DSV Library for Lisp

Gene Michael Stover

created Sunday, 2005 June 19
updated Monday, 2005 July 11

Copyright copyright 2005 Gene Michael Stover. All rights reserved. Permission to copy, store, & view this document unmodified & in its entirety is granted.

Contents

1 What is this?

This is a description of a Lisp library for reading Delimeter Separated Values (DSV). The library is called CyberTiggyr DSV.

2 To Do

Document `do-escaped`. Actually, I wrote it in a rush, on a whim, so it'd be worth re-considering it. It does its job, but maybe there is a better way. Or maybe not. Whatever. After deciding on something, doc it.

3 What is DSV

DSV is Delimeter Separated Values. Comma Separated Values (CSV) is a kind of DSV. The unix `/etc/passwd` file is a DSV file.

DSV file formats are explained well in the "Data file Metaformats" chapter of *The Art of Unix Programming* by Eric S. Raymond. ([2])

CyberTiggyr DSV converts the records of the file into lists of strings in Lisp. An alternative would be to use a regular expression library & treat the records as lines of text. (And if doing that, Perl could be a better language choice than Lisp.)

4 Examples

A programming library's documentation should have an Examples section near the front so you can determine whether the library does what you want in a way you want without having to read an entire manual.

CyberTiggyr DSV can read unix-style DSV files that have an escape character. The LOAD-ESCAPED function returns the entire contents of such a file at once. The separator, escape character, & end-of-record character default to colon, backslash, & newline, respectively, so you could read a file such as `/etc/passwd` like this:

```
;; Requires CyberTiggyr Test
> (load "../lut/test.lisp")
T
> (load "dsv.lisp")
T
> (use-package "CYBERTIGGYR-DSV")
T
> (load-escaped "/etc/passwd")
(("root" "x" "0" "0" "root" "/root" "/bin/sh")
 ("uucp" "x" "10" "14" "uucp" "/var/spool/uucp" "/sbin/nologin")
 ("fido" "x" "501" "501" "fidonet national mail hour" "/home/fido"
  "/home/bin/fido"))
```

You can specify your own field separator character & end-of-record character. For example, at my dayjob just today (I swear), I had a file that separated fields with tabs & ended records with the underbar. Here's an example of that nonsense (using consecutive white space to simulate a tab):

```
Joe     123 Sesame St
Virginia, USA_Steve     345 Suite Street
DC, US A
phone 123-456-7890_
```

You can read a file like that by specifying the field separator & end-of-record characters for LOAD-ESCAPED, like this:

```
> (load-escaped "addresses.dsv"
                :field-separator #\Tab
                :end-of-record #\_)
(("Joe" "123 Sesame St
Virginia, USA")
 ("Steve" "345 Suite Street
DC, US A
phone 123-456-7890"))
```

You can change the default field separator, end-of-record, & escape characters so you don't need to specify them each time you call LOAD-ESCAPED.

If you have a stream, not a file, you can read a record at a time from it with READ-ESCAPED.

In the future, CyberTiggyr DSV will support quoted-style DSV files. That's what Microsloth XL uses when it writes CSV files.

5 License

CyberTiggyr DSV is released according to the *Gnu Lesser General Public License* ([1]).

6 Obtaining

You need just one file: dsv.lisp[1] .

The complete source code is also in Appendix A.

7 Reference

7.1 Package CyberTiggyr DSV

The Lisp package is called CYBERTIGGYR-DSV (all upcase).

It requires COMMON-LISP & CYBERTIGGYR-TEST. You can get CyberTiggyr Test from ../lut/[2] .

CyberTiggyr DSV exports these symbols:

[1] http://cybertiggyr.com/gene/dsv/dsv.lisp

[2] http://cybertiggyr.com/gene/lut/

- *END-OF-RECORD*
- *ESCAPE*
- *FIELD-SEPARATOR*
- LOAD-ESCAPED
- READ-ESCAPED

7.2 *end-of-record*

`defvar *end-of-record* #`
`Newline`

END-OF-RECORD must be bound to the character which ends a record. By default, it's a newline. When you do not specify an end-of-record character when you call READ-ESCAPED or LOAD-ESCAPED, the function you call will get its default end-of-record character from *END-OF-RECORD*.

7.3 *escape*

`defvar *escape* #`

ESCAPE is bound to the default escape character that READ-ESCAPED & LOAD-ESCAPED will use. By default, it's a backslash.

To disable escapes, bind NIL to *ESCAPE*. Since NIL is a symbol, not a character, no character will ever be EQL to it, so no character will ever be used as the escape character.

