6. Variables

The next basic building block we need to look at are variables. Common Lisp supports two kinds of variables: lexical and dynamic.[1] These two types correspond roughly to "local" and "global" variables in other languages. However, the correspondence is only approximate. On one hand, some languages' "local" variables are in fact much like Common Lisp's dynamic variables.[2] And on the other, some languages' local variables are *lexically scoped* without providing all the capabilities provided by Common Lisp's lexical variables. In particular, not all languages that provide lexically scoped variables support closures.

To make matters a bit more confusing, many of the forms that deal with variables can be used with both lexical and dynamic variables. So I'll start by discussing a few aspects of Lisp's variables that apply to both kinds and then cover the specific characteristics of lexical and dynamic variables. Then I'll discuss Common Lisp's general-purpose assignment operator, **`SETF`**, which is used to assign new values to variables and just about every other place that can hold a value.

Variable Basics

As in other languages, in Common Lisp variables are named places that can hold a value. However, in Common Lisp, variables aren't typed the way they are in languages such as Java or C++. That is, you don't need to declare the type of object that each variable can hold. Instead, a variable can hold values of any type and the values carry type information that can be used to check types at runtime. Thus, Common Lisp is *dynamically typed*--type errors are detected dynamically. For instance, if you pass something other than a number to the **`+`** function, Common Lisp will signal a type error. On the other hand, Common Lisp *is* a *strongly typed* language in the sense that all type errors will be detected--there's no way to treat an object as an instance of a class that it's not.[3]

All values in Common Lisp are, conceptually at least, references to objects.[4] Consequently, assigning a variable a new value changes *what* object the variable refers to but has no effect on the previously referenced object. However, if a variable holds a reference to a mutable object, you can use that reference to modify the object, and the modification will be visible to any code that has a reference to the same object.

One way to introduce new variables you've already used is to define function parameters. As you saw in the previous chapter, when you define a function with **`DEFUN`**, the parameter list defines the variables that will hold the function's arguments when it's called. For example, this function defines three variables--`x`, `y`, and `z`--to hold its arguments.

```
(defun foo (x y z) (+ x y z))
```

Each time a function is called, Lisp creates new *bindings* to hold the arguments passed by the function's caller. A binding is the runtime manifestation of a variable. A single variable--the thing you can point to in the program's source code--can have many different bindings during a run of the program. A single variable can even have multiple bindings at the same time; parameters to a recursive function, for example, are rebound for each call to the function.

As with all Common Lisp variables, function parameters hold object references.[5] Thus, you can assign a new value to a function parameter within the body of the function, and it will not affect the bindings created for another call to the same function. But if the object passed to a function is mutable and you change it in the function, the changes will be visible to the caller since both the caller and the callee will be referencing the same object.

Another form that introduces new variables is the **`LET`** special operator. The skeleton of a **`LET`** form looks like this:

```
(let (variable*)
  body-form*)
```

where each *variable* is a variable initialization form. Each initialization form is either a list containing a variable name and an initial value form or--as a shorthand for initializing the variable to **`NIL`**--a plain variable name. The following **`LET`** form, for example, binds the three variables `x`, `y`, and `z` with initial values 10, 20, and **`NIL`**:

```
(let ((x 10) (y 20) z)
  ...)
```

When the **`LET`** form is evaluated, all the initial value forms are first evaluated. Then new bindings are created and initialized to the appropriate initial values before the body forms are executed. Within the body of the **`LET`**, the variable names refer to the newly created bindings. After the **`LET`**, the names refer to whatever, if anything, they referred to before the **`LET`**.

The value of the last expression in the body is returned as the value of the **`LET`** expression. Like function parameters, variables introduced with **`LET`** are rebound each time the **`LET`** is entered.[6]

The *scope* of function parameters and **`LET`** variables--the area of the program where the variable name can be used to refer to the variable's binding--is delimited by the form that introduces the variable. This form--the function definition or the **`LET`**--is called the *binding form*. As you'll see in a bit, the two types of variables--lexical and dynamic--use two slightly different scoping mechanisms, but in both cases the scope is delimited by the binding form.

