Mitsubishi Electric Research Laboratories

Technical Report 93-17 **August, 1993**

Some Useful Lisp Algorithms: Part 2

by

Richard C. Waters

Abstract

This technical report gathers together three papers that were written during 1992 and 1993 and submitted for publication in *ACM Lisp Pointers*.

Chapter 1 "Using the New Common Lisp Pretty Printer" explains how the pretty printing facilities that have been adopted as part of the forthcoming Common Lisp standard can be used to gain detailed control over the printing of lists. As an example, it shows how the pretty printer can be used to print a subset of Lisp as Pascal.

Chapter 2 "Macroexpand-All: An Example of a Simple Lisp Code Walker" presents a function `macroexpand-all` for expanding all the macro calls in a Lisp expression. This is useful when debugging macros and can be helpful as a subroutine when writing complex macros. In addition, the chapter serves as an introduction to code walkers—the general class of programs of which `macroexpand-all` is an example. Code walkers are important because they are a vital part of the foundation of many Lisp programming tools and macro packages.

Chapter 3 "To NReverse When Consing a List or By Pointer Manipulation, To Avoid It; That Is the Question" discusses a question that Lisp programmers have argued about for decades. When creating an ordered list of elements it is often convenient to push the items onto the list one at a time and then call `nreverse` to put resulting list in the correct order. By writing more complex code, you can enter the elements in the list in the correct order in the first place. It seems that this latter approach should be better since it avoids calling `nreverse`, but is it?

Submitted to *ACM Lisp Pointers*.

201 Broadway
Cambridge Massachusetts 02139

Publication History:-

1. First printing, TR 92-17, August 1993

2. Chapter 1 published as "Using the New Common Lisp Pretty Printer" *ACM Lisp Pointers*, 5(2):27–34, April 1992.

1. Using the New Common Lisp Pretty Printer

Richard C. Waters

Although not part of the initial definition of the language, pretty printing has been an important feature of Lisp programming environments for twenty years or more [1]. By the time Common Lisp was being defined, the importance of pretty printing was clear enough that pretty printing was made a formal part of the language [2]. However, little was done beyond recognizing the least common denominator of the pretty printing facilities available at the time—[2] specifies how pretty printing can be turned on and off, but says very little else. In particular, no provision was made for allowing the user to control what the pretty printer does.

Since the late 1970s, efficient pretty printers that allow extensive user control over the format of the output produced have been a particular interest of mine. In 1989, my XP pretty printer [4] was adopted as part of the proposed Common Lisp standard [5]. This adds a number of very useful facilities to Common Lisp, however, some study and experimentation on the part of the user is required to make the best use of these facilities.

The purpose of this short paper is to show how the user format-control facilities provided by the new Common Lisp pretty printer can be used to advantage. It is intended as an extension to the documentation in [5], rather than a replacement for it. To make the best use of this article, it is advisable to read [5] and obtain a copy of the new Common Lisp pretty printer as outlined at the end of this article, so that you can play with the examples.

Printing Lisp as Pascal

As a convenient context for discussing the format-control facilities provided by the new Common Lisp pretty printer, this article uses the problem of displaying Lisp code using Pascal syntax. In particular, the article shows how the pretty printer can be used to print a simple mathematical subset of Lisp as Pascal. For example, the following Lisp function definition

```
(defun sqt (n &aux sqt)
  (declare (float n) (float sqt))
  (setq sqt 1.0)
  (loop (when (< (abs (- (* sqt sqt) n))
                 1.0E-4)
          (return nil))
        (setq sqt
              (/ (+ sqt (/ n sqt)) 2.0)))
  sqt)
```

is printed as shown below.

```
function Sqt (N: Real): Real;
begin
  Sqt := 1.0;
  while not (Abs(Sqt*Sqt-N) < 1.0E-4) do
    Sqt := (Sqt+N/Sqt)/2.0
end
```

The Lisp-as-Pascal printing system is best viewed as a means for presenting the pretty printing facilities, rather than any kind of serious attempt at program translation. However, it is worthy of note that the system is not totally contrived. A similar system was used as part of the Knowledge-Based Editor in Emacs [3] to display a Lisp-like internal representation as Ada code.

Figures 1–4 contain definitions that cause

```
(in-package "USER")

(defvar *PD* (copy-pprint-dispatch))
(proclaim '(special *B*))

(defun pascal-write (sexpr &rest args)
  (let ((*B* 0))
    (apply #'write sexpr :pretty t
           :pprint-dispatch *PD* args)))

(defun pr-string (s string)
  (setq string (string string))
  (write-char #\' s)
  (dotimes (i (length string))
    (let ((char (aref string i)))
      (write-char char s)
      (when (char= char #\')
        (write-char #\' s))))
  (write-char #\' s))

(set-pprint-dispatch 'string
  #'pr-string 0 *PD*)

(set-pprint-dispatch 'character
  #'pr-string 0 *PD*)

(set-pprint-dispatch 'symbol
  #'(lambda (s id)
      (write-string
        (remove #\-
          (string-capitalize
           (string id)))
        s))
  0 *PD*)

(set-pprint-dispatch
  '(and rational (not integer))
  #'(lambda (s n)
      (write (float n) :stream s))
  0 *PD*)
```

Figure 1: Code for printing atoms.

the pretty printer to print Lisp as Pascal. The pretty printer operates under the control of a dispatch table that specifies how various kinds of objects should be printed. The second form in Figure 1 defines a variable `*PD*` and initializes it with a copy of the default pretty printing dispatch table. The function `pascal-write` prints a Lisp expression as Pascal by triggering pretty printing and using the dispatch table `*PD*`. (As discussed in conjunction with Figure 2, the variable `*B*` is used to control the printing of parentheses in Pascal expressions.)

The function `pr-string` prints strings as required by Pascal.

```
"Bob's house"   prints as   'Bob''s house'
"say \"Hi\""    prints as   'say "Hi"'
```

The top two forms on the right of Figure 1 cause `pr-string` to be used for printing strings and characters. (The first line of `pr-string` is included so that `pr-string` can be used to print character objects in addition to strings.)

```
#\s             prints as   's'
```

The third form on the right of Figure 1 specifies how variables and function names should be printed in Pascal. In particular, it specifies that whenever an object of type `symbol` is encountered, it should be pretty printed using the indicated function. This function capitalizes the first letter of each word and removes any hyphens.

```
first-num       prints as   FirstNum
break-level-2   prints as   BreakLevel2
```

(As with most of the code being presented here, this does not guarantee that every relevant Lisp construct will be translated into a valid Pascal construct. However, it is sufficient to translate Lisp constructs that are intended to be displayed as Pascal into valid Pascal.)

A key thing to notice is that the symbol printing function uses `write-string` to print the string it computes, rather than `princ`. The reason for this is that `princ` applies pretty printing dispatching to its argument while `write-string` does not. If the function `princ` were used in the symbol printing function, the symbol `first-num` would be printed as `'FirstNum'`, because the string created by the symbol printing function would be printed as specified by the pretty printing dispatch entry for strings.

The last form in Figure 1 specifies how to print rational numbers. Nothing special has to be said about integers and floating point numbers, because the standard Lisp printer prints them in a way that is compatible with Pascal.