7.4 *field-separator*

`defvar *field-separator* #:`

FIELD-SEPARATOR is bound to the character which by default separates fields in a record. If you do not specify a field separator character when you call READ-ESCAPED & LOAD-ESCAPED, the function will use the character bound to *FIELD-SEPARATOR*. By default, it's a colon.

7.5 load-escaped

`defun load-escaped` *pathname* `&key (field-separator *field-separator*) (end-of-record *end-of-record*) (escape *escape*) (trace nil)`

LOAD-ESCAPED reads all the DSV records from the specified file & returns them in a list.

If you specify a stream for *trace*, LOAD-ESCAPED will print a progress messages as it goes. (It isn't pretty, so you probably don't want to use that feature when an end user will see the output.)

7.6 read-escaped

`defun read-escaped` *strm* `&key (field-separator *field-separator*) (end-of-record *end-of-record*) (escape *escape*)`

READ-ESCAPED consumes & returns the next record from the DSV stream. On end-of-input, returns *strm*.

strm must be a stream that supports READ-CHAR & PEEK-CHAR.

A The Source Code

```
;;;
;;; $Header: /home/gene/library/website/docsrc/dsv/RCS/dsv.tex,v 395.1 2008/04/20 17:25:46 gene Ex
;;;
;;; Copyright (c) 2005 Gene Michael Stover.  All rights reserved.
;;;
;;; This program is free software; you can redistribute it and/or modify
;;; it under the terms of the GNU Lesser General Public License as
;;; published by the Free Software Foundation; either version 2 of the
;;; License, or (at your option) any later version.
;;;
;;; This program is distributed in the hope that it will be useful,
;;; but WITHOUT ANY WARRANTY; without even the implied warranty of
;;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
;;; GNU Lesser General Public License for more details.
;;;
;;; You should have received a copy of the GNU Lesser General Public
;;; License along with this program; if not, write to the Free Software
;;; Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301
;;; USA
;;;

;;;
;;; Complete documentation is at http://cybertiggyr.com/gene/dsv/
;;;

(defpackage "CYBERTIGGYR-DSV"
  (:use "COMMON-LISP")
  (:import-from "CYBERTIGGYR-TEST" "DEFTEST")
  (:export "*END-OF-RECORD*"
   "*ESCAPE*"
   "*FIELD-SEPARATOR*"
   "LOAD-ESCAPED"
   "READ-ESCAPED"))

(in-package "CYBERTIGGYR-DSV")

;;;
;;; UNEXPORTED HELPER FUNCTIONS & STOOF
;;;
```

```
(defun xpeek (strm)
  "Return the next character without consuming it, or return STRM on
end-of-input or other error."
  (peek-char nil strm nil strm))

(defun consume-leading-crap (strm crap)
  "Read (consume) newlines until the next character is not a newline or
there is no next character (end-of-input, which isn't an error)."
  (loop while (eql (xpeek strm) crap) do (read-char strm))
  'consume-leading-crap)

(defun read-escaped-field (strm terminators escape)
  "Return the next field as a string.  Return STRM if there is no next
field, which is when the stream is already at its end.  Assumes caller
has already consumed white-space crap that might preceed the field.
Consumes the character which ends the field.  TERMINATORS is a list
of characters & the stream which could terminate the field."
  (if (eq (xpeek strm) strm)
      strm                              ; already at end-of-input
    ;; else, Consume & collect characters until we find a terminator (field
    ;; terminator, record terminator, or end-of-input).  Do not collect
    ;; the terminator.
    (coerce
     (loop until (member (xpeek strm) terminators)
   collect (if (eql (xpeek strm) escape)
       ;; It's an escape, so discard it & use the next
       ;; character, verbatim.
       (progn (read-char strm) (read-char strm))
     ;; else, Use this character.
     (read-char strm)))
     'string)))

;;;
;;;
;;;

(defvar *field-separator* #\:
  "The default field separator character.  It defaults to colon (:).")

(defvar *end-of-record* #\Newline
  "The end-of-record character.  Defaults to Newline.")

(defvar *escape* #\\
  "The default escape character for unix-style DSV files.  It uses a single
escape character to allow the field separator character to occur
within fields.  The escape character can be used to allow an end-of-line
character or an escape character to occur in fields, too.
Defaults to backslash (\\).  You can change it with SETQ.  If you do not
want to allow separator characters at all, bind it to NIL.")
```

