Let Over Lambda

Let Over Lambda -- 50 Years of Lisp

by Doug Hoyte

Programs That Program

Lisp Is Not Functional

One of the most common mischaracterisations of lisp is calling it a functional programming language. Lisp is not functional. In fact it can be argued that lisp is one of the least functional languages ever created. This misunderstanding has interesting roots and is a good example of how a small misnomer can have long-lasting influence and end up causing confusion long after the reasons behind the misnomer have passed into irrelevance. What is a functional language? The only definition that makes sense is

A functional language is a programming language made up of functions.

So what is a function? A function is a concept from *math* that has been with us for centuries:

A function is a static, well-defined mapping from input values to output values.

It seems like we use **defun** to define new functions in lisp. For example, the following seems to be a function that uses summation to map the set of all numbers to a new set, one that also includes all numbers:

```
(defun adder (x)
  (+ x 1))
```

Obviously we can apply this object to any number and get the mapped result from its return value, but is **adder** really a function? Well, confusingly, lisp tells us that it is[1] :

```
* (describe #'adder)

#<Interpreted Function> is a function.
```

[1] *If you haven't already seen the COMMON LISP* ***describe*** *function before, try it right now. Describe a function, a special form,*

But calling this object a function is one of the deepest-rooted misnomers in lisp history. What **defun** and lambda forms actually create are *procedures* or, more accurately still, *funcallable instances*[AMOP]. Why the distinction? Procedures don't necessarily have anything to do with mapping values, but instead are pieces of code, possibly with stored environment, that can be executed (funcalled). When lisp programmers write in a certain style called *functional style*, then the resulting procedures can be thought of and combined together pretending that they are math-style functional maps.

The reason why lisp is so often incorrectly described as a functional language has to do with history. Believe it or not but there was once a time when most languages didn't even support the concept of a procedure that modern programmers in any language take for granted. Very early languages didn't provide a suitable abstraction for locally naming arguments in a piece of re-usable code and programmers had to manually allocate registers and manipulate stacks to achieve this behaviour. Lisp, a language that had procedures all along, seemed way more functional than these languages.

Next, once procedural abstraction was given the attention it deserved and incorporated into almost all programming languages, people started to slowly run up against barriers due to the limited nature of the procedures they had implemented. Programmers started to realise that they would like to be able to return procedures from other procedures, embed them in new environments, aggregate them in data-structures, and, in general, treat them as any old regular value. A slogan sprang up to mobilise programmers for this next big abstraction push: a society without classes, *first-class procedures*. In comparison to languages that relegated the procedure to an inferior second class, lisp, a language that has had these first-class procedures all along, seemed way more functional.

Finally, many languages make a pointless distinction between expression and statement, usually to support some horrible Blub syntax like infix assignment. In lisp, everything returns something[2] and there are no (syntactic) restrictions on what you can nest or combine. It is a simple question with an obvious answer: what is more important in a language, newbie-friendly syntax or real flexibility? Any language that uses infix syntax is reducing the possibilities of its abstractions in many ways. Luckily most modern languages have decided to trust their users enough to allow them to combine expressions mostly as they see fit. In comparison to languages that make these sorts of brain-dead syntax decisions, lisp seems way more functional.

[2] *Except* ***(values)*** *which returns nothing. But even this is coerced to* ***nil*** *and so can be used in*

After having become comfortable with this ubiquitous yet misguided terminology, programmers started to realise that the notion of function that had been used in the great functional vs non-functional programming language debate is not only confusing, but is actually fundamentally backwards. To correct this, programmers and academics alike went back to the drawing board, returning to the mathematical definition of a function: a map from input values to output values. If lisp is in any way a functional language, it is only as much so as modern languages such as Perl and Javascript.

Clearly lisp procedures are not functions. Lisp procedures can return non-static values, that is you can call them multiple times with the same arguments and receive different return values each time. As in our examples from previous chapters, lisp procedures can store state by closing over variables. Procedures like **rplaca** can change values in memory or elsewhere (such as registers). Lisp procedures like **terpri** and **format** create output (newlines in the case of **terpri**) directed to terminals or files[3] . Lisp procedures like **yes-or-no-p** can read input from a terminal and return values depending on what the user entered. Are these procedures static, well-defined mappings?

[3] ***Terpri*** *and* ***rplaca****—both have reasonable claim to the distinction of worst-named COMMON LISP operator.*

Because lisp procedures are not mathematical functions, lisp is not a functional language. In fact, a strong argument can be made that lisp is even less functional than most other languages. In most languages, expressions that look like procedure calls are enforced by the syntax of the language to be procedure calls. In lisp, we have macros. As we've seen, macros can invisibly change the meaning of certain forms from being function calls into arbitrary lisp expressions, a technique which is capable of violating referential transparency in many ways that simply aren't possible in other languages.

After considering that most languages are in fact not at all functional, some language designers decided to find out what programming in a really functional language would be like. As you would expect, programming functional languages is mostly annoying and impractical. Almost no real-world problems can be usefully expressed as static, well-defined mappings from input values to output values. That being said, functional programming is not without merit and many languages have been designed to take advantage of a functional style of programming. What that means is finding a convenient way of isolating the functional parts of a program from the (actually interesting) non-functional parts. Languages like Haskell and Ocaml use this isolation as a means of making aggressive optimisation assumptions.

But this is lisp. We're very non-functional and very proud of it. To the extent that this isolation of side-effects is useful, lisp programmers can and do implement it with macros. The real purpose behind functional programming is to separate the functional description of what should happen from the mechanics of how it actually does happen. Lisp is definitely not functional, but, because of macros, there is no better platform or material for implementing functional languages than lisp.

Top-Down Programming

You cannot teach beginners top-down programming, because they don't know which end is up. —C.A.R. Hoare

In section 3.2, Domain Specific Languages when we first looked at domain specific languages, we created a simple macro, **unit-of-time**. This macro lets us

conveniently specify periods of time in various units with an intuitive, symbol-based syntax:

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

86400
```

Unit-of-time is a handy domain specific language because the programmer isn't required to remember, for instance, how many seconds are in a day. **Unit-of-time** is implemented with a simple macro that uses a case statement as the heart of the underlying expansion.

An important principle of macro design is *top-down* programming. When designing a lisp macro, you want to start with the abstraction first. You want to write programs that use the macro long before you write the macro itself. Somewhat paradoxically, you need to know how to program in a language before you can write a concise definition/implementation for that language.

So the first step in serious macro construction is always to write *use cases* of the macro, even though there is no way you can test or use them. If the programs written in the new language are comprehensive enough, a good idea of what will be required to implement a compiler or interpreter for the language will follow.

Considering our **unit-of-time** macro, is there a way we can bump this up another level of specification and create a language for creating these sorts of unit convenience macros? Well, **unit-of-time** is a macro, so to do this we will need a macro defining macro...

Stop! That end is up.

We are not starting by considering the implementation of a language, but rather asking ourselves what we would like to use this language for. The answer is that we would like a simple way to define these sorts of unit conversion helpers. In the following example use case, we would like to be able to take a type of unit, **time**, a base unit, seconds, represented here by **s**, and a collection of pairs of a unit and the conversion factor for this unit to the base unit:

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

Defunits% could then expand into code that defines a macro like **unit-of-time** we wrote in section 3.2, Domain Specific Languages, allowing us to convert arbitrary time units to seconds. Could it get any better?

This is the point in the design brainstorming that innovation halts in most programming languages. We just created a way to map multiplier values for different units into code that allows us to convert units at our convenience. But

a professional lisp programmer recognises that this mapping is a program itself and can be enhanced through the means we regularly enhance lisp programs.

When we are entering many different units it might be helpful to specify units in terms of other units. Let's mandate that the factor used to multiply the unit could also be a list with a value relative to another unit, like so:

```
(defunits%% time s
  m 60
  h (60 m)
  d (24 h)
 ms (1/1000 s)
 us (1/1000 ms))
```

This *chaining* of units just feels natural. Minutes are specified in terms of our base unit, seconds, hours in terms of minutes, and days in terms of hours. To implement this macro in an iterative fashion we first implement unchained behaviour with **defunits%** and next implement chaining with **defunits%%** before add appropriate error checking and settle on our final version, **defunits**.

Notice that this new language can provide more than just a convenient syntax for adding new types of units. This language also allows us to delay the impact of *rounding* in our calculations and also lets lisp use as much exact arithmetic as possible. For instance, the furlong is defined to be exactly 1/8th of a mile, so if we encoded it that way using chaining instead of, say, a metric approximation, we could end up with more accurate results or, perhaps more importantly, results that are as consistent as possible with other calculations using miles. Because we can just add the most precise conversion factor we find and don't have to bother doing any converting ourselves, this macro will let us build conversion routines at a level of expression not possible in other languages.

Using our automatic gensym behaviour described in section 3.5, Unwanted Capture, **defunits%** is fairly easy to write. Graham's **symb** function lets us generate the new name for the conversion macro. For example, if the symbol **time** is the type of unit provided then the new conversion macro will be **unit-of-time**. **Defunits%** was constructed by taking our original definition of **unit-of-time**, defined in section 3.2, Domain Specific Languages, surrounding it with a **defmacro!** and a backquote, and replacing the parts that need to be regenerated each time the macro is invoked.

DEFUNITS-1

```
(defmacro! defunits% (quantity base-unit &rest units)
  `(defmacro ,(symb 'unit-of- quantity) (,g!val ,g!un)
     `(* ,,g!val
         ,(case ,g!un
           ((,base-unit) 1)
           ,@(mapcar (lambda (x)
                       `((,(car x)) ,(cadr x)))
```

```
(group units 2))))))
```

Defunits% uses a *nested backquote*—a construct notoriously difficult to understand. Programming with backquotes is almost like having one more dimension of meaning in your code. In other languages, a given programming statement usually has very simple evaluation semantics. You know when every bit of code will run because every bit of code is forced to run at the same time: run-time. But in lisp by nesting backquotes we can scale up and down a *ladder of quotation*. Every backquote we write takes us one step up this ladder: our code is a list that we may or may not evaluate later. But inside the raw list, every comma encountered takes us back down the quotation ladder and actually executes the code from the appropriate step on the ladder[CLTL2-P967].

So there is a simple algorithm for determining when any bit of code in a lisp form will be evaluated. Simply start from the root of the expression and after encountering a backquote, mark up one level of quotation. For every comma encountered, mark down one level of quotation. As Steele notes[CLTL2-P530], following this level of quotation can be challenging. This difficulty, the need to track your current quotation depth, is what makes using backquote feel like another dimension has been added onto regular programming. In other languages you can walk north, south, east, and west, but lisp also gives you the option of going up.

Defunits% is a good first step, but still doesn't implement chaining. Right now, the macro implementing this language is mostly a straightforward substitution. Implementing our chaining behaviour will require more sophisticated program logic. Simple substitution won't work because parts of the macro depend on other parts of the macro, so when we build our expansion we need to process the forms provided to the macro completely, not just thinking about it in terms of individual pieces we can splice in.

DEFUNITS-CHAINING-1

```
(defun defunits-chaining% (u units)
  (let ((spec (find u units :key #'car)))
    (if (null spec)
      (error "Unknown unit ~a" u)
      (let ((chain (cadr spec)))
        (if (listp chain)
          (* (car chain)
             (defunits-chaining%
               (cadr chain)
               units))
          chain)))))
```

Remembering that macros are really just functions, we create a utility function for use in the macro definition, **defunits-chaining%**. This utility function takes a unit, as specified by symbols like **S**, **M**, or **H**, and a list of unit specs. We let unit specs either have a single number, which is interpreted as meaning

base units, like **(M 60)**, or a list with an indirection to another unit in a chain, like **(H (60 M))**.

This utility function is recursive. In order to find the multiplier against the base unit, we multiply each step in the chain with another call to the utility function to work out the rest of the chain. When the call stack winds back, we get the multiplier used to convert values of a given unit into the base unit. For instance, when we're constructing the multiplier for hours, we find that there are 60 minutes in an hour. We recurse and find there are 60 seconds in a minute. We recurse again and find that seconds is the end of a chain—minutes were specified directly in terms of the base unit. So, winding back from the recursion, we evaluate **(* 60 (* 60 1))**, which is 3600: there are 3600 seconds in an hour.

DEFUNITS-2

```
(defmacro! defunits%% (quantity base-unit &rest units)
  `(defmacro ,(symb 'unit-of- quantity) (,g!val ,g!un)
     `(* ,,g!val
         ,(case ,g!un
           ((,base-unit) 1)
           ,@(mapcar (lambda (x)
                       `((,(car x))
                           ,(defunits-chaining%
                              (car x)
                              (cons `(,base-unit 1)
                                (group units 2)))))
                     (group units 2))))))
```

With this utility function defined, working out the multiplier for each unit requires only a simple modification to **defunits%**, which we have done in **defunits%%**. Instead of splicing in the value straight from the unit spec, we pass each unit and the entire unit spec to the **defunits-chaining%** utility. As described above, this function recursively works out the multiplier required to turn each unit into the base unit. With this multiplier, **defunits%%** can splice the value into a case statement just like **defunits%**.

As yet, these macros are incomplete. The **defunits%** macro didn't support chaining. **Defunits%%** supported chaining but lacks *error checking*. A professional macro writer always takes care to handle any error conditions that might arise. Cases that involve infinite loops, or are otherwise difficult to debug in the REPL, are especially important.

DEFUNITS

```
(defun defunits-chaining (u units prev)
  (if (member u prev)
    (error "~{ ~a~^ depends on~}"
      (cons u prev)))
  (let ((spec (find u units :key #'car)))
```

```
    (if (null spec)
      (error "Unknown unit ~a" u)
      (let ((chain (cadr spec)))
        (if (listp chain)
          (* (car chain)
             (defunits-chaining
               (cadr chain)
               units
               (cons u prev)))
          chain)))))

(defmacro! defunits (quantity base-unit &rest units)
  `(defmacro ,(symb 'unit-of- quantity)
             (,g!val ,g!un)
     `(* ,,g!val
         ,(case ,g!un
           ((,base-unit) 1)
           ,@(mapcar (lambda (x)
                       `((,(car x))
                         ,(defunits-chaining
                            (car x)
                            (cons
                              `(,base-unit 1)
                              (group units 2))
                            nil)))
                     (group units 2))))))
```

The problem with **defunits%%** is actually a property of the language we designed: it is possible to write programs that contain cycles. For example:

```
(defunits time s
  m (1/60 h)
  h (60 m))
```

In order to provide proper debugging output, we need to enhance our implementation slightly. Our final version, **defunits**, allows chaining and also provides useful debugging output should a user of the language specify a program with such a cyclic dependency. It can do this because it uses **defunits-chaining**, an improved version of **defunits-chaining%** that maintains a list of all the units that have previously been visited. This way, if we ever visit that unit again through chaining, we can throw an error that concisely describes the problem:

```
* (defunits time s
    m (1/60 h)
    h (60 m))

Error in function DEFUNITS-CHAINING:
  M depends on H depends on M
```

The **defunits** macro is identical to **defunits%%** except that it passes an additional argument **nil** to **defunits-chaining** which is the end of the list representing the history of units that we've already visited. If a new unit is searched for and we've already visited it, a cycle has been detected. We can use this history of units visited to display a helpful message to users of the macro, very probably ourselves, that accidentally write cycles.

So **defunits** is a language specific to the domain of entering units to a conversion routine. Actually it is specific to a much finer domain than that; there are many possible ways it could have been written. Because it is hard to create languages in Blub, and it is easy in lisp, lisp programmers usually don't bother trying to cram everything into one domain. Instead, they just make the language more and more specific to the domain in question until the eventual goal becomes trivial.

UNIT-OF-DISTANCE