Printing Expressions

Figure 2 shows the pretty printing control code that specifies how expressions should be printed. The most interesting aspect of this code is the way it handles the printing of paren-

```
(defvar *unary*
  '((+ "+" 2) (- "-" 2) (not "not " 4)))

(defun unary-p (x)
  (and (consp x)
       (assoc (car x) *unary*)
       (= (length x) 2)))

(set-pprint-dispatch '(satisfies unary-p)
  #'(lambda (s list)
      (let* ((info (cdr (assoc (car list)
                               *unary*)))
             (nest (<= (cadr info) *B*))
             (*B* (cadr info)))
        (when nest (write-char #\( s))
        (write-string (car info) s)
        (write (cadr list) :stream s)
        (when nest (write-char #\) s))))
  0 *PD*)

(defvar *builtin*
  '((atan "ArcTan") (code-char "Chr")
    (log "Ln") (oddp "Odd")
    (char-code "Ord") (truncate "Trunc")
    (prin1 "Write") (terpri "Writeln")))

(defun builtin-p (x)
  (and (consp x)
       (assoc (car x) *builtin*)))

(defun pr-arglist (s args)
  (when args
    (let ((*B* 0))
      (format s #"~:<~@{~W~^, ~_~}~:>"
              args))))

(set-pprint-dispatch '(satisfies builtin-p)
  #'(lambda (s list)
      (write-string
        (cadr (assoc (car list)
                     *builtin*))
        s)
      (pr-arglist s (cdr list)))
  0 *PD*)
```

```
(defvar *bin*
  '((* "*" 3) (/ "/" 3)
    (mod " mod " 3)
    (round " div " 3)
    (and " and " 3)
    (+ "+" 2) (- "-" 2)
    (or " or " 2)
    (= " = " 1)
    (< " < " 1) (> " > " 1)
    (/= " <> " 1) (<= " <= " 1)
    (>= " >= " 1) (eq " = " 1)
    (eql " = " 1) (equal " = " 1)))

(defun bin-p (x)
  (and (consp x)
       (assoc (car x) *bin*)
       (= (length x) 3)))

(set-pprint-dispatch
  '(satisfies bin-p)
  #'(lambda (s list)
      (let* ((info (cdr (assoc (car list)
                               *bin*)))
             (nest (<= (cadr info) *B*)))
        (pprint-logical-block
            (s (cdr list)
               :prefix (if nest "(" "")
               :suffix (if nest ")" ""))
          (let ((*B* (1- (cadr info))))
            (write (pprint-pop)
                   :stream s))
          (pprint-newline :linear s)
          (write-string (car info) s)
          (let ((*B* (cadr info)))
            (write (pprint-pop)
                   :stream s)))))
      0 *PD*)

(set-pprint-dispatch 'cons
  #'(lambda (s list)
      (write (car list) :stream s)
      (pr-arglist s (cdr list)))
  -1 *PD*)
```

Figure 2: Code for printing expressions.

theses. At each moment, the variable `*B*` contains the binding strength of the current context on a scale of 0 (weakest) to 4 (strongest). Unless the binding strength of an operator is stronger than the surrounding context, parentheses are printed to specify the proper nesting of expressions.

Consider the top left of Figure 2. The variable `*unary*` contains information about the relationship between unary operators in Lisp and Pascal. Each triple contains a Lisp function, the corresponding Pascal operator, and the binding strength of the operator in Pascal. The function `unary-p` tests whether something is a list that is an application of a unary operator.

The printing function for unary operators determines whether the expression should be nested in parentheses by comparing the binding strength of the operator with the binding strength of the surrounding context; changes the value of `*B*` to reflect the binding strength of the operator; prints parentheses if needed; prints the appropriate Pascal operator; and calls `write` to print the argument appropriately.

The top three forms on the right of Figure 2 specify how binary operators should be handled. This is done in a way that is closely analogous to the handling of unary operators, however, two additional complexities have to be handled by the binary printing function.

The left associative nature of Pascal must be taken into account when deciding where to place parentheses. This is done by reducing the binding strength of the context when printing the first argument of a binary operation.

`(* (+ 1 2) 3)`	prints as	`(1+2)*3`
`(* (* 1 2) 3)`	prints as	`1*2*3`
`(* 1 (* 2 3))`	prints as	`1*(2*3)`

To allow the pretty printer to adjust the output based on the line width available, the printing function for binary operators creates a logical block and introduces a conditional newline. As discussed at length in [4, 5], logical blocks are a central feature of the pretty printing algorithm. Each logical block is printed on a single line if possible. However, if this is not possible, a block is broken across multiple lines as specified by the conditional newlines within it and appropriate indentation is inserted. For example, the Lisp expression

```
(> threshold (+ new-val delta))
```

prints as follows if the line width is sufficient.

```
Threshold > NewVal+Delta
```

If somewhat less space is available it prints as:

```
Threshold
 > NewVal+Delta
```

If even less space is available it prints as:

```
Threshold
 > NewVal
   +Delta
```

Logical blocks and conditional newlines allow a single printing function to produce aesthetic output for a wide range of line widths.

The bottom four forms on the left of Figure 2 support the printing of built-in functions where the name used in Pascal is different from the Lisp name. The most interesting thing here is the function `pr-arglist`. This function prints out zero or more arguments of a Pascal function call. Note that nothing is printed if there are zero arguments and `*B*` is set to 0 reflecting the fact that the printing of the arguments does not have to be sensitive to the binding strength of the outer context.

The function `pr-arglist` and the printing function for binary operators each create a logical block and specify conditional newlines. However, they do so using different forms. The form `pprint-logical-block` is the most general form for creating a logical block. It must be used in situations where complex computation is required to determine what should be printed within the block. In simple situations (e.g., in `pr-arglist`) the format directive `"~<...~:>"` can be used instead. The directive `"~_"` is used to specify conditional newlines within a format string.

The last form on the right of Figure 2 supports the printing of user-defined functions and built-in functions where the name is the same in Lisp and Pascal (e.g., `cos` and `round`). Note that the dispatching entry is given a priority of -1 instead of 0 as in the other entries in Figure 2. A different priority is required because the type specifier associated with the entry (`cons`) is not disjoint from the other type specifiers in Figure 2. A lesser priority is used so that the entry will act as a catch-all that only applies in situations where no other entry applies.

As examples of the way function calls are printed, consider the following:

`(terpri)`	prints as	`Writeln`
`(log x)`	prints as	`Ln(X)`
`(my-fn a b c)`	prints as	`MyFn(A, B, C)`

The logical block introduced by `pr-arglist` causes the Lisp expression

```
(my-fn epsilon (+ end delta) total)
```

to print either as

```
MyFn(Epsilon, End+Delta, Total)
```

or

```
MyFn(Epsilon,
     End+Delta,
     Total)
```

depending on the space available.

```
(set-pprint-dispatch
  '(cons (member setq))
  #"~<~*~1@{~W :=~_ ~W~}~:>"
  0 *PD*)

(set-pprint-dispatch
  '(cons (member progn))
  #"~<~*~1@{begin ~2i~_~@{~W~^; ~_~} ~
    ~I~_end~}~:>"
  0 *PD*)

(defun pr-if (s list)
  (let ((then (caddr list))
        (else (cadddr list)))
    (when
      (and else (consp then)
           (or (member (car then)
                       '(when unless))
               (and (eq (car then) 'if)
                    (null (cdddr then)))))
      (setq then `(progn ,then)))
    (format s #"~@<if ~W ~i~:_~3I~
                 then ~_~W~@[ ~I~_~3I~
                 else ~_~W~]~:>"
            (cadr list) then else)))
```

```
(set-pprint-dispatch
  '(cons (member if))
  #'pr-if 0 *PD*)

(defun maybe-progn (list)
  (if (cdr list)
      `(progn ., list)
      (car list)))

(set-pprint-dispatch
  '(cons (member when))
  #'(lambda (s list)
      (pr-if s `(if ,(cadr list)
                    ,(maybe-progn
                       (cddr list)))))
  0 *PD*)

(set-pprint-dispatch
  '(cons (member unless))
  #'(lambda (s list)
      (pr-if s `(if (not ,(cadr list))
                    ,(maybe-progn
                       (cddr list)))))
  0 *PD*)
```

Figure 3: Code for printing simple statements.

Printing Statements

Figure 3 shows the pretty printing control code that specifies how simple statements are printed. The first two forms print assignment statements and `begin...end` blocks. They are specified very compactly by using the extended form of the cons type specifier included as part of the propsed Common Lisp standard and the reader macro `#"..."`, which creates a function corresponding to a format string. Both forms use logical blocks to control the output.

The function `pr-if` is used to print conditional statements. Some complexity is involved, because the function must ensure that nested conditionals print correctly. In particular, the expression

```
(if a (if b c) d)
```

must be printed as

```
if A then begin if B then C end else D
```

instead of

```
if A then if B then C else D
```

to distinguish it from

```
(if a (if b c d))
```

Checking for this problem requires `pr-if` to inspect the `then` clause of the conditional being printed.

The last three forms on the right of Figure 3 specify how to print the Lisp forms `when` and `unless`. This is done by converting them to equivalent `if`s.

