Parenthetically Speaking with Kent M. Pitman

Ambitious Evaluation

A New Reading of an Old Issue

Much has been written about Lazy Evaluation in Lisp—less about the other end of the spectrum—Ambitious Evaluation. Ambition is a very subjective concept, though, and if you have some preconceived idea of what you think an Ambitious Evaluator might be about, you might want to set it aside for a few minutes because this probably isn't going to be what you expect.

A Familiar Question

It all started in 1986 when I was on staff at the MIT AI Lab and an undergrad studying Lisp asked me why the following piece of code didn't "work":

```
(PROGN (SETQ *READ-BASE* 8.) 10)
⇒ 10.  ; not 8.
```

I chuckled for a moment with the kind of easy superiority that I have since learned to take as a sign that it's *me*, not the person I'm answering questions for, that really has something to learn.

I answered the question in the obvious way, explaining how *of course* Lisp has to read the entire form into memory before it can execute any of it. So by the time the expression above has been processed by `READ` and is ready to have `EVAL` called, it is too late for any change to the setting of `*READ-BASE*` to affect it. The student was mostly happy with the explanation, but the more I thought about it, I was not.

Increasingly, I've always found that the phrase "of course" is bandied about far too much, especially in computer science, a "science" which sometimes deals in provable facts (like algorithmic complexity) but that often deals in much more subjective, near-religious issues (like "Fred implemented it this way, and Fred is generally right on such matters so everyone should do the same.") as if they were facts.

Too often in computer science, I think, we assume that the first person to implement something has a lock on The Right Way To Think About It™, and we define any expectation to the contrary as an obvious illustration of why the person with that expectation is a novice and we're experts. It's not always so, and often enough so that I think we need to make a more regular practice of recognizing and questioning our design assumptions. What distinguishes experts from novices is not that they make the choices, but that their choices will provoke fewer questions and their choices will tend to stand up better to those questions that do come along afterward.

Parenthetically Speaking expresses opinions and analysis about the Lisp family of languages. Except as explicitly indicated otherwise, the opinions expressed are those of the author and do not necessarily reflect the official positions of any organization or company with which the author is affiliated. Kent M. Pitman can be reached via the Internet as KMP@Harlequin.COM, or by U.S. mail at Harlequin, Inc., Suite 904, One Cambridge Center, Cambridge, MA 02142 U.S.A.

Flashback

In the late 1970's and early 1980's, a technology arose at MIT, which many referred to informally as "lisp machine style rubout handling". It allowed a very useful modularity that fully separates parser technology from rubout handling (or general-purpose input editing). Since that time, this technology has become widespread in Lisp-based systems that do rubout handling.

To see why fully separating parsing from input editing is important, consider the following program that parses a word from a stream of input:

```
(DEFVAR *WORD-DELIMITERS* '(#\Newline #\Space #\, #\.))

(DEFUN READ-A-CHAR ()  ; This function will make sense later.
  (READ-CHAR))

(DEFUN READ-A-WORD-1 ()
  (WITH-OUTPUT-TO-STRING (STR)
    (LOOP (LET ((C (READ-A-CHAR)))
            (IF (MEMBER C *WORD-DELIMITERS*) (RETURN))
            (WRITE-CHAR C STR)))))
```

This function will work as expected when reading from non-interactive streams where there is no need for rubout handling, but if a user typing text makes a mistake, this will attempt to accumulate Rubout characters along with other characters as part of the string to be accumulated, rather than using them to repair preceding typographical errors.