```
(defunits distance m
  km 1000
  cm 1/100
  mm (1/10 cm)
  nm (1/1000 mm)

  yard 9144/10000 ; Defined in 1956
  foot (1/3 yard)
  inch (1/12 foot)
  mile (1760 yard)
  furlong (1/8 mile)

  fathom (2 yard) ; Defined in 1929
  nautical-mile 1852
  cable (1/10 nautical-mile)

  old-brit-nautical-mile ; Dropped in 1970
    (6080/3 yard)
  old-brit-cable
    (1/10 old-brit-nautical-mile)
  old-brit-fathom
    (1/100 old-brit-cable))
```

An example use of **defunits** is **unit-of-distance**. Just in case you ever wondered, the 1970 adoption of the international nautical mile shortened the fathom, at least to British sailors, by 1/76th:

```
* (/ (unit-of-distance 1 fathom)
     (unit-of-distance 1 old-brit-fathom))

75/76
```

Which is just over 2 centimetres:

```
* (coerce
    (unit-of-distance 1/76 old-brit-fathom)
    'float)

0.024384
```

Implicit Contexts

Macros can leverage a technique called *implicit context*. In code that is used frequently, or absolutely must be concise and lacking any surrounding book-keeping cruft, we sometimes choose to implicitly add lisp code around portions of an expression so we don't have to write it every time we make use of an abstraction. We've talked before about implicit contexts and it should be clear that even when not programming macros they are a fundamental part of lisp programming: let and lambda forms have an *implicit progn* because they evaluate, in sequence, the forms in their body and return the last result. **Defun** adds an *implicit lambda* around forms so you don't need to use a lambda form for named functions.

This section describes the derivation and construction of a *code-walking* macro used later in this book, **tree-leaves**[4] *Also see the COMMON LISP function* ***subst****.* . Like **flatten**, this macro inspects a piece of lisp code, considering it a tree, then performs some modifications and returns a new tree. The list structure of the original expression is not modified: **flatten** and **tree-leaves** both cons new structure. The difference between the two is that while the purpose of **flatten** is to remove nested lists and return a flat list that isn't really lisp code, **tree-leaves** preserves the shape of the expression but changes the values of particular atoms.

TREE-LEAVES-1

```
(defun tree-leaves% (tree result)
  (if tree
    (if (listp tree)
      (cons
        (tree-leaves% (car tree)
                      result)
        (tree-leaves% (cdr tree)
                      result))
      result)))
```

Let's start off with a simple sketch. **Tree-leaves%** is a function that recurses across the provided **tree** expression, consing a new list structure with the same shape[5] *An empty else clause in an if form returns* ***nil****, which is also the empty list.* . When it discovers an atom, instead of returning that atom it returns the value of the **result** argument:

```
* (tree-leaves%
```

```
    '(2 (nil t (a . b)))
    'leaf)

(LEAF (NIL LEAF (LEAF . LEAF)))
```

So **tree-leaves%** returns a new tree with all atoms converted into our provided symbol, **leaf**. Notice that the atom **nil** in a car position of a cons cell is not changed, just as it isn't changed when it resides in the cdr position (and represents an empty list).

Changing every element is, of course, pretty useless. What we really want is a way to pick and choose specific atoms and selectively apply transformations to them to get new atoms to insert into the new list structure, leaving atoms we aren't interested in untouched. In lisp, the most straightforward way to write a customisable utility function is to allow *plug-ins* where the user can use custom code to control the utility's behaviour. The **sort** function included with COMMON LISP is an example of this. Here, the less-than function is plugged-in to **sort**:

```
* (sort '(5 1 2 4 3 8 9 6 7) #'<)

(1 2 3 4 5 6 7 8 9)
```

This concept of taking a function to control behaviour is especially convenient because we can create anonymous functions suited to the task at hand. Or, when we need even more power, we can create functions that create these anonymous functions for us. This is known as *function composition*. While function composition is not nearly as interesting as macro composition[6] *Which is why function composition only gets a couple paragraphs where macro composition gets most of this book.* , it is still an extremely useful technique that all professional lisp programmers must master.

PREDICATE-SPLITTER

```
(defun predicate-splitter (orderp splitp)
  (lambda (a b)
    (let ((s (funcall splitp a)))
      (if (eq s (funcall splitp b))
        (funcall orderp a b)
        s))))
```

A simple example of function composition is **predicate-splitter**. This function exists to combine two predicates into a single, new predicate. The first predicate takes two arguments and is used for ordering elements. The second predicate takes one argument and determines whether an element is in the special class of elements you want to split your predicate over. For example, in the following we use **predicate-splitter** to create a new predicate that works exactly like less-than except for the fact that even numbers are considered less than odd numbers:

```
* (sort '(5 1 2 4 3 8 9 6 7)
        (predicate-splitter #'< #'evenp))

(2 4 6 8 1 3 5 7 9)
```

TREE-LEAVES-2

```
(defun tree-leaves%% (tree test result)
  (if tree
    (if (listp tree)
      (cons
        (tree-leaves%% (car tree) test result)
        (tree-leaves%% (cdr tree) test result))
      (if (funcall test tree)
        (funcall result tree)
        tree))))
```

So how can we use functions as plug-ins to control how **tree-leaves%** works? In an updated version of **tree-leaves%**, **tree-leaves%%**, we add two different plug-in functions, one to control which leaves to change and one to specify how to convert an old leaf into a new leaf, respectively called **test** and **result**.

We can use **tree-leaves%%** by passing it two lambda expressions, both of which must take a single argument, **x**. In this case we want a new tree: one with identical list structure to our **tree** argument except that all even numbers are changed into the symbol **even-number**:

```
* (tree-leaves%%
    '(1 2 (3 4 (5 6)))
    (lambda (x)
      (and (numberp x) (evenp x)))
    (lambda (x)
      'even-number))

; Note: Variable X defined but never used.

(1 EVEN-NUMBER (3 EVEN-NUMBER (5 EVEN-NUMBER)))
```

It seems to work except lisp correctly calls attention to the fact that we don't make use of the **x** variable in our second plug-in function. When we don't use a variable it is often a sign of a problem in the code. Even when it is deliberate, like here, the compiler appreciates information on which variables should be ignored. Normally we will make use of this variable but there are cases, like this one, where we actually do not want to. It's too bad we have to take an argument for this function—after all we're just ignoring that argument anyways. This situation comes up often when writing macros designed to be flexible. The solution is to declare to the compiler that it is acceptable to ignore the variable **x**. Because it doesn't hurt to declare a variable ignorable and still use

it[7] *Lisp will figure out that it actually can't be ignored.* , we may as well declare both **x** variables to be ignorable:

```
* (tree-leaves%%
    '(1 2 (3 4 (5 6)))
    (lambda (x)
      (declare (ignorable x))
      (and (numberp x) (evenp x)))
    (lambda (x)
      (declare (ignorable x))
      'even-number))

(1 EVEN-NUMBER (3 EVEN-NUMBER (5 EVEN-NUMBER)))
```

Here is where this tutorial gets interesting. It seems like **tree-leaves%%** will work fine for us. We can change any leaves in the tree by providing plug-in functions which verify whether the leaf should be changed and what it should be changed to. In a programming language other than lisp, this would be where improvement to the utility stops. But with lisp, we can do better.

Although **tree-leaves%%** provides all the functionality we desire, its interface is inconvenient and redundant. The easier it is to experiment with a utility the more likely we are to find interesting future uses for it. To reduce the surrounding clutter of our code-walking utility, we create a macro that provides implicit context for its users (probably ourselves).

But instead of something simple like an implicit progn or an implicit lambda, we want an entire implicit lexical context that saves us all the overhead of creating these plug-in functions and only requires us to enter the bare minimum of code when doing common tasks like translating trees. This implicit lexical context is not like simple implicits in the sense that we didn't just find another use for a common implicit pattern. Instead, we developed a not-so-common pattern, step by step, when we developed our **tree-leaves%%** walking interface.

TREE-LEAVES