The first five forms in Figure 4 specify how to print while and repeat loops. The primary complexity revolves around the need to inspect Lisp `loop` forms and determine what Pascal statements should be used to represent them. (The code shown assumes that every Lisp loop it encounters can be displayed as either a `while` or `repeat` loop in Pascal). An example of the way a `while` loop is printed is shown at the beginning of this paper. The following `repeat` loop

```
(loop (setq result (* result x))
      (setq count (- count 1))
      (when (= count 0) (return nil)))
```

is printed as shown below.

```
repeat
  Result := Result*X;
  Count := Count-1
until Count = 0
```

```
(defun while-loop-p (x)
  (and (consp x) (eq (car x) 'loop)
       (exit-p (cadr x))))

(defun exit-p (x)
  (and (consp x)
       (member (car x) '(if when))
       (equal (cddr x) '((return nil)))))

(set-pprint-dispatch
  '(satisfies while-loop-p)
  #'(lambda (s list)
      (format s "~@<while ~W ~
                 ~:_do ~2I~_~W~:>"
              `(not ,(cadadr list))
              (maybe-progn (cddr list))))
  0 *PD*)

(defun repeat-loop-p (x)
  (and (consp x) (eq (car x) 'loop)
       (exit-p (car (last x)))))

(set-pprint-dispatch
  '(satisfies repeat-loop-p)
  #'(lambda (s list)
      (format s "~@<~<repeat ~2I~
                 ~@{~^~_~W~^; ~}~:> ~I~_~
                 until ~W~:>"
              (butlast (cdr list))
              (cadar (last list))))
  0 *PD*)

(proclaim '(special *decls*))

(defun pr-decl (s var &rest ignore)
  (declare (ignore ignore))
  (format s #"~W: ~W"
          var (declared-type var)))

(defun declared-type (var)
  (cdr (assoc
         (dolist (d *decls* 'integer)
           (when (member var (cdr d))
             (return (car d))))
         `((float . real)
           (single-float . real)
           (integer . integer)
           (fixnum . integer)
           (character . char)
           (string-char . char)))))
```

```
(defun pr-defun (s list)
  (let* ((name (cadr list))
         (args (caddr list))
         (body (cdddr list))
         (*decls* nil)
         (fn? (and (member name args)
                   (eq name
                       (car (last body)))))
         (locals
           (delete name
             (cdr (member '&aux args))))
         (parameters
           (ldiff args
             (member '&aux args)))
         (*B* 0))
   (loop
     (unless (eq (caar body) 'declare)
       (return nil))
     (setq *decls*
           (append *decls* (cdar body)))
     (pop body))
   (pprint-logical-block (s (cdr list))
     (write-string
       (if fn? "function " "procedure ")
       s)
     (write name :stream s)
     (format s #" ~:<~@{~/pr-decl/~^~
                        ; ~:_~}~:>"
             parameters)
     (when fn?
       (format s #": ~W"
               (declared-type name)))
     (format s #";~:@_")
     (when locals
       (format s #"  var ~4I~
                  ~{~:@_~/pr-decl/;~}~
                  ~0I~:@_"
               locals))
     (format s #"begin ~2i~:@_~{~W~^; ~_~}~
                 ~I~_end"
             (if fn?
                 (butlast body)
                 body)))))

(set-pprint-dispatch
  '(cons (member defun))
  #'pr-defun 0 *PD*)
```

Figure 4: Code for printing loops and function definitions.

Printing Function Definitions

The remainder of Figure 4 controls the printing of function definitions. The primary complexity here is that parameters, local variables, and type declarations are specified in Pascal very differently from the way they are specified in Lisp. The function `pr-defun` effectively has to parse the `defun` to be printed and then re-express the information using either a `procedure` or `function` statement in Pascal.

An interesting thing to note is the function `pr-decl`. This function prints a variable followed by its type and is used as a user-defined format directive in `pr-defun`. The type to print is determined by the function `declared-type`, which looks at declaration information stored in the variable `*decl*` by `pr-defun`.

The results produced by `pr-defun` are illustrated by the first example in this paper and the following

```
(defun print-exp (x i &aux count result)
  (declare (integer x count result))
  (setq count i)
  (setq result 1)
  (loop (setq result (* result x))
        (setq count (- count 1))
        (when (= count 0) (return nil)))
  (print result))
```

which is printed as shown below.

```
procedure PrintExp (X: Integer;
                    I: Integer);
  var
    Count: Integer;
    Result: Integer;
begin
  Count := I;
  Result := 1;
  repeat
    Result := Result*X;
    Count := Count-1
  until Count = 0;
  Print(Result)
end
```

Conclusion

You can get a lot of value out of the proposed Common Lisp pretty printer by merely setting `*print-pretty*` to `t`. However, this only scratches the surface of the value that can be obtained. The next level of use comes from defining special pretty printing functions for particular data structures you define. This allows the pretty printer to be much more useful during debugging. However, the usefulness of the pretty printer is not limited to being part of the Lisp programming environment.

An entirely new level of use comes from using the pretty printer as a component of a system you are building, as in the example presented here. The pretty printer's ability to tailor output to fit the space available makes it valuable in a wide variety of situations where output is being produced. In particular, it allows a modular approach to the creation of output.

For example, in the code shown in Figures 1–4, each dispatching entry specifies how to print a single kind of form. Except for a small amount of contextual information (e.g., the information required to decide where parentheses should be printed) each entry operates on a local basis without having to know anything about any other form. However, because each entry specifies how the relevant form should be printed if it will fit on one line and what should be done if it cannot be printed on one line, the pretty printer is able to dynamically combine the entries and automatically adjust the output to fit aesthetically in a wide range of line widths.

Obtaining the Example

The example above is written in Common Lisp and has been tested in several different Common Lisp implementations. The full source is shown in Figures 1–4. In addition, the source can be obtained over the INTERNET by using FTP. Connect to `FTP.AI.MIT.EDU` (INTERNET number 128.52.32.6). Login as "anonymous" and copy the files shown below.

In the directory `/pub/lptrs/`

`xpx-code.lisp`	source code
`xpx-test.lisp`	test suite

Since the example makes use of the pretty printing facilities in the proposed Common Lisp standard, it requires a Common Lisp implementation that supports these facilities. If the Common Lisp implementation you use does not yet support these facilities, you can obtain an implementation over the INTERNET. Connect to `FTP.AI.MIT.EDU`, login as "anonymous", and copy the files shown below.

In the directory `/pub/xp/`

`xp.lisp`	source code
`xp-test.lisp`	test suite
`xp-doc.lisp`	brief documentation

References

[1] Goldstein I., "Pretty Printing, Converting List to Linear Structure", MIT/AIM-279, February 1973.

[2] Steele G.L.Jr., *Common Lisp: the Language*, Digital Press, Maynard MA, 1984.

[3] R.C. Waters, "The Programmer's Apprentice: A Session With KBEmacs", *IEEE Transactions on Software Engineering*, 11(11):1296–1320, November 1985.

[4] Waters R.C., *XP: A Common Lisp Pretty Printing System*, MIT AI Laboratory technical memo MIT/AIM-1102a, September 1989.

[5] Waters R.C., "Pretty Printing," in *Common Lisp: the Language*, Second Edition, 748–769, Steele G.L.Jr., Digital Press, Burlington MA, 1990.

2. Macroexpand-All: An Example of a Simple Lisp Code Walker

Richard C. Waters

If you like to write Lisp macros, or even just use the macros other people write, you have no doubt felt the desire to see what particular macro calls expand into. The standard Common Lisp function `macroexpand` is very useful in this regard; however, since it only expands the topmost form in an expression, it does not necessarily show you the full result of a macro expansion.

For example, suppose that you wrote (or have available to you) the following implementation of the standard Lisp macro `cond`.[1]

```
(defmacro cond (&rest clauses)
  (when clauses
    `(if ,(caar clauses)
         (progn ,@(cdar clauses))
         (cond ,@(cdr clauses)))))
```

If you evaluate

```
(macroexpand '(cond (a b) (c d)))
```

you obtain the result

```
(if a (progn b) (cond (c d)))
```

which is not as informative as you might wish, because it does not show you the complete result that will be obtained when the nested instance of `cond` that is created by the macro is eventually expanded.

As shown below, it is trivial to write a program that applies `macroexpand` to every sublist in a Lisp expression.

