The Idiot's Guide to Special Variables and Lexical Closures[1]

Erann Gat

Copyright © 2003 by the author. Permission is hereby granted for non-commercial use provided this notice is retained.

Variables and bindings

To understand special variables we have to start by understanding variables in general.

A *variable* is a *place* that is denoted by an *identifier* where a *value* can be stored. In some programming languages (like early dialects of BASIC and FORTRAN) there is a one-to-one-correspondence between identifiers and variables. But in most programming languages the same name can used to denote more than one variable. For example, in C:

```
{
  int x = 1;  // Create a variable named X
  x = 2;      // Change the value of X ...
  {
    float x = 3.1;  // Create a second variable also named X
    x = 4.2;        // Change the value of that second variable
    ...
  }
}
```

The phrase "the variable named X" means the same thing as "the variable denoted by the identifier X".

Note that there is a difference between *creating* a variable for the first time, and *changing the value* stored in a variable that already exists. Some languages (like Python) hide this distinction, using the same syntax for both operations. While this may seem convenient at first glance, it is generally considered a bad idea because it makes it impossible to tell the difference between the creation of a new variable and an assignment to an existing variable that contains a typographical error. This in turn often leads to hard-to-find bugs.

The association between an identifier and a place to store values is called a *binding*. In the example above there are two bindings for the identifier X.

---

[1] Version 4, July 2003

Sometimes the term “binding” is used to refer to the storage location itself rather than the association between the identifier and the storage location. This is not strictly correct, but rarely leads to confusion. The important thing to keep in mind is that there is a distinction between the *storage location* (or the binding) and the *value* stored in that storage location.

Aside

(This sub-section contains some extra tidbits about C variables, and can safely be skipped.)

In C, a variable's binding is *not* the same thing as its address. A binding is a storage location. An address is a *pointer*, which is a first-class data object in C. Pointers can be passed as arguments to functions, and returned as values from functions. Bindings are not first-class data objects.

Also, the same physical storage location can be used for different bindings at different times. For example:

```
int *p1, *p2;
{
  int x = 1;
  p1 = &x;
}
{
  int y = 2;
  p2 = &y;
}
```

At the end of this code snippet, p1 and p2 might be equal despite the fact that each one pointed to a different variable. (Note that pointers cause trouble in C precisely because they seem to correspond to bindings but they really don't. The language therefore places a burden on the programmer to track the correspondence between pointers and bindings, and when the programmer gets it wrong, which is easy to do, the result is invariably disastrous.)

Example:

To illustrate the difference between values and bindings, consider the following code:

```
(let ((x 1))    ; Establish a binding for the identifier X
  (print x)
  (let ((x 2))  ; Establish a second binding for X
    (print x)
    (setq x 3)  ; Change the value of the second binding
    (print x))
  (print x))    ; The value of the first binding is unchanged
```

This will print "1 2 3 1". The SETQ changes the value of the second (or inner) binding of X, not the first (or outer) binding.

Notation:

Because talking about multiple bindings for the same identifier can get confusing, I'm going to adopt the following notation: **X*n** means the n^{th} binding of identifier X. Using this notation we can describe the above code as follows: the first LET form establishes binding X*1. The second LET form established binding X*2. The SETQ changes the value of X*2 but not X*1.

(Actually, LET establishes new bindings every time it is run, so the first time you run the above code you get X*1 and X*2. The second time you get X*3 and X*4, and so on. Some forms establish bindings at compile time, not run time, so you get the same binding every time the code is run.)

Value bindings and function bindings

In C (and in Scheme) an identifier can have only one binding at a time. In Common Lisp an identifier can have many different kinds of bindings simultaneously. For example:

```
(let ((x 1))                ; Create a variable named X
  (flet ((x (y) (+ x y)))   ; Create a function named X
    (x x)))                 ; Call the function X on the variable X
```

Here the name X is associated simultaneously with a value and a function. To distinguish them we speak of X's *value binding* and X's *function binding*. (Common Lisp identifiers have other kinds of bindings as well.)