```
(defmacro tree-leaves (tree test result)
  `(tree-leaves%%
     ,tree
     (lambda (x)
       (declare (ignorable x))
       ,test)
     (lambda (x)
       (declare (ignorable x))
       ,result)))
```

For the construction of our implicit macro, the **tree-leaves%%** use case in the REPL above was effectively copy-and-pasted directly into the definition of **tree-leaves** and then the parts that we expect to change upon different uses

of this macro were parameterised using backquote. Now, through this macro, we have a less redundant interface for using the utility **tree-leaves%%**. This interface is, of course, completely arbitrary since there are many possible ways it could have been written. However, this seems to be the most intuitive, fat-free way, at least for the uses we have so-far envisioned. Macros have allowed us to create an efficient programmer interface in a simple, direct way impossible in other languages. Here is how we can use the macro:

```
* (tree-leaves
    '(1 2 (3 4 (5 . 6)))
    (and (numberp x) (evenp x))
    'even-number)

(1 EVEN-NUMBER (3 EVEN-NUMBER (5 . EVEN-NUMBER)))
```

Notice the variable **x** is actually used without it appearing to have been defined. That is because there is an *implicit lexical variable* bound around each of the last two expressions. This introduction of a variable without it being visible is said to violate *lexical transparency*. Another way to say it is that an *anaphor* named **x** is introduced for those forms to make use of. We will develop this idea much, much further in chapter 6, Anaphoric Macros.

Code-Walking with Macrolet

Lisp isn't a language, it's a building material. —Alan Kay

Forms of expression that are written and seldom spoken, like computer code, often breed diverse pronunciation habits. Most programmers run a dialogue in their heads, reasoning expressions and pronouncing operators, sometimes consciously, often not. For example, the most obvious way to pronounce the name of the lisp special form **macrolet** is simply the audible concatenation of its two lispy component words: macro and let. But after reading Steele's observation[CLTL2-P153] that some lisp programmers pronounce it in a way that rhymes with *Chevrolet*, it can be difficult to get this humorous pronunciation out of your programming dialogue.

However it is pronounced, **macrolet** is a vital part of advanced lisp programming. **Macrolet** is a COMMON LISP special form that introduces new macros into its enclosed lexical scope. Writing **macrolet** syntax transformations is done in the same way as defining global macros with **defmacro**. Just as lisp will expand occurrences of **defmacro**-defined macros in your code, **macrolet**-defined macros will be expanded by the lisp system when it *code-walks* your expressions.

But **macrolet** is not just a convenience. It offers a number of important advantages over using **defmacro** to define macros. First, if you want to have uses of a macro expand differently given their lexical contexts in an expression, creating different **macrolet** contexts is required. **Defmacro** just won't work.

Most importantly, **macrolet** is useful because of how difficult it is to code-walk

COMMON LISP expressions. Often we have an arbitrary tree of lisp code, say because we are macro-processing it, and we would like to change the values or meaning of different branches of the tree. To do things like implement temporary meanings for certain forms, and to temporarily over-ride specific macros, perhaps only in particular portions of the lexical context of an expression, we need to walk code. Specifically, we need to recurse through the code, looking for our desired macro or function name in a position where it is being evaluated, and substitute our own expression in its place.

Easy, right? The difficulty is that many legitimate lisp code fragments will break a naive code-walker implementation. Consider if we would like to perform special substitutions for evaluations of a function of a certain symbol, say, **blah**. If we are given the following expression, it is easy to tell that the substitution should occur:

```
(blah t)
```

Blah appears in the function position of a list that will get evaluated when the expression is evaluated so we should obviously make the substitution. So far so good. But what if we are passed this form:

```
'(blah t)
```

Because it is quoted, this bit of code is meant to return a literal list. Performing the substitution here would be incorrect. So our code-walker must know to stop when it hits a quote and make no substitutions in the form being quoted. Fine, that is easy enough. But let's consider if there are any other situations where it would be incorrect to expand **blah**. What if the code is using **blah** as the name for a lexical variable?

```
(let ((blah t))
  blah)
```

Even though **blah** appears as the first element of a list, here it appears in a local binding for a let form and a use of that binding so should not be expanded. Even this doesn't seem too bad. We could add some special case logic to our code-walker so it knows what to do when it encounters a let form. Unfortunately, we still have 23 more ANSI COMMON LISP special forms[8] that need to have special case logic added for them. What's more, many special forms are complicated to correctly walk. **Let**, as we've already seen, can be tricky, and it gets even worse. The following potentially legal COMMON LISP form contains one use of **blah** that should be expanded. Which one?

[8] *There are 25 special forms in ANSI CL, 23 without* ***let*** *and* ***quote****.*

```
(let (blah (blah (blah blah)))
  blah)
```

So code-walking is difficult because correctly handling all the special forms is difficult (also see [SPECIAL-FORMS] and [USEFUL-LISP-ALGOS2]). Notice that we don't need any special case logic for forms defined as macros. When a macro

is encountered, we can simply expand it until it turns into a function call or a special form. If it is a function, we know it follows lambda's once-only, left-to-right evaluation semantics. It is the special forms that we need to develop special case logic for.

Sounds like a lot of work, doesn't it? It is. A complete COMMON LISP code-walker, especially if designed to be portable, is a large, complicated piece of code. So why doesn't COMMON LISP provide us an interface for code-walking COMMON LISP code? Well, it turns out, in a way, it does, and, in a way, it's called **macrolet**. Code-walking is exactly what your COMMON LISP system needs to do before it can evaluate or compile an expression. Just like our hypothetical code-walker, COMMON LISP needs to understand and handle the special semantics of **let** and other special forms.

Since COMMON LISP has to walk our code in order to evaluate it, there is usually little need for a separate code-walking program. If we want to make selective transformations to expressions in a way that is smart about what will actually be evaluated, we can simply encode our transformation as a macro and then wrap a **macrolet** form around the expression. COMMON LISP will code-walk this expression when it is evaluated or compiled and will apply the macro transformations specified by the **macrolet**. Of course since **macrolet** defines macros, it does not impose any additional overhead at run-time. Macrolet is for communicating with COMMON LISP's code-walker and the only guarantee that COMMON LISP makes as to when a macro will be expanded is that it will be done before the run-time of a compiled function[CLTL2-P685][ON-LISP-P25].

One of the most common scenarios for using **macrolet** is when you want to pretend that a function is bound in a certain lexical context but you would like the behaviour of using this form to be something other than a function call. **Flet** and **labels** are out—they can only define functions. So our choices are to write a code-walker to look for invocations of this function and replace them with something else, define a global macro with **defmacro** so the "function" will expand into something else, or wrap the form in a **macrolet** and let the system's code-walker sort it out for us.

As mentioned above, writing a code-walker is hard. If at all possible we should avoid that route. Using a global **defmacro** is sometimes possible but often problematic. The largest issue is that COMMON LISP makes few guarantees about when—or how often—macros will be expanded, so we can't reliably use the same name to have different meanings in different lexical contexts. When we overwrite a global macro we can't know if COMMON LISP has expanded—or will want to again expand in the future—old uses of the macro.

For an example of where this code-walking is useful, let's re-visit an issue that we swept under the rug in section 3.3, Control Structures. Our initial version of the Scheme named let macro, **nlet**, used the **labels** special form to create a new type of control structure. This use of **labels** allowed us to temporarily define a function for use inside the named let body that would allow us to recurse, as if we

started the **let** over again with new values for the **let** bindings. When we defined this function, we mentioned that because COMMON LISP does not guarantee that it will optimise away tail calls, it is possible that each iteration of this named let control structure will take up needless additional stack space. In other words, unlike in Scheme, COMMON LISP function calls are not guaranteed to be *tail call optimised*.

Even though most decent COMMON LISP compilers will perform proper tail call optimisation, sometimes we need to be certain that the optimisation is being made. The easiest portable way to accomplish this is to change the **nlet** macro so that the expansions it generates necessarily use no unnecessary stack space.

NLET-TAIL

