Notes

This section is also intended as a bibliography. All the books and papers listed here should be considered recommended reading.

v Foderaro, John K. Introduction to the Special Lisp Section. *CACM* 34, 9 (September 1991), p. 27.

viii The final Prolog implementation is 94 lines of code. It uses 90 lines of utilities from previous chapters. The ATN compiler adds 33 lines, for a total of 217. Since Lisp has no formal notion of a line, there is a large margin for error when measuring the length of a Lisp program in lines.

ix Steele, Guy L., Jr. *Common Lisp: the Language*, 2nd Edition. Digital Press, Bedford (MA), 1990.

5 Brooks, Frederick P. *The Mythical Man-Month.* Addison-Wesley, Reading (MA), 1975, p. 16.

18 Abelson, Harold, and Gerald Jay Sussman, with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, 1985.

21 More precisely, we cannot define a recursive function with a single lambda-expression. We can, however, generate a recursive function by writing a function to take *itself* as an additional argument,

```
(setq fact
      #'(lambda (f n)
          (if (= n 0)
              1
              (* n (funcall f f (- n 1))))))
```

and then passing it to a function that will return a closure in which original function is called on itself:

```
(defun recurser (fn)
  #'(lambda (&rest args)
      (apply fn fn args)))
```

Passing `fact` to this function yields a regular factorial function,

```
> (funcall (recurser fact) 8)
40320
```

which could have been expressed directly as:

```
((lambda (f) #'(lambda (n) (funcall f f n)))
 #'(lambda (f n)
     (if (= n 0)
         1
         (* n (funcall f f (- n 1))))))
```

Many Common Lisp users will find `labels` or `alambda` more convenient.

23 Gabriel, Richard P. Performance and Standardization. *Proceedings of the First International Workshop on Lisp Evolution and Standardization,* 1988, p. 60.

Testing `triangle` in one implementation, Gabriel found that "even when the C compiler is provided with hand-generated register allocation information, the Lisp code is 17% faster than an iterative C version of this function." His paper mentions several other programs which ran faster in Lisp than in C, including one that was 42% faster.

24 If you wanted to compile all the named functions currently loaded, you could do it by calling `compall`:

```
(defun compall ()
  (do-symbols (s)
    (when (fboundp s)
      (unless (compiled-function-p (symbol-function s))
        (print s)
        (compile s)))))
```

This function also prints the name of each function as it is compiled.

26 You may be able to see whether `inline` declarations are being obeyed by calling `(disassemble 'foo)`, which displays some representation of the object code of function `foo`. This is also one way to check whether tail-recursion optimization is being done.

31 One could imagine `nreverse` defined as:

```
(defun our-nreverse (lst)
  (if (null (cdr lst))
      lst
      (prog1 (nr2 lst)
             (setf (cdr lst) nil))))
```

```
(defun nr2 (lst)
  (let ((c (cdr lst)))
    (prog1 (if (null (cdr c))
               c
               (nr2 c))
           (setf (cdr c) lst))))
```

43 Good design always puts a premium on economy, but there is an additional reason that programs should be dense. When a program is dense, you can see more of it at once.

People know intuitively that design is easier when one has a broad view of one's work. This is why easel painters use long-handled brushes, and often step back from their work. This is why generals position themselves on high ground, even if they are thereby exposed to enemy fire. And it is why programmers spend a lot of money to look at their programs on large displays instead of small ones.

Dense programs make the most of one's field of vision. A general cannot shrink a battle to fit on a table-top, but Lisp allows you to perform corresponding feats of abstraction in programs. And the more you can see of your program at once, the more likely it is to turn out as a unified whole.

This is not to say that one should make one's programs shorter at any cost. If you take all the newlines out of a function, you can fit it on one line, but this does not make it easier to read. Dense code means code which has been made smaller by abstraction, not text-editing.

Imagine how hard it would be to program if you had to look at your code on a display half the size of the one you're used to. Making your code twice as dense will make programming that much easier.

