title)  title)  t)
       (if artist   (equal (getf cd :artist) artist) t)
       (if rating   (equal (getf cd :rating) rating) t)
       (if ripped-p (equal (getf cd :ripped) ripped) t))))
```

This function returns an anonymous function that returns the logical `AND` of one clause per field in our CD records. Each clause checks if the appropriate argument was passed in and then either compares it to the value in the corresponding field in the CD record or returns `t`, Lisp's version of truth, if the parameter wasn't passed in. Thus, the selector function will return `t` only for CDs that match all the arguments passed to `where`.[7] Note that you need to use a three-item list to specify the keyword parameter `ripped` because you need to know whether the caller actually passed `:ripped nil`, meaning, "Select CDs whose ripped field is nil," or whether they left out `:ripped` altogether, meaning "I don't care what the value of the ripped field is."

Updating Existing Records--Another Use for WHERE

Now that you've got nice generalized `select` and `where` functions, you're in a good position to write the next feature that every database needs--a way to update particular records. In SQL the `update` command is used to update a set of records matching a particular `where` clause. That seems like a good model, especially since you've already got a `where`-clause generator. In fact, the `update` function is mostly just the application of a few ideas you've already seen: using a passed-in selector function to choose the records to update and using keyword arguments to specify the values to change. The main new bit is the use of a function **`MAPCAR`** that maps

over a list, `*db*` in this case, and returns a new list containing the results of calling a function on each item in the original list.

```
(defun update (selector-fn &key title artist rating (ripped nil ripped-p))
  (setf *db*
        (mapcar
         #'(lambda (row)
             (when (funcall selector-fn row)
               (if title    (setf (getf row :title) title))
               (if artist   (setf (getf row :artist) artist))
               (if rating   (setf (getf row :rating) rating))
               (if ripped-p (setf (getf row :ripped) ripped)))
             row) *db*)))
```

One other new bit here is the use of **`SETF`** on a complex form such as `(getf row :title)`. I'll discuss **`SETF`** in greater detail in Chapter 6, but for now you just need to know that it's a general assignment operator that can be used to assign lots of "places" other than just variables. (It's a coincidence that **`SETF`** and **`GETF`** have such similar names--they don't have any special relationship.) For now it's enough to know that after `(setf (getf row :title) title)`, the plist referenced by row will have the value of the variable `title` following the property name `:title`. With this `update` function if you decide that you *really* dig the Dixie Chicks and that all their albums should go to 11, you can evaluate the following form:

```
CL-USER> (update (where :artist "Dixie Chicks") :rating 11)
NIL
```

And it is so.

```
CL-USER> (select (where :artist "Dixie Chicks"))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 11 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 11 :RIPPED T))
```

You can even more easily add a function to delete rows from the database.

```
(defun delete-rows (selector-fn)
  (setf *db* (remove-if selector-fn *db*)))
```

The function **`REMOVE-IF`** is the complement of **`REMOVE-IF-NOT`**; it returns a list with all the elements that do match the predicate removed. Like **`REMOVE-IF-NOT`**, it doesn't actually affect the list it's passed but by saving the result back into `*db*`, `delete-rows`[8] actually changes the contents of the database.[9]

Removing Duplication and Winning Big

So far *all* the database code supporting insert, select, update, and delete, not to mention a command-line user interface for adding new records and dumping out the contents, is just a little more than 50 lines. Total.[10]

Yet there's still some annoying code duplication. And it turns out you can remove the duplication and make the code more flexible at the same time. The duplication I'm thinking of is in the

where function. The body of the `where` function is a bunch of clauses like this, one per field:

```
(if title (equal (getf cd :title) title) t)
```

Right now it's not so bad, but like all code duplication it has the same cost: if you want to change how it works, you have to change multiple copies. And if you change the fields in a CD, you'll have to add or remove clauses to `where`. And `update` suffers from the same kind of duplication. It's doubly annoying since the whole point of the `where` function is to dynamically generate a bit of code that checks the values you care about; why should it have to do work at runtime checking whether `title` was even passed in?

Imagine that you were trying to optimize this code and discovered that it was spending too much time checking whether `title` and the rest of the keyword parameters to `where` were even set? [11] If you really wanted to remove all those runtime checks, you could go through a program and find all the places you call `where` and look at exactly what arguments you're passing. Then you could replace each call to `where` with an anonymous function that does only the computation necessary. For instance, if you found this snippet of code:

```
(select (where :title "Give Us a Break" :ripped t))
```

you could change it to this:

```
(select
 #'(lambda (cd)
     (and (equal (getf cd :title) "Give Us a Break")
          (equal (getf cd :ripped) t))))
```

Note that the anonymous function is different from the one that `where` would have returned; you're not trying to save the call to `where` but rather to provide a more efficient selector function. This anonymous function has clauses only for the fields that you actually care about at this call site, so it doesn't do any extra work the way a function returned by `where` might.

You can probably imagine going through all your source code and fixing up all the calls to `where` in this way. But you can probably also imagine that it would be a huge pain. If there were enough of them, and it was important enough, it might even be worthwhile to write some kind of preprocessor that converts `where` calls to the code you'd write by hand.

The Lisp feature that makes this trivially easy is its macro system. I can't emphasize enough that the Common Lisp macro shares essentially nothing but the name with the text-based macros found in C and C++. Where the C pre-processor operates by textual substitution and understands almost nothing of the structure of C and C++, a Lisp macro is essentially a code generator that gets run for you automatically by the compiler.[12] When a Lisp expression contains a call to a macro, instead of evaluating the arguments and passing them to the function, the Lisp compiler passes the arguments, unevaluated, to the macro code, which returns a new Lisp expression that is then evaluated in place of the original macro call.

I'll start with a simple, and silly, example and then show how you can replace the `where` function with a `where` macro. Before I can write this example macro, I need to quickly introduce one new function: `REVERSE` takes a list as an argument and returns a new list that is its reverse. So `(reverse '(1 2 3))` evaluates to `(3 2 1)`. Now let's create a macro.

```
(defmacro backwards (expr) (reverse expr))
```

The main syntactic difference between a function and a macro is that you define a macro with **`DEFMACRO`** instead of **`DEFUN`**. After that a macro definition consists of a name, just like a function, a parameter list, and a body of expressions, both also like a function. However, a macro has a totally different effect. You can use this macro as follows:

```
CL-USER> (backwards ("hello, world" t format))
hello, world
NIL
```

How did that work? When the REPL started to evaluate the `backwards` expression, it recognized that `backwards` is the name of a macro. So it left the expression `("hello, world" t format)` unevaluated, which is good because it isn't a legal Lisp form. It then passed that list to the `backwards` code. The code in `backwards` passed the list to **`REVERSE`**, which returned the list `(format t "hello, world")`. `backwards` then passed that value back out to the REPL, which then evaluated it in place of the original expression.

The `backwards` macro thus defines a new language that's a lot like Lisp--just backward--that you can drop into anytime simply by wrapping a backward Lisp expression in a call to the `backwards` macro. And, in a compiled Lisp program, that new language is just as efficient as normal Lisp because all the macro code--the code that generates the new expression--runs at compile time. In other words, the compiler will generate exactly the same code whether you write `(backwards ("hello, world" t format))` or `(format t "hello, world")`.

So how does that help with the code duplication in `where`? Well, you can write a macro that generates exactly the code you need for each particular call to `where`. Again, the best approach is to build our code bottom up. In the hand-optimized selector function, you had an expression of the following form for each actual field referred to in the original call to `where`:

```
(equal (getf cd field) value)
```

So let's write a function that, given the name of a field and a value, returns such an expression. Since an expression is just a list, you might think you could write something like this:

```
(defun make-comparison-expr (field value)    ; wrong
  (list equal (list getf cd field) value))
```

However, there's one trick here: as you know, when Lisp sees a simple name such as `field` or `value` other than as the first element of a list, it assumes it's the name of a variable and looks up

its value. That's fine for `field` and `value`; it's exactly what you want. But it will treat `equal`, `getf`, and `cd` the same way, which *isn't* what you want. However, you also know how to stop Lisp from evaluating a form: stick a single forward quote (`'`) in front of it. So if you write `make-comparison-expr` like this, it will do what you want:

```
(defun make-comparison-expr (field value)
  (list 'equal (list 'getf 'cd field) value))
```

You can test it out in the REPL.