```
(defmacro! nlet-tail (n letargs &rest body)
  (let ((gs (loop for i in letargs
                  collect (gensym))))
    `(macrolet
       ((,n ,gs
          `(progn
             (psetq
               ,@(apply #'nconc
                        (mapcar
                          #'list
                          ',(mapcar #'car letargs)
                          (list ,@gs))))
             (go ,',g!n))))
       (block ,g!b
         (let ,letargs
           (tagbody
             ,g!n (return-from
                    ,g!b (progn ,@body)))))))))
```

In **nlet-tail**, we surround the supplied body of the macro and wrap it up inside a few forms. We use **block** and **return-from** statements to return the value of the final expression because we are trying to mimic the behaviour of a let form and its implicit progn. Notice that we use a gensym for the name of this block and a gensym for each **let** parameter to avoid unwanted capture and the loop macro[9] to collect these gensyms.

[9] ***Loop*** *is oddly one of the most controversial issues in COMMON LISP. Most of the objections to it are, however, completely*

Nlet-tail is used in the same way as our original **nlet**, except that invocations of the named let in non-tail positions are prohibited since they will be expanded to tail calls. Here is the same unimaginative example we used when we presented **nlet**, except that it is guaranteed, even in a lisp that doesn't perform tail call optimisation, to not consume extra stack space:

```
(defun nlet-tail-fact (n)
  (nlet-tail fact ((n n) (acc 1))
```

```
(if (zerop n)
  acc
  (fact (- n 1) (* acc n)))))
```

As this is the motivating example for this section, notice that we use **macrolet** to code-walk the supplied body looking for uses of **fact**. Where our original **nlet** uses the **labels** special form to bind a function, we would like to guarantee that no additional stack space is consumed when we invoke the named let. Technically, we would like to change some bindings in our lexical environment and perform a jump back to the top of the named let. So **nlet-tail** takes the provided name of the let, **fact** in our above usage example, and creates a *local macro* which only takes effect inside the supplied body. This macro expands into code that uses **psetq** to set the bindings of the **let** to the new provided values and then jumps back to the top, no stack space required. And most importantly, we can use the name **fact** for other unrelated macros in our program[10] *What kind of programming book doesn't have a few factorial implementations anyways?* .

To implement this jumping, **nlet-tail** uses a combination of the lisp special forms **tagbody** and **go**. These two forms offer a *goto* system. Although the problems gotos impose upon *structured programming*, whatever that means, are widely discussed, COMMON LISP offers these special forms for exactly the reason we use them here. By controlling the *program counter*—the current location in the code we are executing—we can create very efficient macro expansions. While gotos are usually discouraged in modern *high-level* languages, a quick look at any bit of assembly code will show that gotos are very much alive and kicking at the lowest level of our computer software. Even the most adamant anti-goto advocates do not suggest ridding *low-level* languages like C and assembly of goto and jump instructions. It seems that at a low level of programming, we just sort of need gotos, at least to write efficient code.

However, as Alan Kay says, lisp isn't a language—it is a building material. Talking about whether lisp is high or low level just doesn't make sense. There is very high-level lisp like our domain specific languages. With the macros we write to handle these languages, we convert their uses into a lower level of lisp. These expansions are, of course, lisp code too, just not as compressed as our original version. Next, we typically give this medium-level lisp code to a compiler which successively converts it to lower and lower levels of lisp. It won't be long at all before concepts like gotos, conditional branching, and bit fiddling make it into the code, but even then it will still be lisp. Eventually, with a native code compiler, your high level lisp program will have been converted down to assembly language. But even then, chances are that your program will still be lisp. Since most lisp assemblers are written in lisp itself it is only natural to store these assembly programs as lisp objects, resulting in really low-level programs that really are still lisp. It is only once the program is actually assembled into binary machine code that it ceases to be lisp. Or does it?

High and low level distinctions are not applicable to lisp; the level of a lisp

program is all a matter of perspective. Lisp is not a language, but instead the most flexible software building material yet discovered.

Recursive Expansions

When teaching lisp by example to beginners, a question that inevitably arises soon into the lesson is

WTF is a cadr?

There are two ways to proceed at this point. The first is to explain to the student that lisp lists are built of cons cells, each of which has a pointer called car and a pointer called cdr. Once this concept is understood, it is easy to show how the accessor functions for these pointers, also named car and cdr, can be combined into a function called **cadr** that traverses a list and retrieves the second element.

The second approach is to point the student to the COMMON LISP function **second** and ignore **cadr** altogether. Both **cadr** and **second** accomplish the same task: retrieving the second element of a list. The difference is that **second** is named for what it does, and **cadr** is named for how it does it. **Cadr** is *transparently specified*. While **second** is an easy to remember name for the function, it undesirably obscures the meaning of the operation[11] *Particularly because **second** is exactly the same as **cadr**: you can't use it to get the second elements of other sequences like ve*. Transparent specifications are often better because we can think about using the **cadr** function for more than just taking the second element of a list. For instance, we can transparently use **cadr** as a concept for getting the argument destructuring list of a lambda form. **Cadr** and **second** both perform the same task underneath, but can conceptually represent different operations[CLTL2-P530].

Even more important than a philosophical preference for transparent specifications, combinations of **car** and **cdr** can represent more list accessor operations, and more consistently, than the handful of english accessor words. **Car** and **cdr** are useful because we can combine them into new, arbitrary functions. For instance, **(cadadr x)** is the same as **(car (cdr (car (cdr x))))**. COMMON LISP mandates that all combinations of **car** and **cdr** of length four or less must be defined. So although there is no function **second-of-second** for taking the second element of a list and then treating it as a list and retrieving its second element, we can use **cadadr** for this purpose.

It is especially convenient to have these pre-defined combinations of **car** and **cdr** available for functions that take a **:key** accessor argument, like **find**:

```
* (find 'a
    '(((a b) (c d)) ((c d) (b a)))
    :key #'cadadr)

((C D) (B A))
```

Using our pre-defined **cadadr** accessor is a more concise than constructing the

equivalent lambda expression of a composition of english accessor combinations:

```
* (find 'a
    '(((a b) (c d)) ((c d) (b a)))
    :key (lambda (e)
           (second (second e))))

((C D) (B A))
```

COMMON LISP also provides the functions **nth** and **nthcdr** which can be used as generic accessors if, for instance, we don't know exactly which element we would like to take at compile-time. **Nth** is defined simply: take **n** cdrs of the list and then a car. So **(nth 2 list)**[12] is the same as **(caddr list)** and **(third list)**. **Nthcdr** is identical except that it doesn't do the final car: **(nthcdr 2 list)** is the same as **(cddr list)**.

[12] *We can comfortably use **list** as a variable name because of COMMON LISP's second namespace. Such examples would be pr*

But if the location in a cons structure is not accessible by one of the above patterns like **nth** or **nthcdr**, we need to combine accessors. Having to combine inconsistent abstractions to accomplish a task is often an indication of incompleteness. Can we define a domain specific language for the domain of accessing lists in order to unite these **car** and **cdr** combining functions, the english accessors, and functions like **nth** and **nthcdr**?

Since **car** and **cdr** are the fundamental operators, our language should involve combining these two accessors in a fully general way. Because there are an infinite number of such combinations, continuing the combinations by defining functions for every possible accessor is plainly infeasible. What we really want is a single macro that can expand into efficient list traversal code.

The syntax of specifying a list accessor function by naming it starting with a C, followed by one or more A or D characters and ending with an R, is very intuitive and is roughly what we would like to copy for our language. The macro **cxr%** is a pun on these accessors with the one or more A or D characters replaced with an x[13] *Except in spirit, **cxr** is unrelated to the **cxr** in Maclisp that accessed hunk slots.* . With **cxr%**, these As and Ds are specified in a list given as the first argument to the macro. This list is an alternating combination of numbers and either of the symbols A or D.

CXR-1

