Let Over Lambda

Let Over Lambda -- 50 Years of Lisp

by Doug Hoyte

Macro Basics

Iterative Development

Lisp has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts. —Edsger Dijkstra

The construction of a macro is an iterative process: all complex macros come from simpler macros. After starting with an idea, a rough implementation can be created from which the ultimate macro will emerge, like a sculpture from a block of stone. If the rough implementation isn't flexible enough, or results in inefficient or dangerous expansions, the professional macro programmer will slightly modify the macro, adding features or removing bugs until it satisfies all requirements.

The necessity of this iterative process for macro construction is partly because this is the most efficient programming style in general and partly because programming macros is more complicated than other types of programming. Because macro programming requires the programmer to think about multiple levels of code executed at multiple points in time, the complexity issues scale more rapidly than other types of programming. An iterative process helps ensure that your conceptual model is more closely aligned to what is actually being created than if the entire macro was written without this constant feedback.

In this chapter we will write some basic macros by introducing two common macro concepts: *domain specific languages* and *control structures*. Once these general macro areas are described, we take a step back and discuss the process of writing macros itself. Techniques like variable capture and free variable injection are introduced, along with the definition of a new, slightly more convenient syntax for defining lisp macros that is used throughout the remainder of this book.

Domain Specific Languages

COMMON LISP, along with most other programming environments, provides a function **sleep** which will pause execution of the process for **n** seconds, where **n** is a non-negative, non-complex, numeric argument. For instance, we might want to sleep for 3 minutes (180 seconds), in which case we could evaluate this form:

```
(sleep 180)
```

Or if we would rather think about sleeping in terms of minutes, we could instead use

```
(sleep (* 3 60))
```

Because compilers know how to *fold constants*, these two invocations are just as efficient. To be even more explicit about what we're doing, we might define a function **sleep-minutes**:

```
(defun sleep-minutes (m)
  (sleep (* m 60)))
```

Defining new functions for every unit of time that we might want to use is clunky and inconvenient. What we would really like is some sort of abstraction that lets us specify the unit of time along with the value. What we really want is a *domain specific language*.

So far, the lisp solution is the same as that of any other language: create a function that accepts a value and a unit and returns the value multiplied by some constant related to the given unit. But a possible lispy improvement becomes apparent when we consider our options for representing this unit. In languages like C it is customary to use an underlying data type like int and assign arbitrary values corresponding to the different units:

```
#define UNIT_SECONDS 1
#define UNIT_MINUTES 2
#define UNIT_HOURS 3

int sleep_units(int value, int unit) {
  switch(value) {
    case UNIT_SECONDS: return value;
    case UNIT_MINUTES: return value*60;
    case UNIT_HOURS: return value*3600;
  }
}
```

SLEEP-UNITS-1

```
(defun sleep-units% (value unit)
  (sleep
    (* value
       (case unit
```

```
((s) 1)
((m) 60)
((h) 3600)
((d) 86400)
((ms) 1/1000)
((us) 1/1000000)))))
```

But in lisp the most obvious way to signal the desired unit is to use a symbol. A symbol in lisp exists mostly to be something not **eq** to other symbols. **Eq** is the fastest lisp comparison operator and roughly corresponds to a pointer comparison. Since pointers can be compared very quickly, symbols provide a very fast and convenient way to let two or more different lisp expressions know you're referring to the same thing. In lisp we might define the **sleep-units%** function so we can specify the units in our forms:

```
(sleep-units% 2 'm)
(sleep-units% 500 'us)
```

Because comparing symbols requires only a pointer comparison, **sleep-units%** will be compiled into a very fast run-time dispatch:

```
...
524:       CMP     ESI, [#x586FC4D0]    ; 'S
52A:       JEQ     L11
530:       CMP     ESI, [#x586FC4D4]    ; 'M
536:       JEQ     L10
538:       CMP     ESI, [#x586FC4D8]    ; 'H
53E:       JEQ     L9
540:       CMP     ESI, [#x586FC4DC]    ; 'D
546:       JEQ     L8
...
```

Notice how the unit given to **sleep-units%** must be quoted. This is because when lisp evaluates a function it first evaluates all arguments and then binds the results to variables for use inside the function. Numbers and strings and some other primitives evaluate to themselves which is why we don't need to quote the numeric values given to **sleep-units%**. But notice that they are evaluated so you are permitted to quote them if you like:

```
(sleep-units% '.5 'h)
```

Symbols, however, don't typically evaluate to themselves[1] *As a rule, no rule without exception. Some symbols do evaluate t*. When lisp evaluates a symbol it assumes you are referring to a variable and tries to look up the value associated with that variable given your lexical context (unless the variable is declared special, in which case the dynamic environment).

SLEEP-UNITS

```
(defmacro sleep-units (value unit)
  `(sleep
```

```
    (* ,value
       ,(case unit
          ((s) 1)
          ((m) 60)
          ((h) 3600)
          ((d) 86400)
          ((ms) 1/1000)
          ((us) 1/1000000)))))
```

To avoid quoting the unit, we need a macro. Unlike a function, a macro does not evaluate its arguments. Taking advantage of this fact, we replace the **sleep-units%** function with the **sleep-units** macro. Now we don't need to quote the unit:

```
(sleep-units .5 h)
```

Although the main purpose of this macro is to avoid having to quote the **unit** argument, this macro is even more efficient than the function because there is no run-time dispatch at all: the unit and therefore the multiplier are known at compile time. Of course whenever we discover this sort of too-good-to-be-true situation, it probably really is too good to be true. This gain in efficiency isn't free. By foregoing the run-time dispatch we have lost the ability to determine the time unit at run-time. This makes it impossible to execute the following code using our macro:

```
(sleep-units 1 (if super-slow-mode 'd 'h))
```

This will not work because **sleep-units** expects the second argument to be one of the symbols in our case statement but instead it is a list with the first element the symbol **if**.

UNIT-OF-TIME

```
(defmacro unit-of-time (value unit)
  `(* ,value
      ,(case unit
         ((s) 1)
         ((m) 60)
         ((h) 3600)
         ((d) 86400)
         ((ms) 1/1000)
         ((us) 1/1000000))))
```

Recall that most macros are written to create more convenient and useful programming abstractions, not to improve the efficiency of the underlying code. Is it possible to extract any idioms from this code to make it more useful for the rest of our program (and possibly other future programs)? Even now we can foresee wanting to do other things with time values than just calling **sleep** on them. The macro **unit-of-time** abstracts functionality from the **sleep-units** macro, returning a value instead of calling **sleep** on it. The **value** parameter

can be determined at run-time because it is evaluated then, but **unit** cannot because we require the information at compile-time, just like **sleep-units**. Here is an example:

```
* (unit-of-time 1 d)

86400
```

Simple macros like **unit-of-time** provide a better syntax for solving a specific domain of problems and can give substantial productivity and correctness advantages. We will continue the development of this unit language further in section 5.2, Top-Down Programming. Unlike most programming languages, lisp gives you the same tools available to the people who created your programming environment. Macros are good enough for implementing the COMMON LISP language and they are good enough for implementing your own domain specific languages.