44 Steele, Guy L., Jr. Debunking the "Expensive Procedure Call" Myth or, Procedural Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO. *Proceedings of the National Conference of the ACM*, 1977, p. 157.

48 For reference, here are simpler definitions of some of the functions in Figures 4.2 and 4.3. All are substantially (at least 10%) slower:

```
(defun filter (fn lst)
  (delete nil (mapcar fn lst)))

(defun filter (fn lst)
  (mapcan #'(lambda (x)
              (let ((val (funcall fn x)))
                (if val (list val))))
          lst))

(defun group (source n)
  (if (endp source)
      nil
      (let ((rest (nthcdr n source)))
        (cons (if (consp rest) (subseq source 0 n) source)
              (group rest n)))))
```

```
(defun flatten (x)
  (mapcan #'(lambda (x)
              (if (atom x) (mklist x) (flatten x)))
          x))

(defun prune (test tree)
  (if (atom tree)
      tree
      (mapcar #'(lambda (x)
                  (prune test x))
              (remove-if #'(lambda (y)
                             (and (atom y)
                                  (funcall test y)))
                         tree))))
```

49 Written as it is, `find2` will generate an error if it runs off the end of a dotted list:

```
> (find2 #'oddp '(2 . 3))
>>Error: 3 is not a list.
```

CLTL2 (p. 31) says that it is an error to give a dotted list to a function expecting a list. Implementations are not required to detect this error; some do, some don't.

The situation gets murky with functions that take sequences generally. A dotted list is a cons, and conses are sequences, so a strict reading of CLTL would seem to require that

```
(find-if #'oddp '(2 . 3))
```

return `nil` instead of generating an error, because `find-if` is supposed to take a sequence as an argument.

Implementations vary here. Some generate an error anyway, and others return `nil`. However, even implementations which follow the strict reading in the case above tend to deviate in e.g. the case of `(concatenate 'cons '(a . b) '(c . d))`, which is likely to return `(a c . d)` instead of `(a c)`.

In this book, the utilities which expect lists expect proper lists. Those which operate on sequences will accept dotted lists. However, in general it would be asking for trouble to pass dotted lists to any function that wasn't specifically intended for use on them.

66 If we could tell how many parameters each function had, we could write a version of `compose` so that, in $f \circ g$, multiple values returned by g would become the corresponding arguments to f. In CLTL2, the new function `function-lambda-expression` returns a lambda-expression representing the original source code of a function. However, it has the option of returning `nil`, and usually does so for built-in functions. What we really need is a function that would take a function as an argument and return its parameter list.

73 A version of `rfind-if` which searches for whole subtrees could be defined as follows:

```
(defun rfind-if (fn tree)
  (if (funcall fn tree)
      tree
      (if (atom tree)
          nil
          (or (rfind-if fn (car tree))
              (and (cdr tree) (rfind-if fn (cdr tree)))))))
```

The function passed as the first argument would then have to apply to both atoms and lists:

```
> (rfind-if (fint #'atom #'oddp) '(2 (3 4) 5))
3
> (rfind-if (fint #'listp #'cddr) '(a (b c d e)))
(B C D E)
```

95 McCarthy, John, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *Lisp 1.5 Programmer's Manual,* 2nd Edition. MIT Press, Cambridge, 1965, pp. 70-71.

106 When Section 8.1 says that a certain kind of operator can only be written as a macro, it means, can only be written by the user as a macro. Special forms can do everything macros can, but there is no way to define new ones.

A special form is so called because its evaluation is treated as a special case. In an interpreter, you could imagine `eval` as a big `cond` expression:

```
(defun eval (expr env)
  (cond ...
        ((eq (car expr) 'quote) (cadr expr))
        ...
        (t (apply (symbol-function (car expr))
                  (mapcar #'(lambda (x)
                              (eval x env))
                          (cdr expr))))))
```

Most expressions are handled by the default clause, which says to get the function referred to in the car, evaluate all the arguments in the cdr, and return the result of applying the former to the latter. However, an expression of the form `(quote x)` should not be treated this way: the whole point of a quote is that its argument is not evaluated. So `eval` has to have one clause which deals specifically with `quote`.