```
(defmacro cxr% (x tree)
  (if (null x)
    tree
    `(,(cond
         ((eq 'a (cadr x)) 'car)
         ((eq 'd (cadr x)) 'cdr)
         (t (error "Non A/D symbol")))
      ,(if (= 1 (car x))
```

```
        `(cxr% ,(cddr x) ,tree)
        `(cxr% ,(cons (- (car x) 1) (cdr x))
               ,tree)))))
```

For example, even though COMMON LISP doesn't provide us an english function to access the eleventh element of a list, we can easily define it:

```
(defun eleventh (x)
  (cxr% (1 a 10 d) x))
```

The point of this section is to illustrate a realistic use for *recursive expansions*. A recursive expansion comes up when a macro expands a form into a new form that also contains a use of the macro in question. Just as with all recursion, this process must terminate on a *base case*. Hopefully the macro will eventually expand into a form that doesn't contain a use of the macro in question and the expansion will finish.

Here we macroexpand an instance of a **cxr%** macro into a form that also uses **cxr%**:

```
* (macroexpand
    '(cxr% (1 a 2 d) some-list))

(CAR (CXR% (2 D) SOME-LIST))
T
```

And when we copy this new recursive form and macroexpand it, we find yet another recursion:

```
* (macroexpand
    '(CXR% (2 D) SOME-LIST))

(CDR (CXR% (1 D) SOME-LIST))
T
```

The results of the next recursion illustrate another possible usage of **cxr%**: the null list accessor[14] *If COMMON LISP included this, it might be called* ***cr****.* :

```
* (macroexpand
    '(CXR% (1 D) SOME-LIST))

(CDR (CXR% NIL SOME-LIST))
T
```

A null list accessor is our base case and expands directly into the accessed list:

```
* (macroexpand
   '(CXR% NIL SOME-LIST))

SOME-LIST
T
```

Using the CMUCL extension **macroexpand-all**, a component of a complete code-walker, we can see the entire expansion of our original **cxr%** form:

```
* (walker:macroexpand-all
    '(cxr% (1 a 2 d) some-list))

(CAR (CDR (CDR SOME-LIST)))
```

Thanks to our excellent lisp compilers, for all intents and purposes this use of **cxr%** is identical to the functions **caddr** and **third**.

But, as the name suggests, **cxr%** is incomplete. It is merely a first sketch of our ultimate macro, **cxr**. The first problem with our sketch is that it only accepts integers as the count of As or Ds. With this specification, there are things that **nth** and **nthcdr** can do that our macro can't.

We need to check for the case where a non-integer is given as the number prefix to an A or D symbol. In this case, our expansion code should evaluate what is provided and use this value[15] as the quantity for the number of cars or cdrs to traverse.

[15] *Hopefully this value should be a number. In lisp, we can safely leave this situation for lisp's exception*

The second problem with **cxr%** is that when given extremely large numbers as the prefix to A or D symbols it will *inline* all the car and cdr combinations. For small numbers, performance can be increased with inlining but generally it doesn't make sense to inline excessively large numbers of cars and cdrs; instead we should use a looping function like **nth** or **nthcdr**.

To fix both of these cases, we add an alternate expansion. If the parameter preceding an A or D symbol isn't an integer, this new behaviour must be used, and if we would rather not inline a large number of cars or cdrs, this behaviour can be selected as well. Arbitrarily choosing this *inline threshold* to be 10, this new behaviour is provided with the macro **cxr**.

CXR

```
(defvar cxr-inline-thresh 10)

(defmacro! cxr (x tree)
  (if (null x)
    tree
    (let ((op (cond
                ((eq 'a (cadr x)) 'car)
                ((eq 'd (cadr x)) 'cdr)
                (t (error "Non A/D symbol")))))
      (if (and (integerp (car x))
               (<= 1 (car x) cxr-inline-thresh))
          (if (= 1 (car x))
            `(,op (cxr ,(cddr x) ,tree))
            `(,op (cxr ,(cons (- (car x) 1) (cdr x))
                       ,tree)))
```

```
        `(nlet-tail
           ,g!name ((,g!count ,(car x))
                    (,g!val (cxr ,(cddr x) ,tree)))
           (if (>= 0 ,g!count)
             ,g!val
             ;; Will be a tail:
             (,g!name (- ,g!count 1)
                      (,op ,g!val)))))))
```

With **cxr** we could define **nthcdr** directly in terms of its transparent car and cdr specification:

```
(defun nthcdr% (n list)
  (cxr (n d) list))
```

And, similarly, **nth**:

```
(defun nth% (n list)
  (cxr (1 a n d) list))
```

Because macro writing is an iterative, layer-based process, we are often prompted to *combine* or *compose* macros that we previously implemented. For instance, in the definition of **cxr**, our alternate expansion makes use of the macro we defined in the previous section: **nlet-tail**. **Nlet-tail** is convenient because it allows us to give a name to an iteration construct and, since we only plan on iterating as a tail call, we are certain that we can use it to avoid needless stack consumption.

Here is how the use of **cxr** in **nthcdr%** expands:

```
* (macroexpand
   '(cxr (n d) list))

(LET ()
  (NLET-TAIL #:NAME1632
             ((#:COUNT1633 N)
              (#:VAL1634 (CXR NIL LIST)))
    (IF (>= 0 #:COUNT1633)
      #:VAL1634
      (#:NAME1632 (- #:COUNT1633 1)
                  (CDR #:VAL1634)))))
T
```

Notice that complex macro expansions often write code that a human programmer never would. Especially note the use of nil **cxr**s and the use of a useless **let**, both left for further macroexpansions and the compiler to optimise away.

Because macros can make more of the expansion visible to the user of the macro, transparent specification is often possible in ways impossible in other languages. For instance, as per the design of **cxr**, parameters preceding As and Ds that are integers less than **cxr-inline-thresh** will be inlined as calls to **car** and **cdr**:

```
* (macroexpand '(cxr (9 d) list))

(LET ()
  (CDR (CXR (8 D) LIST)))
T
```

But thanks to **cxr**'s transparent specification, we can pass a value that, although not an integer itself, will, when evaluated, become an integer. When we do this, we know that no inlining has taken place because the macro will result in an **nlet-tail** expansion. The simplest form that evaluates to an integer is simply that integer, quoted:

```
* (macroexpand '(cxr ('9 d) list))

(LET ()
  (NLET-TAIL #:NAME1638
             ((#:COUNT1639 '9)
              (#:VAL1640 (CXR NIL LIST)))
    (IF (>= 0 #:COUNT1639)
      #:VAL1640
      (#:NAME1638 (- #:COUNT1639 1)
                  (CDR #:VAL1640)))))
T
```

We often find it useful to combine macros together: **cxr** can expand into a macro we wrote earlier called **nlet-tail**. Similarly, sometimes it is useful to combine a macro with itself, resulting in a recursive expansion.

Recursive Solutions

It seems the macro we defined in the previous section, **cxr**, has subsumed combinations of the functions **car** and **cdr**, as well as the general flat list accessor functions **nth** and **nthcdr**. But what about english accessors like **first**, **second**, and **tenth**? Are these functions useless? Definitely not. When representing the operation of accessing the fourth element in a list, using **fourth** sure beats counting the three Ds in **cadddr**, both for writing and reading efficiency.

In fact, the largest problem with the english accessors is the limitation of having only 10 of them, **first** through **tenth**, in COMMON LISP. But one of the themes of this section, and indeed this book, is that every layer of the lisp onion can use every other layer. In lisp there are no primitives. If we want to define more english accessors, like **eleventh**, we can easily do so, as demonstrated above. The **eleventh** function we defined with **defun** is no different from accessors like **first** and **tenth** specified by ANSI. Since there are no primitives, and we can use all of lisp in our macro definitions, we can take advantage of advanced features like **loop** and **format**[16] in our macro definitions.

[16] ***Format*** *is a somewhat controversial feature of COMMON LISP. However, like the objections to* ***loop****, most are based on mis*

DEF-ENGLISH-LIST-ACCESSORS