To make Rubout characters work interactively, the following naive solution presents itself:

```
(DEFVAR *WORD-DELIMITERS* '(#\Newline #\Space #\, #\.))

;;; In real code, a character like #\Rubout would be used for the
;;; 'delete-character' character, and #\Control-U or #\Clear-Input might be
;;; used for the 'delete-line' character.  For this article, using printing
;;; characters will make things clearer.
(DEFVAR *DELETE-CHARACTER-CHAR* #\%)
(DEFVAR *DELETE-LINE-CHAR* #\@)

(DEFUN READ-A-CHAR ()  ; To be explained soon.
  (READ-CHAR))

;;; In a real system, this would call out to display operations to make
;;; the character disappear.  But for this article, this will be clearer.
(DEFUN UNDISPLAY-CHAR (CHAR)
  (FORMAT T "[~C]" CHAR))

(DEFUN READ-A-WORD-2 ()
  (LET ((CHARS '()))
    (LOOP (LET ((CHAR (READ-CHAR)))
            (COND ((EQL CHAR *DELETE-CHARACTER-CHAR*)
                   (UNDISPLAY-CHAR CHAR)
                   (POP CHARS))
                  ((EQL CHAR *DELETE-LINE-CHAR*)
                   (MAPC #'UNDISPLAY-CHAR CHARS)
                   (SETQ CHARS '()))
                  ((MEMBER CHAR *WORD-DELIMITERS*)
                   (UNREAD-CHAR CHAR)
                   (RETURN (COERCE (NREVERSE CHARS) 'STRING)))
                  (T
                   (PUSH CHAR CHARS)))))))
```

Contrasting READ-A-WORD-1 and READ-A-WORD-2, we see several bad effects of including rubout handling within the definition of the word reader function itself:

- Each parser function must do rubout handling. This is a lot of coding effort, and also bloats the size of any delivered product.
- If users are not to be driven crazy, each parser function must *agree* on how to handle rubout and other editing characters. This is a lot of coordination effort. The more bells and whistles one wants, the more it will be impossible to keep all of them up to date. The result is a very irregular user interface.
- In order to handle sophisticated editing, pretty abstractions like WITH-OUTPUT-TO-STRING may need to be sacrificed because they hide the data structure which is accumulating the parser result.

In order to avoid these problems, lisp machine style rubout handling uses a modularity of full separation between input editing, in which a parser function is passed as an argument to a rubout handler, as in:

```
(INVOKE-RUBOUT-HANDLER #'READ-A-WORD-1)
```

The simplest form of a rubout handler is one that simply prompts for input and then permits an activation character (such as Return or Enter) to indicate that parsing should begin. The chief problems with this solution are these:

- In the case of doing line-at-a-time input using Return, such systems do not typically allow one to use rubout to move back up to a previously completed line.
- There is a problem about what to do with extra input if multiple forms are entered in response to a single prompt on a line.
- Syntax errors are not detected in a timely way. For example, in the case of typing a multi-line expression with each line ended by Return, the entire expression must be retyped to correct an error on any line.

If we could make rubout handling occur at the same time as parsing, then error detection could be made to occur at a point of error.

The Magic Revealed

"How can a program that has no understanding of rubout be magically transformed into one that does by a function which cannot see its internals?" The answer likes in how this function gets characters from the input stream. It's not really magic—just a little "slight of handler."

In a real system, INVOKE-RUBOUT-HANDLER establishes an input stream which has a special method for READ-CHAR which returns characters only if they are not editing characters. After all, there is no difference between reading from a file and reading from the terminal of someone who happens never to type an incorrect character.

But what happens when an editing character is typed? In that case, rather than return normally, the stream operation for READ-CHAR throws out of the call to READ-CHAR, past the call to READ-A-WORD-1, and all the way to INVOKE-RUBOUT-HANDLER. Secretly, we now learn, this READ-CHAR has also been remembering all of our input since we started. It then performs the editing on its remembered characters, and it restarts the read in a way that first reads characters from the remembered (and now edited) sequence of characters until it is ready for terminal input again. From the user's point of view, it is as if the editing process backed up or was changed, but really it has been started over with fewer or different characters.