Language designers regard special forms as something like constitutional amendments. It is necessary to have a certain number, but the fewer the better. The special forms in Common Lisp are listed in CLTL2, p. 73.

The preceding sketch of `eval` is inaccurate in that it retrieves the function before evaluating the arguments, whereas in Common Lisp the order of these two operations is deliberately unspecified. For a sketch of `eval` in Scheme, see Abelson and Sussman, p. 299.

115 It's reasonable to say that a utility function is justified when it pays for itself in brevity. Utilities written as macros may have to meet a stricter standard. Reading macro calls can be more difficult than reading function calls, because they can violate the Lisp evaluation rule. In Common Lisp, this rule says that the value of an expression is the result of calling the function named in the car on the arguments given in the cdr, evaluated left-to-right. Since functions all follow this rule, it is no more difficult to understand a call to `find2` than to `find-books` (page 42).

However, macros generally do not preserve the Lisp evaluation rule. (If one did, you could have used a function instead.) In principle, each macro defines its own evaluation rule, and the reader can't know what it is without reading the macro's definition. So a macro, depending on how clear it is, may have to save much more than its own length in order to justify its existence.

126 The definition of `for` given in Figure 9.2, like several others defined in this book, is correct on the assumption that the initforms in a `do` expression will be evaluated left-to-right. CLTL2 (p. 165) says that this holds for the stepforms, but says nothing one way or the other about the initforms.

There is good cause to believe that this is merely an oversight. Usually if the order of some operations is unspecified, CLTL will say so. And there is no reason that the order of evaluation of the initforms of a `do` should be unspecified, since the evaluation of a `let` is left-to-right, and so is the evaluation of the stepforms in `do` itself.

128 Common Lisp's `gentemp` is like `gensym` except that it interns the symbol it creates. Like `gensym`, `gentemp` maintains an internal counter which it uses to make print names. If the symbol it wants to create already exists in the current package, it increments the counter and tries again:

```
> (gentemp)
T1
> (setq t2 1)
1
> (gentemp)
T3
```

and so tries to ensure that the symbol created will be unique. However, it is still possible to imagine name conflicts involving symbols created by `gentemp`. Though `gentemp` can guarantee to produce a symbol not seen before, it cannot foresee what symbols might be encountered in the future. Since `gensym`s work perfectly well and are always safe, why use `gentemp`? Indeed, for macros the only advantage of `gentemp` is that the symbols it makes can be written out and read back in, and in such cases they are certainly not guaranteed to be unique.

131 The capture of function names would be a more serious problem in Scheme, due to its single name-space. Not until 1991 did the Scheme standard suggest any official way of defining macros. Scheme's current provision for hygienic macros differs greatly from `defmacro`. For details, and a bibliography of recent research on the subject, see the most recent Scheme report.

137 Miller, Molly M., and Eric Benson. *Lisp Style and Design*. Digital Press, Bedford (MA), 1990, p. 86.

158 Instead of writing `mvpsetq`, it would be cleaner to define an inversion for `values`. Then instead of

```
(mvpsetq (w x) (values y z) ...)
```

we could say

```
(psetf (values w x) (values y z) ...)
```

Defining an inversion for `values` would also render `multiple-value-setq` unnecessary. Unfortunately, as things stand in Common Lisp it is impossible to define such an inversion; `get-setf-method` won't return more than one store variable, and presumably the expansion function of `psetf` wouldn't know what to do with them if it did.

180 One of the lessons of `setf` is that certain classes of macros can hide truly enormous amounts of computation and yet leave the source code perfectly comprehensible. Eventually `setf` may be just one of a class of macros for programming with assertions.

For example, it might be useful to have a macro `insist` which took certain expressions of the form (*predicate* . *arguments*), and would make them true if they weren't already. As `setf` has to be told how to invert references, this macro would have to be told how to make expressions true. In the general case, such a macro call might amount to a call to Prolog.