```
(defmacro def-english-list-accessors (start end)
  (if (not (<= 1 start end))
    (error "Bad start/end range"))
  `(progn
     ,@(loop for i from start to end collect
         `(defun
            ,(symb
               (map 'string
                    (lambda (c)
                      (if (alpha-char-p c)
                        (char-upcase c)
                        #\-))
                    (format nil "~:r" i)))
            (arg)
            (cxr (1 a ,(- i 1) d) arg)))))
```

The macro **def-english-list-accessors** uses the format string "~:r" to convert a number, **i**, to a string containing the corresponding english word. We change all non-alphabetic characters, as is customary in lisp, to hyphens. We then convert this string to a symbol and use it in a **defun** form which implements the appropriate accessor functionality with our **cxr** macro.

For instance, say we suddenly realise we need to access the eleventh element of a list. We can use **nth** or combinations of **cdr** and english accessors, but this results in an inconsistent coding style. We could rewrite our code to avoid using english accessors altogether, but there was probably a reason why we chose to use that abstraction in the first place.

Finally, we could define the necessary missing accessors ourself. In other languages, this usually means a lot of copy-pasting or maybe some special-case code generation scripts—neither of which are particularly elegant. But with lisp, we have macros:

```
* (macroexpand
   '(def-english-list-accessors 11 20))

(PROGN
  (DEFUN ELEVENTH (ARG) (CXR (1 A 10 D) ARG))
  (DEFUN TWELFTH (ARG) (CXR (1 A 11 D) ARG))
  (DEFUN THIRTEENTH (ARG) (CXR (1 A 12 D) ARG))
  (DEFUN FOURTEENTH (ARG) (CXR (1 A 13 D) ARG))
  (DEFUN FIFTEENTH (ARG) (CXR (1 A 14 D) ARG))
  (DEFUN SIXTEENTH (ARG) (CXR (1 A 15 D) ARG))
  (DEFUN SEVENTEENTH (ARG) (CXR (1 A 16 D) ARG))
  (DEFUN EIGHTEENTH (ARG) (CXR (1 A 17 D) ARG))
  (DEFUN NINETEENTH (ARG) (CXR (1 A 18 D) ARG))
```

```
  (DEFUN TWENTIETH (ARG) (CXR (1 A 19 D) ARG)))
T
```

Being able to create these english accessors reduces the impact of the ten accessor limitation of ANSI COMMON LISP. If we ever need more english accessors, we just create them with the **def-english-list-accessors** macro.

How about ANSI's limitation of only defining combinations of **car** and **cdr** up to a depth of four? Sometimes, when programming complicated list processing programs, we wish there was an accessor function defined that isn't. For instance, if we are using the function **cadadr**, **second-of-second**, to access a list and we change our data representation so that the references now need to be **second-of-third**, or **cadaddr**, we encounter this COMMON LISP limitation.

Like we did with the english accessors, we could write a program that defines extra combinations of **car** and **cdr**. The problem is that, unlike english accessors, an increase in the depth of a combination function like **caddr** results in an exponential increase in the number of functions that need to be defined. Specifically, the number of accessors that need to be defined to cover a depth of **n** can be found by using the function **cxr-calculator**.

CXR-CALCULATOR

```
(defun cxr-calculator (n)
  (loop for i from 1 to n
        sum (expt 2 i)))
```

We see that ANSI specifies 30 combinations:

```
* (cxr-calculator 4)

30
```

To give you an idea of how quickly the number of functions required grows:

```
* (loop for i from 1 to 16
        collect (cxr-calculator i))

(2 6 14 30 62 126 254 510 1022 2046
 4094 8190 16382 32766 65534 131070)
```

Obviously to cover all the combinations of **car** and **cdr** in deep **cxr** functions we need an approach different from how we tackled the english accessor problem. Defining all combinations of **car** and **cdr** up to some acceptable depth is infeasible.

CXR-SYMBOL-P

```
(defun cxr-symbol-p (s)
  (if (symbolp s)
    (let ((chars (coerce
                   (symbol-name s)
```

```
                'list)))
      (and
        (< 6 (length chars))
        (char= #\C (car chars))
        (char= #\R (car (last chars)))
        (null (remove-if
                (lambda (c)
                  (or (char= c #\A)
                      (char= c #\D)))
                (cdr (butlast chars)))))))
```

To start with, we should have a solid specification of what a **cxr** symbol is. **Cxr-symbol-p** is a concise definition: all symbols that start with C, end with R, and contain 5 or more As or Ds in-between. We don't want to consider cxr symbols with less than 5 As or Ds because those functions are already guaranteed to be defined by COMMON LISP[17] *Re-binding functions specified by COMMON LISP is forbidden.* .

CXR-SYMBOL-TO-CXR-LIST

```
(defun cxr-symbol-to-cxr-list (s)
  (labels ((collect (l)
             (if l
               (list*
                 1
                 (if (char= (car l) #\A)
                   'A
                   'D)
                 (collect (cdr l))))))
    (collect
      (cdr       ; chop off C
        (butlast ; chop off R
          (coerce
            (symbol-name s)
            'list))))))
```

Next, because we plan on using **cxr** to implement the functionality of arbitrary **car** and **cdr** combinations, we create a function **cxr-symbol-to-cxr-list** to convert a cxr symbol, as defined by **cxr-symbol-p**, into a list that can be used as the first argument to **cxr**[18] *Amusingly, the deprecated function* ***explode*** *might prove useful in this situation but was left out of COMM* . Here is an example of its use:

```
* (cxr-symbol-to-cxr-list
    'caddadr)

(1 A 1 D 1 D 1 A 1 D)
```

Notice the use of the function **list*** in **cxr-symbol-to-cxr-list**. **List*** is almost the same as **list** except that its last argument is inserted into the cdr position

of the last cons cell in the created list. **List*** is very convenient when writing recursive functions that build up a list where each stack frame might want to add more than one element to the list. In our case, each frame wants to add two elements to the list: the number 1 and one of the symbols **A** or **D**.

WITH-ALL-CXRS

```
(defmacro with-all-cxrs (&rest forms)
  `(labels
     (,@(mapcar
          (lambda (s)
            `(,s (l)
               (cxr ,(cxr-symbol-to-cxr-list s)
                    l)))
          (remove-duplicates
            (remove-if-not
              #'cxr-symbol-p
              (flatten forms)))))
     ,@forms))
```

Finally, we decide the only way to effectively provide cxr functions of an arbitrary depth is to code-walk provided expressions and define only the necessary functions. The **with-all-cxrs** macro uses Graham's **flatten** utility to code-walk the provided expressions in the same way that the **defmacro/g!** macro from section 3.5, Unwanted Capture does. **With-all-cxrs** finds all the symbols satisfying **cxr-symbol-p**, creates the functions they refer to using the **cxr** macro, and then binds these functions around the provided code with a **labels** form[19] *The one problem with this approach is that such accessors will not be* ***setf****able.* .

Now we can enclose expressions within the forms passed to **with-all-cxrs** and pretend that these expressions have access to any possible cxr function. We can, if we choose, simply return these functions for use elsewhere:

```
* (with-all-cxrs #'cadadadadadr)

#<Interpreted Function>
```

Or, as shown in the following macro expansion, we can embed arbitrarily complex lisp code that makes use of this infinite class:

```
* (macroexpand
   '(with-all-cxrs
      (cons
        (cadadadr list)
        (caaaaaaaar list))))

(LABELS
  ((CADADADR (L)
     (CXR (1 A 1 D 1 A 1 D 1 A 1 D) L))
```

```
   (CAAAAAAAAR (L)
     (CXR (1 A 1 A 1 A 1 A 1 A 1 A 1 A 1 A) L)))
  (CONS
    (CADADADR LIST)
    (CAAAAAAAAR LIST)))
T
```

Often a task that sounds difficult—like defining the infinite classes of english list accessors and **car-cdr** combinations—is really just a collection of simpler problems grouped together. In contrast to single problems that happen to be difficult, collections of simpler problems can be tackled by approaching the problem recursively. By thinking of ways to convert a problem into a collection of simpler problems, we employ a tried-and-true approach to problem-solving: *divide and conquer*.

Dlambda

In our discussion of closures we alluded to how a closure can be used as an object, and how, generally, indefinite extent and lexical scope can replace complicated object systems. But one feature that objects frequently provide that we have mostly ignored up until now is multiple *methods*. In other words, while our simple counter closure example only allows one operation, increment, objects usually want to be able to respond to different *messages* with different behaviours.

Although a closure can be thought of as an object with exactly one method—**apply**—that one method can be designed so as to have different behaviours based on the arguments passed to it. For instance, if we designate the first argument to be a symbol representing the message being passed, we can provide multiple behaviours with a simple **case** statement based on the first argument.

To implement a counter with an increment method and a decrement method, we might use this:

```
(let ((count 0))
  (lambda (msg)
    (case msg
      ((:inc)
        (incf count))
      ((:dec)
        (decf count)))))
```

Notice that we have chosen *keyword symbols*, that is symbols that begin with a : and always evaluate to themselves, to indicate messages. Keywords are convenient because we don't have to quote them or export them from packages, and are also intuitive because they are designed to perform this and other sorts of *destructuring*. Often in a lambda or defmacro form keywords are not destructured at run-time. But since we are implementing a message passing system, which is a type of *run-time destructuring*, we leave the keyword processing operation to

be performed at run-time. As previously discussed, destructuring on symbols is an efficient operation: a mere pointer comparison. When our counter example is compiled it might be reduced down to the following machine code:

```
2FC:       MOV     EAX, [#x582701E4]    ; :INC
302:       CMP     [EBP-12], EAX
305:       JEQ     L3
307:       MOV     EAX, [#x582701E8]    ; :DEC
30D:       CMP     [EBP-12], EAX
310:       JEQ     L2
```

DLAMBDA

```
(defmacro! dlambda (&rest ds)
  `(lambda (&rest ,g!args)
     (case (car ,g!args)
       ,@(mapcar
           (lambda (d)
             `(,(if (eq t (car d))
                    t
                    (list (car d)))
                (apply (lambda ,@(cdr d))
                       ,(if (eq t (car d))
                            g!args
                            `(cdr ,g!args)))))
           ds))))
```

But to make this convenient, we would like to avoid having to write the case statement for every object or class we create. Situations like this deserve macros. The macro I like to use is **dlambda**, which expands into a lambda form. This expansion includes a way for one of many different branches of code to be executed depending on the arguments it is applied to. This sort of run-time destructuring is what gives **dlambda** its name: it is either a *destructuring* or a *dispatching* version of **lambda**.

Dlambda is designed to be passed a keyword symbol as the first argument. Depending on which keyword symbol was used, **dlambda** will execute a corresponding piece of code. For instance, our favourite example of a closure—the simple counter—can be extended so as to either increment or decrement the count based on the first argument using **dlambda**. This is known as the *let over dlambda* pattern:

```
* (setf (symbol-function 'count-test)
    (let ((count 0))
      (dlambda
        (:inc () (incf count))
        (:dec () (decf count)))))

#<Interpreted Function>
```

We can increment

```
* (count-test :inc)

1
```

and decrement

```
* (count-test :dec)

0
```

the closure depending on the first argument passed. Although left empty in the above let over dlambda, the lists following the keyword symbols are actually *lambda destructuring* lists. Each dispatch case, in other words each keyword argument, can have its own distinct lambda destructuring list, as in the following enhancement to the counter closure:

```
* (setf (symbol-function 'count-test)
    (let ((count 0))
      (dlambda
        (:reset () (setf count 0))
        (:inc (n) (incf count n))
        (:dec (n) (decf count n))
        (:bound (lo hi)
          (setf count
            (min hi
                 (max lo
                      count)))))))

#<Interpreted Function>
```

We now have several different possible lambda destructuring lists that might be used, depending on our first keyword argument. **:reset** requires no arguments and brings **count** back to 0:

```
* (count-test :reset)

0
```

:inc and **:dec** both take a numeric argument, **n**:

```
* (count-test :inc 100)

100
```

And **:bound** ensures that the value of **count** is between two *boundary values*, **lo** and **hi**. If **count** falls outside this boundary it is changed to the closest boundary value:

```
* (count-test :bound -10 10)
```

```
10
```

An important property of **dlambda** is that it uses lambda for all destructuring so as to preserve the normal error checking and debugging support provided by our COMMON LISP environment. For instance, if we give only one argument to **count-test** we will get an error directly comparable to an incorrect *arity* lambda application:

```
* (count-test :bound -10)

ERROR: Wrong argument count, wanted 2 and got 1.
```

Especially when **dlambda** is embedded into a lexical environment forming a closure, **dlambda** allows us to program—in object oriented jargon—as though we are creating an object with multiple *methods*. **Dlambda** is tailored so as to make this functionality easily accessible while not departing from the syntax and usage of lambda. **Dlambda** still expands into a single lambda form and, as such, its evaluation results in exactly what evaluating **lambda** results in: an *anonymous function* that can be saved, applied, and, most importantly, used as the lambda component of a lexical closure.

But **dlambda** takes this synchronisation with lambda one step further. In order for **dlambda** to provide as smooth a transition from code containing the **lambda** macro as possible, **dlambda** also allows us to process invocations of the anonymous function that don't pass a keyword argument as the first symbol. When we have large amounts of code written using the closure through a normal lambda interface we would appreciate being able to add special case dlambda methods without changing how the rest of the code uses the interface.

If the last possible method is given the symbol **t** instead of a keyword argument, the provided method will always be invoked if none of the special case keyword argument methods are found to apply. Here is a contrived example:

```
* (setf (symbol-function 'dlambda-test)
    (dlambda
      (:something-special ()
        (format t "SPECIAL~%"))
      (t (&rest args)
        (format t "DEFAULT: ~a~%" args))))

#<Interpreted Function>
```

With this definition, the majority of ways to call this function invoke the default case. Our default case uses the **&rest** lambda destructuring argument to accept all possible arguments; we are free to narrow the accepted arguments by providing more specific lambda destructuring parameters.

```
* (dlambda-test 1 2 3)
DEFAULT: (1 2 3)
```

```
NIL
* (dlambda-test)
DEFAULT: NIL
NIL
```

However, even though this anonymous function acts mostly like a regular lambda form defined with the default case, we can pass a keyword argument to invoke the special method.

```
* (dlambda-test :something-special)
SPECIAL
NIL
```

A key feature, one that will be exploited heavily by the following chapter, is that both the default method and all the special methods are, of course, invoked in the lexical context of the encompassing **dlambda**. Because of how closely **dlambda** is integrated with lambda notation, this lets us bring multi-method techniques to the domain of creating and extending lexical closures.