Because the mechanisms for extending a stream are complicated and implementation dependent, the code I'll show here will illustrate the behavior by instead redefining the READ-A-CHAR function we've

been using. Calls to READ-A-CHAR will represent the operation on the rubout-handling stream, and calls to Common Lisp's READ-CHAR will represent calls to the underlying non-rubout-handling stream. Doing the examples this way should mean you can run this code in any Common Lisp implementation:

```
(DEFVAR *INPUT-SEEN-STACK* '())
(DEFVAR *INPUT-WAITING-QUEUE* '())
(DEFVAR *RUBOUT-HANDLING* NIL)

(DEFUN INVOKE-RUBOUT-HANDLER (PARSER-FUNCTION)
  (LET ((*INPUT-SEEN-STACK* '())
        (*INPUT-WAITING-QUEUE* '())
        (*RUBOUT-HANDLING* T))
    (LOOP (CATCH 'EDIT-CHARACTER-TYPED
            (RETURN-FROM INVOKE-RUBOUT-HANDLER
              (FUNCALL PARSER-FUNCTION)))
          (HANDLE-EDITING)
          (SETQ *INPUT-WAITING-QUEUE* (REVERSE *INPUT-SEEN-STACK*)))))

(DEFVAR *DELETE-CHARACTER-CHAR* #\%)
(DEFVAR *DELETE-LINE-CHAR* #\@)

(DEFUN UNDISPLAY-CHAR (CHAR) (FORMAT T "[~C]" CHAR))

(DEFUN HANDLE-EDITING ()
  (LOOP (LET ((CHAR (READ-CHAR)))
          (COND ((EQL CHAR *DELETE-CHARACTER-CHAR*)
                 (WHEN *INPUT-SEEN-STACK*
                   (UNDISPLAY-CHAR (POP *INPUT-SEEN-STACK*))))
                ((EQL CHAR *DELETE-LINE-CHAR*)
                 (MAPC #'UNDISPLAY-CHAR *INPUT-SEEN-STACK*)
                 (SETQ *INPUT-SEEN-STACK* NIL))
                (T ;; Must be done editing.
                 (UNREAD-CHAR CHAR)
                 (RETURN))))))

;;; Normally, this would be a different method on READ-CHAR, but we're just
;;; indirecting through a different function name for presentational simplicity.
(DEFUN READ-A-CHAR ()
  (IF *RUBOUT-HANDLING*
      (IF *INPUT-WAITING-QUEUE*
          (POP *INPUT-WAITING-QUEUE*)
          (LET ((C (PEEK-CHAR NIL)))
            (COND ((OR (EQL C *DELETE-CHARACTER-CHAR*)
                       (EQL C *DELETE-LINE-CHAR*))
                   (THROW 'EDIT-CHARACTER-TYPED NIL))
                  (T
                   (READ-CHAR) ;for effect
                   (PUSH C *INPUT-SEEN-STACK*)
                   C))))
      (READ-CHAR)))

(DEFUN UNREAD-A-CHAR (CHAR)
  (IF *RUBOUT-HANDLING*
      (PUSH CHAR *INPUT-WAITING-QUEUE*)
      (UNREAD-CHAR CHAR))
  CHAR)
```

Given all of this code, one might expect an interaction like:

```
(INVOKE-RUBOUT-HANDLER #'READ-A-WORD-1)
abc%[c]%[b]lpha@[a][h][p][l][a]bex%[x]ta
=> "beta"
```

The Rubout Handler Meets #.

By extension of what we've seen, if you either imagine that READ was written with READ-A-CHAR instead of READ-CHAR or imagine a version of INVOKE-RUBOUT-HANDLER which can make the right kind of stream so that native READ-CHAR can be called, then it works to do:

```
(INVOKE-RUBOUT-HANDLER #'READ)
```

But what happens when we use #. in conjunction with READ using this style of rubout handler? Consider the expression:

```
(FOO #.(PRINT 'BAR) BAZ
```

The instant you type the ")" on the (PRINT 'BAR) expression, you will see BAR. But moreover, each time you're in non-editing mode and type Rubout, the next time you type a non-editing character it will restart the read and you'll see the BAR printed again! This is because as soon as you type a non-editing character, the READ restarts and the #.(PRINT 'BAR) will be reexecuted.

