Let Over Lambda

Let Over Lambda -- 50 Years of Lisp

by Doug Hoyte

Read Macros

Run-Time at Read-Time

Syntactic sugar causes cancer of the semicolon. —Alan Perlis

Not only does lisp provide direct access to code that has been parsed into a cons cell structure, but it also provides access to the characters that make up your programs before they even reach that stage. Although regular macros work on programs in the form of trees, a special type of macro, called a *read macro*, operates on the raw characters that make up your program.

In lisp, when we want to define a non-lisp syntax, it doesn't make sense to use the lisp reader—that is only for reading lisp. The read macro is the device we use to process non-lisp syntax before the lisp reader gets its paws on it. The reason why the lisp reader is more powerful than that of other languages is that lisp gives you *hooks* for controlling every aspect of its behaviour. In particular, lisp lets you *extend* the reader, so that non-lisp objects are actually read in as lisp objects. Just as you build your applications on top of lisp, extending it with macros and functions, lisp applications can, and frequently do, ooze out into this dimension of extensibility as well. When this happens, any character based syntax can be read with the lisp reader, meaning that you have turned the syntax into lisp.

While the transformations of code done by regular macros are only used for turning lisp code into new lisp code, read macros can be created so as to turn non-lisp code into lisp code. Like regular macros, read macros are implemented with functions underneath so we have available the full power of the lisp environment. Like macros that increase productivity because they create more concise domain specific languages for the programmer to use, read macros boost productivity by allowing expressions to be abbreviated to the point where they aren't even lisp expressions anymore. Or are they?

If all we have to do to parse these non-lisp domain specific languages is write a short read macro, maybe these non-lisp languages really are lisp, just in a clever disguise. If XML can be directly read in by the lisp reader[XML-AS-READ-MACRO], maybe XML, in a twisted sort of sense, is actually lisp. Similarly, read macros can be used to read regular expressions and SQL queries directly into lisp, so maybe these languages really are lisp too. This fuzzy distinction between code and data, lisp and non-lisp, is the source of many interesting philosophical issues that have perplexed lisp programmers since the very beginning.

A basic read macro that comes built in with COMMON LISP is the #. read-time eval macro. This read macro lets you embed objects into the forms you read that can't be serialised, but can be created with a bit of lisp code. One fun example is making forms that become different values each time they are read:

```
* '(football-game
     (game-started-at
       #.(get-internal-real-time))
     (coin-flip
       #.(if (zerop (random 2)) 'heads 'tails)))

(FOOTBALL-GAME
  (GAME-STARTED-AT 187)
  (COIN-FLIP HEADS))
```

Even though it is the same expression, this form reads in differently each time:

```
* '(football-game
     (game-started-at
       #.(get-internal-real-time))
     (coin-flip
       #.(if (zerop (random 2)) 'heads 'tails)))

(FOOTBALL-GAME
  (GAME-STARTED-AT 309)
  (COIN-FLIP TAILS))
```

Note that the two forms surrounded by #. are evaluated at read time, not when the form is evaluated. The complete list has been formed after they are evaluated, and the before-after equivalence (as defined by **equal**) can be seen by re-evaluating the last form read in and comparing it with the previous results, using the * and + convenience variables of the REPL[1] *The * variable contains the value that resulted from evaluating the previous form, and the + variable contains that form.*
:

```
* (equal * (eval +))

T
```

Notice that because these forms are actually evaluated at read-time, this is different from using backquote, which we will look at more closely in the following section. We can evaluate a similar form that uses backquotes:

```
* `(football-game
     (game-started-at
       ,(get-internal-real-time))
     (coin-flip
       ,(if (zerop (random 2)) 'heads 'tails)))

(FOOTBALL-GAME
  (GAME-STARTED-AT 791)
  (COIN-FLIP HEADS))
```

but it will evaluate to a different result when we re-evaluate it, since backquote reads in as code for evaluation:

```
* (equal * (eval +))

NIL ; unless you're really fast and lucky
```

Backquote

Backquote, sometimes known as *quasiquote*[2] *Scheme programmers call it quasiquote and COMMON LISP programmers call it*
, and displayed as ‘, is a relative new-comer to mainstream lisp programming, and the concept is still almost completely foreign to languages other than lisp.

Backquote has a bizarre history of development in parallel with lisp. It is reported[QUASIQUOTATION] that early on nobody believed that nested backquotes worked right until a sharp programmer realised that they actually did work right—people's ideas of what was right were wrong. The nested backquote is notoriously difficult to understand. Even Steele, the father of COMMON LISP, complains about it[CLTL2-P530].

In principle, lisp doesn't need backquote. Anything that can be done with backquote can be done with other list building functions. However, backquote is so useful for macro programming, which in lisp means all programming, that lisp professionals have come to rely on it heavily.

First off, we need to understand regular quotation. In lisp, when we prefix a form with the quote character (') we are informing lisp that the following form should be treated as raw data, and not code to be evaluated. Rather, quote reads in as code that, when evaluated, returns a form. We sometimes say that quote *stops* or *turns off* the evaluation of a form.

Backquote can be used as a substitute for quote in lisp. Unless certain special characters, called *unquote* characters, appear in a form, backquote stops evaluation in the same way as quote. As the name suggests, these unquote characters reverse the evaluation semantics. We sometimes say that an unquote *restarts* or *turns back on* the evaluation of a form.