[1]This is an ugly implementation of `cond`, because it produces a lot of excess code. Worse, it is an erroneous implementation of `cond`, because it assumes that every clause contains at least two elements. However, it is a convenient example for the purposes of this discussion.

```
(defun macroexpand-tree (tree)
  (setq tree (macroexpand tree))
  (if (atom tree)
      tree
      (mapcar #'macroexpand-tree tree)))
```

This can be used to show you the complete macroexpansion of a form in many situations. For instance,

```
(macroexpand-tree '(cond (a b) (c d)))
```

yields

```
(if a (progn b) (if c (progn d) nil))
```

Unfortunately, `macroexpand-tree` is severely flawed, because it does not operate in the same manner as the Common Lisp compiler and evaluator. In particular, while `macroexpand-tree` macroexpands every sublist in a Lisp expression, the evaluator and compiler only macroexpand sublists that are in positions where they can be evaluated.

For example, when it encounters the form

```
(mapcar #'(lambda (cond) (car cond)) list)
```

the compiler does not do any macroexpansion. However, applying `macroexpand-tree` produces

```
(mapcar #'(lambda nil (car cond)) list)
```

since `(macroexpand '(cond))` is `nil`.

To macroexpand everything that should be expanded in a Lisp expression, and nothing else, you have to write a function that understands which parts of which Lisp forms are evaluated and which parts are not. A function called `macroexpand-all` that does this is presented in the next section.

Taken by itself, `macroexpand-all` is useful, but not all that interesting. However, the way that `macroexpand-all` is written is quite interesting, because it is an example of an important class of programs known as code walkers.

One of the beauties of Lisp is that everything that any programming tool has to know about the syntax of Lisp can be stated in a couple of short paragraphs. Further, this information is built into the Lisp reader so that nobody has to expend much effort dealing with Lisp syntax.

In comparison to many other programming languages, the semantics of Lisp is also very simple, because almost everything is a mere function call or a macro that expands into function calls. However, there is a residue of some 25 special forms each of which has its own special semantics.[2]

Unfortunately, 25 is a pretty large number when you consider that each tool that manipulates programs in non-trivial ways has to have embedded knowledge of all 25 special forms. The way this typically comes about is that the tool has to traverse a Lisp expression and either change parts of it (e.g., macroexpanding subforms or renaming variables) or collecting information (e.g., about free variables or bound variables). This kind of process is generally referred to as *code walking*.

People have attempted to implement general code walkers that encode everything any tool has to know about Lisp semantics (see for example, the PCL code walker described in [1]). However, none of the resulting code walkers have been generally accepted as really containing everything anyone would need. As a result, the writers of Lisp programming tools and complex macros are typically required to write there own code walkers. The implementation of `macroexpand-all` is a valuable tutorial introduction to how this can be done.

[2]Due primarily to the urgings of Kent Pitman [2], a key advance of Common Lisp over its predecessors (e.g., MacLisp) was reducing the number of special forms to only 25 and preventing users from defining new ones. A significant attraction of Scheme is that it goes even farther in the direction of semantic simplification.

Macroexpand-All

Except as noted below, `macroexpand-all` is addressed solely to Common Lisp as defined in *Common Lisp the Language, first edition* (CLtL1) [3], rather than the proposed standard version described in *Common Lisp the Language, second edition* (CLtL2) [4]. The decision to stick to CLtL1 was motivated by two issues. First, except where noted below, moving to CLtL2 would make `macroexpand-all` a bit more complex, because there are a few more special forms, but it would not make it any more interesting. In addition, like many people, I still have to work in CLtL1 and so this is the version of `macroexpand-all` I am using.

The main body of the code for `macroexpand-all` is shown in Figure 5. `Macroexpand-all` (at the top left of the figure) takes two arguments: a form and an optional macro environment (i.e., the same kind of environment that `macroexpand` takes). `Macroexpand-all` copies the form to be expanded to protect the code that contains the form from being destructively modified during macro expansion, and then calls `mexp` to do the real work. (Some implementations of Common Lisp implement `copy-tree` recursively. If this is the case in the Lisp you use, you will have to write an iterative implementation of `copy-tree` to use in `macroexpand-all` or risk stack overflow occurring.)

`Mexp` is the central control point of the code walking process. It calls `macroexpand-1` repeatedly until the form has been converted into a use of a special form whose semantics is understood by `mexp` or reduced to an ordinary function call or other vanilla object. `Mexp` then recurses by calling an appropriate handler as discussed shortly. (`Mexp` checks for special forms each time before calling `macroexpand-1`, because some implementations of Common Lisp implement some special forms as macros.)

It should be noted that while `mexp` is a very simple code walker, every code walker has to have essentially the same structure. A code walker has to expand every macro call, because the only way for it to understand the semantics of a macro call is to determine what it expands

```
(in-package :mexp)
(export '(macroexpand-all))

(defun macroexpand-all (f &optional env)
  (mexp (copy-tree f) env))

(defun mexp (f env &aux (flag t) m)
  (loop
    (cond ((atom f)
           (return f))
          ((not (symbolp (car f)))
           (return (all-mexp f env)))
          ((setq m (get (car f) 'mexp))
           (return (funcall m f env)))
          ((not flag)
           (return (funcall-mexp f env))))
    (multiple-value-setq (f flag)
      (macroexpand-1 f env))))

(defun all-mexp (list env)
  (do ((f list (cdr f))
       (r () (cons (mexp (car f) env) r)))
      ((atom f) (nreconc r f))))

(defun funcall-mexp (f env)
  `(,(car f) ,@(all-mexp (cdr f) env)))

(defun quote-mexp (f env)
  (declare (ignore env))
  f)

(defun block-mexp (f env)
  `(,(car f)
    ,(cadr f)
    ,@(all-mexp (cddr f) env)))

(defun let-mexp (f env)
  `(,(car f)
    ,(mapcar #'(lambda (p)
                 (bind-mexp p env))
      (cadr f))
    ,@(all-mexp (cddr f) env)))

(defun bind-mexp (p env)
  (if (and (consp p) (consp (cdr p)))
      (list (car p) (mexp (cadr p) env))
      p))

(defun lambda-mexp (f env)
  `(,(car f)
    ,(mapcar #'(lambda (p)
                 (arg-mexp p env))
             (cadr f))
    ,@(all-mexp (cddr f) env)))

(defun arg-mexp (arg env)
  (if (and (consp arg) (consp (cdr arg)))
      `(,(car arg)
        ,(mexp (cadr arg) env)
        ,@(cddr arg))
      arg))

(defun get-var (b)
  (if (consp b) (car b) b))

(defun get-val (b)
  (eval (if (consp b) (cadr b) nil)))

(defun compiler-let-mexp (f env)
  (progv (mapcar #'get-var (cadr f))
         (mapcar #'get-val (cadr f))
    (mexp
      (if (null (cdddr f))
          (caddr f)
          `(let nil ,@(cddr f)))
      env)))

(defun macrolet-mexp (f env)
  (with-env env `(macrolet ,(cadr f))
            #'mexp
            (if (null (cdddr f))
                (caddr f)
                `(let nil ,@(cddr f)))))

(defun flet-mexp (f env)
  `(flet
    ,(all-lambda-mexp (cadr f) env)
    ,@(with-env env `(flet ,(cadr f))
                #'all-mexp
                (cddr f))))

(defun labels-mexp (f env)
  (with-env env `(labels ,(cadr f))
            #'labels-mexp-2 f))

(defun labels-mexp-2 (f env)
  `(labels
    ,(all-lambda-mexp (cadr f) env)
    ,@(all-mexp (cddr f) env)))

(defun all-lambda-mexp (list env)
  (mapcar #'(lambda (f)
              (lambda-mexp f env))
          list))

(mapc #'(lambda (x)
          (setf (get (car x) 'mexp)
                (eval (cadr x))))
  '((block          #'block-mexp)
    (catch          #'funcall-mexp)
    (compiler-let   #'compiler-let-mexp)
    (declare        #'quote-mexp)
    (eval-when      #'block-mexp)
    (flet           #'flet-mexp)
    (function       #'funcall-mexp)
    (go             #'quote-mexp)
    (if             #'funcall-mexp)
    (labels         #'labels-mexp)
    (lambda         #'lambda-mexp)
    (let            #'let-mexp)
    (let*           #'let-mexp)
    (macrolet       #'macrolet-mexp)
    (multiple-value-call #'funcall-mexp)
    (multiple-value-prog1 #'funcall-mexp)
    (progn          #'funcall-mexp)
    (progv          #'funcall-mexp)
    (quote          #'quote-mexp)
    (return-from    #'block-mexp)
    (setq           #'funcall-mexp)
    (tagbody        #'funcall-mexp)
    (the            #'block-mexp)
    (throw          #'funcall-mexp)
    (unwind-protect #'funcall-mexp)))
```

Figure 5: The main body of the code for `macroexpand-all`.

into. When confronted by a form that is not a macro call, any code walker has to have special-purpose handlers for each kind of form since the various special forms are totally idiosyncratic.