This is where things start to get a little confusing. Remember that there is a difference between a binding (the memory location) and its value (what is stored there). So there is a difference between X's function binding and the VALUE of X's function binding. Likewise, there is a difference between X's value binding and the value of X's value binding.

In Common Lisp, the phrase "the value of X" means "the value of the value binding of the identifier X."

Fortunately, the concept of special variables (which is the topic at hand in case you'd forgotten) deals only with value bindings, so we can forget about all the other bindings that exist in Common Lisp and just use the term "binding" from here on out to mean "value binding". The "value of a binding" then means the "value of the value binding", that is, the contents of the memory location where the value of the variable is stored.

Scope

The *scope* of a binding is the parts of the program from which that binding is *visible*, that is, from which the value of the binding can be accessed. For example, the scope of a binding established by a LET is the body of the LET, e.g.:

```
(defun foo()
  (let ((x 1) (y 2))          ; Establish binding X*1 and Y*1
    (frotz x y)               ; X refers to X*1, Y refers to Y*1
    (let ((y 2) (z 3))        ; Establish binding Y*2 and Z*1
      (baz x y z))            ; Refers to X*1, Y*2 and Z*1.
```

```
                              ;  Y*1 cannot be accessed here.
                              ;  It is SHADOWED by Y*2.
      z))                     ; This does NOT refer to Z*1.
                              ;  Z*1 is out of scope.
```

What about that Z in the last line? Z is what is referred to as a *free variable*. It is a variable which has no *lexically apparent binding*, that is, it has no binding established by its surrounding code. Normally this would mean that when we call FOO we would get an error:

```
(foo) --> ERROR: Z is unbound
```

But we can create a binding for Z using, for example, the DEFVAR special form:

```
(defvar z 3)  ; Establish binding Z*D1
```

Now FOO no longer generates an error:

```
(foo) --> 3   ; Refers to the value of Z*D1
```

Note that the binding established by DEFVAR behaves differently from a binding established by LET: bindings established by LET go away when the LET form returns[2]. By contrast, a binding established by DEFVAR continues to exist after the DEFVAR returns. Also, whether a particular reference to Z refers to the binding established by DEFVAR depends on the order in which the code runs: if the reference to Z happens before the DEFVAR then it's a error. If it happens afterwards, then it refers to the binding established by the DEFVAR.

DEFVAR does in fact establish a different sort of binding, known as a DYNAMIC binding, which is why I referred to it as Z*D1 instead of just Z*1. But we're getting ahead of ourselves again.

Lexical versus dynamic scope

Now consider the following code:

```
(defvar z 1)      ; Establish binding Z*D1

(defun foo () z)

(defun baz ()
  (let ((z 2))    ; Establish binding Z*2
    (foo)))
```

What does (BAZ) return? Well, it returns the result of calling FOO, which returns the value of Z, but which one? There are two different bindings of Z at this point - one established by the DEFVAR (Z*D1) and one established by the LET (Z*2).

---

[2] It is possible to stop these bindings from "going away". That's what lexical closures do. But we're getting ahead of ourselves.

Which binding is in scope when FOO is called from inside BAZ? Does Z*2 shadow Z*D1 in the same way that Y*2 shadowed Y*1?

The rule for LET bindings is that their scope is the body of the LET, but that can mean two different things. It can mean the body of the LET as defined by the *code*, or the body of the LET as defined by the *execution path*. The reference to Z is inside the body of the LET according to the execution path, but not according to the structure of the code (since the code for FOO is outside the code for BAZ).

(IMPORTANT: Re-read the preceding paragraph until you're sure you understand it. It's the key to understanding everything else.)