198 Gelernter, David H., and Suresh Jagannathan. *Programming Linguistics*. MIT Press, Cambridge, 1990, p. 305.

199 Norvig, Peter. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, San Mateo (CA), 1992, p. 856.

213 The constant `least-negative-normalized-double-float` and its three cousins have the longest names in Common Lisp, with 38 characters each. The operator with the longest name is `get-setf-method-multiple-value`, with 30.

The following expression returns a list, from longest to shortest, of all the symbols visible in the current package:

```
(let ((syms nil))
  (do-symbols (s)
    (push s syms))
  (sort syms
        #'(lambda (x y)
            (> (length (symbol-name x))
               (length (symbol-name y))))))
```

217 As of CLTL2, the expansion function of a macro is supposed to be defined in the environment where the `defmacro` expression appears. This should make it possible to give `propmacro` the cleaner definition:

```
(defmacro propmacro (propname)
  `(defmacro ,propname (obj)
     `(get ,obj ',propname)))
```

But CLTL2 does not explicitly state whether the `propname` form originally passed to `propmacro` is part of the lexical environment in which the inner `defmacro` occurs. In principle, it seems that if `color` were defined with `(propmacro color)`, it should be equivalent to:

```
(let ((propname 'color))
  (defmacro color (obj)
    `(get ,obj ',propname)))
```

or

```
(let ((propname 'color))
  (defmacro color (obj)
    (list 'get obj (list 'quote propname))))
```

However, in at least some CLTL2 implementations, the new version of `propmacro` does not work.

In CLTL1, the expansion function of a macro was considered to be defined in the null lexical environment. So for maximum portability, macro definitions should avoid using the enclosing environment anyway.

238 Functions like `match` are sometimes described as doing unification. They don't, quite; `match` will successfully match `(f ?x)` and `?x`, but those two expressions should not unify.

For a description of unification, see: Nilsson, Nils J. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, New York, 1971, pp. 175-178.

244 It's not really necessary to set unbound variables to gensyms, or to call `gensym?` at runtime. The expansion-generating code in Figures 18.7 and 18.8 could be written to keep track of the variables for which binding code had already been generated. To do this the code would have to be turned inside-out, however: instead of generating the expansion on the way back up the recursion, it would have to be accumulated on the way down.

244 A symbol like `?x` occurring in the pattern of an `if-match` always denotes a new variable, just as a symbol in the car of a `let` binding clause does. So although Lisp variables can be used in patterns, pattern variables from outer queries cannot—you can use the same *symbol*, but it will denote a new variable. To test that two lists have the same first element, it wouldn't work to write:

```
(if-match (?x . ?rest1) lst1
    (if-match (?x . ?rest2) lst2
        ?x))
```

In this case, the second `?x` is a new variable. If both `lst1` and `lst2` had at least one element, this expression would always return the car of `lst2`.

However, since you can use (non-`?`ed) Lisp variables in the pattern of an `if-match`, you can get the desired effect by writing:

```
(if-match (?x . ?rest1) lst1
    (let ((x ?x))
      (if-match (x . ?rest2) lst2
          ?x)))
```

The restriction, and the solution, apply to the `with-answer` and `with-inference` macros defined in Chapters 19 and 24 as well.

254 If it were a problem that "unbound" pattern variables were `nil`, you could have them bound to a distinct gensym by saying `(defconstant unbound (gensym))` and then replacing the line

```
`(,v (binding ',v ,binds)))
```

in `with-answer` with:

```
`(,v (aif2 (binding ',v ,binds) it unbound))
```

258 Scheme was invented by Guy L. Steele Jr. and Gerald J. Sussman in 1975. The language is currently defined by: Clinger, William, and Jonathan A. Rees (Eds.). *Revised*[4] *Report on the Algorithmic Language Scheme.* 1991.

This report, and various implementations of Scheme, were at the time of printing available by anonymous FTP from `altdorf.ai.mit.edu:pub`.