If you nest binding forms that introduce variables with the same name, then the bindings of the innermost variable *shadows* the outer bindings. For instance, when the following function is called, a binding is created for the parameter `x` to hold the function's argument. Then the first **`LET`** creates a new binding with the initial value 2, and the inner **`LET`** creates yet another binding, this one with the initial value 3. The bars on the right mark the scope of each binding.

```
(defun foo (x)
  (format t "Parameter: ~a~%" x)      ; |<------ x is argument
  (let ((x 2))                        ; |
    (format t "Outer LET: ~a~%" x)    ; | |<---- x is 2
    (let ((x 3))                      ; | |
      (format t "Inner LET: ~a~%" x)) ; | | |<-- x is 3
    (format t "Outer LET: ~a~%" x))   ; | |
  (format t "Parameter: ~a~%" x))     ; |
```

Each reference to `x` will refer to the binding with the smallest enclosing scope. Once control leaves the scope of one binding form, the binding from the immediately enclosing scope is unshadowed and `x` refers to it instead. Thus, calling `foo` results in this output:

```
CL-USER> (foo 1)
Parameter: 1
Outer LET: 2
Inner LET: 3
Outer LET: 2
Parameter: 1
NIL
```

In future chapters I'll discuss other constructs that also serve as binding forms--any construct that introduces a new variable name that's usable only within the construct is a binding form.

For instance, in Chapter 7 you'll meet the **`DOTIMES`** loop, a basic counting loop. It introduces a variable that holds the value of a counter that's incremented each time through the loop. The following loop, for example, which prints the numbers from 0 to 9, binds the variable `x`:

```
(dotimes (x 10) (format t "~d " x))
```

Another binding form is a variant of **`LET`**, **`LET*`**. The difference is that in a **`LET`**, the variable names can be used only in the body of the **`LET`**--the part of the **`LET`** after the variables list--but in a **`LET*`**, the initial value forms for each variable can refer to variables introduced earlier in the variables list. Thus, you can write the following:

```
(let* ((x 10)
       (y (+ x 10)))
  (list x y))
```

but not this:

```
(let ((x 10)
      (y (+ x 10)))
  (list x y))
```

However, you could achieve the same result with nested **`LET`**s.

```
(let ((x 10))
  (let ((y (+ x 10)))
    (list x y)))
```

Lexical Variables and Closures

By default all binding forms in Common Lisp introduce *lexically scoped* variables. Lexically scoped variables can be referred to only by code that's textually within the binding form. Lexical scoping should be familiar to anyone who has programmed in Java, C, Perl, or Python since they all provide lexically scoped "local" variables. For that matter, Algol programmers should also feel right at home, as Algol first introduced lexical scoping in the 1960s.

However, Common Lisp's lexical variables are lexical variables with a twist, at least compared to the original Algol model. The twist is provided by the combination of lexical scoping with nested functions. By the rules of lexical scoping, only code textually within the binding form can refer to a lexical variable. But what happens when an anonymous function contains a reference to a lexical variable from an enclosing scope? For instance, in this expression:

```
(let ((count 0)) #'(lambda () (setf count (1+ count))))
```

the reference to `count` inside the **`LAMBDA`** form should be legal according to the rules of lexical scoping. Yet the anonymous function containing the reference will be returned as the value of the **`LET`** form and can be invoked, via **`FUNCALL`**, by code that's *not* in the scope of the **`LET`**. So what happens? As it turns out, when `count` is a lexical variable, it just works. The binding of `count` created when the flow of control entered the **`LET`** form will stick around for as long as needed, in this case for as long as someone holds onto a reference to the function object returned by the **`LET`** form. The anonymous function is called a *closure* because it "closes over" the binding created by the **`LET`**.

The key thing to understand about closures is that it's the binding, not the value of the variable, that's captured. Thus, a closure can not only access the value of the variables it closes over but can also assign new values that will persist between calls to the closure. For instance, you can capture the closure created by the previous expression in a global variable like this:

```
(defparameter *fn* (let ((count 0)) #'(lambda () (setf count (1+ count)))))
```

Then each time you invoke it, the value of count will increase by one.

```
CL-USER> (funcall *fn*)
1
CL-USER> (funcall *fn*)
2
CL-USER> (funcall *fn*)
3
```

A single closure can close over many variable bindings simply by referring to them. Or multiple closures can capture the same binding. For instance, the following expression returns a list of three closures, one that increments the value of the closed over `count` binding, one that decrements it, and one that returns the current value:

```
(let ((count 0))
  (list
   #'(lambda () (incf count))
   #'(lambda () (decf count))
   #'(lambda () count)))
```

Dynamic, a.k.a. Special, Variables

Lexically scoped bindings help keep code understandable by limiting the scope, literally, in which a given name has meaning. This is why most modern languages use lexical scoping for local variables. Sometimes, however, you really want a global variable--a variable that you can refer to from anywhere in your program. While it's true that indiscriminate use of global variables can turn code into spaghetti nearly as quickly as unrestrained use of `goto`, global variables do have legitimate uses and exist in one form or another in almost every programming language.[7] And as you'll see in a moment, Lisp's version of global variables, dynamic variables, are both more useful and more manageable.