There are three main types of unquote: regular unquote, splicing unquote, and destructive splicing unquote.

To perform a regular unquote, we use the comma operator:

```
* (let ((s 'hello))
    `(,s world))

(HELLO WORLD)
```

Although the expression we are unquoting is simply a symbol to evaluate, **s**, it can instead be any lisp expression that evaluates to something meaningful for whatever context it appears in the backquote template. Whatever the results are, they are inserted into the resulting list in the car position of where they appear in the backquote template.

In lisp form notation, we can use . to indicate that we want to explicitly put something in the cdr of the list structure we are creating. If we put a list there, the resulting form from the backquote will remain a valid list. But if we put something else there, we will get a new, non-list structure.

We have this ability inside of backquote like everywhere else[3] *Because backquote uses the standard* ***read*** *function like (nearly*
. Thanks to the design of backquote, we can even unquote things in this position:

```
* (let ((s '(b c d)))
    `(a . ,s))

(A B C D)
```

Inserting lists into the cdr position of a list being created from a backquote template is so common that backquote takes it a step further with splicing unquote. The ., combination above is useful, but is incapable of inserting elements into the middle of a list. For that, we have the splicing unquote operator:

```
* (let ((s '(b c d)))
    `(a ,@s e))

(A B C D E)
```

Neither ., nor ,@ modify the list being spliced in. For instance, after evaluating the backquote in both of the previous forms, **s** will still be bound to the three element list **(B C D)**. Although it is not strictly required to by the standard, the **(B C D)** in the **(A B C D)** form above is allowed to share structure with the spliced-in list, **s**. However, in the list **(A B C D E)**, this list structure is guaranteed to be freshly allocated when the backquote is evaluated since ,@ is forbidden to modify the lists being spliced in. Splicing unquote is non-destructive because generally we want to think about backquote as being a re-usable template for creating lists. Destructively modifying the list structure of data that isn't freshly allocated on every evaluation of the backquote code can have undesirable effects upon future expansions.

However, COMMON LISP also provides a destructive version of splicing unquote which can be used anywhere splicing unquote can. To splice destructively, use ,. instead. Destructive splicing works the same as regular splicing except that the list being spliced in may be modified during the evaluation of the backquote template. As well as being only one character different from regular splicing, this notation is a clever re-use of the . character from the ., cdr position unquoting we looked at above.

To see this in action, here we destructively modify the list pointed to by to-splice:

```
* (defvar to-splice '(B C D))

TO-SPLICE
* `(A ,.to-splice E)

(A B C D E)
* to-splice

(B C D E)
```

Destructively modifying lists to be spliced in can be dangerous. Consider the following use of destructive splicing:

```
(defun dangerous-use-of-bq ()
  `(a ,.'(b c d) e))
```

The first time **dangerous-use-of-bq** is called, the expected answer is returned: **(A B C D E)**. But since it uses destructive splicing and modifies a list that isn't freshly generated—the quoted list—we can expect various undesirable consequences. In this case, the second time **dangerous-use-of-bq** is evaluated, the **(B C D)** form is now really a **(B C D E)** form, and when backquote tries to destructively splice this list onto the remainder of the backquote template, **(E)**—its own tail—it creates a list containing a *cycle*. We discuss cycles in more detail in section 4.5, Cyclic Expressions.

However, there are many cases where destructive splicing is perfectly safe. Don't let **dangerous-use-of-bq** scare you if you need more efficiency in your backquote forms. There are many operations that create fresh list structure that you are probably planning on throwing out anyways. For instance, splicing the results of a mapcar is so common and safe that the following could probably become a programming idiom:

```
(defun safer-use-of-bq ()
  `(a
    ,.(mapcar #'identity '(b c d))
    e))
```

But there is a reason it hasn't. The most common use of backquote is for authoring macros, the part of lisp programming where speed matters least and where clarity matters most. If thinking about the *side-effects* of your splicing operations distracts you even for a split second while creating and interpreting macros, it is probably not worth the trouble. This book sticks with regular splicing. The most common use of backquote is in macro construction but this is not its only use. Backquote is actually a useful domain specific language for the domain of mashing together lists, one made even more useful given the possibility of destructive splicing.

How does backquote work? Backquote is a read macro. Backquoted forms read in as code that, when evaluated, becomes the desired list. Going back to the example of the previous section on read-time evaluation, we can turn off *pretty printing*, quote the value of the backquote form, and print it out to see exactly how backquote forms read[4] *We return **t** so we don't see the value returned from **print**. **(values)** is also common.* :

```
* (let (*print-pretty*) ; bind to nil
    (print
      '`(football-game
          (game-started-at
            ,(get-internal-real-time))
          (coin-flip
            ,(if (zerop (random 2))
               'heads
               'tails))))
    t)

(LISP::BACKQ-LIST
  (QUOTE FOOTBALL-GAME)
  (LISP::BACKQ-LIST
    (QUOTE GAME-STARTED-AT)
    (GET-INTERNAL-REAL-TIME))
  (LISP::BACKQ-LIST
    (QUOTE COIN-FLIP)
    (IF (ZEROP (RANDOM 2))
      (QUOTE HEADS)
      (QUOTE TAILS))))
T
```

In the above *ugly printed* form, the function **LISP::BACKQ-LIST** is identical to list, except for its pretty printing behaviour. Notice that the comma operators are gone. COMMON LISP is fairly liberal in what it allows backquote to read in as, particularly for operations where shared structure is permitted.