Ambitious Evaluation, Revisited

Armed with all of this information, we can finally revisit my original point about the design decisions inherent in a Read-Eval-Print loop. We are taught that Lisp will do:

```
(DEFUN READ-EVAL-PRINT-LOOP ()
  (LOOP (PRINT (EVAL (INVOKE-RUBOUT-HANDLER #'READ)))))
```

but it's important to understand that nature doesn't call for this behavior. It's just a choice. We could, for example, do:

```
(DEFUN READ-EVAL-PRINT-LOOP-2 ()
  (LOOP (PRINT (INVOKE-RUBOUT-HANDLER #'(LAMBDA () (EVAL (READ)))))))
```

On the other hand, this would only restart while you were typing so would give the same result. But could we write a variant of this which would co-routine EVAL with READ so that expressions were evaluated the instant enough information was available to evaluate them, rather than later? The answer is yes. I call such a facility an Ambitious Evaluator, and I offer a toy definition of a function called EVAL-WHILE-READING as a proof of concept.

A Long-Overdue Apology

One interesting consequence of using an Ambitious Evaluator is the following:

```
(INVOKE-RUBOUT-HANDLER #'EVAL-WHILE-READING)
(PROGN (SETQ *READ-BASE* 8.) 10)
⇒ 8.  ; not 10.
```

So to that unnamed undergraduate, whose name I have sadly forgotten, I hereby publicly apologize for having ever said "*of course*, the answer has to be 10." That turns out to have been a naive response on my part. I should perhaps have said, "Because Lisp made the design choice of separating the reader from the evaluator, the result happens to be 10." There were other possible choices, as is almost always the case if you look closely enough.

It was wrong of me to have so hastily dismissed the possibility that there were other points of view. I still believe that 10, not 8, is a better value for this to return, because I believe that separate, modular definitions of READ and EVAL are superior to a combined definition—for reasons similar to why I believe rubout handling and parsing should be separated. And *of course* modularity is good.

An Aside about Conditions

In our sample rubout handler above, we didn't really show how lisp machine style rubout handling offers more timely correction of errors. To see how that would work, consider that a command like Rubout is a voluntary admission on the part of the user that a syntax error has occurred—one the evaluator might not notice until later. In some cases, however, the user might not be aware that an error has occurred but the parser might. In that case, the parser can cooperate with the rubout handler to force the user involuntarily into a mode in which editing is required. This might be done by:

```
(DEFINE-CONDITION SIMPLE-PARSER-ERROR (SIMPLE-CONDITION))

(DEFUN PARSE-ERROR (FORMAT-STRING &REST FORMAT-ARGUMENTS)
  (ERROR 'SIMPLE-PARSER-ERROR
     #.(IF (FIND-SYMBOL "SIMPLE-CONDITION-FORMAT-STRING" "CL")
         :FORMAT-STRING :FORMAT-CONTROL) ;ignore CLtL2/ANSI differences
     FORMAT-STRING
    :FORMAT-ARGUMENTS FORMAT-ARGUMENTS))
```

This would permit the rubout handler to notice conditions relevant to parsing errors and permit user intervention. For example, we might modify the definition of our rubout handler as follows:

```
(DEFUN INVOKE-RUBOUT-HANDLER-2 (PARSER-FUNCTION)
  (LET ((*INPUT-SEEN-STACK* '())
        (*INPUT-WAITING-QUEUE* '())
        (*RUBOUT-HANDLING* T))
    (LOOP (CATCH 'EDIT-CHARACTER-TYPED
            (HANDLER-BIND ((SIMPLE-PARSER-ERROR
                            #'(LAMBDA (C)
                                (FORMAT T "~&~A~%~C" C *DELETE-CHARACTER-CHAR*)
                                (UNDISPLAY-CHAR (POP *INPUT-SEEN-STACK*))
                                (THROW 'EDIT-CHARACTER-TYPED)))) ;white lie
              (RETURN-FROM INVOKE-RUBOUT-HANDLER
                (FUNCALL PARSER-FUNCTION))))
          (HANDLE-EDITING)
          (SETQ *INPUT-WAITING-QUEUE* (REVERSE *INPUT-SEEN-STACK*)))))
```

Using such a modified rubout handler, any reader errors would be detected at the time they occurred, and an opportunity to recover from them would be immediately available:

```
(invoke-rubout-handler #'eval-while-reading)
(cond ((zerop 3) (
This COND branch failed.
Don't bother typing the body of the failing clause, huh? It's boring.
%[(]) (t 3))
⇒ 3
```

Conclusion

My Ambitious Evaluator has really no practical value at all, really—unless you're writing a column for *Lisp Pointers*. And even then, someone's already done that topic so it's not much good there either.