Common Lisp provides two ways to create global variables: **DEFVAR** and **DEFPARAMETER**. Both forms take a variable name, an initial value, and an optional documentation string. After it has been **DEFVAR**ed or **DEFPARAMETER**ed, the name can be used anywhere to refer to the current binding of the global variable. As you've seen in previous chapters, global variables are conventionally named with names that start and end with `*`. You'll see later in this section why it's quite important to follow that naming convention. Examples of **DEFVAR** and **DEFPARAMETER** look like this:

```
(defvar *count* 0
  "Count of widgets made so far.")

(defparameter *gap-tolerance* 0.001
  "Tolerance to be allowed in widget gaps.")
```

The difference between the two forms is that **DEFPARAMETER** always assigns the initial value to the named variable while **DEFVAR** does so only if the variable is undefined. A **DEFVAR** form can also be used with no initial value to define a global variable without giving it a value. Such a variable is said to be *unbound*.

Practically speaking, you should use **DEFVAR** to define variables that will contain data you'd want to keep even if you made a change to the source code that uses the variable. For instance, suppose the two variables defined previously are part of an application for controlling a widget factory. It's appropriate to define the `*count*` variable with **DEFVAR** because the number of widgets made so far isn't invalidated just because you make some changes to the widget-making code.[8]

On the other hand, the variable `*gap-tolerance*` presumably has some effect on the behavior of the widget-making code itself. If you decide you need a tighter or looser tolerance and change the value in the **`DEFPARAMETER`** form, you'd like the change to take effect when you recompile and reload the file.

After defining a variable with **`DEFVAR`** or **`DEFPARAMETER`**, you can refer to it from anywhere. For instance, you might define this function to increment the count of widgets made:

```
(defun increment-widget-count () (incf *count*))
```

The advantage of global variables is that you don't have to pass them around. Most languages store the standard input and output streams in global variables for exactly this reason--you never know when you're going to want to print something to standard out, and you don't want every function to have to accept and pass on arguments containing those streams just in case someone further down the line needs them.

However, once a value, such as the standard output stream, is stored in a global variable and you have written code that references that global variable, it's tempting to try to temporarily modify the behavior of that code by changing the variable's value.

For instance, suppose you're working on a program that contains some low-level logging functions that print to the stream in the global variable `*standard-output*`. Now suppose that in part of the program you want to capture all the output generated by those functions into a file. You might open a file and assign the resulting stream to `*standard-output*`. Now the low-level functions will send their output to the file.

This works fine until you forget to set `*standard-output*` back to the original stream when you're done. If you forget to reset `*standard-output*`, all the other code in the program that uses `*standard-output*` will also send its output to the file.[9]

What you really want, it seems, is a way to wrap a piece of code in something that says, "All code below here--all the functions it calls, all the functions they call, and so on, down to the lowest-level functions--should use *this* value for the global variable `*standard-output*`." Then when the high-level function returns, the old value of `*standard-output*` should be automatically restored.

It turns out that that's exactly what Common Lisp's other kind of variable--dynamic variables--let you do. When you bind a dynamic variable--for example, with a **`LET`** variable or a function parameter--the binding that's created on entry to the binding form replaces the global binding for the duration of the binding form. Unlike a lexical binding, which can be referenced by code only within the lexical scope of the binding form, a dynamic binding can be referenced by any code that's invoked during the execution of the binding form.[10] And it turns out that all global variables are, in fact, dynamic variables.