Backquote also provides many interesting solutions to the amusing *non-problem* of writing a lisp expression that evaluates to itself. These expressions are commonly called *quines* after Willard Quine who studied them extensively and who, in fact, coined the term quasiquote—an alternative name for backquote[FOUNDATIONS-P31-FOOTNOTE3]. Here is a fun example of a quine that is attributed to Mike McMahon in [QUASIQUOTATION]:

```
* (let ((let '`(let ((let ',let))
                 ,let)))
    `(let ((let ',let)) ,let))

(LET ((LET '`(LET ((LET ',LET))
               ,LET)))
  `(LET ((LET ',LET)) ,LET))
```

To save you the *mental code-walk*:

```
* (equal * +)

T
```

Exercise: In the following evaluation, why is the backquote expanded into a regular quote? Isn't it quoted?

```
* '`q

'Q
```

Reading Strings

In lisp, strings are delimited by the double-quote (") character. Although strings can contain any character in the character set of your lisp implementation, you can't directly insert certain characters into the string. If you want to insert the " character or the \ character, you will need to prefix them with \ characters. This is called *escaping* the characters. For example, here is how to enter a string containing " and \ characters:

```
* "Contains \" and \\."

"Contains \" and \\."
```

This obviously works, but sometimes typing these \ characters becomes tedious and error prone. This is lisp, of course, and if we don't like something we are free, even encouraged, to change it. In this spirit, this book presents a read macro called #", or sharp-double-quote. This read macro is for creating strings containing " and \ characters without invoking escapes.

SHARP-DOUBLE-QUOTE

```
(defun |#"-reader| (stream sub-char numarg)
  (declare (ignore sub-char numarg))
  (let (chars)
    (do ((prev (read-char stream) curr)
         (curr (read-char stream) (read-char stream)))
        ((and (char= prev #\") (char= curr #\#)))
      (push prev chars))
    (coerce (nreverse chars) 'string)))

(set-dispatch-macro-character
  #\# #\" #'|#"-reader|)
```

Sharp-double-quote[5] *Our convention of naming the underlying functions of read macros with a symbol based on the read macro's* will start reading the string immediately after its invocation characters: # and ". It will continue to read, character by character, until it encounters the two characters " and #, in sequence. When it finds this terminating sequence, it returns the string represented by all the characters between the #" and "#. The sharp-double-quote read macro used to be used for bit-strings, but COMMON LISP freed up this useful macro character for us by moving bit-strings to the #* read macro[EARLY-CL-VOTES].

Here is an example evaluation of our new sharp-double-quote:

```
* #"Contains " and \."#

"Contains \" and \\."
```

Notice that when the REPL prints the string, it still uses the " character as a delimiter, so the " and \ characters are still escaped in the printed representation of the string. These strings are simply read in as if you escaped the characters manually.

But sometimes #" isn't good enough. For instance, in this U-language paragraph you are reading right now, I have included the following sequence of characters: "#. Because of this, this paragraph couldn't have been delimited with #" and "#. And because I hate escaping things, take my word that it wasn't delimited with regular double quotes.

SHARP-GREATER-THAN

```
(defun |#>-reader| (stream sub-char numarg)
  (declare (ignore sub-char numarg))
  (let (chars)
    (do ((curr (read-char stream)
               (read-char stream)))
        ((char= #\newline curr))
      (push curr chars))
    (let* ((pattern (nreverse chars))
           (pointer pattern)
           (output))
      (do ((curr (read-char stream)
                 (read-char stream)))
          ((null pointer))
        (push curr output)
        (setf pointer
              (if (char= (car pointer) curr)
                (cdr pointer)
                pattern))
        (if (null pointer)
          (return)))
      (coerce
        (nreverse
          (nthcdr (length pattern) output))
        'string))))

(set-dispatch-macro-character
  #\# #\> #'|#>-reader|)
```

We need a read macro that allows us to customise the delimiter for each specific context where we use it. As is often the case, we need to look no further than Larry Wall's Perl language for inspiration on the design of programming shortcuts. Perl is a beautiful, wonderfully designed language and possesses many great ideas that are ripe for *pilfering* by lisp. Lisp is, in some sense, a big blob, a snowball perhaps, that rolls around assimilating ideas from other programming languages, making them its own[6] *The most cited example of this is objects, but there are countless other examples such as the **format** function from FORTRAN*
.

The #> read macro is directly inspired by Perl's « operator. This operator allows Perl programmers to specify a string of text to act as the end delimiter for a quoted string. #> reads characters until it finds a newline character, then reads characters, one-by-one, until it encounters a sequence of characters identical to the characters it found immediately after #> and before the newline.

For example:

```
* #>END
I can put anything here: ", \, "#, and ># are
no problem. The only thing that will terminate
the reading of this string is...END

"I can put anything here: \", \\, \"#, and ># are
no problem. The only thing that will terminate
the reading of this string is..."
```

CL-PPCRE

CL-PPCRE[CL-PPCRE] is a high-performance regular expression library written on COMMON LISP. It was created by the widely respected lisp hacker Edi Weitz. On behalf of the lisp professionals everywhere who have benefited so much from CL-PPCRE and his other software, this section is dedicated to Edi

Weitz. When other people are talking, Edi is coding; code speaks louder than argument.