```
CL-USER> (make-comparison-expr :rating 10)
(EQUAL (GETF CD :RATING) 10)
CL-USER> (make-comparison-expr :title "Give Us a Break")
(EQUAL (GETF CD :TITLE) "Give Us a Break")
```

It turns out that there's an even better way to do it. What you'd really like is a way to write an expression that's mostly not evaluated and then have some way to pick out a few expressions that you *do* want evaluated. And, of course, there's just such a mechanism. A back quote (`` ` ``) before an expression stops evaluation just like a forward quote.

```
CL-USER> `(1 2 3)
(1 2 3)
CL-USER> '(1 2 3)
(1 2 3)
```

However, in a back-quoted expression, any subexpression that's preceded by a comma is evaluated. Notice the effect of the comma in the second expression:

```
`(1 2 (+ 1 2))          ==> (1 2 (+ 1 2))
`(1 2 ,(+ 1 2))         ==> (1 2 3)
```

Using a back quote, you can write `make-comparison-expr` like this:

```
(defun make-comparison-expr (field value)
  `(equal (getf cd ,field) ,value))
```

Now if you look back to the hand-optimized selector function, you can see that the body of the function consisted of one comparison expression per field/value pair, all wrapped in an **`AND`** expression. Assume for the moment that you'll arrange for the arguments to the `where` macro to be passed as a single list. You'll need a function that can take the elements of such a list pairwise and collect the results of calling `make-comparison-expr` on each pair. To implement that function, you can dip into the bag of advanced Lisp tricks and pull out the mighty and powerful `LOOP` macro.

```
(defun make-comparisons-list (fields)
  (loop while fields
     collecting (make-comparison-expr (pop fields) (pop fields))))
```

A full discussion of `LOOP` will have to wait until Chapter 22; for now just note that this `LOOP` expression does exactly what you need: it loops while there are elements left in the `fields` list, popping off two at a time, passing them to `make-comparison-expr`, and collecting the

results to be returned at the end of the loop. The **POP** macro performs the inverse operation of the PUSH macro you used to add records to *db*.

Now you just need to wrap up the list returned by make-comparison-list in an AND and an anonymous function, which you can do in the where macro itself. Using a back quote to make a template that you fill in by interpolating the value of make-comparisons-list, it's trivial.

```
(defmacro where (&rest clauses)
  `#'(lambda (cd) (and ,@(make-comparisons-list clauses))))
```

This macro uses a variant of , (namely, the ,@) before the call to make-comparisons-list. The ,@ "splices" the value of the following expression--which must evaluate to a list--into the enclosing list. You can see the difference between , and ,@ in the following two expressions:

```
`(and ,(list 1 2 3))   ==> (AND (1 2 3))
`(and ,@(list 1 2 3))  ==> (AND 1 2 3)
```

You can also use ,@ to splice into the middle of a list.

```
`(and ,@(list 1 2 3) 4) ==> (AND 1 2 3 4)
```

The other important feature of the where macro is the use of &rest in the argument list. Like &key, &rest modifies the way arguments are parsed. With a &rest in its parameter list, a function or macro can take an arbitrary number of arguments, which are collected into a single list that becomes the value of the variable whose name follows the &rest. So if you call where like this:

```
(where :title "Give Us a Break" :ripped t)
```

the variable clauses will contain the list.

```
(:title "Give Us a Break" :ripped t)
```

This list is passed to make-comparisons-list, which returns a list of comparison expressions. You can see exactly what code a call to where will generate using the function **MACROEXPAND-1**. If you pass **MACROEXPAND-1**, a form representing a macro call, it will call the macro code with appropriate arguments and return the expansion. So you can check out the previous where call like this:

```
CL-USER> (macroexpand-1 '(where :title "Give Us a Break" :ripped t))
#'(LAMBDA (CD)
    (AND (EQUAL (GETF CD :TITLE) "Give Us a Break")
         (EQUAL (GETF CD :RIPPED) T)))
T
```

Looks good. Let's try it for real.