It turns out that you have a choice. Common Lisp provides both scoping disciplines. If the scope is defined by the execution path that is known as *dynamic scope*. If the scope is defined by the structure of the code that is known as *lexical scope*.[3]

How do you choose between dynamic and lexical scope? Unfortunately, this is yet another point where things get a little complicated in Common Lisp. In order to ease into the explanation I'm going to first illustrate with a hypothetical language (which I'm told resembles ISLISP, though I couldn't say since I don't know ISLISP). In this hypothetical language, there are two LET forms, L-LET and D-LET. L-LET always introduced lexical bindings (i.e. bindings whose scope is the lexical body of the L-LET) and D-LET always introduces dynamic bindings (that is, bindings whose scope is *any* code that runs inside the body of the D-LET regardless of where it is defined). For example:

```
(defvar z 1)      ; Establish binding Z*D1
(defun foo () z)  ; Return the current dynamic binding of Z
(l-let ((z 2))    ; Create Z*1, a lexical binding
  (foo)) --> 1    ;   which does not shadow Z*D1
(d-let ((z 2))    ; Create Z*D2
  (foo)) --> 2    ;   which does shadow Z*D1
```

Now consider:

```
(d-let ((z 1))
  (l-let ((z 2))
    z))
```

or

```
(l-let ((z 1))
```

---

[3] The lexical scope of a binding can always be determined at compile time, while the dynamic scope cannot in general be determined until the program runs. Because of this, it is possible at compile-time to eliminate the creation of symbols for lexical variables, and most compilers do unless you specifically ask them not to in order to make debugging easier. However, the standard does not require this behavior. People sometimes say that dynamic variables have to do with symbols while lexical variables do not. This is true only true as a result of common practice, not as a requirement of the language design.

```
(d-let ((z 2))
  z))
```

Now we have *two* bindings for Z, one lexical and one dynamic, in scope at the same time. Which one are we referring to? Well, we could allow ourselves to refer to both with hypothetical L-VAL and D-VAL constructs that allow us to refer to the values of the lexical and dynamic bindings respectively:

```
(defvar z 1)      ; Establish binding Z*D1
(defun foo () z)  ; Z must refer to the current dynamic binding
                  ; of Z because there is no lexical binding

(d-let ((z 2))
  (llet ((z 3))   ; Simply referring to Z here would be ambiguous
    (list
      (l-val 'z)  ; The value of the current lexical binding of Z
      (d-val 'z)  ; The value of the current dynamic binding of Z
      (foo)       ; and our old friend for good measure
      )))   --> (3 2 2)
```

Now the reality.

Common Lisp's LET combines the functionality of both LLET and DLET. Variable references can be either lexical or dynamic depending on the context. And there is a DVAR (called SYMBOL-VALUE[4]) but no LVAR.

By default, LET creates lexical bindings (i.e. bindings whose scope are defined by the lexical structure of the code, not by the runtime execution). To create a dynamic binding you have to use a special declaration:

```
(let ( (x 1) (y 2) )      ; Lexical bindings by default
  (declare (special x))   ;   unless you say otherwise ...)
```

In Common Lisp, SPECIAL means the same things as dynamic.

So, for example:

```
(defun baz ()
  (let ((x 2))
    (list x (symbol-value 'x))))  ; Same as
                                  ;   (list (l-val 'x) (d-val 'x))

(defun foo ()
  (let ( (x 1) )             ; Bind X
    (declare (special x))    ;   ... as a dynamic binding
```

---

[4] Note that the function is called SYMBOL-VALUE rather than IDENTIFIER-VALUE. To a first-order approximation you can think of symbols and identifiers to be the same thing. The differences between them are subtle and somewhat controversial. Again to first order, the distinction between a symbol and an identifier is that a symbol is a first-class data object at run time while an identifier is not.

```
    (baz)))

(foo) --> (2 1)
```

What happens when we call FOO is that we establish a dynamic binding for X with value 1. We then call BAZ, which establishes a lexical binding for X with value 2. The reference to X by itself returns the value of the lexical binding, while SYMBOL-VALUE gives us access to the dynamic binding.

The pervasiveness of DEFVAR

There is one final wrinkle. Consider:

```
(defvar x 1)

(defun baz () x)

(defun foo ()
  (let ((x 2))  ; Bind X, no special declaration
                ;    means it's lexical, but...
    (baz)))
```

Based on what I've told you so far you should expect (foo) to return 1. But in fact (foo) returns 2. Why? Because DEFVAR does more than just establish a dynamic binding for X. It pervasively declares all references to X and all subsequent bindings for X to be dynamic (or special -- same thing). In other words, DEFVAR turns its argument (permanently and pervasively) into a special variable.