PPCRE, for those who aren't already familiar, stands for Portable Perl Compatible Regular Expressions. CL-PPCRE, like the code in this book, is *portable* because it can run in any ANSI-compliant COMMON LISP environment. CL-PPCRE, also like the code in this book, is open-source and freely available. Although CL-PPCRE is almost perfectly compatible with Perl, it is different from Perl in a few important ways. CL-PPCRE provides several notable lispy enhancements to regular expressions. There are three substantial ways that CL-PPCRE is different from the implementation of regular expressions in Perl.

First, CL-PPCRE is fast. Really fast. When compiled with a good native code compiler, benchmarks suggest that for most regular expressions CL-PPCRE is roughly twice as fast as Perl, often much faster. And Perl has one of the fastest non-lisp regular expression engines around: a highly optimised engine written in C. How is this possible? Surely Perl's low-level implementation should have a performance edge over anything written in a high-level language like lisp.

This misconception is known as the *performance myth*, the general version of which is the following: low level languages result in faster code because you can program closer to the hardware. As this book hopes to explain, for complicated systems this myth is false. Examples like CL-PPCRE demonstrate this. The more low-level a language is, the more it prevents you and your compiler from making the efficiency optimisations that actually matter.

With CL-PPCRE, the technical reason for the performance boost is simple: COMMON LISP, the language used to implement CL-PPCRE, is a more powerful language than C, the language used to implement Perl. When Perl reads in a regular expression, it can perform analysis and optimisation but eventually the regular expression will be stored into some sort of C data structure for the static regular expression engine to use when it attempts the matching. But in COMMON LISP—the most powerful language—it is essentially no more difficult to take this regular expression, convert it into a lisp program, and pass that lisp program to the optimising, native-code lisp compiler used to build the rest of your lisp system[7] *CL-PPCRE is actually more sophisticated than described here. It has its own compilation function and usually (unless you're*

. Because programs compiled with a C compiler don't have access to the C compiler, Perl is unable to compile regular expressions all the way down to machine code. Lisp's compilation model is in a different class from C altogether. In COMMON LISP, compiling things at run-time (as at anytime) is portable, seamless, done in the same process as your lisp image, garbage collected when no longer needed, and, due to its incremental nature, highly efficient.

The second major difference between CL-PPCRE and Perl is that CL-PPCRE isn't tied to a string-based notation for regular expressions. CL-PPCRE has been freed from a character representation and permits us to encode regular expressions as lisp forms (sometimes called *S-expressions*). Since such forms are the very notation we use for writing lisp programs and macros, we are allowed many more opportunities for *cohesion* in our abstractions. See the documentation and code of CL-PPCRE[CL-PPCRE] for details on using this regular expression notation, and also for an example of a well-designed, lispy domain specific language.

Sure, CL-PPCRE is great, but why are we discussing it in a chapter about read macros? The answer lies in the third and last way that CL-PPCRE is substantially different from Perl. In Perl, regular expressions are closely tied into the language. While lisp's syntax is the way it is to accommodate meta-programming, Perl's syntax is the way it is to accommodate regular expressions and other sorts of syntactic shortcuts. Part of the reason we use regular expressions so often in Perl code is due to the experience of writing them being so brief and painless.

To add a convenient programmer interface in a Perlish style, read macros come in very handy. Because programming read macros is programming lisp, we start off with a utility function: **segment-reader**. Given a stream, a delimiter character, and a count, **segment-reader** will read characters from the stream until the delimiter character is encountered. If the count is greater than 1, **segment-reader** will return a cons. The car of this cons is a string and the cdr is the result of a recursive invocation of **segment-reader** given a decremented count parameter to get the next segment[8] *In COMMON LISP, when the test clause of an if form missing the else clause is found false,* ***nil*** *is returned from the if. Exp*
.

SEGMENT-READER

```
(defun segment-reader (stream ch n)
  (if (> n 0)
    (let ((chars))
      (do ((curr (read-char stream)
                 (read-char stream)))
          ((char= ch curr))
        (push curr chars))
      (cons (coerce (nreverse chars) 'string)
            (segment-reader stream ch (- n 1))))))
```

For example, reading 3 segments from the stream t[9] *The stream* ***t*** *corresponds to standard input when done from the REPL h*
with a delimiter character of / is done like so:

```
* (segment-reader t #\/ 3)
abc/def/ghi/

("abc" "def" "ghi")
```

Perl programmers will probably see exactly where this is going. The idea is, with full apologies to Larry Wall, to *pilfer* the syntax for two handy Perl regular expression operators. In Perl, if we want to try matching a regular expression to a variable, we can write

```
$my_boolean = ($var =~ m/^\w+/);
```

to see if the contents of **$var** begin with one or more alphanumeric characters. Similarly, if we want to apply a *substitution* regular expression, we can also use the Perl =~ operator to apply a substitution regular expression to change the first occurrence of dog to cat in our string variable **$var**:

```
$var =~ s/dog/cat/;
```