As summarized in Table 5-1 on page 57 of [3], CLtL1 has 24 special forms. However, from the perspective of tools that operate on programs `lambda` should be added to this list, since it can appear in code and certainly has special semantics. `Mexp` maintains an index between special forms and their handlers by storing the handler functions as properties of the special form symbols. This is set up by the expression in the lower right of Figure 5. The remainder of the figure shows the handlers themselves.

Since `mexp` primarily only cares about what parts of a special form macroexpansion should be applied to and what parts it should not be applied to, most of the handlers are very simple, and many of the special forms are treated in the same way. For example, the handler `funcall-mexp` specifies that everything except the first element in a form should be macroexpanded. For the purposes of `mexp` this is appropriate for handling ordinary function calls and 11 of the 25 special forms.

A code walker that is more complex than `mexp` will require more complex handlers that keep track of additional information such as what variables are bound. However, the handlers will be basically upward compatible from the ones shown here.

There are only three classes of `mexp`'s handlers that are at all complex. The handlers for forms that bind variables (i.e., `let{*}` and `lambda`) are a bit complex due to the somewhat complex syntax that is used to specify variables and values for them.

The handler for `compiler-let` is complex because it must cause a change in the variable bindings that are in effect while macro expansion proceeds. Conveniently, only special variables are involved, so the change can be straightforwardly made by using `progv` to change the evaluation environment before recursing into the body of the `compiler-let`.

The handlers for `flet`, `labels`, and `macrolet` are by far the most interesting. They are complicated because they potentially change the environment that controls the way macros expand. `Flet` and `labels` can shadow a macro definition with a function definition. `Macrolet` can introduce a new macro definition.

For example, consider the form

```
(flet ((cond (x) (cond (x (1+ x)))))
  (cond (car y)))
```

Assuming the definition of `cond` used above, this should macroexpand into

```
(flet ((cond (x)
         (if x (progn (1+ x)) nil)))
  (cond (car y)))
```

The use of `cond` in the body of the local function definition is an instance of the macro `cond` defined above, but the instance of `cond` in the body of the `flet` is an instance of the locally defined function instead.

A similar situation arises with `macrolet`.

```
(macrolet ((cond (x) (cond (x `(1+ ,x)))))
  (cond (car y)))
```

macroexpands into

```
(1+ (car y))
```

The `macrolet` form itself does not need to be retained once macro expansion has occurred. The information it specifies is only relevant to the expansion of macro calls syntactically nested within it and these calls have all been eliminated by expanding them.

The handlers for `flet`, `labels`, and `macrolet` are each implemented using a function called `with-env`, which takes four arguments, a macro environment, a form that potentially modifies this environment, a function `fn`, and an argument `x` to apply the function to. `With-env` updates the macro environment as specified by the form and then applies `fn` to `x` and the modified environment. `With-env` returns whatever `fn` returns.

For example, `flet-mexp` uses `all-lambda-mexp`, which calls `lambda-mexp`, to macroexpand the local function definitions. It then uses `with-env` to create the altered macro environment that corresponds to the `flet` and uses `all-mexp` to macroexpand the body of the `flet` in this new environment. `Labels-mexp` operates the same way as `flet-mexp` except that it uses the altered environment when macroexpanding the local function definitions.

Before discussing how `with-env` works, it is useful to note that all the code in Figure 5 is portable Common lisp. Unfortunately, this is not true for `with-env`.

Evaluation and Macro Environments

Lisp evaluation is controlled by an evaluation environment that specifies the values of variables and what functions and macros symbols refer to. In order not to over constrain implementors, Common Lisp documentation says almost nothing about this environment. CLtL1 merely describes a couple of situations where evaluation environments appear. In particular, if an `*evalhook*` function is specified, then whenever an attempt is made to evaluate something, the `*evalhook*` function will be called and passed the form to evaluate and an appropriate evaluation environment. Just about the only thing that this environment can be used for is as an argument to the function `evalhook`, which can be used to resume evaluation of the form passed to the `*evalhook*` function. (Stepping and tracing tools can be implemented using `*Evalhook*` functions and `evalhook`.)

The expansion of macros is controlled by a macro environment that specifies which symbols refer to macros and which do not. As above, Common Lisp documentation says almost nothing about this environment. CLtL1 merely describes two situations where macro environments appear. When a macro function (a function that implements a macro) is called, it is passed the macro environment that is appropriate for the place in the source program where the macro call appeared. The function `macroexpand` can be passed a macro environment that specifies the context that should be used when expanding the specified form. This is needed so that a macro (e.g., `setf`) can call `macroexpand` on part of its argument and get the results that are appropriate for the place where the original macro call appeared.

`With-env` has to modify the macro environment given to it to reflect the changes implied by the specified form. This is difficult to do, because CLtL1 does not provide any functions for creating or inspecting either evaluation or macro environments. All that it provides is a few obscure functions that are not intended to be at all relevant to our task.

Solving the Puzzle

For those that delight in getting Lisp to do things that the builders of the language never dreamed that you would want to do, successfully extending a macro environment is a puzzle much too interesting to pass up. The key to solving the puzzle is realizing that whatever a macro environment is, the Lisp evaluator succeeds in extending it appropriately whenever it encounters an `flet`, `labels`, or `macrolet`. We can get the evaluator to make the modification we want, by simply passing it the form we have.

Unfortunately, there is a problem with this simple idea. The evaluator descends into an expression creating appropriate evaluation and macro environments, but while you can specify an initial execution environment with `evalhook`, there is no way to specify an initial macro environment. In contrast, you can specify an initial macro environment to `macroexpand`, but `macroexpand` does not descend into an expression and therefore does not lead to the construction of an extended macro environment. As a result, while it is easy to get the evaluator to extend an evaluation environment, it is not clear how to get it to extend a macro environment for us.

The function `evalhook` can be used to both prime the evaluator with an initial evaluation environment and to access an extended evaluation environment. For example, suppose you have an evaluation environment *E*.

```
(evalhook '(macrolet ((h (a) `(1+ ,a))) t)
          #'(lambda (x env)
              (print env)
              (eval x))
          nil
          E)
```

```
(defmacro grab-env (fn x
                    &environment env)
  '',(funcall fn x env))
```

```
(defun aug-env (env form fn x)
  (evalhook `(,@ form (grab-env ,fn ,x))
            nil nil env))
```

Figure 6: Manipulating execution and macro environments.

```
#+(or :SYMBOLICS :AKCL :CORAL :FRANZ-INC)
(defun with-env (env form fn x)
  (aug-env (convert-env env) form fn x))

#+(or :SYMBOLICS :AKCL)
(defun convert-env (env)
  env)
```

```
#+:CORAL
(defun convert-env (env)
  (list nil env nil nil nil nil))

#+:FRANZ-INC
(defun convert-env (env)
  (list nil env nil nil))
```

Figure 7: Extending macro environments that are similar to execution environments.

shows you the result of extending E with the information in the `macrolet`. Exactly what this environment is like differs radically from one Common Lisp implementation to another. (If you want to type the expression above at top level, you can use the value `nil` for E, which stands for the top-level environment.)

What we need is some way to convert evaluation environments into macro environments and vice versa. The first of these conversions can be done straightforwardly with a macro, by utilizing an `&environment` argument. In particular, the macro `grab-env` in Figure 6 applies a function to an argument `x` and the macro environment corresponding to the evaluation environment in effect at the place where the macro call appears. It then returns whatever the function returns. For example, the expression

```
(grab-env #,#'(lambda (x env) (print env))
          nil)
```

will show you the macro environment corresponding to the place where the expression appears. If you type this at top level you will see the top-level macro environment. If you type it nested in a form you will see a more complex macro environment.

You can use `evalhook` and `grab-env` together to access the macro environment that corresponds to extending an evaluation environment.

```
(evalhook '(macrolet ((h (a) `(1+ , a)))
             (grab-env #,#'(lambda (x env)
                               (print env))
                        nil))
          nil nil E)
```

shows you the macro environment that results from extending E with the information in the `macrolet`. Exactly what this is like differs radically from one Common Lisp implementation to another. Further, while it is possible that this macro environment will be the same as the extended evaluation environment, there is no guarantee that they will be anything like each other.