It is actually illegal to do a top-level assignment of a variable unless it is special because Common Lisp does not have top-level lexical bindings[5]. This is why in most Common Lisp implementations if you type (SETQ X 1) without a DEFVAR you will get a warning. Strictly speaking, SETQ does assignment, that is, it modifies a binding (a symbol's value binding to be precise). But at the top level a symbol doesn't *have* a value binding until you create one with DEFVAR.

Most implementations assume that since there are no lexically apparent bindings at the top level, that any top-level assignment is intended to modify the dynamic binding, and so they will create one for you automatically if you do a SETQ without a DEFVAR. But at least one implementation (CMUCL) goes further: if you do a SETQ at the top level it will not only create a dynamic binding for that symbol, but it will also globally declare that symbol to be special just as if you had done a DEFVAR. This can lead to very surprising behavior if you don't have a deep understanding of what is going on. Once a symbol has been DEFVARd it is no longer possible to create a lexical binding for it.

There is a sneaky trick for doing a top-level assignment to a variable without risking having it defvar'd for you:

---

[5] It is possible to add top-level lexicals, but there are some subtle design issues. See http://www.nhplace.com/kent/CL/Issues/proclaim-lexical.html and http://www.flownet.com/gat/locales.pdf for more information.

```
(locally (declare (special x)) (setq x ...))
```

To avoid confusion, the following typographical convention is customarily followed: all pervasively special variables should have names that are preceded and followed by asterisks, e.g. *X*.

Lexical closures

We've now covered everything you need to know to understand lexical closures. Consider:

```
(defun foo ()
  (let ((x 1))   ; Create a (lexical) binding for X
    x            ; X is in scope here
  ))           ; X goes out of scope here
```

FOO returns the value of X, not the binding. It is not possible to return a binding because bindings are not first-class entities. However, it is possible to "capture" a binding inside of a function object and return *that* instead! For example:

```
(defun baz ()
  (let ((x 1))           ; Create a (lexical) binding for X
    (lambda ()
      (setq x (+ x 1)))  ; X is in scope here ))
```

Now when we call (baz) we get back a function object (created by the LAMBDA special form). This function object has a reference to X inside it which refers to the lexical binding created by the LET.

Note that I haven't followed my usual pattern and given this binding the name X*1. This is because a new binding is created every time BAZ is called:

```
(setq *x1* (baz)) ; Returns a closure over binding X*1
(setq *x2* (baz)) ; Returns a closure over binding X*2
(funcall *x1*) --> 2  ; Increment X*1
(funcall *x1*) --> 3  ; again
(funcall *x1*) --> 4  ; one more time
(funcall *x2*) --> 2  ; Increment X*2
```

One last example:

```
(defun up-n-down ()
  (let ((x 0))
    (list
      (lambda () (setq x (+ x 1)))
      (lambda () (setq x (- x 1))))))

(setq *l1* (up-n-down))        ; X bound to X*1
(setq *l2* (up-n-down))        ; X bound to X*2
(funcall (first *l1*)) --> 1
(funcall (first *l1*)) --> 2
(funcall (first *l1*)) --> 3
(funcall (second *l1*)) --> 2
```

```
(funcall (first *l2*)) --> 1
(funcall (first *l2*)) --> 2
(funcall (first *l2*)) --> 3
(funcall (second *l2*)) --> 2
```

By now this should all make perfect sense. When we call up-n-down we get a new binding for X (established by the LET). This binding is then captured by the LAMBDA forms. Both lambdas in the list capture the *same* binding every time up-n-down is called, and every time we call up-n-down we get a (one) new binding.

Notice how much the list returned by up-n-down resembles an object with one slot and two methods. That is in fact precisely what it is. This is why when a Lisp programmer learns a new programming language one of the first questions they will ask is, "does it have closures?" Because if you have closures you can build an OO system (and just about anything else you need) out of them. It is, alas, much harder to go the other way.