```
CL-USER> (select (where :title "Give Us a Break" :ripped t))
((:TITLE "Give Us a Break" :ARTIST "Limpopo" :RATING 10 :RIPPED T))
```

It works. And the `where` macro with its two helper functions is actually one line shorter than the old `where` function. And it's more general in that it's no longer tied to the specific fields in our CD records.

Wrapping Up

Now, an interesting thing has happened. You removed duplication and made the code more efficient *and* more general at the same time. That's often the way it goes with a well-chosen macro. This makes sense because a macro is just another mechanism for creating abstractions--abstraction at the syntactic level, and abstractions are by definition more concise ways of expressing underlying generalities. Now the only code in the mini-database that's specific to CDs and the fields in them is in the `make-cd`, `prompt-for-cd`, and `add-cd` functions. In fact, our new `where` macro would work with any plist-based database.

However, this is still far from being a complete database. You can probably think of plenty of features to add, such as supporting multiple tables or more elaborate queries. In Chapter 27 we'll build an MP3 database that incorporates some of those features.

The point of this chapter was to give you a quick introduction to just a handful of Lisp's features and show how they're used to write code that's a bit more interesting than "hello, world." In the next chapter we'll begin a more systematic overview of Lisp.

---

[1]Before I proceed, however, it's crucially important that you forget anything you may know about #define-style "macros" as implemented in the C pre-processor. Lisp macros are a totally different beast.

[2]Using a global variable also has some drawbacks--for instance, you can have only one database at a time. In Chapter 27, with more of the language under your belt, you'll be ready to build a more flexible database. You'll also see, in Chapter 6, how even using a global variable is more flexible in Common Lisp than it may be in other languages.

[3]One of the coolest **`FORMAT`** directives is the `~R` directive. Ever want to know how to say a really big number in English words? Lisp knows. Evaluate this:

```
(format nil "~r" 160693804425899027554196209 2)
```

and you should get back (wrapped for legibility):

> "one octillion six hundred six septillion nine hundred thirty-eight sextillion forty-four quintillion two hundred fifty-eight quadrillion nine hundred ninety trillion two hundred seventy-five billion five hundred forty-one million nine hundred sixty-two thousand ninety-two"

[4]Windows actually understands forward slashes in filenames even though it normally uses a backslash as the directory separator. This is convenient since otherwise you have to write double backslashes because backslash is the escape character in Lisp strings.

[5]The word lambda is used in Lisp because of an early connection to the lambda calculus, a mathematical formalism invented for studying mathematical functions.

[6]The technical term for a function that references a variable in its enclosing scope is a closure because the function "closes over" the variable. I'll discuss closures in more detail in Chapter 6.

[7]Note that in Lisp, an IF form, like everything else, is an expression that returns a value. It's actually more like the ternary operator (`?:`) in Perl, Java, and C in that this is legal in those languages:

```
some_var = some_boolean ? value1 : value2;
```

while this isn't:

```
some_var = if (some_boolean) value1; else value2;
```

because in those languages, `if` is a statement, not an expression.

[8]You need to use the name `delete-rows` rather than the more obvious `delete` because there's already a function in Common Lisp called **`DELETE`**. The Lisp package system gives you a way to deal with such naming conflicts, so you could have a function named delete if you wanted. But I'm not ready to explain packages just yet.

[9]If you're worried that this code creates a memory leak, rest assured: Lisp was the language that invented garbage collection (and heap allocation for that matter). The memory used by the old value of `*db*` will be automatically reclaimed, assuming no one else is holding on to a reference to it, which none of this code is.

[10]A friend of mine was once interviewing an engineer for a programming job and asked him a typical interview question: how do you know when a function or method is too big? Well, said the candidate, I don't like any method to be bigger than my head. You mean you can't keep all the details in your head? No, I mean I put my head up against my monitor, and the code shouldn't be bigger than my head.

[11]It's unlikely that the cost of checking whether keyword parameters had been passed would be a detectible drag on performance since checking whether a variable is `NIL` is going to be pretty cheap. On the other hand, these functions returned by `where` are going to be right in the middle of the inner loop of any `select`, `update`, or `delete-rows` call, as they have to be called once per entry in the database. Anyway, for illustrative purposes, this will have to do.

[12]Macros are also run by the interpreter--however, it's easier to understand the point of macros when you think about compiled code. As with everything else in this chapter, I'll cover this in greater detail in future chapters.