Thus, if you want to temporarily redefine `*standard-output*`, the way to do it is simply to rebind it, say, with a **`LET`**.

```
(let ((*standard-output* *some-other-stream*))
  (stuff))
```

In any code that runs as a result of the call to `stuff`, references to `*standard-output*` will use the binding established by the **`LET`**. And when `stuff` returns and control leaves the **`LET`**, the new binding of `*standard-output*` will go away and subsequent references to `*standard-output*` will see the binding that was current before the **`LET`**. At any given time, the most recently established binding shadows all other bindings. Conceptually, each new binding for a given dynamic variable is pushed onto a stack of bindings for that variable, and references to the variable always use the most recent binding. As binding forms return, the bindings they created are popped off the stack, exposing previous bindings.[11]

A simple example shows how this works.

```
(defvar *x* 10)
(defun foo () (format t "X: ~d~%" *x*))
```

The **`DEFVAR`** creates a global binding for the variable `*x*` with the value 10. The reference to `*x*` in `foo` will look up the current binding dynamically. If you call `foo` from the top level, the global binding created by the **`DEFVAR`** is the only binding available, so it prints 10.

```
CL-USER> (foo)
X: 10
NIL
```

But you can use **`LET`** to create a new binding that temporarily shadows the global binding, and `foo` will print a different value.

```
CL-USER> (let ((*x* 20)) (foo))
X: 20
NIL
```

Now call `foo` again, with no **`LET`**, and it again sees the global binding.

```
CL-USER> (foo)
X: 10
NIL
```

Now define another function.

```
(defun bar ()
  (foo)
  (let ((*x* 20)) (foo))
  (foo))
```

Note that the middle call to `foo` is wrapped in a **`LET`** that binds `*x*` to the new value 20. When you run `bar`, you get this result:

```
CL-USER> (bar)
X: 10
```

```
X: 20
X: 10
NIL
```

As you can see, the first call to `foo` sees the global binding, with its value of 10. The middle call, however, sees the new binding, with the value 20. But after the **`LET`**, `foo` once again sees the global binding.

As with lexical bindings, assigning a new value affects only the current binding. To see this, you can redefine `foo` to include an assignment to `*x*`.

```
(defun foo ()
  (format t "Before assignment~18tX: ~d~%" *x*)
  (setf *x* (+ 1 *x*))
  (format t "After assignment~18tX: ~d~%" *x*))
```

Now `foo` prints the value of `*x*`, increments it, and prints it again. If you just run `foo`, you'll see this:

```
CL-USER> (foo)
Before assignment X: 10
After assignment  X: 11
NIL
```

Not too surprising. Now run `bar`.

```
CL-USER> (bar)
Before assignment X: 11
After assignment  X: 12
Before assignment X: 20
After assignment  X: 21
Before assignment X: 12
After assignment  X: 13
NIL
```

Notice that `*x*` started at 11--the earlier call to `foo` really did change the global value. The first call to `foo` from `bar` increments the global binding to 12. The middle call doesn't see the global binding because of the **`LET`**. Then the last call can see the global binding again and increments it from 12 to 13.

So how does this work? How does **`LET`** know that when it binds `*x*` it's supposed to create a dynamic binding rather than a normal lexical binding? It knows because the name has been declared *special*.[12] The name of every variable defined with **`DEFVAR`** and **`DEFPARAMETER`** is automatically declared globally special. This means whenever you use such a name in a binding form--in a **`LET`** or as a function parameter or any other construct that creates a new variable binding--the binding that's created will be a dynamic binding. This is why the `*naming* *convention*` is so important--it'd be bad news if you used a name for what you thought was a lexical variable and that variable happened to be globally special. On the one hand, code you call could change the value of the binding out from under you; on the other, you might be shadowing a binding established by code higher up on the stack. If you always name global variables according to the * naming convention, you'll never accidentally use a dynamic binding where you intend to establish a lexical binding.

It's also possible to declare a name locally special. If, in a binding form, you declare a name special, then the binding created for that variable will be dynamic rather than lexical. Other code can locally declare a name special in order to refer to the dynamic binding. However, locally special variables are relatively rare, so you needn't worry about them.[13]

Dynamic bindings make global variables much more manageable, but it's important to notice they still allow action at a distance. Binding a global variable has two at a distance effects--it can change the behavior of downstream code, and it also opens the possibility that downstream code will assign a new value to a binding established higher up on the stack. You should use dynamic variables only when you need to take advantage of one or both of these characteristics.