Control Structures

Although this book is focused on COMMON LISP, it is written for and about the Scheme programming language as well. Scheme is a wonderful language that, although lacking many features lisp programmers take for granted, still offers a flexible enough core for the professional lisp programmer to extend as necessary[2] *Scheme and COMMON LISP differ mostly in the communities they cater to. Scheme programmers like to talk about how grea*. Similarly, there are a few features Scheme programmers rely on heavily that COMMON LISP doesn't specifically address. But comparisons between the features offered by each language are, with a few exceptions, meaningless. The gaps between the two languages can be, and frequently are, bridged. The bridges we use to cross between the two languages are, of course, macros.

Scheme's **let** form is in one respect more powerful than its COMMON LISP counterpart. Scheme's **let** form supports something called a *named let*. In Scheme, you can optionally insert a symbol before the bind list of a **let** form and Scheme will bind a function named by the provided symbol around the let body[3] *Scheme only has one namespace so this function is bound there.* . This function accepts new arguments for the values provided in the **let** bindings, providing a very convenient way to express loops.

NLET

```
(defmacro nlet (n letargs &rest body)
  `(labels ((,n ,(mapcar #'car letargs)
                ,@body))
     (,n ,@(mapcar #'cadr letargs))))
```

Luckily we can build a bridge between Scheme and COMMON LISP with the **nlet** macro. **Nlet** lets us code in a Scheme style by emulating Scheme's named lets. In **nlet-fact**, **nlet** is used to define the factorial function by using a named let:

```
(defun nlet-fact (n)
  (nlet fact ((n n))
    (if (zerop n)
      1
      (* n (fact (- n 1))))))
```

Because **nlet** is one of our first macros, let's slow down and analyse it in depth. Sometimes to understand a macro it helps to *macroexpand* an example use of this macro[4] *The terminology of expansion is actually rather unfortunate. Nothing says that macroexpanding something will result in lar*. To do that, provide a list representing this macro invocation to the **macroexpand** function. Notice that **macroexpand** will only expand macros that have their symbols in the first element of the list and will not expand nested macro invocations for you[5] *But* ***macroexpand*** *will continue to expand the macro until the first element no longer represents a macro.* ***Macroex***. In the following, we've copied an invocation of **nlet** directly from **nlet-fact**, quoted it, and passed it to **macroexpand**:

```
* (macroexpand
    '(nlet fact ((n n))
       (if (zerop n)
         1
         (* n (fact (- n 1))))))

(LABELS ((FACT (N)
           (IF (ZEROP N)
             1
             (* N (FACT (- N 1))))))
  (FACT N))
T
```

The expansion uses the **labels** special form to bind a function around the provided body. The function is named according to the symbol used in the named let form. It takes as arguments the values bound with **nlet**, here only **n**. Since this function can be recursive, **nlet** implements a useful iteration construct.

Although simple macros might just be filling out backquote templates, most complicated macros at minimum make use of lisp's extensive list processing functions. **Mapcar**, applying a function to every element in a list and returning a list of the resulting values, turns up especially often in macros. Tellingly, **mapcar** turns up often in regular lisp code as well. Lisp has been tuned to be as useful as possible for processing lists. In all kinds of lisp programming, including macro construction, we splice, merge, reduce, map, and filter lists. The only difference is that when programming macros, the output subsequently gets passed to a compiler or interpreter. Programming macros in lisp is actually the same process as programming regular lisp.

But what does it mean to say that **nlet** is a new control structure? A control structure is just a fancy way of describing some construct that doesn't follow the behaviour of a function. A function will evaluate each argument from left to

right, bind the results in an environment, and execute machine code specified by some **lambda** form. Since **nlet** doesn't evaluate its arguments directly, instead splicing them into some chunk of lisp code, we have changed the flow of evaluation for **nlet** forms and thus have created a new control structure.

By this broad definition, virtually all macros—at least all interesting macros—define new control structures. When people say "only use macros when functions won't do", they mean that for any definition where you don't want to evaluate certain arguments, or you want to evaluate them out of order, or more than once, you will need to use a macro. Functions, no matter how cleverly written, simply will not work.

The **nlet** macro demonstrates one way that COMMON LISP was designed for macro writers. In binding forms such as **let**, it is a common convention for a variable to be bound to nil if no value is specified along with the variable name. In other words, **(let ((a)) a)** will be nil[6] *COMMON LISP even allows us to write* ***(let (a) a)*** *for the same effect.* . In Scheme, a language slightly less macro-writer friendly, this case must be checked for as a special case when iterating through such bindings because **(car nil)** and **(cdr nil)** raise type errors. In COMMON LISP, **(car nil)**, **(cdr nil)**, and therefore **(car (cdr nil))** and **(cadr nil)** are defined to return **nil**, allowing the second **mapcar** in **nlet** to work even if the empty let variable convention is used. This COMMON LISP feature is from Interlisp[INTERLISP].

Our **nlet** macro is different from Scheme's named lets in one subtle way. In this case, the interface to the macro is acceptable but the expansion may not be. As is common when programming across multiple levels, our mental model of the code can easily be slightly different from reality. In Scheme, a tail call of a named let is guaranteed to take up no additional stack space since Scheme is required, by the standard, to make this specific optimisation. This is not the case in COMMON LISP, however, so it is possible for stack overflows to occur in our COMMON LISP version of **nlet** that would not happen with named lets in Scheme. In section 5.4, Code-Walking with Macrolet we will see how to write a version of **nlet** with an identical interface but a potentially more efficient expansion[7] *In practice, this version of* ***nlet*** *is usually sufficient since a COMMON LISP compiler will almost certainly optimise tail calls* .