The function `aug-env` in Figure 6 embodies the trick shown above. It applies a function to an argument and the macro environment that results from extending an initial evaluation environment as specified by the given form. `Aug-env` then returns whatever the function returns. For example,

```
(aug-env E
         '(macrolet ((h (a) `(1+ ,a))))
        #'(lambda (x env) (print env))
        nil)
```

is identical to the last example, in that it constructs exactly the same form and evaluates it in the same environment.

It does not appear that there is any implementation independent way in CLtL1 to convert a macro environment into an evaluation environment. However, in a given implementation it is usually easy to do. In particular, I used the expressions shown above to inspect macro and execution environments in various implementations of Common Lisp, and determined that in most of them, execution and macro environments are very similar. When this is the case, the function `with-env` needed in Figure 5

```
#+:LUCID
(defun with-env (env form fn x)
  (aug-env nil
           form
           #'with-appended-env
           (list env fn x)))
```

```
#+:LUCID
(defun with-appended-env (z delta)
  (let ((env (car z))
        (fn (cadr z))
        (x (caddr z)))
    (funcall fn x (append delta env))))
```

Figure 8: Extending macro environments that are stacks implemented as lists.

```
(defun with-env (env form bind fn body)
  (funcall fn body
    (if (eq form 'macrolet)
        (augment-env env :macro
                     (mapcar #'parse bind))
        (augment-env env :function
                     (mapcar #'car bind)))))
```

```
(defun parse (b)
  (list (car b)
        (parse-macro (car b)
                     (cadr b)
                     (cddr b)
                     env)))
```

Figure 9: Extending macro environments in CLtL2.

can be directly implemented using `aug-env` as shown in Figure 7.

In particular, in two of the Common Lisps I looked at, execution and macro environments are identical. In the other two, an execution environment is a list, one of whose components is a macro environment. Therefore in all four cases, converting a macro environment into an equivalent execution environment is trivial.

In the fifth Common Lisp I looked at, the relationship between execution environments and macro environments is obscured by the use of implementation-specific data structures. However, I noticed that in this implementation a macro environment is a stack implemented as a list. This opens up an alternate approach to modifying a macro environment.

Rather than converting a macro environment to an execution environment and then letting the evaluator extend it, one can determine what extension should be applied and do the extension yourself. This depends on knowing how extension can be done.

If a macro environment is a stack implemented as a list, then a macro environment can be extended using `append`. Further, if the top-level macro environment is the empty stack `nil`, than the change introduced by a form can be determined by determining what macro environment is created by evaluating the form at top level. These observations lead to the implementation of `with-env` shown in Figure 8.

In the figure, `aug-env` is used to determine the change in the macro environment that results from evaluating the specified form in isolation. The function `with-append-env` then combines this change with the original macro environment, creating an extended macro environment, which is passed to the specified function.

Improvements In CLtL2

The problem posed above can be solved in a portable way in CLtL2, because CLtL2 specifies a suite of functions that can extract information from and add information to environments. As a result, a macro environment can be directly extended as shown in Figure 9.

The CLtL2 function `augment-env` is used to add information into an environment. In the figure, it is used to add specifications for the macro definitions in a `macrolet` or the function definitions in an `flet` or `labels`. The function `parse` uses the CLtL2 function `parse-macro` to convert the local macro definitions in a `macrolet` into the form expected by `augment-env`.

Conclusion

The function `macroexpand-all` is a tool that can be useful for anyone who writes or uses complex macros. The code for `macroexpand-all` is primarily implementation independent. However, in order to use it, you have to supply a definition of the critical function `with-env`. If you are using one of the implementations of Common Lisp where `macroexpand-all` has already been tested, this has been done for you. If not, you have three choices.

First, by inspecting evaluation and macro environments in the Common Lisp you use, you should be able discover enough about the structure of these environments, in order to implement `with-env` in a way that is analogous to Figure 7 or 8.

Second, you can include a vestigial definition of `with-env`, such as

```
(defun with-env (env form fn x)
  (declare (ignore form))
  (funcall fn x env))
```

and live with the fact that `macroexpand-all` will occasionally produce incorrect results. This might be a reasonable thing to do if you are going to use `macroexpand-all` merely as a debugging aid. However, it is not tolerable if you intended to use `macroexpand-all` as part of a macro definition.

Third, you can wait until you have an implementation of CLtL2 available and use the implementation of `with-env` shown in Figure 9.

In addition to being a useful tool in its own right, Figure 5 can be viewed as the minimal skeletal structure on which more complex code walkers can be written. Everything shown in the figure is necessary, because a code walker must expand all the macro calls in an expression in order to work. The code has to be extended in order for the walker to keep track of information about an expression such as what variables are bound. In CLtL2, this is much easier than in CLtL1, because the code walker can retrieve information from the environments used by the implementation, rather than keeping track of it redundantly.

Obtaining Macroexpand-All

The example above is written in Common Lisp and has been tested in several different Common Lisp implementations. The full source is shown in Figures 5–9. In addition, the source can be obtained over the INTERNET by using FTP. Connect to `MERL.COM` (INTERNET number 140.237.1.1). Login as "anonymous" and copy the files shown below.

In the directory `/pub/lptrs/`

`mexp-code.lisp`	source code
`mexp-test.lisp`	test suite
`mexp-doc.txt`	brief documentation

References

[1] Curtis, P., "Algorithms", *ACM Lisp Pointers*, 3(1):48–61, March 1990.

[2] Pitman, K.M., "Special Forms in Lisp", in *Proc. 1980 Lisp Conference*, 179–187, August 1980.

[3] Steele G.L.Jr., *Common Lisp: the Language*, Digital Press, Maynard MA, 1984.

[4] Steele G.L.Jr., *Common Lisp: the Language*, second edition, Digital Press, Maynard MA, 1990.

3. To NReverse When Consing a List or By Pointer Manipulation, To Avoid It; That Is the Question

Richard C. Waters

A situation that arises all the time in Lisp is the need to create a list of elements where the order of the elements in the list is the same as the order that they are created in time—i.e., the first element computed is the first element in the list, the second element computed is the second element in the list, etc. There are two basic ways of doing this: the `nreverse` approach and the `rplacd` approach. In the `nreverse` approach, you push the elements onto the list as they are computed and then use `nreverse` to put the list into the correct order after all of the elements have been computed. In the `rplacd` approach, you maintain a pointer to the end of the list and use `rplacd` to put each element directly into its proper place in the list.

Which of the two approaches to creating a list is better?

Over the two decades that I have been writing Lisp programs, I have overheard (and participated in) quite a number of arguments about this question. Some people argue vehemently that the `rplacd` approach is **obviously** much faster and therefore better. Others argue just as vehemently that the `nreverse` approach is actually faster and given its greater simplicity is therefore **obviously** better. However, I have seen very little in the way of hard facts.

As discussed in detail below, the facts suggest that neither approach is **obviously** faster. It is just as easy to imagine Lisp implementations where one approach is faster as implementations where the other is faster. It is easiest of all to imagine implementations where the two approaches run at more or less the same speed.

Experimentation suggests that the `nreverse` approach is actually faster in many if not most Lisp implementations. However, more importantly, it supports the idea that the speed difference is not enough to be important. Therefore, given that the `nreverse` approach is easier to write and understand, I recommend using `nreverse` when creating lists.

A Specific Example

As a precise foundation for comparing the two approaches, it is best to look at a specific example. Consider implementing a simplified version of the standard Common Lisp function `maplist` that takes only a single list argument. This example is convenient because it contains very little computation other than the creation of the output list.