My *real* goal here was partly to expose people to the way parsers and input editors are usually modularized in Lisp. And it was also partly to keep us all reminded that the reason we do things the way we do (e.g., separating input editing from parsing, and separating READ from EVAL) is not because it's the only way to do things, but because it's one of many design choices we made.

The world is full of choices. Sometime an examination of the choices leads you to the same conclusion as you'd have arrived at without examining them. But looking over all of your options when making a design decision is always a good idea if you can spare the time.

Acknowledgments

Andy Latto, Becky Spainhower, and Chris Stacy read drafts of this paper, and provided much useful feedback. It's likely that people somewhere have written papers on Ambitious Evaluation that attach a different, more serious, meaning to the topic. If so, I'd like to thank them in advance for having a sense of humor about the whimsy nature of *my* Ambitious Evaluator.

Proof of Concept: An Ambitious Evaluator

This is code that I wrote for fun in 1986. I've dusted it off slightly and updated the syntax so it runs in Common Lisp:

```
(DEFUN AMBITIOUS-READ-EVAL-PRINT ()
  (LOOP (WRITE-CHAR #\Newline)
   (PRINT (INVOKE-RUBOUT-HANDLER #'EVAL-WHILE-READING))))

(DEFVAR *IMPATIENT* T)
(DEFVAR *PAREN* (GENSYM "CLOSE-PAREN"))

(DEFUN PEEK-CHAR-GOBBLING-WHITESPACE ()
  (DO ((CHAR (READ-A-CHAR) (READ-A-CHAR)))
      ((NOT (MEMBER CHAR '(#\Space #\Tab #\Return)))
       (UNREAD-A-CHAR CHAR) CHAR)))

(DEFUN READ-FORM-OR-CLOSE-PAREN ()
  (IF (EQL (PEEK-CHAR-GOBBLING-WHITESPACE) #\))
      (PROGN (READ-A-CHAR) *PAREN*)
      (READ-LITERAL)))

(DEFUN READ-EVALED-FORM-OR-CLOSE-PAREN ()
  (IF (EQL (PEEK-CHAR-GOBBLING-WHITESPACE) #\))
      (PROGN (READ-A-CHAR) *PAREN*)
      (EVAL-WHILE-READING)))

(DEFUN EVAL-WHILE-READING ()
  (CASE (PEEK-CHAR-GOBBLING-WHITESPACE)
    (#\) (PARSE-ERROR "Close paren seen where not expected."))
    (#\; (READ-LINE) (EVAL-WHILE-READING))
    (#\' (READ-A-CHAR) (READ-LITERAL))
    (#\( (READ-A-CHAR)
         (IF (EQL (PEEK-CHAR-GOBBLING-WHITESPACE) #\)) '()
             (LET ((FN (READ-LITERAL)))
               (CASE FN
                 ((COND)     (EVAL-COND-WHILE-READING))
                 ((QUOTE)    (EVAL-QUOTE-WHILE-READING))
                 ((PROGN)    (EVAL-PROGN-WHILE-READING))
                 ((SETQ)     (EVAL-SETQ-WHILE-READING))
                 ((DEFUN)    (EVAL-DEFUN-WHILE-READING))
                 (OTHERWISE (EVAL-FUNCTION-WHILE-READING FN))))))
    (OTHERWISE (EVAL (READ-LITERAL)))))

(DEFUN EVAL-SETQ-WHILE-READING ()
  (DO ((RV NIL) (VAR (READ-FORM-OR-CLOSE-PAREN) (READ-FORM-OR-CLOSE-PAREN)))
      ((EQL VAR *PAREN*) RV)
    (LET ((VALUE (READ-EVALED-FORM-OR-CLOSE-PAREN)))
      (COND ((EQL VALUE *PAREN*) (PARSE-ERROR "Wrong number of args to SETQ"))