Free Variables

A *free variable* is any variable or function referenced in an expression that doesn't have a global special binding or an enclosing lexical binding. In the following expression, **x** is free:

```
(+ 1 x)
```

But in the following, we create a binding around the form which *captures* the variable **x**, depriving it of its freedom:

```
(let ((x 1))
```

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

The terminology of freedom and capture may seem strange at first. After all, freedom implies consciousness and an ability to make decisions—something a simple expression is obviously lacking. But freedom doesn't refer to what the expression can do, rather what we, as programmers, can do with the expression. For example, we can take the expression **(+ 1 x)** and embed it anywhere we want, allowing our expression to access a binding named **x** in the surrounding code. We then say that the code has *captured* our free variable. After the free variables in an expression are captured, as in the above **let** form, other surrounding code has no option of capturing our variable **x**. Our formerly free variable has already been captured. It is now completely unambiguous which **x** it refers to. Because of this, there isn't really any need for lisp to keep the reference to the symbol **x** in the code at all. As was described in detail in section 2.3, Lexical and Dynamic Scope, lisp compilers will forget the symbols that were used to represent lexical variables.

Although any language with expressions can have expressions with free variables, lisp's macro capabilities mean that free variables are much more useful in lisp than in other languages. In most languages we are forced to obey *referential transparency*. If there is no global or object variable **x** defined in a Blub program, the following code is unconditionally incorrect:

```
some_function_or_method() {
  anything(1 + x);
}
```

There is no way that **some_function_or_method** can create an *implicit binding* for **x**. In Blub, any use of a variable must have a textually apparent definition[8] *Or, sometimes, in object oriented Blub, a class or object definition.* . Languages with primitive macro systems (like C) can accomplish some of this in a very limited sense. But just as general purpose macros are impractical or impossible to write in C, so are the special cases involving free variables.

In lisp we can push around expressions with free variables as we please and either splice them into new expressions for them to be captured by surrounding code, or define global special variables to capture them. We can also write macros to modify which variables are free in an expression, either by re-writing the expression so it has fewer free variables (say by wrapping it in a let form, as above) or by modifying the expression in a way that adds new free variables. Such addition of free variables is the opposite of capturing variables and is called *free variable injection*.

The simplest possible free variable injection is a macro that expands into a symbol reference:

```
(defmacro x-injector ()
  'x)
```

Because a macro is just a function, it executes its body as a regular lisp form.

The above injector macro evaluates the quoted symbol and, of course, returns a symbol—a free variable—to be spliced into any expression that uses the **x-injector** macro. Discussing such free variable injection in *On Lisp*, Paul Graham writes

This kind of lexical intercourse is usually viewed more as a source of contagion than a source of pleasure. Usually it would be bad style to write such a macro. Of all the macros in this book, only [two isolated cases] use the calling environment in this way.

By contrast, this book gets much pleasure from this sort of lexical intercourse. Free variable injection—writing a macro with full knowledge of the lexical environment it will be expanded in—is just another approach to lisp macro programming, one that is especially useful when there are a few slightly different lexical contexts that you would like to write mostly identical code inside. Although often the main advantage of a function call is that you throw out your lexical environment, sometimes, to lisp programmers, this is just a guide-line that can be ignored through the use of macros. In fact, once accustomed to it, some lisp programmers try to always write macros, extending the lexical context as far as possible, using a function only when they need to evaluate arguments or just chicken out and want a new lexical context. In section 3.6, Once Only we will see a way to avoid throwing out your lexical environment when you need arguments evaluated. Keeping the lexical environment around as much as possible allows for very interesting macro *combinations*, where a macro adds lexical context around a use of one or more other macros. Expanding into code that uses the very macro being defined is a special case of macro combination and is treated in section 5.5, Recursive Expansions.

The shortest distance between two points is a straight line. Free variables and, more generally, extended lexical contexts are often the easiest way to programmatically construct a program. Using macros in this way might seem like a hack, and might feel objectionable on stylistic grounds, but it works conveniently and reliably. Especially after we consider **macrolet** in section 5.4, Code-Walking with Macrolet, this style of programming—combining macros—will begin to feel more comfortable. Just remember that macro programming is not about style; it is about power. Macros allow us to do many things that are impossible in other languages. Free variable injection is one of them.

Unwanted Capture

There are two perspectives on variable capture. Variable capture is the source of some very unpredictable bugs but when used properly can also be a highly desirable macro feature. Let's start our consideration of variable capture with a simple macro defined by Graham in *On Lisp*: **nif**. **Nif** is a *numeric if* which has four required clauses, compared to the regular boolean **if** that has two required clauses and an optional third clause. **Nif**, or rather the code that **nif** expands into, evaluates the first clause and assumes the result to be a non-complex

number. It then evaluates one of the three respective clauses, depending on whether the result is positive (**plusp**), zero (**zerop**), or negative (otherwise). We can use it to test the variable **x** like so:

```
(nif x "positive" "zero" "negative")
```

Nif is the ideal function for our discussion of variable capture and we will use it to illustrate a few key points and also as a test case for a new notation for macro construction. Before we present the version of nif defined by Graham, let's define a nearly correct, but slightly buggy version:

```
(defmacro nif-buggy (expr pos zero neg)
  `(let ((obscure-name ,expr))
     (cond ((plusp obscure-name) ,pos)
           ((zerop obscure-name) ,zero)
           (t ,neg))))
```

Nif-buggy expands into a bit of code that uses **let** to bind the result of evaluating the user's supplied **expr** form. We need to do this because it is possible that evaluating **expr** will incur *side-effects* and we need to use its value for two separate things: passing it to **plusp** and passing it to **zerop**. But what do we call this temporary binding? To introduce a subtle bug we chose an arbitrary symbol, **obscure-name**. Unless someone looks at the macro expansion, nobody will ever see this name anyways, so it's no big deal, right?

Nif-buggy will appear to work like **nif** in almost all cases. As long as the symbol **obscure-name** is never used in the forms supplied to **nif-buggy**[9] then there is no possibility for unwanted variable capture. But what happens if **obscure-name** does appear in forms passed? In many cases, there is still no bug:

[9] *Or in the macro expansions of forms passed to it. See sub-lexical scope.*

```
(nif-buggy
  x
  (let ((obscure-name 'pos))
    obscure-name)
  'zero
  'neg)
```

Even if **x** turns out to be positive, and even though we have injected the forbidden symbol into **nif-buggy**'s macroexpansion, this code still works as intended. When a new binding is created, and the references inside that binding always refer to the created binding, no unwanted variable capture occurs. The problem only appears when our usage of **obscure-name** *crosses over* its use in the expansion. Here is an example of unwanted variable capture:

```
(let ((obscure-name 'pos))
  (nif-buggy
    x
    obscure-name
```

```
'zero
'neg))
```

In this case, **obscure-name** will be bound to the result of the evaluation of **x**, so the symbol **pos** will not be returned as was intended[10] *In truth, of course, this buggy behaviour was exactly what was intend* . This is because our use of a symbol crossed over an invisible use of a binding. Sometimes code with invisible bindings like this is said to not be *referentially transparent*.