The program `maplist-nreverse` shows how a simplified `maplist` can be implemented using the `nreverse` approach to creating the output list. The code is clear and concise. It enumerates each sublist in the list, calls the function on each sublist, and creates a list (in reverse order) of the results. As the final step of the code, the list is reversed.

```
(defun maplist-nreverse (f list)
  (do ((sub list (cdr sub))
       (r nil (cons (funcall f sub) r)))
      ((null sub) (nreverse r))))
```

The program `maplist-rplacd` shows how a simplified `maplist` can be implemented using the `rplacd` approach to creating the output list. The code is less clear and less concise, but avoids calling `nreverse`.

```
(defun maplist-rplacd (f list)
  (let ((r (cons nil nil)))
    (do ((sub list (cdr sub))
         (end r (let ((x (cons
                          (funcall f sub)
                          nil)))
                  (rplacd end x)
                  x)))
        ((null sub) (cdr r)))))
```

The code starts by creating a dummy cons cell that is later discarded. This makes the main loop simpler and faster, because it avoids the need for code that handles the first output element specially. The savings is greater than the cost of the extra cons unless the input list is extremely short. (In the Lisp I use, the break even point is at an input length of five.)

In order to compare `maplist-nreverse` with `maplist-rplacd` one must look in detail at the function `nreverse`, since this should properly be considered as part of `maplist-nreverse`. As shown below, `nreverse` is a very simple function. It merely runs down a list applying `cdr` and `rplacd` once to each cons cell.

```
(defun nreverse (list)
  (prog ((prev nil) next)
        (when (null list) (return nil))
     lp (setq next (cdr list))
        (rplacd list prev)
        (when (not next) (return list))
        (setq prev list)
        (setq list next)
        (go lp)))
```

The key observation to make is that the code for `maplist-rplacd` is very much the same as the code for `maplist-nreverse` plus the code for `nreverse`. In particular, each approach calls `cons` to create the cells of the output list, and uses `rplacd` to place the cells in the correct order. The only difference is that taken together `maplist-nreverse` and `nreverse` traverse the output list twice as opposed to once for `maplist-rplacd`. This is a real difference, but not a large enough difference to be important.

Counting Instructions

To sharpen the comparison of the functions above, it is interesting to consider the best possible ways that the functions can be implemented using low level machine instructions. Since I do my work on an HP-9000 series machine and am more familiar with it than with other current machines, I will use HP's PA-RISC architecture [1] as the basis for the examples below.

`Maplist_nreverse` approximates the best PA-RISC implementation of `maplist-nreverse`. It is approximate because it makes many assumptions about the associated Lisp implementation.[3] In particular, it assumes that the implementation is using the standard PA-RISC calling conventions and that cons cells are implemented as a 4-byte car pointer followed by a 4-byte cdr, with `nil` implemented as `0`.

```
maplist_nreverse
.CALLINFO CALLER,SAVE_RP,ENTER_GR=5
r   .reg  %r3
sub .reg  %r4
f   .reg  %r5
    .ENTER
    LDI         0,r          ;(setq r nil)
    MOVB,=,n    %arg1,sub,DN
    MOVB        %arg0,f
 LP MOVB        sub,%arg0
    BLR         f,%rp
    MOVB        %ret0,%arg0 ;1st cons arg
    MOVB        r,%arg1     ;2nd cons arg
    BL          cons,%rp    ;cons
    MOVB        %ret0,r     ;(setq r ...)
    LDW         4(sub),sub
    COMIB,<>,n  0,sub,loop
 DN MOVB        r,%arg0     ;1st arg
    BL          nreverse,%rp;nreverse
    .LEAVE
```

As long as quantities are stored in registers, operations like `car`, `cdr`, `rplacd`, and `setq` can be implemented as single PA-RISC instructions. As a result, `maplist_nreverse` is very compact. The parts of the code that concern us are the seven instructions that create the output list. The correspondence between these instructions and parts of `maplist-nreverse` are indicated by

[3]The machine code shown is also approximate because it was not practical to test it. As a result, there might be minor errors; however, this should not effect the basic comparisons being made.

comments.

The list `r` being constructed is stored in a register. A load immediate instruction (`LDI`) is used to initialize `r` to `nil`. Two move instructions (`MOVB`) are used to set up the arguments of `cons`. A branch and link instruction (`BL`) is used to call the `cons` subroutine. A move is used to store the result returned by `cons` in `r`. The last two instructions call `nreverse`, whose result is returned as the result of `maplist_nreverse`.

`Maplist_rplacd` approximates the best PA-RISC implementation of `maplist-rplacd`. As above, the relationship between the instructions that create the output list and the code in `maplist-rplacd` is indicated by comments.

```
maplist_rplacd
.CALLINFO FRAME=4,CALLER,SAVE_RP,ENTER_GR=5
end .reg  %r3
sub .reg  %r4
f   .reg  %r5
    .ENTER
    STW         0,-52(%sp)  ;set cdr r nil
    ADDI        -56,%sp,end ;(setq end r)
    MOVB,=,n    %arg1,sub,DN
    MOVB        %arg0,f
 LP MOVB        sub,%arg0
    BLR         f,%rp
    MOVB        %ret0,%arg0 ;1st cons arg
    LDI         0,%arg1     ;2nd cons arg
    BL          cons,%rp    ;cons
    STW         %ret0,4(end);(rplacd end x)
    MOVB        %ret0,end   ;(setq end x)
    LDW         4(sub),sub
    COMIB,<>,n  0,sub,LP
 DN LDW         -52(%sp),%ret0;return cdr r
    .LEAVE
```

To save on overhead, the dummy header cons cell is simulated on the PA-RISC stack instead of calling cons. The first store instruction (`STW`) initializes the cdr of this cons cell to `nil` by storing `0` in the appropriate stack frame slot. The add immediate instruction (`ADDI`) initializes `end` to point to four bytes in front of this cdr. The car part of the cons cell never actually has to exist since it is never referred to. A store instruction is used to implement the required `rplacd`.

Comparing `maplist_rplacd` with `maplist_nreverse` shows that the cost of eliminating the call on `nreverse` is only one instruction execution per cons cell created in the output list. This clearly opens the door to a savings in run time. However, it turns out that `nreverse` is so cheap to compute that the door is not opened very far.

To see how inexpensive `nreverse` is, it is useful to look at the modified implementation shown in `nreverse-unrolled`. By unrolling the loop so that three consecutive cons cells are handled on each cycle of the loop, one can eliminate the pointer shuffling that is required in the implementation shown above. This reduces the number of basic operations per cons cell from five to three.

```
(defun nreverse-unrolled (list)
  (prog ((prev nil) next)
        (when (null list) (return nil))
     lp (setq next (cdr list))
        (rplacd list prev)
        (when (not next) (return list))
        (setq prev (cdr next))
        (rplacd next list)
        (when (not prev) (return next))
        (setq list (cdr prev))
        (rplacd prev next)
        (when (not list) (return prev))
        (go lp)))
```

`Nreverse_unrolled` approximates the best PA-RISC implementation of `nreverse-unrolled` and therefore `nreverse`. `Cdr` and `rplacd` are implemented with load and store instructions, as above. The tests for the end of the list are implemented using compare immediate and

```
nreverse_unrolled
prev .reg %ret1
list .reg %ret0
next .reg %arg0
     .CALLINFO
     .ENTER
     LDI         0,prev
     MOVB,=,n    %arg0,list,DN2
 LP  LDW         4(list),next   ; cdr
     STW         prev,4(list)   ; rplacd
     COMIB,=,n   0,next,DN2     ; when
     LDW         4(next),prev   ; cdr
     STW         list,4(next)   ; rplacd
     COMIB,=,n   0,prev,DN1     ; when
     LDW         4(prev),list   ; cdr
     STW         next,4(prev)   ; rplacd
     COMIB,<>,n  0,next,LP      ; when
     MOVB,TR,n   prev,%ret0,DN2
 DN1 MOVB        next,%ret0
 DN2 .LEAVE
```

branch instructions (`COMIB`). In the loop, only three instructions per cons cell are require to reverse the input list.

Just as in `maplist_rplacd`, one instruction per cons cell is all that is needed to store the correct cdr pointers. The only overhead in comparison with `maplist_rplacd` is that `nreverse_unrolled` has to traverse the list a second time. This requires two instructions per cons cell.

One way to summarize the results in this section is to say that the `rplacd` approach to creating a list has a clear theoretical advantage of two instructions per cons cell over the `nreverse` approach.

However, a better way to summarize the results is to consider the percentage improvement. Taken together, `maplist_nreverse` and `nreverse_unrolled` use seven instructions per cons cell plus the computation that is performed by `cons`. In the best of all possible worlds, `cons` will require several instructions even if it is coded in line. Therefore, the speed advantage of the `rplacd` approach is at most 20% or so.