Constants

One other kind of variable I haven't mentioned at all is the oxymoronic "constant variable." All constants are global and are defined with **DEFCONSTANT**. The basic form of **DEFCONSTANT** is like **DEFPARAMETER**.

```
(defconstant name initial-value-form [ documentation-string ])
```

As with **DEFVAR** and **DEFPARAMETER**, **DEFCONSTANT** has a global effect on the name used--thereafter the name can be used only to refer to the constant; it can't be used as a function parameter or rebound with any other binding form. Thus, many Lisp programmers follow a naming convention of using names starting and ending with + for constants. This convention is somewhat less universally followed than the *-naming convention for globally special names but is a good idea for the same reason.[14]

Another thing to note about **DEFCONSTANT** is that while the language allows you to redefine a constant by reevaluating a **DEFCONSTANT** with a different initial-value-form, what exactly happens after the redefinition isn't defined. In practice, most implementations will require you to reevaluate any code that refers to the constant in order to see the new value since the old value may well have been inlined. Consequently, it's a good idea to use **DEFCONSTANT** only to define things that are *really* constant, such as the value of NIL. For things you might ever want to change, you should use **DEFPARAMETER** instead.

Assignment

Once you've created a binding, you can do two things with it: get the current value and set it to a new value. As you saw in Chapter 4, a symbol evaluates to the value of the variable it names, so you can get the current value simply by referring to the variable. To assign a new value to a binding, you use the **SETF** macro, Common Lisp's general-purpose assignment operator. The basic form of **SETF** is as follows:

```
(setf place value)
```

Because **SETF** is a macro, it can examine the form of the *place* it's assigning to and expand into appropriate lower-level operations to manipulate that place. When the place is a variable, it expands into a call to the special operator **SETQ**, which, as a special operator, has access to both lexical and dynamic bindings.[15] For instance, to assign the value 10 to the variable `x`, you can write this:

```
(setf x 10)
```

As I discussed earlier, assigning a new value to a binding has no effect on any other bindings of that variable. And it doesn't have any effect on the value that was stored in the binding prior to the assignment. Thus, the **SETF** in this function:

```
(defun foo (x) (setf x 10))
```

will have no effect on any value outside of `foo`. The binding that was created when `foo` was called is set to 10, immediately replacing whatever value was passed as an argument. In particular, a form such as the following:

```
(let ((y 20))
  (foo y)
  (print y))
```

will print 20, not 10, as it's the value of `y` that's passed to `foo` where it's briefly the value of the variable `x` before the **SETF** gives `x` a new value.

SETF can also assign to multiple places in sequence. For instance, instead of the following:

```
(setf x 1)
(setf y 2)
```

you can write this:

```
(setf x 1 y 2)
```

SETF returns the newly assigned value, so you can also nest calls to **SETF** as in the following expression, which assigns both `x` and `y` the same random value:

```
(setf x (setf y (random 10)))
```

Generalized Assignment

Variable bindings, of course, aren't the only places that can hold values. Common Lisp supports composite data structures such as arrays, hash tables, and lists, as well as user-defined data structures, all of which consist of multiple places that can each hold a value.

I'll cover those data structures in future chapters, but while we're on the topic of assignment, you should note that **SETF** can assign any place a value. As I cover the different composite data structures, I'll point out which functions can serve as "**SETF**able places." The short version, however, is if you need to assign a value to a place, **SETF** is almost certainly the tool to use. It's even possible to extend **SETF** to allow it to assign to user-defined places though I won't cover that.[16]

In this regard **SETF** is no different from the = assignment operator in most C-derived languages. In those languages, the = operator assigns new values to variables, array elements, and fields of classes. In languages such as Perl and Python that support hash tables as a built-in data type, = can also set the values of individual hash table entries. Table 6-1 summarizes the various ways = is used in those languages.

Table 6-1. Assignment with = in Other Languages

Assigning to ...	Java, C, C++	Perl	Python
... variable	`x = 10;`	`$x = 10;`	`x = 10`
... array element	`a[0] = 10;`	`$a[0] = 10;`	`a[0] = 10`
... hash table entry	`--`	`$hash{'key'} = 10;`	`hash['key'] = 10`
... field in object	`o.field = 10;`	`$o->{'field'} = 10;`	`o.field = 10`

SETF works the same way--the first "argument" to **SETF** is a place to store the value, and the second argument provides the value. As with the = operator in these languages, you use the same form to express the place as you'd normally use to fetch the value.[17] Thus, the Lisp equivalents of the assignments in Table 6-1--given that **AREF** is the array access function, **GETHASH** does a hash table lookup, and `field` might be a function that accesses a slot named `field` of a user-defined object--are as follows:

```
Simple variable:    (setf x 10)
Array:              (setf (aref a 0) 10)
Hash table:         (setf (gethash 'key hash) 10)
Slot named 'field': (setf (field o) 10)
```