But isn't this just an academic issue? Surely we can think of rare enough names so that the problem never shows up. Yes, in many cases, packages and smart variable naming can solve the problem of variable capture. However, most serious variable capture bugs don't arise in code directly created by a programmer. Most variable capture problems only surface when other macros use your macro (combine with your macro) in ways you didn't anticipate. Paul Graham's has a direct answer for why to protect against unwanted variable capture:

Why write programs with small bugs when you could write programs with no bugs?

I think we can distill the issue even further: no matter how subtle, why do something incorrectly when you can do it correctly?

Luckily, it turns out that variable capture, to the extent that it is a problem, is a solved problem with an easy solution. That last sentence is a controversial statement to many people, especially those who have decided they don't like the obvious solution and have dedicated large portions of time looking for a better one. As a professional macro programmer you will come into contact with many of these variable capture solutions. The current popular solution is to use so-called *hygienic macros*[11] *Another popular term used to be "macros by example".* . These solutions try to limit or eliminate the impact of unwanted variable capture but unfortunately do so at the expense of wanted, desirable variable capture. Almost all approaches taken to reducing the impact of variable capture serve only to reduce what you can do with **defmacro**. Hygienic macros are, in the best of situations, a beginner's safety guard-rail; in the worst of situations they form an electric fence, trapping their victims in a sanitised, capture-safe prison. Furthermore, recent research has shown that hygienic macro systems like those specified by various Scheme revisions can still be vulnerable to many interesting capture problems[SYNTAX-RULES-INSANE][SYNTAX-RULES-UNHYGIENIC].

The real solution to variable capture is known as the *generated symbol*, or *gensym* for short. A gensym is a way of having lisp pick the name of a variable for us. But instead of picking lame names like **obscure-name** as we did previously, lisp picks good names. Really good names. These names are so good and unique that there is no way anyone (even **gensym** itself) will ever pick the same names again. How is this possible? In COMMON LISP, symbols (names) are associated with *packages*. A package is a collection of symbols from which you can get pointers to by providing strings, their **symbol-name** strings. The most important property

of these pointers (usually just called symbols) is that they will be **eq** to all other pointers (symbols) that have been looked up in that package with that same **symbol-name**. A gensym is a symbol that doesn't exist in any package, so there is no possible **symbol-name** that will return a symbol **eq** to it. Gensyms are for when you want to indicate to lisp that some symbol should be **eq** to some other symbol in an expression without having to name anything at all. Because you aren't naming anything, name collisions just can't happen.

So by following these three simple, very important rules, avoiding unwanted variable capture in COMMON LISP is easy:

Whenever you wrap a lexical or dynamic binding around code provided to your macro, name this binding with a gensym unless you want to capture it from the code you are wrapping.

Whenever you wrap a function binding or a ***macrolet*** *or* ***symbol-macrolet*** *macro around code provided to your macro, name this function or macro with a gensym unless you want to capture it from the code you are wrapping. Verify that this binding doesn't conflict with any of the special forms, macros, or functions defined by the standard.*

Never assign or re-bind a special form, macro, or function specified by COMMON LISP.

Some lisps other than COMMON LISP, like Scheme, have the unfortunate property of combining the variable namespace with the function/macro namespace. Sometimes these lisps are termed *lisp-1* lisps, while COMMON LISP, with its separate namespaces, is termed a *lisp-2* lisp. With a hypothetical *lisp-1* COMMON LISP we would also be obliged to follow these two additional rules when constructing macros:

Verify that intentionally introduced lexical or dynamic bindings do not collide with intentionally introduced function or macro bindings, or any of the special forms, macros, or functions defined by the standard.

Verify that intentionally introduced function or macro bindings do not collide with intentionally introduced lexical or dynamic bindings.

COMMON LISP's wise design decision to separate the variable namespace from the function namespace eliminates an entire dimension of unwanted variable capture problems. Of course lisp-1 lisps do not suffer any theoretical barrier to macro creation: if we follow the previous two rules, we can avoid variable capture in the same way as we do in COMMON LISP. However, when programming sophisticated macros it can be hard enough to keep track of symbols in a single, isolated namespace. Having any cross-pollination of names to consider just makes macro writing more difficult than it needs to be.

More so than any other property except possibly its incomplete standard[12] *Especially as it relates to macros and exceptions.* , it is this defect of a single namespace that makes Scheme, an otherwise excellent language, unfit for serious

macro construction[13] *Though as we will see throughout this book, there are many reasons to prefer COMMON LISP over Scheme* . Richard Gabriel and Kent Pitman summarise the issue with the following memorable quote[LISP2-4LIFE]:

There are two ways to look at the arguments regarding macros and namespaces. The first is that a single namespace is of fundamental importance, and therefore macros are problematic. The second is that macros are fundamental, and therefore a single namespace is problematic.

Because there is little of less importance than the quantity of namespaces, and little of more importance than the enabling of macros, it can only be concluded that Scheme made the *wrong* decision and COMMON LISP made the *right* decision.

Still, calling **gensym** every single time we want a nameless symbol is clunky and inconvenient. It is no wonder that the Scheme designers have experimented with so-called *hygienic* macro systems to avoid having to type **gensym** all over the place. The wrong turn that Scheme took was to promote a domain specific language for the purpose of macro construction. While Scheme's mini-language is undeniably powerful, it misses the entire point of macros: macros are great because they are written in lisp, not some dumbed down pre-processor language.

This book presents a new syntax for gensyms that should be more palatable to the brevity-conscious yet remains a thin film over traditional lisp expressions. Our new notation for gensyms, which we will use as the foundation for most of the macros in this book, is most clearly described by peeling off the layers of a simple macro which uses the features our notation offers. Let's continue with the **nif** example from the previous section. Here is how Graham defines a capture-safe **nif**:

```
(defmacro nif (expr pos zero neg)
  (let ((g (gensym)))
    `(let ((,g ,expr))
       (cond ((plusp ,g) ,pos)
             ((zerop ,g) ,zero)
             (t ,neg)))))
```

This is how to use **gensym** correctly. As we saw in the previous section, a macro that can expand user input into something that could interfere with one of its variables must take care against variable capture. Graham presents a macro abbreviation **with-gensyms** that is somewhat more concise for situations where a number of gensyms need to be created:

```
(with-gensyms (a b c)
  ...)
```

Expands into

```
(let ((a (gensym))
      (b (gensym))
```

```
      (c (gensym)))
  ...)
```

Because needing gensyms in a **defmacro** form is so common, we decide to pursue the abbreviation further. In particular, notice that we have to type the temporary name for each of the gensyms (like **a**, **b**, and **c**) at least twice: once when we declare it a gensym and again when we use it. Can we eliminate this redundancy?