266 As another example of the technique presented in Chapter 16, here is the derivation of the `defmacro` template within the definition of `=defun`:

```
(defmacro fun (x)
  `(=fun *cont* ,x))

(defmacro fun (x)
  (let ((fn '=fun))
    `(,fn *cont* ,x)))

`(defmacro ,name ,parms
   (let ((fn ',f))
     `(,fn *cont* ,,@parms)))

`(defmacro ,name ,parms
   `(,',f *cont* ,,@parms))
```

267 If you wanted to see multiple return values in the toplevel, you could say instead:

```
(setq *cont*
      #'(lambda (&rest args)
          (if (cdr args) args (car args))))
```

273 This example is based on one given in: Wand, Mitchell. Continuation-Based Program Transformation Strategies. *JACM* 27, 1 (January 1980), pp. 166.

273 A program to transform Scheme code into continuation-passing style appears in: Steele, Guy L., Jr. LAMBDA: The Ultimate *Declarative.* MIT Artificial Intelligence Memo 379, November 1976, pp. 30-38.

292 These implementations of `choose` and `fail` would be clearer in T, a dialect of Scheme which has `push` and `pop`, and allows `define` in non-toplevel contexts:

```
(define *paths* ())
(define failsym '@)

(define (choose choices)
  (if (null? choices)
      (fail)
      (call-with-current-continuation
        (lambda (cc)
          (push *paths*
                (lambda () (cc (choose (cdr choices)))))
          (car choices)))))

(call-with-current-continuation
  (lambda (cc)
    (define (fail)
      (if (null? *paths*)
          (cc failsym)
          ((pop *paths*))))))
```

For more on T, see: Rees, Jonathan A., Norman I. Adams, and James R. Meehan. *The T Manual*, 5th Edition. Yale University Computer Science Department, New Haven, 1988.

The T manual, and T itself, were at the time of printing available by anonymous FTP from `hing.lcs.mit.edu:pub/t3.1`.

293 Floyd, Robert W. Nondeterministic Algorithms. *JACM* 14, 4 (October 1967), pp. 636-644.

298 The continuation-passing macros defined in Chapter 20 depend heavily on the optimization of tail calls. Without it they may not work for large problems. For example, at the time of printing, few computers have enough memory to allow the Prolog defined in Chapter 24 to run the zebra benchmark without the optimization of tail calls. (Warning: some Lisps crash when they run out of stack space.)

303 It's also possible to define a depth-first correct *choose* that works by explicitly avoiding circular paths. Here is a definition in T:

```
(define *paths* ())
(define failsym '@)
(define *choice-pts* (make-symbol-table))

(define-syntax (true-choose choices)
  `(choose-fn ,choices ',(generate-symbol t)))
```

```
(define (choose-fn choices tag)
  (if (null? choices)
      (fail)
      (call-with-current-continuation
        (lambda (cc)
          (push *paths*
                (lambda () (cc (choose-fn (cdr choices)
                                          tag))))
          (if (mem equal? (car choices)
                          (table-entry *choice-pts* tag))
              (fail)
              (car (push (table-entry *choice-pts* tag)
                         (car choices)))))))))
```

In this version, `true-choose` becomes a macro. (The T `define-syntax` is like `defmacro` except that the macro name is put in the car of the parameter list.) This macro expands into a call to `choose-fn`, a function like the depth-first `choose` defined in Figure 22.4, except that it takes an additional `tag` argument to identify choice-points. Each value returned by a `true-choose` is recorded in the global hash-table `*choice-pts*`. If a given `true-choose` is about to return a value it has already returned, it fails instead. There is no need to change `fail` itself; we can use the `fail` defined on page 396.

This implementation assumes that paths are of finite length. For example, it would allow `path` as defined in Figure 22.13 to find a path from `a` to `e` in the graph displayed in Figure 22.11 (though not necessarily a direct one). But the `true-choose` defined above wouldn't work for programs with an infinite search-space:

```
(define (guess x)
  (guess-iter x 0))

(define (guess-iter x g)
  (if (= x g)
      g
      (guess-iter x (+ g (true-choose '(-1 0 1))))))
```

With `true-choose` defined as above, `(guess` *n*`)` would only terminate for non-positive *n*.