Note that **SETF**ing a place that's part of a larger object has the same semantics as **SETF**ing a variable: the place is modified without any effect on the object that was previously stored in the place. Again, this is similar to how **=** behaves in Java, Perl, and Python.[18]

Other Ways to Modify Places

While all assignments can be expressed with **SETF**, certain patterns involving assigning a new value based on the current value are sufficiently common to warrant their own operators. For instance, while you could increment a number with **SETF**, like this:

```
(setf x (+ x 1))
```

or decrement it with this:

```
(setf x (- x 1))
```

it's a bit tedious, compared to the C-style `++x` and `--x`. Instead, you can use the macros **`INCF`** and **`DECF`**, which increment and decrement a place by a certain amount that defaults to 1.

```
(incf x)    === (setf x (+ x 1))
(decf x)    === (setf x (- x 1))
(incf x 10) === (setf x (+ x 10))
```

`INCF` and **`DECF`** are examples of a kind of macro called *modify macros*. Modify macros are macros built on top of **`SETF`** that modify places by assigning a new value based on the current value of the place. The main benefit of modify macros is that they're more concise than the same modification written out using **`SETF`**. Additionally, modify macros are defined in a way that makes them safe to use with places where the place expression must be evaluated only once. A silly example is this expression, which increments the value of an arbitrary element of an array:

```
(incf (aref *array* (random (length *array*))))
```

A naive translation of that into a **`SETF`** expression might look like this:

```
(setf (aref *array* (random (length *array*)))
      (1+ (aref *array* (random (length *array*)))))
```

However, that doesn't work because the two calls to **`RANDOM`** won't necessarily return the same value--this expression will likely grab the value of one element of the array, increment it, and then store it back as the new value of a different element. The **`INCF`** expression, however, does the right thing because it knows how to take apart this expression:

```
(aref *array* (random (length *array*)))
```

to pull out the parts that could possibly have side effects to make sure they're evaluated only once. In this case, it would probably expand into something more or less equivalent to this:

```
(let ((tmp (random (length *array*))))
  (setf (aref *array* tmp) (1+ (aref *array* tmp))))
```

In general, modify macros are guaranteed to evaluate both their arguments and the subforms of the place form exactly once each, in left-to-right order.

The macro **`PUSH`**, which you used in the mini-database to add elements to the `*db*` variable, is another modify macro. You'll take a closer look at how it and its counterparts **`POP`** and **`PUSHNEW`** work in Chapter 12 when I talk about how lists are represented in Lisp.

Finally, two slightly esoteric but useful modify macros are **`ROTATEF`** and **`SHIFTF`**. **`ROTATEF`** rotates values between places. For instance, if you have two variables, `a` and `b`, this call:

```
(rotatef a b)
```

swaps the values of the two variables and returns **`NIL`**. Since `a` and `b` are variables and you don't have to worry about side effects, the previous **`ROTATEF`** expression is equivalent to this:

```
(let ((tmp a)) (setf a b b tmp) nil)
```

With other kinds of places, the equivalent expression using **SETF** would be quite a bit more complex.

SHIFTF is similar except instead of rotating values it shifts them to the left--the last argument provides a value that's moved to the second-to-last argument while the rest of the values are moved one to the left. The original value of the first argument is simply returned. Thus, the following:

```
(shiftf a b 10)
```

is equivalent--again, since you don't have to worry about side effects--to this:

```
(let ((tmp a)) (setf a b b 10) tmp)
```

Both **ROTATEF** and **SHIFTF** can be used with any number of arguments and, like all modify macros, are guaranteed to evaluate them exactly once, in left to right order.

With the basics of Common Lisp's functions and variables under your belt, now you're ready to move onto the feature that continues to differentiate Lisp from other languages: macros.

---

[1]Dynamic variables are also sometimes called *special variables* for reasons you'll see later in this chapter. It's important to be aware of this synonym, as some folks (and Lisp implementations) use one term while others use the other.