First, consider how the **nif** macro uses gensyms. When the **nif** macro is expanded, it calls **gensym** which returns a generated symbol. Because this symbol is guaranteed to be unique, we can safely splice it into a macro expansion knowing that it will never capture any unintended references. But we still need to name this gensym in the definition of the macro so we are able to splice it into the expansion in the right places. Graham, for the scope of the **nif** macro definition, names this gensym **g**. Notice that this name never actually appears in the macro expansion of **nif**:

```
* (macroexpand '(nif x 'pos 'zero 'neg))

(LET ((#:G1605 X))
  (COND ((PLUSP #:G1605) 'POS)
        ((ZEROP #:G1605) 'ZERO)
        (T 'NEG)))
T
```

The name **g** disappears in the macro expansion. Because **g** was only bound in our expander environment, the name given to such a variable is irrelevant with respect to capture in the expansions. All occurrences of **g** have, in the expansion, been replaced by a symbol with a print name of **G1605**. It is prefixed by #: because the symbol is not *interned* in any package—it is a gensym. When printing out forms, it is necessary to prefix gensyms in such a way because we want lisp to break if we ever use (evaluate) this form after reading it back in again. We want lisp to break because we can't know by looking at the print names of two gensyms if they should be **eq** or not—that is their purpose. Lisp breaks in an interesting way: because each time a #: symbol is read in a new symbol is created, and because **(eq '#:a '#:a)** is never true, the inner **#:G1605** symbols in the above expansion do not refer to the binding created by the **let** form so lisp considers the expression to have a free variable, indicating to us that a form with gensyms was read back in again.

Despite the default printing behaviour for such uninterned symbols, it is still possible to save and reload macro expansions. For a more accurate printed representation of a form with gensyms, we can turn on ***print-circle*** mode when we print the results[14] :

[14] *We return* ***t*** *so that we don't see the form returned by* ***print****. Returning* ***(values)*** *is also common.*

```
* (let ((*print-circle* t))
    (print
```

```
      (macroexpand '(nif x 'pos 'zero 'neg)))
    t)

(LET ((#1=#:G1606 X))
  (COND ((PLUSP #1#) 'POS)
        ((ZEROP #1#) 'ZERO)
        (T 'NEG)))
T
```

In the above form, the lisp printer uses the #= and ## *read macros*. These read macros allow us to create *self-referential* forms which we will discuss in more depth in section 4.5, Cyclic Expressions. If we read in the above form, the symbols used inside will actually be the same as the symbol used in the **let** binding and the expansion will still work. It seems as though the above definition has avoided the dual-naming redundancy. Is there a way we can pull this back up into a macro writing macro template?

G-BANG-SYMBOL-PREDICATE

```
(defun g!-symbol-p (s)
  (and (symbolp s)
       (> (length (symbol-name s)) 2)
       (string= (symbol-name s)
                "G!"
                :start1 0
                :end1 2)))
```

Remember that we can name our gensyms anything in the macro definition, even, as Graham does, simple names like **g**, and they will disappear in the macro expansion. Because of this freedom in naming, let's standardise on a naming convention for gensyms. As a compromise between brevity and uniqueness, any symbol that starts with the two characters **G!**, and is followed by at least one other character is considered to be a special gensym referencing symbol called a *G-bang symbol*. We define a predicate, **g!-symbol-p**, which is a predicate for determining whether a given atom is a G-bang symbol.

DEFMACRO-WITH-G-BANG

```
(defmacro defmacro/g! (name args &rest body)
  (let ((syms (remove-duplicates
               (remove-if-not #'g!-symbol-p
                              (flatten body)))))
    `(defmacro ,name ,args
       (let ,(mapcar
              (lambda (s)
                `(,s (gensym ,(subseq
                               (symbol-name s)
                               2))))
              syms)
```

```
,@body))))
```

Now that we have G-bang symbols standardised, we can create a macro that writes macro definitions for us and exploits a macro writing shortcut known as *automatic gensyms*. The macro **defmacro/g!** defines a domain specific language for the domain of macro writing, but manages to retain all of lisp's power. **Defmacro/g!** is simple, but how to use it, and how it works, may be non-obvious. Because of this, and because this is one of the first real macros we've presented in this book, we take the analysis of **defmacro/g!** slowly.

When dissecting any macro, the first step is to *stop*. Don't think of a macro as a syntax transformation or any other such nonsense abstraction. Think of a macro as a function. A macro is a function underneath, and works in the exact same way. The function is given the unevaluated expressions provided to it as arguments and is expected to return code for lisp to insert into other expressions.

So, thinking about **defmacro/g!** as a function, consider its execution. Because we are programming a regular lisp function, we have access to all of lisp's features, even utilities we've since added to the language. In **defmacro/g!**, we use Graham's **flatten** utility, lisp's **remove-if-not** and **remove-duplicates** functions, and our G-bang symbol predicate **g!-symbol-p** to create a new list consisting of all the G-bang symbols found inside the body form that was passed to our macro. Next, we use a backquote template to return a list representing the code we would like the macro to expand into. In our case, because we're writing an improvement to **defmacro**, we would like our code to expand to a defmacro form itself. But we are adding new convenience features to the defmacro language and want to create a slightly more sophisticated expansion. In order to give each G-bang symbol found in the macro's body a fresh gensym, we use **mapcar** to map a function over the list of collected G-bang symbols, creating a new list that can be spliced into the **let** form, establishing bindings for each gensym[15] *The **gensym** function can optionally be passed a single string argument. This changes the gensym's print name which is help*
.

Notice how the lambda that we map contains an expression created with the backquote operator, resulting in what appears to be—but is not—a *nested backquote* situation. Because the **mapcar** that applies this function is *unquoted*, the unquoted expressions in the nested backquote are still evaluated in our original context. Nested backquotes are notoriously difficult to understand and we will return to this concept when we look at backquote in more depth in chapter 4, Read Macros.

So what, exactly, does **defmacro/g!** let us do? It lets us exploit this technique of automatic gensyms, a way of checking for the presence of particular symbols in the lexical scope of code provided to the macro[16] *This is, for now, a simplification. See the section on sub-lexical scope.* . If we don't use any G-bang symbols, we can use **defmacro/g!** exactly like **defmacro**. But any G-bang symbols that occur in the body of the macro expansion are interpreted to mean:

I want a gensym to be bound around this expression, and I've already given the symbol. Make it happen.