How we define a correct *choose* also depends on what we call a choice point. This version treats each (textual) call to `true-choose` as a choice point. That might be too restrictive for some applications. For example, if `two-numbers` (page 291) used this version of *choose*, it would never return the same pair of numbers twice, even if it was called by several different functions. That might or might not be what we want, depending on the application.

Note also that this version is intended for use only in compiled code. In interpreted code, the macro call might be expanded repeatedly, each time generating a new gensymed tag.

305 Woods, William A. Transition Network Grammars for Natural Language Analysis. *CACM* 3, 10 (October 1970), pp. 591-606.

312 The original ATN system included operators for manipulating registers on the stack while in a sub-network. These could easily be added, but there is also a more general solution: to insert a lambda-expression to be applied to the register stack directly into the code of an arc body. For example, if the node `mods` (page 316) had the following line inserted into the body of its outgoing arc,

```
(defnode mods
  (cat n mods/n
    ((lambda (regs)
       (append (butlast regs) (setr a 1 (last regs)))))
    (setr mods *)))
```

then following the arc (however deep) would set the the topmost instance of the register `a` (the one visible when traversing the topmost ATN) to `1`.

323 If necessary, it would be easy to modify the Prolog to take advantage of an existing database of facts. The solution would be to make `prove` (page 336) a nested *choose:*

```
(=defun prove (query binds)
  (choose
     (choose-bind b2 (lookup (car query) (cdr query) binds)
       (=values b2))
     (choose-bind r *rules*
       (=funcall r query binds))))
```

325 To test quickly whether there is any match for a query, you could use the following macro:

```
(defmacro check (expr)
  `(block nil
     (with-inference ,expr
       (return t))))
```

344 The examples in this section are translated from ones given in: Sterling, Leon, and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, 1986.

349 The lack of a distinct name for the concepts underlying Lisp may be a serious barrier to the language's acceptance. Somehow one can say "We need to use C++ because we want to do object-oriented programming," but it doesn't sound nearly as convincing to say "We need to use Lisp because we want to do Lisp programming."

To administrative ears, this sounds like circular reasoning. Such ears would rather hear that Lisp's value hinged on a single, easily understood concept. For years we have tried to oblige them, with little success. Lisp has been described as a "list-processing language," a language for "symbolic computation," and most recently, a "dynamic language." None of these phrases captures more than a fraction of what Lisp is about. When retailed through college textbooks on programming languages, they become positively misleading.

Efforts to sum up Lisp in a single phrase are probably doomed to failure, because the power of Lisp arises from the combination of at least five or six features. Perhaps

we should resign ourselves to the fact that the only accurate name for what Lisp offers is *Lisp*.

352 For efficiency, `sort` doesn't guarantee to preserve the order of sequence elements judged equal by the function given as the second argument. For example, a valid Common Lisp implementation could do this:

```
> (let ((v #((2 . a) (3 . b) (1 . c) (1 . d))))
    (sort (copy-seq v) #'< :key #'car))
#((1 . D) (1 . C) (2 . A) (3 . B))
```

Note that the relative order of the first two elements has been reversed.

The built-in `stable-sort` provides a way of sorting which won't reorder equal elements:

```
> (let ((v #((2 . a) (3 . b) (1 . c) (1 . d))))
    (stable-sort (copy-seq v) #'< :key #'car))
#((1 . C) (1 . D) (2 . A) (3 . B))
```

It is a common error to assume that `sort` works like `stable-sort`. Another common error is to assume that `sort` is nondestructive. In fact, both `sort` and `stable-sort` can alter the sequence they are told to sort. If you don't want this to happen, you should sort a copy. The call to `stable-sort` in `get-ancestors` is safe because the list to be sorted has been freshly made.