Perlish Lisp

Gene Michael Stover

created Thursday, 21 October 2004
updated Thursday, 21 October 2004

Contents

1 Introduction

Another programmer mentioned that, though he likes Lisp, he found Perl to be easier for basic data processing tasks. Perl is ideal & damned near optimal for processing input files that are one-record-per-line. For example, the classic Perl loop:

```
# an example of the classic Perl loop
while(<>) {
    chomp;
    @Line = split "\t",$_;
    if ($Line[4] =~ /\.hooha$/) {
```

```
        ...
    }
}
```

Just for fun, we thought about how we might accomplish similar things with similar clarity & conciseness in Lisp.

1.1 Audience

This article is probably most suited to

- beginning Lisp programmers who want to see more examples of Lisp code or
- more experienced Lisp programmers who have a jones to read something about programming, no matter how trivial.

2 Foundations

Perl lets you read a string from `*standard-input*` with its `<>` operator. It returns the next line of input or false.

Conceptually, we can achieve the same thing in Lisp with "`(read-line strm nil nil)`", but it's not as concise. Let's make a `next-line` Lisp function that is more concise.

```
(defun next-line (&optional (strm *standard-input*))
  "Return the next line of input as a string, or Nil."
  (declare (type stream strm))
  (assert (input-stream-p strm))
  (read-line strm nil nil))
```

Another part of the classic Perl main loop is the `chomp` function. It removes leading & trailing white space from its single argument.

In Lisp, I'd rather not modify a string itself, but we could simulate the Perl function `chop`, which returns the contetns of is argument without leading & trailing white space, but `chop` does not modify its argument. Here is a `chop` function for Lisp.

```
(defun chop (str)
  "Return the contents of STR, which must be a string,
without leading \& trailing spaces, tabs, & end-of-line
characters.  Does not modify STR."
  (declare (type string str))
  ;; Common Lisp's STRING-TRIM function already does
  ;; most of the work for us.
  (string-trim '(#\Tab #\Space #\Newline) str))
```

Finally, we need a Lisp equivalent of Perl's `split` function. Here is an implementation I copied from my own CyberTiggyr Tigris ([1,)] library.

```
(defun split (sep str)
  "Return a list of the substrings from STR that are separated by
SEP.  STR is a string.  SEP is a character.  STR will not be
modified.  From CyberTiggyr Tigris."
  ;; See the SUBSEQ a few lines below?  I think it might
  ;; be more efficient to use MAKE-ARRAY & to displace the
  ;; new vector to the original string.  It might reduce the
  ;; amount of copying.
  (do ((s 0 (1+ i))
       (i (position sep str) (position sep str :start (1+ i)))
       (lst () (cons (subseq str s i) lst)))
      ((null i) (nreverse (cons (subseq str s) lst)))))
```

Whew! So much for the fundamentals. We these functions, we can do much of what Perl does, though it remains to be seen if we can do it as conveniently.

3 Regular Expressions

I don't have a regular expression library for Lisp. I know they are out their, & I'm sure they are fine, but I haven't found one I like, & I haven't found a strong enough need for one to motivate me to find one I like.

If you were going to use what I show in this article, & you have a regular expression library, use it! I don't have one, so I'll assume we always have a function called `match` which indicates whether a given input record is appropriate for processing (whatever "processing" means).

Even without regular expressions, you can write a reasonably un-cluttered `match` function for most criteria. For example, if we wanted to process all records whose fourth field ended with "dowah", we would define `match` like this:

```
(defun match? (lst)
  "Return true if & only if the fourth element of LST
is a string that ends with 'dowah'.  In other words,
the fourth element of LST would match the regular
expression 'dowah$'."
  (let ((pattern "dowah"))
    (and (listp lst)
         (stringp (elt lst 3))
         (>= (length (elt lst 3)) (length pattern))
         (equal (subseq (elt lst 3)
                        (- (length (elt lst 3))
                           (length pattern)))
                pattern))))
```

It's as pretty as Perl's regular expressions, but it's not horrid.

If you needed to detect many patterns at the end of fields, It'd be simple to write a function that returns a function that does the work.

```
(defun make-ends-with-fn (pattern)
  "Return a function which returns true if a string
ends with PATTERN."
  (declare (type string pattern))
  #'(lambda (str)
      ;; Returns true if & only if STR ends with
      ;; PATTERN
      (declare (type string str))
      (and (>= (length str) (length pattern))
           (equal (subseq str (- (length str) (length pattern)))
                  pattern))))
```

Anyway, enough of that. I just wanted to show that if you don't have regular expressions, you can encapsulate the pattern check into a function so that you needn't see the non-regular-expression code.

4 Processing

I'll assume that, once our program finds a line that is worth of processing, it uses a function called `process` to, well, *process* the line. By the time we call `process`, we've split the line into fields, so the single argument to `process` is a non-empty list of fields, where each field is a string.

Here's a silly example of a process function:

```
(defvar *line-number* 0)
(defun process (lst)
  (declare (type list lst))
  (format t "~&~D: ~S" (incf *line-number*) lst))
```

The point is that I'll assume there is a `process` function, & it can do whatever it needs to do.

5 First Loop

As one who generally pefers `do` to `loop`, here is my first idea of a Perlish loop for Lisp.

```
(do* ((line (chop (next-line)) (chop (next-line)))
      (lst (and line (split line)) (and line (split line))))
     ((null line))
     (when (match lst)
       (process lst)))
```

The body of this `do*` loop isn't bad, but the two initialization forms are an eye-sore. We could remove the `and`s with a function, but we'd still call it twice, & we'd call the `next-line` function twice. That would still be icky.

6 Second Loop

Maybe this is a case for the `loop` übermacro.

```
;; one of the few times when LOOP is a real improvement
;; over DO
(loop for line = (chop (next-line))
      while line
      do (when (match (split line #\:))
           (process (split line #\:))))
```

It's still not as terse as the classic main loop in Perl, but it's not bad.

Notice a subtle difference in logic between this loop & a loop in Perl. In this loop, I put the `chop` around the `next-line` input function. You can't do that in Perl because an empty string in Perl is a kind of *false*. So in Perl, you must `chop` the line after checking for end-of-file.

7 Conclusion

I suspect that with a global "default variable", it would be possible to come close to the convenience of the classic main loop in Perl, but it would sacrifice some serious Lispiness. Some really clever macro magic might be a better approach. The second loop I've shown here is pretty close to the convenience of Perl.

A Other File Formats

- This document is available in multi-file HTML format at http://lisp-p.org/perlish/.
- This document is available in Pointless Document Format at http://lisp-p.org/perlish/perlish.pdf.

References

[1] Gene Michael Stover. *CyberTiggyr Tigris, a Library of Miscellaneous Functions.* personal web site, April 2003. http://cybertiggyr.com/gene/tigris/.

[2] ANSI Committee X3J13. Common lisp hyperspec. Xanalys Web site. http://www.lispworks.com/reference/HyperSpec/.