We can use this to save having to explicitly create a gensym in this re-definition of **nif**:

```
(defmacro/g! nif (expr pos zero neg)
  `(let ((,g!result ,expr))
     (cond ((plusp ,g!result) ,pos)
           ((zerop ,g!result) ,zero)
            (t ,neg))))
```

When we want to use a gensym we just use it. We need to be careful, of course, that all references to G-bang symbols are only evaluated by the macro expansion because that is the only place where the gensym will be bound[17] *G-bang symbols especially shouldn't appear in the expansion itself—that is exactly what we are trying to avoid with gensyn* . Unquoting the G-bang symbols that occur inside a backquote, like above, is the most obvious way to do this, and we can see the direct parallel to the unquoting of the symbol **g** in Graham's original definition of **nif**.

So we have defined a macro **nif** that appears to function the same as Graham's, but this improvement almost seems too good to be true. Does it really work? Let's look at the macro expansion[18] *We use* ***macroexpand-1*** *so we only expand the* ***defmacro/g!*** *macro and not the* ***defmacro*** *it expand* before we decide:

```
* (macroexpand-1
    '(defmacro/g! nif (expr pos zero neg)
       `(let ((,g!result ,expr))
          (cond ((plusp ,g!result) ,pos)
                ((zerop ,g!result) ,zero)
                (t ,neg)))))

(DEFMACRO NIF (EXPR POS ZERO NEG)
  (LET ((G!RESULT (GENSYM "RESULT")))
    `(LET ((,G!RESULT ,EXPR))
       (COND ((PLUSP ,G!RESULT) ,POS)
             ((ZEROP ,G!RESULT) ,ZERO)
             (T ,NEG)))))
T
```

It seems that **defmacro/g!** wrote essentially the same code that Graham did when he wrote the original version of **nif**. Seeing this example use of **defmacro/g!**, we see that no non-gensym bindings will be created in its expansions. **Nif**, defined with **defmacro/g!** like this, is free from variable capture problems.

But since **defmacro/g!** is a macro itself, is it possible that there could be unwanted capture or substitution problems in the macro expansion environment? As with any sufficiently complex abstraction, the behaviour is, to an extent, arbitrary. In the same sense that variable capture itself is a flaw, certain

properties of **defmacro/g!** that might appear to be flaws could simply be inherent to its design[19] *Though it is never safe to rule out programmer error either.* . As always, the best solution is to understand the abstraction completely.

An interesting *corner-case* of **defmacro/g!** is in G-bang macro defining G-bang macros. All **defmacro/g!** does is introduce a set of bindings into the expansion environment, each of which is bound to a gensym that the macro can use, if it wants. In cases where there are multiple possibilities of where the gensym could be bound, they are always distinguishable because of context. In other words, you can always specify which environment's gensym should be used based on which environment you evaluate it in. Take this contrived example:

```
(defmacro/g! junk-outer ()
  `(defmacro/g! junk-inner ()
     `(let ((,g!abc))
        ,g!abc)))
```

Here there are two gensyms created. The uses of **g!abc** are preceded by only one unquote (comma) so we know that the expansion refers to the inner gensym created by the expansion of **junk-inner**. If each had instead two unquotes, they would refer to the outer gensym created by the expansion of **junk-outer**.

Defmacro/g! uses Graham's **flatten** function. **Flatten**, as described in section 1.3, The Lisp Utility, takes a tree cons structure—our lisp code—and returns a new list of all the leaves/atoms. The use of **flatten** in **defmacro/g!** is a simple example of *code-walking*, a topic we will revisit throughout this book.

Exercise: In the above G-bang macro defining G-bang macro, what would be the problem if the first gensym was prefixed with one unquote and the other was prefixed with two?

Once Only

Peter Norvig is a brilliant programmer and author. His books on AI, especially *Artificial Intelligence: A Modern Approach*[AIMA], are required reading before tackling many of the most difficult problems we currently face as computer scientists. Norvig is perhaps better known to lisp programmers for his book *Paradigms Of Artificial Intelligence Programming: Case Studies in COMMON LISP*[PAIP]. This book is dated but still required reading for serious lisp students and contains many important lisp insights[20] *One bit of COMMON LISP advice from PAIP that is timelessly true is to never mix &optional and &key arguments to lan* . This section is dedicated to Peter Norvig and is even named after a macro described in PAIP. In its last few pages, tucked away in a description of sequence function implementation, is

Once-only: A Lesson in Macrology

Which is shortly followed by an even more intriguing sentence:

[I]f you can understand how to write and when to use once-only, then you truly understand macros.

As we now know, nobody truly understands macros. Understanding a particular macro, even one as important as **once-only**, gets you no further to understanding macros than understanding an important theorem gets you to truly understanding mathematics. Because their possibilities so far seem infinite, truly understanding math or macros is truly impossible.

We will not give the definition of Norvig's **once-only** here, but it is a reasonably complex macro with some interesting properties that we will implement slightly differently. **Once-only** was originally written for the deceased *lisp machine* programming environment and was left out of COMMON LISP for inconsequential reasons.

The idea behind **once-only** is to surround a macro expansion with code that will create a new binding. When the macro expansion is evaluated, this binding will be initialised with the result of evaluating one of the forms passed to the macro as an argument. The code in the body of **once-only** can then use the binding which, of course, does not re-evaluate the form that was passed to the macro. The form passed as an argument to the macro is only and always evaluated once. Once-only.

As an example of **once-only**, Norvig shows a **square** macro. Its expansion takes one argument and returns the product of that argument with itself:

```
(defmacro square (x)
  `(* ,x ,x))
```

This will work when we pass a lot of things to it: most variables, numbers, and other forms that can freely be evaluated as many times as necessary. But as soon as forms that have *side-effects* are passed to this version of **square**, all bets are off. The behaviour is, of course, still deterministic, but can be decidedly difficult to determine. With this particular macro, the form will be evaluated exactly twice. But because these things get complicated quickly, in the general case, all bets are off. Making it convenient and easy to avoid these unwanted side-effects is the point of **once-only**. Notice that if we use a function, we get this behaviour for free. After we depart the land of contrived text-book examples, come to our senses, and define **square** as a function, it ends up looking like this:

```
(defun square (x)
  (* x x))
```

Because of how lambda works, we can use any form as the argument to this function definition of **square**. Since this argument will be evaluated exactly once, our notions and conceptual models of side-effects are satisfied. In most instances we expect an expression that we've written only once to be evaluated only once. Conversely, one of the main powers of a macro is to violate this assumption by manipulating the frequency and order of evaluation. In things like loops, for instance, we might want expressions to be evaluated more than

once. We might even want them to never be evaluated, say because we want from them something other than their evaluation.