The great thing about the Perl syntax is that the delimiter character can be any character that is convenient for the programmer. If we wanted to use a regular expression or a substitution containing the / character, we could use a different character to avoid any conflicts[10] *This may not be due to Perl; TeX's verbatim quotations provide something similar.*
:

```
$var =~ s|/usr/bin/rsh|/usr/bin/ssh|;
```

MATCH-MODES

```
#+cl-ppcre
(defmacro! match-mode-ppcre-lambda-form (o!args)
 ``(lambda (,',g!str)
     (cl-ppcre:scan
       ,(car ,g!args)
       ,',g!str)))

#+cl-ppcre
(defmacro! subst-mode-ppcre-lambda-form (o!args)
 ``(lambda (,',g!str)
     (cl-ppcre:regex-replace-all
       ,(car ,g!args)
       ,',g!str
       ,(cadr ,g!args))))
```

Defining a read macro to copy these two Perl syntaxes gives us a chance to demonstrate an interesting macro technique, the double backquote. The idea is that sometimes, as in the **match-mode-ppcre-lambda-form** and **subst-mode-ppcre-lambda-form** macros, we want to write code that generates lists. Notice that when you normally define a macro and use a single backquote, you are generating a list representing code and returning it from the macro for it to be spliced into expressions for evaluation. With a double backquote you are still generating a list representing code, but this code will, when evaluated, itself use code built by a backquote in order to return a list. In our case, these two macros expand into code that you can evaluate to create lambda forms that are useful for applying CL-PPCRE regular expressions.

We prefix these macros, and some other expressions below, with a #+

read macro. This read macro tests whether we have CL-PPCRE available[11] *It tests for CL-PPCRE by searching for the presence of the keyword symbol* ***:CL-PPCRE*** *in the list stored in the* ****features**** *v*
before evaluating the following form. If CL-PPCRE isn't available when loading the source code from this book, the functionality of this section will not be available.

CL-PPCRE-READER

```
#+cl-ppcre
(defun |#~-reader| (stream sub-char numarg)
  (declare (ignore sub-char numarg))
  (let ((mode-char (read-char stream)))
    (cond
      ((char= mode-char #\m)
         (match-mode-ppcre-lambda-form
           (segment-reader stream
                           (read-char stream)
                           1)))
      ((char= mode-char #\s)
         (subst-mode-ppcre-lambda-form
           (segment-reader stream
                           (read-char stream)
                           2)))
      (t (error "Unknown #~~ mode character")))))

#+cl-ppcre
(set-dispatch-macro-character #\# #\~ #'|#~-reader|)
```

Finally, we can define a reader function to pull together these utilities then add this function to our macro dispatch table. We chose to use the #~ read macro because it is a nice analog to the Perl =~, the source of inspiration for our syntax.

The #~ read macro is designed to be convenient. Here is how we can create a regular expression matching function:

```
* #~m/abc/

#<Interpreted Function>
```

We can now apply this function to a string just as a normal function call[12] *The * variable is bound to the value returned from the evaluation of the last expression entered in the REPL. Here it is bounc*
:

```
* (funcall * "123abc")

3
6
#()
```

```
#()
```

The values returned are from the **cl-ppcre:scan** function, documentation for which can be found in [CL-PPCRE]. If you are only interested in whether the string matched, the fact that the first value returned is not **nil** means that it did. Generalised booleans, and why they are an important feature of COMMON LISP, are discussed further in chapter 6, Anaphoric Macros.

We can also create substitution regular expression functions. A slight difference between Perl and our read macro is that substitution regular expression functions do not modify their arguments. They will return new strings, which are copies of the original strings with the substitutions made. Another difference is that, by default, this read macro substitutes all occurrences of the pattern instead of just the first in the string. In Perl you need to add a global modifier to your regular expression to get this behaviour, but not here:

```
* (funcall #~s/abc/def/ "Testing abc testing abc")

"Testing def testing def"
```

So how does this work? What do #~ expressions, which are clearly not lisp expressions, read in as? On the surface, it appears as though they read in as functions, but this turns out to not be the case. Let's quote one of these forms so we can see what it is according to the lisp reader:

```
* '#~m|\w+tp://|

(LAMBDA (#:STR1)
  (CL-PPCRE:SCAN "\\w+tp://" #:STR1))
```

Substitutions are similar:

```
* '#~s/abc/def/

(LAMBDA (#:STR2)
  (CL-PPCRE:REGEX-REPLACE-ALL
    "abc"
    #:STR2
    "def"))
```

They are read in as lambda forms. So as far as the lisp reader is concerned, we didn't write it in some funny non-lisp language after all. This is a function designator. Since our expressions are simply lists with the first element the symbol **lambda**, recall from section 2.4, Let It Be Lambda how we can use lambda forms in the first argument of a function call to invoke anonymous functions:

```
* (if (#~m/^[\w-.]+$/ "hcsw.org")
    'kinda-looks-like-a-domain
    'no-chance!)
```

```
KINDA-LOOKS-LIKE-A-DOMAIN
```

When we use **funcall** or **apply** to use the objects read in by #~, we make use of the ANSI **lambda** macro but not when the form is the first in the list: a useful *duality of syntax*. If our #~ expressions read in as sharp-quoted lambda forms we wouldn't be able to use them in the function position of an expression—only function names and lambda forms can go there. So for both tasks there only needs to be one read macro, which is fortunate because it is a large and complicated one. Taking advantage of dual syntax lets us focus on getting the correct expansion instead of tracking different syntax requirements. Instead of one interesting macro, we got two. To save effort, make your syntax as similar as possible.