It should be noted that the results presented above are not distorted by the fact that we have looked at only one specific hardware architecture. The PA-RISC architecture is at an intermediate level of complexity. There are RISC machines with much simpler instruction sets. There are non-RISC machines with much more complex instruction sets.

Switching to a simpler architecture would increase the number of instructions in the examples above. However, the programs are so similar that it is hard to imagine that the relative lengths of the critical loops would change much. The same can be said about switching to a more complex architecture.

Hand Tailored Code

It must be kept in mind that the speed advantage of the `rplacd` approach presented in the last section is only theoretical, because the hyper-efficient code shown is the result of careful hand coding, rather than being the output of a Lisp compiler. It is unlikely that any compiler will produce code that is anywhere near as efficient.

To start with, the typical compiler is likely to implement operations like `rplacd` as subroutine calls rather than inline instructions. In addition, it may store some intermediate values on the stack rather than in registers. Together, these and other factors are liable to lead to compiled code that is several times larger than the idealized code above.

The deficiencies of compilers are unfortunate in many ways, but in the main, they are not relevant to the current discussion. There is no reason to believe that the compiler will work better for any one function than for the others. Therefore, the quality of the compiler should not effect the comparisons being made here, except for one important thing.

Since `nreverse` is a built-in function, implementors may choose to write it using special implementation-specific subprimitives and/or to hand compile it. Either way, this could tilt the performance balance in favor of the `nreverse` approach, because the hand tailored code in `nreverse` could perform a good deal better than the equivalent user code required by the `rplacd` approach. Given that the typical compiler produces relatively voluminous code, this difference can be quite significant.

Cache Performance

Over the past decade, processor speed has increased much faster than main memory speed. This has progressed to the point where the instruction cycle time is 1/10 of the memory cycle time or even less. This mismatch is overcome by using fast cache memory between the processor and the main memory. However, to work well, this requires good memory locality in order to minimize cache misses.

This could tilt the performance balance in favor of the `rplacd` approach, because that approach processes the cons cells created in a very local way. In contrast, the traversal of the output list initiated by `nreverse` does not begin until after the entire list has been created. If the output list is long enough, some of the cons cells in it may have fallen out of the cache before they are revisited by `nreverse`. If this happens, the main loop of `nreverse` could slow down by the equivalent of 10 additional instructions or more for each of these cons cells.

However, it is unlikely that this would be a significant difference for two reasons. First, since the `nreverse` visits the most recently created cons cells first, the initial cells it visits must be in the cache. Second, given that the typical compiler produces relatively voluminous code, 10 instructions is not liable to be a significant percentage difference.

Some Experiments

To assess the relative practical significance of the arguments above, I performed a set of experiments in two very different Lisp implementations: Lucid Common Lisp on an HP-730 machine and Allegro Common Lisp on an old (slow) Apple MacIntosh.

On the HP machine, the `maplist-nreverse` and `maplist-rplacd` compiled into 54 and 65 instructions respectively—4 to 5 times the size of the idealized code. Using the compiled code, I determined the average time required per cons cell for various size input lists. I used `#'identity` as the map function to maximize the percentage of time spent actually creating the output list. The results of these experiments are shown in the table below.

	input list length					
	1	10	10^2	10^3	10^4	10^5
`nreverse`	8.3	3.1	2.6	2.6	2.9	6.2
`rplacd`	8.5	3.3	2.8	2.8	3.1	6.4

The numbers in the body of the table are the computation time in terms of microseconds per cons created. Startup costs cause both approaches to perform poorly on a per-cons basis when applied to lists of length one. Both approaches also began to behave badly on lists of length one hundred thousand, perhaps due to cache misses or some other memory phenomenon.

Interestingly, the `nreverse` approach is consistently faster than the `rplacd` approach, but by at most 8%. It appears that this is due to hand-coding of `nreverse`.

Comparing the speeds of the system implementation of `nreverse` and the result of compiling `nreverse-unrolled` revealed that the user compiled version is 66% slower (1.0 microseconds per cons cell versus .6 microseconds per cons cell). This suggests that something was done to improve the machine code for `nreverse` in comparison with what a user can easily get the compiler to generate. The hand-coding benefit obtained (.4 microseconds per cons cell) is easily large enough to account for the fact that the `nreverse` approach is faster than the `rplacd` approach, and to suggest that without the hand-coding benefit, the `rplacd` approach would be faster.

The data does not reveal any cache-miss penalty for the `nreverse` approach on long lists.

On the MacIntosh, the `maplist-nreverse` and `maplist-rplacd` compiled into 29 and 42 instructions respectively. This reflects the fact that the MacIntosh is not a RISC machine. Since I know very little about the machine instructions the MacIntosh uses, I cannot comment on how close this is to the best that is possible.

Timing experiments identical to the ones above revealed the following.

	input list length					
	1	10	10^2	10^3	10^4	10^5
`nreverse`	280	95	75	75	75	73
`rplacd`	280	110	92	91	90	90

As above, startup costs cause poor performance on a per-cons basis for lists of length one. However, there is no diminution of speed even on very long lists.

For all but the shortest lists, the `nreverse` approach is faster than the `rplacd` approach—in general, 15-18% faster. It appears that this is due to a major hand-coding effect for `nreverse`.

Comparing the speeds of the system implementation of `nreverse` and the result of compiling `nreverse-unrolled` reveals that the user compiled version is 208% slower (40 microseconds per cons cell versus 13 microseconds per cons cell). This suggests that `nreverse` has been very carefully hand coded. As above, the hand-coding benefit obtained (27 microseconds per cons cell) is easily large enough to account for the fact that the `nreverse` approach is faster than the `rplacd` approach, and to suggest that without the hand-coding benefit, the `rplacd` approach would be faster.

The data does not suggest any cache-miss penalty for either approach on long lists.

The experiments suggest that of the three sources of speed difference between the two approaches (a theoretical advantage for `rplacd`, a hand-coding advantage for `nreverse`, and a cache performance advantage for `rplacd`) the hand-coding advantage wins out and therefore the `nreverse` approach is fastest.

However, more than this, it is clear that the speed difference between the two approaches is not very large. If the computation being performed to compute the elements being consed together involved much more than just computing `identity`, the difference would recede into complete insignificance.

If you are interested in such things, you might run an experiment in your Lisp to see if any significant speed difference can be found. However, in the absence of clear evidence for such a difference, I recommend assuming that there is none.

Conclusion

The `rplacd` approach to creating an output list has a theoretical speed advantage, but as a practical matter this appears to be overwhelmed by the fact that `nreverse` is a system function that can be hand coded by the system implementors. As a result, the `nreverse` approach is probably fastest in most Lisp implementations. Even if the `rplacd` approach is faster in a given Lisp, it is unlikely to be much faster. Therefore, since the `nreverse` approach is simpler and clearer it is the best thing to do in almost every situation.

The only situation where I would consider using the `rplacd` approach is if I were a Lisp system implementor and had the opportunity to write a system function where I could hand tune machine code for creating a list. In this situation, the `rplacd` approach should be able to achieve its theoretical advantages and I would consider trying. However, it should be realized that there would be much more to be gained through the hand tuning itself than through the choice of which approach to tune.

In closing I would like to note that the very best thing to do is to avoid writing code that conses lists altogether. Whenever possible, you should use standard parts of Common Lisp that do the consing for you. In particular, you should use functions like `replace`, `map`, `reduce`, `remove`, `union`, etc. whenever they are appropriate. Beyond this, you should take advantage of looping macro packages such as `loop` and Series.

For example, using the extended features of `loop` that are available in the proposed standard for Common Lisp [2], a simple version of `maplist` could be written as follows.

```
(defun maplist-loop (f list)
  (loop for sub on list
        collect (funcall f sub)))
```

Alternatively, the Series macro package [3] could be used as shown below.

```
(defun maplist-series (f list)
  (collect
    (map-fn t f (scan-sublists list))))
```

Either way, the resulting code is clearer, more compact, and no slower than anything else you can write.

References

[1] Hewlett-Packard, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, Hewlett-Packard, Cupertino CA, 1986.

[2] White, J.L., "Loop", in *Common Lisp: the Language*, Second Edition, 709–747, Steele G.L.Jr., Digital Press, Maynard MA, 1990.

[3] Waters R.C., "Automatic Transformation of Series Expressions into Loops", *ACM Transactions on Programming Languages and Systems*, 13(1):52–98, January 1991.

$\boxed{()}$