Once-only allows us to specify particular parameters in our macro expansion that we would like to only be evaluated once and have their order of evaluation be left-to-right, just like lambda. Here is how we would accomplish this with the traditional **once-only** macro:

```
(defmacro square (x)
  (once-only (x)
    `(* ,x ,x)))
```

But of course if all you ever wanted to do was **once-only** all the arguments of your macro, you would be using a function (lambda) instead. We will return to this point in a moment, but because this book doesn't supply a direct implementation of **once-only**, we introduce an alternate implementation of this functionality for our macro notation. Although there are many interesting implementations of **once-only**[PAIP-P853][PRACTICAL-CL-P95], this section introduces a new technique involving a combination with **defmacro/g!**.

The first step in our once-only implementation is to create some new predicates and utility functions. Again compromising brevity with uniqueness, we reserve another set of symbols for our own use. All symbols starting with the characters **O!** and followed by one or more characters are called *O-bang symbols*.

O-BANG-SYMBOLS

```
(defun o!-symbol-p (s)
  (and (symbolp s)
       (> (length (symbol-name s)) 2)
       (string= (symbol-name s)
                "O!"
                :start1 0
                :end1 2)))

(defun o!-symbol-to-g!-symbol (s)
  (symb "G!"
        (subseq (symbol-name s) 2)))
```

A predicate to distinguish O-bang symbols from other objects is defined: **o!-symbol-p**. Its definition is nearly identical to that of **g!-symbol-p**. We also introduce a convenient utility function that changes an O-bang symbol into a G-bang symbol, preserving the characters after the bang: **o!-symbol-to-g!-symbol**. This utility function uses Graham's handy utility function **symb** to create new symbols.

DEFMACRO-BANG

```
(defmacro defmacro! (name args &rest body)
  (let* ((os (remove-if-not #'o!-symbol-p args))
```

```
      (gs (mapcar #'o!-symbol-to-g!-symbol os)))
  `(defmacro/g! ,name ,args
     `(let ,(mapcar #'list (list ,@gs) (list ,@os))
        ,(progn ,@body)))))
```

Defmacro! represents the final step in our macro defining language—it adds a once-only feature. **Defmacro!** combines with **defmacro/g!** from the previous section. Since **defmacro!** expands directly into a **defmacro/g!** form, it *inherits* the automatic gensym behaviour. Understanding all the pieces being combined is essential for sophisticated combinations. Recall that **defmacro/g!** looks for symbols starting with G-bang and automatically creates gensyms. By expanding into a form with G-bang symbols, **defmacro!** can avoid duplicating gensym behaviour when it implements once-only.

Defmacro! gives a shortcut known as *automatic once-only*. With automatic once-only we can prefix one or more of the symbols in the macro's arguments with an O-bang, making them O-bang symbols as defined by **o!-symbol-p**. When we do this, **defmacro!** will know we mean to create a binding in the produced code that will, when evaluated, contain the results of evaluating the code provided as an argument to the macro. This binding will be accessible to the macro expansion through a gensym. But when creating the expansion how can we refer to this gensym? By using the equivalent G-bang symbol as defined above by **o!-symbol-to-g!-symbol**.

The implementation relies on the capabilities of **defmacro/g!**. With the **o!-symbol-to-g!-symbol** utility, we create new G-bang symbols to add into a **defmacro/g!** form. Once we have automatic gensyms, once-only is easy to implement, as evidenced by the brevity of the **defmacro!** definition.

Come back to the land of contrived textbook examples for a moment and we will re-implement the **square** macro, this time with **defmacro!**:

```
(defmacro! square (o!x)
  `(* ,g!x ,g!x))
```

Which we can macroexpand to:

```
* (macroexpand
    '(square (incf x)))

(LET ((#:X1633 (INCF X)))
  (* #:X1633 #:X1633))
T
```

In the previous section I mentioned that we pass a string value to **gensym** for all G-bang symbols. This makes examining the expansions of such forms much easier. Although there is nothing significant about the name of gensyms like **#:X1633**, if we were writing or debugging the **defmacro!** definition of **square** above, we could directly see the connection between this symbol and the symbol used in the macro definition: **X**. Being able to match symbols

from definition to expansion and vice-versa is much easier if this information is preserved in the print-name of the gensyms used, as done in **defmacro/g!** expansions[21] *This is also the reason for the number in the print-name of a gensym, specified by* ***gensym-counter****. This counter lets us a*
.

Aside from the less verbose usage and more helpful expansion output compared to the traditional **once-only**, **defmacro!** also provides one extra key feature. In the traditional **once-only**, the binding for the gensym used to access the created lexical variable is given the same name as the argument to the macro expansion, which *shadows* the macro argument so it cannot be accessed by the macro definition. Because **defmacro!** splits this into two separate types of symbols, G-bang symbols and O-bang symbols, we can write macro expansions that use both of these values. To demonstrate this, here is yet another definition of the **square** macro:

```
(defmacro! square (o!x)
  `(progn
     (format t "[~a gave ~a]~%"
                 ',o!x   ,g!x)
     (* ,g!x ,g!x)))
```

Which can be used like so:

```
* (defvar x 4)

X
* (square (incf x))
[(INCF X) gave 5]
25
```

Notice that we *quote* the unquoted O-bang symbol in the above **square** definition. We do this because we don't want to evaluate this form again. The expansion generated by **defmacro!** already evaluated it. We simply want to take the form passed to square and use it for another purpose, in this case some kind of crude debugging statement. However, even though we evaluated it once already, and in this case it being incorrect, there is nothing stopping us from evaluating the provided form again, should our desired abstraction demand it.

The **defmacro!** language allows us granular, convenient control over the evaluation of the arguments passed to our macros. If we prefix all the symbols representing arguments in the macro definition with O-bang, and only use the corresponding G-bang symbols in the macro definition, our expansions will be the same as lambda expressions—each form evaluated once, in left to right order. Without any of these symbols in **args** and without using any G-bang symbols in the expansion, **defmacro!** acts just like the regular COMMON LISP **defmacro**.

Defmacro! is most useful during iterative development of a macro. Because it is a simple matter of adding two characters to a macro argument to get lambda

style evaluation, and using gensyms is as easy as, well, writing them, we can change our mind about these decisions instantly. **Defmacro!** feels like an even tighter fitting glove over lambda than does COMMON LISP's **defmacro**. It is for this reason, iterative development, that we will use **defmacro!** as the main macro definition interface for the remainder of this book.