A common problem when using CL-PPCRE is to forget to *escape* backslashes in your regular expressions. Look what happens when you do this:

```
* "\w+"

"w+"
```

This is a string of length 2. Where did the backslash go? Double-quote thought we meant to escape the w character instead of writing a literal \ character. For our #~ read macro that just reads characters and looks for the appropriate delimiter, this is not an issue and we can write regular expressions just as we do in Perl—without escapes. See the quoting of the URL regular expression above.

Although the #~ read macro defined in this section is already very convenient, there is still room for improvement and enhancement. Exercise: Improve it. The most obvious first step is to support regular expression modifiers, such as case insensitivity in matches. If done with the same syntax as Perl, this will involve using the function **unread-char**, which is common in read macros to avoid accidentally *eating* a character that some other read macro might be expecting.

Cyclic Expressions

All our talk about lisp programs being *trees* of cons cells has actually been a small lie. Sorry about that. Lisp programs are actually not trees but are instead *directed acyclic graphs*—trees with possibly shared branches. Since the evaluator doesn't care about where the branches it is evaluating come from, there is nothing wrong with evaluating code with shared structure.

A useful provided read macro is #=. We already saw how lisp can be made to output forms with the #= macro when serialising macro expansions in section 3.5, Unwanted Capture. #= and its partner ## let you create self-referential S-expressions. This allows you to do things like represent shared branches in directed acyclic graphs and other interesting data structures with little or no effort.

But most importantly it allows you to serialise data without having to disassemble and reassemble an efficient in-memory data structure where large portions of the data are shared. Here is an example where the two lisp lists read in are distinct objects (not eq):

```
* (defvar not-shared '((1) (1)))

((1) (1))
* (eq (car not-shared) (cadr not-shared))

NIL
```

But in the following example, with serialised data using the #= read macro, the 2 lists really are the same list:

```
* (defvar shared '(#1=(1) #1#))

((1) (1))
* (eq (car shared) (cadr shared))

T
```

As mentioned, we can give shared, acyclic list structure to the evaluator with no trouble:

```
* (list
    #1=(list 0)
    #1#
    #1#)

((0) (0) (0))
```

If we print the last form we just evaluated, we see it the same way the lisp evaluator does: a regular list with three separate branches:

```
* +

(LIST (LIST 0) (LIST 0) (LIST 0))
```

But if we bind the ***print-circle*** special variable to a non-nil value when we print it, we see that the expression is not really a tree at all, but instead a directed acyclic graph:

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

(LIST #1=(LIST 0) #1# #1#)
T
```

As another fun example, here's how to print an infinite list by pointing the cdr of a cons to itself, forming a so-called *cycle* or *circle*:

```
* (print '#1=(hello . #1#))

(HELLO HELLO HELLO HELLO HELLO HELLO HELLO
 HELLO HELLO HELLO HELLO HELLO HELLO HELLO
 HELLO HELLO HELLO HELLO HELLO HELLO HELLO
 ...
```

So unless you want that to happen, be sure you set ***print-circle*** when *serialising* cyclic data structures:

```
* (let ((*print-circle* t))
    (print '#1=(hello . #1#))
    nil)

#1=(HELLO . #1#)
NIL
```

CYCLIC-P

```
(defun cyclic-p (l)
  (cyclic-p-aux l (make-hash-table)))

(defun cyclic-p-aux (l seen)
  (if (consp l)
    (or (gethash l seen)
        (progn
          (setf (gethash l seen) t)
          (or (cyclic-p-aux (car l) seen)
              (cyclic-p-aux (cdr l) seen))))))
```

Is there an easy way to tell if a portion of list structure is cyclic or contains shared structure? Yes, the provided **cyclic-p** predicate uses the most obvious algorithm for discovering this: recurse across the structure keeping a *hash-table* up to date with all the cons cells you've encountered so far. If you ever come across a cons cell that already exists in your hash-table, you have already been there and therefore have detected a cycle or a shared branch. Notice that because it only recurses across cons cells, **cyclic-p** is incapable of discovering such references in data structures like vectors.

Finally, because most (see [SYNTACTICALLY-RECURSIVE]) lisp compilers forbid you from passing circular forms to the compiler, executing the following is undefined but likely to break your compiler by putting it into an infinite compiling loop:

```
(progn
  (defun ouch ()
    #1=(progn #1#))
  (compile 'ouch))
```

Reader Security

Extensibility, the ability to make things happen that weren't originally intended or anticipated, is almost always a good thing. In fact, encouraging extensibility wherever possible is what has made lisp as great as it is. However, there are times when we would prefer for things to be as inextensible as possible. In particular, we don't want outsiders to extend themselves into our systems without our knowledge or consent. That is known as being *hacked* or *r00ted*. Today, interesting computing is mostly about communication and networking. When you fully control both programs exchanging data, you obviously trust the entire system. But as soon as there is a possibility for some untrusted party to even partially control one of the programs, the trust system breaks down completely, like a toppling house of cards.

The largest source of these *security* problems arise from what programmers jokingly refer to as *impedance mismatch*. Whenever you use something you don't completely understand, there is a possibility you are using it wrong. There are two approaches to combating impedance mismatches: style (don't use **strcpy(3)**) and understanding (actually read that manual page). Lisp is a good language for writing secure software because, more so than any other language, lisp always does what is expected. If you just always follow the assumption that lisp does something *right*, you will hardly ever go wrong. For example, if you attempt to write outside the bounds of a string or vector, an obviously problematic situation, lisp will raise an exception and immediately and loudly notify you of the problem. In fact, lisp does this even more *right* than you might expect: after encountering an exception, you have the option of *restarting* your program at another location in your program, preserving most of the state of your computation. In other words, COMMON LISP's exception system doesn't automatically destroy your computation's stack when an exception occurs: you might still want to use that stack. Mostly due to space constraints, the exception system[13] *Actually called the condition system because it is useful for more than just exceptions.* is not described in much detail in this book. Instead, I recommend Peter Seibel's *Practical COMMON LISP*[PRACTICAL-CL].

But part of learning lisp is discovering that everything is extensible. How on earth can we limit this? It turns out that we are thinking about the problem in the wrong direction. As in all areas of computer security, you can't consider defence until you have considered offence. In all other areas of programming, you can arrive at good results constructively, that is by building and using abstractions. In security, you must think destructively. Instead of waiting for and then fixing bugs, you must try to find bugs by breaking your code.

So what attacks are we concerned with? There is no way to attack a program unless you control *input* to that program in some way. Of course in our networked world most programs are pretty useless unless people can give input to them. There are many protocols for shuffling data around the internet[14] *The **nmap-service-probes** file that I help maintain for the Nmap Security Scanner project is one of the most comprehensive an*

. The variety of things we would like to do is simply too vast to create a universal standard for data interchange. The best that can be done is to provide an extensible framework and allow programmers to customise the protocol to fit the application being created. This will generally mean less network overhead, better transfer algorithms, and more reliability. However, the main advantage is that when we design the protocol we can reduce or eliminate the impedance mismatch which is how to make secure protocols.

The problem with standards for interchanging data is that, in order to support the standard, applications are forbidden from reducing what can be done with the protocol. There is usually some base-line behaviour that must be met in order for an application to conform to the standard. To make secure protocols we need to be able to make sure we accept only what we are certain we can handle and no more.

So what is the lisp way to exchange data? The mechanism for getting data into lisp is called the *lisp reader* and the mechanism for getting it out is called the *lisp printer*. If you have made it this far into the book you already know more than enough to design and use lisp protocols. When you program lisp you are using such a protocol. You interact with lisp by feeding it lisp forms and this often turns out to be the best way to interact with the rest of the world too. Of course you don't trust the rest of the world so precautions must be taken. Remember that to think about security you must think about attacks. The designers of COMMON LISP were thinking about attacks against the reader during design. Earlier in this chapter we described the **#.** read macro that lets the reader execute lisp expressions so we can encode non-serialisable data structures. To mitigate an obvious attack against the lisp reader, COMMON LISP provides ***read-eval***. From CLtL2:

Binding ****read-eval**** *to* ***nil*** *is useful when reading data that came from an untrusted source, such as a network or a user-supplied data file; it prevents the* ***#.*** *read macro from being exploited as a "Trojan Horse" to cause arbitrary forms to be evaluated.*

When the ANSI COMMON LISP committee voted ***read-eval*** into being in June, 1989, they were thinking like attackers. What sort of trojan horse would an attacker include? The correct answer is, from a secure software author's point of view, the worst conceivable one you can think of—or worse. Always think that an attacker would like to be able to completely control your system. Traditionally, that means the trojan horse should be something called *shell code*. This is usually a carefully crafted chunk of machine code that does something like provide a unix shell for an attacker to use to further r00t the victim. Crafting this shell code is really an art-form, especially because of the unusual circumstances that such attacks usually exploit. For instance, most shell code cannot contain null bytes because with C-style strings these bytes terminate the string, preventing the inclusion of further shell code. Here is what an example of lisp shell code might look like, assuming the victim is running CMUCL and has Hobbit's original *netcat* (**nc**) [NETCAT] program installed:

```
#.(ext:run-program
    "/bin/nc" '("-e" "/bin/sh" "-l" "-p" "31337"))
```

The above will start listening for connections on the port 31337 and will supply unix shell access to anyone who connects. With traditional exploits, lots of effort is spent on trying to make them as portable and reliable as possible, that is so they will successfully r00t the most amounts of targets the most often. Often this is extremely difficult. In lisp reader attacks, it is extremely easy. Here is how we might update our shell code to make it portable between CMUCL and SBCL:

```
#.(#+cmu ext:run-program
   #+sbcl sb-ext:run-program
    "/bin/nc" '("-e" "/bin/sh" "-l" "-p" "31337"))
```

So the moral is to always make sure you bind ***read-eval*** to **nil** when processing any data that you even slightly distrust. If you rarely use the **#.** read macro, you might be wise to **setq** it to **nil** and only enable it when you expect to use it.

So we can disable the **#.** read macro fairly easily. But is this enough? It depends on your application and what is considered an effective attack. For interactive programs, this might be sufficient. If we get bad data we will hear about it as soon and loudly as possible. However, for internet servers this is probably not enough. Consider this shell code:

```
)
```

Or this:

```
no-such-package:rewt3d
```

Lisp will normally throw an error because we tried to read in an unbalanced form or lookup a symbol in a package that doesn't exist. Likely our entire application will grind to a halt. This is known as a *denial of service* attack. An even more subtle and more difficult to debug denial of service attack is to pass a circular form using the ## and #= read macros. If our code that processes this data wasn't written with such forms in mind, the result is an impedance mismatch and, likely, a security problem. On the other hand, maybe our application depends on being able to pass circular and shared data structures. The data security requirements depend completely on the application. Luckily, whatever your requirements, the lisp reader and printer are up to the task.

SAFE-READ-FROM-STRING

```
(defvar safe-read-from-string-blacklist
  '(#\# #\: #\|))

(let ((rt (copy-readtable nil)))
  (defun safe-reader-error (stream closech)
    (declare (ignore stream closech))
    (error "safe-read-from-string failure"))
  (dolist (c safe-read-from-string-blacklist)
    (set-macro-character
      c #'safe-reader-error nil rt))

  (defun safe-read-from-string (s &optional fail)
    (if (stringp s)
      (let ((*readtable* rt) *read-eval*)
        (handler-bind
          ((error (lambda (condition)
                    (declare (ignore condition))
                    (return-from
                      safe-read-from-string fail))))
          (read-from-string s)))
      fail)))
```

Safe-read-from-string is a partial answer to the problem of reader security. This function is less ready for production use than most of the other code in this book. You are advised to think carefully about the security requirements of your application and adapt (or even re-write) this code for your application. **Safe-read-from-string** is a very locked down version of **read-from-string**. It has its own copy of the default lisp *readtable*. This copy has had most of the interesting read macros removed, including the **#** dispatching macro. That means that vectors, bit-vectors, gensyms, circular references, **#.**, and all the rest are out. **Safe-read-from-string** will not even allow keywords or foreign package symbols. It will, however, allow any cons structure, not just well formed lists. It also allows numbers[15] *Exercise: What is the one class of numbers not allowed?* and strings.

Safe-read-from-string uses lisp's exception system to catch all errors thrown by the lisp **read-from-string** function. If there is any problem with reading from the string, including encountering unbalanced parenthesis or encountering any of the other read macros we have blacklisted in the **safe-read-from-string-blacklist** variable, **safe-read-from-string** will return the value passed as its second argument, or **nil** if none was provided (remember you might want to read in **nil**). Here is how it is typically used[16] *Though of course if we were using it in a macro we would use **defmacro!** and its automatic gensyms.*
:

```
(let* ((g (gensym))
       (v (safe-read-from-string
            user-supplied-string g)))
  (if (eq g v)
    (log-bad-data ; careful how it's logged!
      user-supplied-string)
    (process v)))
```

Of course this version of **safe-read-from-string** is very limited and will probably require modification for your application. In particular, you will probably want keyword symbols. Allowing them is easy: just bind a list without the **:** character to **safe-read-from-string-blacklist** when you use **safe-read-from-string** and be aware that your symbols might reside in multiple packages (including the **keyword** package). Even if you remove the **:** character, our above package shell code will be thwarted because we catch all errors during reading, including errors indicating nonexistent packages. ***Read-eval*** is always bound to **nil** in case you decide to remove the # character from the blacklist. If you do so, you might want to create a sub-blacklist for the # dispatching macro (might be a large blacklist). The vertical bar character is blacklisted so that we don't read in wacky looking symbols.

So we can lock down the reader as tightly as we feel necessary, in fact as tight as our application will allow. But even after we've made sure that there are no *attack vectors* through the software used to read in a form, how can we minimise the impedance mismatch between what we think the structure of a lisp form is and what it actually can be? We have to verify that it matches up with what we expect. Some data standards call this procedure *validation* against a *schema*, but lisp calls it **destructuring-bind** against an *extended lambda form*. All are terms that try to sound more important than is deserved of the simple concept they represent. The idea is that you want to be sure your data is of the form, or structure, that you expect it to be for some given processing. **Destructuring-bind** checks this structure for us, providing a very useful schema language that includes keywords and optional parameters, and also has the bonus that we get to name the different parts of structure as we go along.

I could give some examples of how to use **destructuring-bind** but it is actually not necessary: we have been using destructuring all along. The argument or parameter list that we insert immediately after the name of a macro when when we use **defmacro**, **defmacro!**, or **destructuring-bind** is called an extended lambda list to highlight the fact that it is more powerful than the destructuring performed for an ordinary lambda list. With extended lambda lists we can *nest* extended lambda lists to destructure list structure of arbitrary depth. Paul Graham's *On Lisp* has an excellent treatment of destructuring. Especially see Graham's **with-places** macro[ON-LISP-P237], preferably after reading section 6.7, Pandoric Macros.

So every time you write a macro or a function, you are, in a sense, treating the arguments that this macro or function will receive as data, and the extended or regular lambda list as the schema. In this light, validation of data seems easy. Lisp can validate that our data is structured as it should be and will raise error conditions if not. As above with the reader, when processing data that we even slightly distrust we should think very carefully about the possible attacks and then use lisp's powerful exception and macro systems to construct a validation scheme that allows only the very bare minimum required by the application and maps directly onto how our application works, reducing or eliminating any impedance mismatch. CL-PPCRE regular expressions are also indispensable for this task. No other language has the potential for secure software that lisp does and this will only become more apparent over time.