            (T (SETQ RV (SET VAR VALUE)))))))

(DEFUN EVAL-FUNCTION-WHILE-READING (FN)
  (DO ((FORM (READ-EVALED-FORM-OR-CLOSE-PAREN) (READ-EVALED-FORM-OR-CLOSE-PAREN))
       (L NIL (CONS FORM L)))
      ((EQL FORM *PAREN*)
       (APPLY FN (NREVERSE L)))))
```

```
(DEFUN EVAL-QUOTE-WHILE-READING ()
  (PROG1 (READ-LITERAL)
         (COND ((EQL (PEEK-CHAR-GOBBLING-WHITESPACE) #\)) (READ-A-CHAR))
               (T (PARSE-ERROR "Wrong number of args to QUOTE")))))

(DEFUN EVAL-COND-WHILE-READING ()
  (PROG1 (TRY-COND-CLAUSES-WHILE-READING)
         (IGNORE-REST-OF-LIST "You already passed the succeeding COND branch.
Save yourself some work and just close off the COND, huh?")))

(DEFUN TRY-COND-CLAUSES-WHILE-READING ()
  (DO ((C (PEEK-CHAR-GOBBLING-WHITESPACE) (PEEK-CHAR-GOBBLING-WHITESPACE)))
      ((EQL C #\)))
    (COND ((EQL C #\()
           (READ-A-CHAR)
           (MULTIPLE-VALUE-BIND (WIN? VALUE)
               (TRY-COND-CLAUSE-WHILE-READING)
             (IF WIN? (RETURN VALUE))))
          (T (PARSE-ERROR "Ill-formed COND clause."))))))

(DEFUN IGNORE-REST-OF-LIST (&OPTIONAL (MSG "Just close off the list.
I'm ignoring stuff from here on at this level anyway."))
  (COND (*IMPATIENT*
         (LET ((C (PEEK-CHAR-GOBBLING-WHITESPACE)))
           (COND ((EQL C #\)) (READ-A-CHAR) NIL)
                 (T (PARSE-ERROR MSG)))))
        (T
         (DO ((FORM (READ-FORM-OR-CLOSE-PAREN) (READ-FORM-OR-CLOSE-PAREN)))
             ((EQL FORM *PAREN*))))))

(DEFUN EVAL-PROGN-WHILE-READING (&OPTIONAL (SEED NIL))
  (DO ((VAL SEED FORM)
       (FORM (READ-EVALED-FORM-OR-CLOSE-PAREN)
             (READ-EVALED-FORM-OR-CLOSE-PAREN)))
      ((EQL FORM *PAREN*) VAL)))

(DEFUN TRY-COND-CLAUSE-WHILE-READING ()
  (LET ((TEST (EVAL-WHILE-READING)))
    (COND ((NOT TEST) (BEEP :LOW)
                      (IGNORE-REST-OF-LIST "This COND branch failed.
Don't bother typing the body of the failing clause, huh? It's boring.")
                      (VALUES NIL NIL))
          (T (BEEP :HIGH) (VALUES TEST (EVAL-PROGN-WHILE-READING TEST))))))

(DEFUN EVAL-DEFUN-WHILE-READING ()
  (DO ((FORM (READ-FORM-OR-CLOSE-PAREN) (READ-FORM-OR-CLOSE-PAREN))
       (L '() (CONS FORM L)))
      ((EQL FORM *PAREN*) ;DEFUN's body can't run until args available in a call
       (EVAL `(DEFUN ,@(NREVERSE L))))))

(DEFUN BEEP (KIND) KIND)  ;Replace with something useful if you like.

(DEFUN PARSE-ERROR (FORMAT-STRING &REST FORMAT-ARGUMENTS)
  (APPLY #'ERROR FORMAT-STRING FORMAT-ARGUMENTS))

(DEFUN READ-LITERAL ()  ;Doesn't do whole job, but enough for this sample.
  (VALUES (READ-FROM-STRING
            (WITH-OUTPUT-TO-STRING (S)
              (DO ((C (READ-A-CHAR) (READ-A-CHAR)))
                  ((AND (NOT (ALPHANUMERICP C)) (NOT (FIND C ".+-*/=><")))
                   (UNREAD-A-CHAR C))
                (WRITE-CHAR C S))))))
```