NIF

```
(defmacro! nif (o!expr pos zero neg)
  `(cond ((plusp ,g!expr) ,pos)
         ((zerop ,g!expr) ,zero)
          (t ,neg)))
```

Let's return to Graham's **nif** macro. When updating this macro for **defmacro!**, we notice that the **expr** argument, the one for which we created a gensym, is evaluated exactly once. Here we use **defmacro!** to indicate that this argument should be evaluated only once by calling it **o!expr**. This implementation of **nif** represents the final step in our evolution of this macro.

Defmacro! blurs the gap between macro and function. It is this feature, the ability to provide some O-bang symbols in the macro argument and some regular symbols, that makes **defmacro!** especially useful. Just as backquote allows you to flip the default quoting behaviour, **defmacro!** allows you to flip the evaluation semantics of macro arguments from regular un-evaluated macro forms to singly evaluated, left-to-right lambda arguments.

Duality of Syntax

One of the most important concepts of lisp is called *duality of syntax*. Understanding how to use dualities and why they are important is an underlying theme of macro writing and of this book. Dualities are sometimes designed and sometimes accidentally discovered. To programmers of non-lisp languages the reality of dual syntax would be too unbelievable to describe at this point in the book so we will for now shy away from a direct definition. Instead, you, the reader, gets to discover it again and again as it is applied slowly and carefully so as to avoid shock. Should you experience headaches or other discomfort through the course of this book, I recommend that you immediately execute a garbage collection cycle (get some sleep), then return with a fresh and open mind.

Referential transparency is sometimes defined as a property of code where any expression can be inserted anywhere and always have the same meaning. Introducing syntactic duals is the conscious violation of referential transparency and discovering them is reaping the fruits of a language that enables such violations. While other languages only let you build with semi-transparent panes of glass, lisp lets you use an assortment of smoke, mirrors, and prisms. The magic dust is made of macros, and most of its best tricks are based on syntactic duals.

This section describes an important dual syntax we have already discussed but

have not yet completely explored: COMMON LISP uses the same syntax for accessing both of its major types of variables, dynamic and lexical. This book tries to illustrate the real power of dynamic and lexical scope and why COMMON LISP's decision to use dual syntax is important.

The purpose of dynamic scope is to provide a way for getting values in and out of lisp expressions based on when the expression is evaluated, not where it is defined or compiled. It just so happens that, thankfully, the syntax that COMMON LISP defines for this is identical to that used to access lexical variables, which are the exact opposite of dynamic variables in that they always refer to the locations they were compiled for, independent of when the access takes place. In fact, without external context in the form of a declaration, you can't tell which type of variable an expression is referring to. This dual syntax violates referential transparency, but rather than being something to avoid, lisp programmers welcome this because just as you can't differentiate an expression without context, neither can a macro. Hold that thought for a second. First, it must be made clear that creating bindings for dynamic variables does not create lexical closures. As an example, let's re-bind the variable **temp-special** that we earlier declared special:

```
* (let ((temp-special 'whatever))
    (lambda () temp-special))

#<Interpreted Function>
```

Even though it is a *let over lambda*, this is not a lexical closure. This is a simple evaluation of a **lambda** macro form in some dynamic context which results in, of course, an anonymous function. This function, when applied, will access whatever current dynamic environment exists and fetch that value of **temp-special**. When the **lambda** macro was evaluated, a dynamic binding of **temp-special** to the symbol **whatever** existed, but who cares? Remember that **lambda** forms are constant objects, just simple machine code pointer returners, so evaluating this lambda form never even accesses the dynamic environment. What happens to our symbol **whatever**? After lisp is done evaluating the lambda form, it removes it from the dynamic environment and throws it away, unused.

Some early lisps did support *dynamic closures*, which meant that every function defined in a non-null dynamic environment had its own (possibly partially shared) stack of dynamic bindings. The effect is similar to COMMON LISP's lexical scope and was implemented with something termed a *spaghetti stack*[SPAGHETTI-STACKS][INTERLISP-TOPS20]. This data structure is no longer a stack data structure, but actually a multiple path, garbage collected network. COMMON LISP does away with spaghetti stacks and only provides lexical closures[MACARONI].

So lexical and dynamic variables are actually completely different, deservedly distinct concepts that just happen to share the same syntax in COMMON LISP

code. Why on earth would we want this so-called duality of syntax? The answer is subtle, and only consciously appreciated by a minority of lisp programmers, but is so fundamental that it merits close study. This dual syntax allows us to a write a macro that has a single, common interface for creating expansions that are useful in both dynamic and lexical contexts. Even though the meanings of expansions of the macro can be completely different given their context, and even though each can mean entirely different things underneath, we can still use the same macro and the same combinations of this macro with other macros. In other words, macros can be made *ambivalent* about not only the contents of their macro arguments, but also about the different meanings of their expansions. We can use the macro just for its understood code transformation, ignoring the semantic meanings of the code, all because the code only has meaning once we use it somewhere—it has no meaning during macro processing. The more dualities of syntax there are, the more powerful an associated macro becomes. Many more examples of the advantages of dual syntax are detailed through this book. The duality between dynamic and lexical variables is a mild (but useful) example of this lispy philosophy. Some macros are created for the specific purpose of having powerful duals, and sometimes there are many more than two possible meanings for an expansion.

A traditional convention in COMMON LISP code is to prefix and postfix the names of special variables with asterisk characters. For example, we might've chosen to name our **temp-special** variable ***temp-special***. Since this convention is almost like having another namespace for dynamic variables, diminishing their duality with lexical variables, this book does not follow it exactly. The asterisks are merely convention and, fortunately, COMMON LISP does not enforce them. Not only can we leave the asterisks off special variable names, but we can add them to lexical variable names. Maybe it is a question of style. Which is a lesser fashion crime: lexical variables with asterisks or special variables without? I tend to think the less verbose of the two. Also, the names of lexical and special variables can be gensyms, a concept that transcends print names on symbols.

So, as mentioned, this book hijacks the usual asterisk convention. Instead of

Asterisked variable names indicate special variables.

this book uses

Asterisked variable names indicate special variables defined by the standard.

My largest motivation for dropping these variable name earmuffs is simple and subjective: I think they are annoying to type and make code look ugly. I will not go so far as to suggest you do this for your own programs, just mention that I have been leaving off the earmuffs for years and am very content with COMMON LISP.