[2]Early Lisps tended to use dynamic variables for local variables, at least when interpreted. Elisp, the Lisp dialect used in Emacs, is a bit of a throwback in this respect, continuing to support only dynamic variables. Other languages have recapitulated this transition from dynamic to lexical variables--Perl's `local` variables, for instance, are dynamic while its `my` variables, introduced in Perl 5, are lexical. Python never had true dynamic variables but only introduced true lexical scoping in version 2.2. (Python's lexical variables are still somewhat limited compared to Lisp's because of the conflation of assignment and binding in the language's syntax.)

[3]Actually, it's not quite true to say that all type errors will always be detected--it's possible to use optional declarations to tell the compiler that certain variables will always contain objects of a particular type and to turn off runtime type checking in certain regions of code. However, declarations of this sort are used to optimize code after it has been developed and debugged, not during normal development.

[4]As an optimization certain kinds of objects, such as integers below a certain size and characters, may be represented directly in memory where other objects would be represented by a pointer to the actual object. However, since integers and characters are immutable, it doesn't matter that there may be multiple copies of "the same" object in different variables. This is the root of the difference between **EQ** and **EQL** discussed in Chapter 4.

[5]In compiler-writer terms Common Lisp functions are "pass-by-value." However, the values that are passed are references to objects. This is similar to how Java and Python work.

[6]The variables in **LET** forms and function parameters are created by exactly the same mechanism. In fact, in some Lisp dialects--though not Common Lisp--**LET** is simply a macro that expands into a call to an anonymous function. That is, in those dialects, the following:

```
(let ((x 10)) (format t "~a" x))
```

is a macro form that expands into this:

```
((lambda (x) (format t "~a" x)) 10)
```

[7]Java disguises global variables as public static fields, C uses `extern` variables, and Python's module-level and Perl's package-level variables can likewise be accessed from anywhere.

[8]If you specifically want to reset a **`DEFVAR`**ed variable, you can either set it directly with **`SETF`** or make it unbound using **`MAKUNBOUND`** and then reevaluate the **`DEFVAR`** form.

[9]The strategy of temporarily reassigning *standard-output* also breaks if the system is multithreaded--if there are multiple threads of control trying to print to different streams at the same time, they'll all try to set the global variable to the stream they want to use, stomping all over each other. You could use a lock to control access to the global variable, but then you're not really getting the benefit of multiple concurrent threads, since whatever thread is printing has to lock out all the other threads until it's done even if they want to print to a different stream.

[10]The technical term for the interval during which references may be made to a binding is its *extent*. Thus, *scope* and *extent* are complementary notions--scope refers to space while extent refers to time. Lexical variables have lexical scope but *indefinite* extent, meaning they stick around for an indefinite interval, determined by how long they're needed. Dynamic variables, by contrast, have indefinite scope since they can be referred to from anywhere but *dynamic* extent. To further confuse matters, the combination of indefinite scope and dynamic extent is frequently referred to by the misnomer *dynamic scope*.

[11]Though the standard doesn't specify how to incorporate multithreading into Common Lisp, implementations that provide multithreading follow the practice established on the Lisp machines and create dynamic bindings on a per-thread basis. A reference to a global variable will find the binding most recently established in the current thread, or the global binding.

[12]This is why dynamic variables are also sometimes called *special variables*.

[13]If you must know, you can look up **`DECLARE`**, **`SPECIAL`**, and **`LOCALLY`** in the HyperSpec.

[14]Several key constants defined by the language itself don't follow this convention--not least of which are **`T`** and **`NIL`**. This is occasionally annoying when one wants to use `t` as a local variable name. Another is **`PI`**, which holds the best long-float approximation of the mathematical constant pi.

[15]Some old-school Lispers prefer to use **`SETQ`** with variables, but modern style tends to use **`SETF`** for all assignments.

[16]Look up **`DEFSETF`**, **`DEFINE-SETF-EXPANDER`** for more information.

[17]The prevalence of Algol-derived syntax for assignment with the "place" on the left side of the = and the new value on the right side has spawned the terminology *lvalue*, short for "left value," meaning something that can be assigned to, and *rvalue*, meaning something that provides a value. A compiler hacker would say, "**`SETF`** treats its first argument as an lvalue."

[18]C programmers may want to think of variables and other places as holding a pointer to the real object; assigning to a variable simply changes what object it points to while assigning to a part of a composite object is similar to indirecting through the pointer to the actual object. C++ programmers should note that the behavior of = in C++ when dealing with objects--namely, a memberwise copy--is quite idiosyncratic.