```
(defun read-escaped (strm &key (field-separator *field-separator*)
  (end-of-record *end-of-record*)
  (escape *escape*))
  "Read (consume) & return the next DSV record from STRM.  The record
will be a list of fields.  The fields will be strings.  Field separator
& end-of-record characters may not occur within fields unless escaped.
If you don't want to allow any kind of escape, use NIL for the escape
character.  Since NIL is not a character, it will never be equal to a
character read from STRM, so there will be no possible escape character.
In fact, you could use any non-character to disable the escape
character.  Ignors empty lines.  On end-of-input, returns STRM.  It is
an error if an escape character is followed by end-of-input."
  (consume-leading-crap strm end-of-record)
  (if (eq (xpeek strm) strm)
      strm ; normal end-of-input
    ;; else, Let's collect fields until we have read an entire record.
    (prog1
(loop until (member (xpeek strm) (list strm end-of-record))
      collect (prog1
  (read-escaped-field strm
      (list strm field-separator
    end-of-record)
      escape)
(when (eql (xpeek strm) field-separator)
  ;; Consume the character which ended the field.
  ;; Notice that we do not consume end-of-record
  ;; characters.
  (read-char strm))))
      (consume-leading-crap strm end-of-record))))

(defun load-escaped (pathname &key (field-separator *field-separator*)
      (end-of-record *end-of-record*)
      (escape *escape*))
  "Return the entire contents of an escaped DSV file as a list of
records.  Each record is a list."
  (with-open-file (strm pathname :direction :input)
    (loop for x = (read-escaped strm :field-separator field-separator
:end-of-record end-of-record
:escape escape)
          while (not (eq x strm))
  collect x)))

;;;
;;; TESTS
;;;

(deftest test0000 ()
  "Null test.  Always succeeds."
  'test0000)
```

```
(deftest test0010 ()
  "Test that XPEEK returns the correct character from a stream, does
not consume the character.  The character is NOT the last in the stream."
  (with-input-from-string (strm "abc")
    (and (eql (xpeek strm) #\a)
 (eql (read-char strm) #\a))))

(deftest test0011 ()
  "Like TEST0011 except that it tests XPEEK on the last character in the
stream.  In other words, tests that XPEEK returns the correct value &
does not consume it, & that character is the last in the stream."
  (with-input-from-string (strm "c")
    (and (eql (xpeek strm) #\c)
 (eql (read-char strm) #\c))))

(deftest test0012 ()
  "Test XPEEK on an empty stream."
  (with-input-from-string (strm "")
    (and (eq (xpeek strm) strm)
 (eq (read-char strm nil strm) strm))))

(deftest test0015 ()
  "Test CONSUME-LEADING-CRAP on a stream that contains nothing but leading
crap."
  (with-input-from-string (strm (format nil "~%~%~%"))
    (and (eql (xpeek strm) #\Newline)   ; not at end
 (consume-leading-crap strm #\Newline) ; doesn't matter what it returns
 (eq (read-char strm nil strm) strm)))) ; now we're at end

(deftest test0016 ()
  "Test CONSUME-LEADING-CRAP on a streeam that starts with leading crap,
then has some non-crap."
  (with-input-from-string (strm (format nil "~%~%~%a"))
    (and (eql (xpeek strm) #\Newline)   ; not at end
 (consume-leading-crap strm #\Newline)
 (eql (read-char strm) #\a))))

(deftest test0017 ()
  "Test CONSUME-LEADING-CRAP on a stream that starts with non-crap, then
has some crap.  CONSUME-LEADING-CRAP should not consume the leading
non-crap."
  (with-input-from-string (strm (format nil "a~%"))
    (and (eql (xpeek strm) #\a)         ; not at end
 (consume-leading-crap strm #\Newline)
 (eql (read-char strm) #\a))))  ; the "a" char should remain

(deftest test0020 ()
  "Test READ-ESCAPED-FIELD on a stream that contains a single field
followed by end-of-input.  Uses the default field separator, end-of-record
```

```
character, & escape character.  Just test that the field is read, not that
the next READ-ESCAPED-FIELD indicates end-of-input."
  (with-input-from-string (strm "abc")
    (equal (read-escaped-field strm
       (list strm *field-separator* *end-of-record*)
       *escape*)
   "abc")))

(deftest test0021 ()
  "Like TEST0020, but also checks that another call to READ-ESCAPED-FIELD
indicates end-of-input by returning STRM."
  (with-input-from-string (strm "abc")
    (let* ((a (read-escaped-field strm
  (list strm *field-separator* *end-of-record*)
  *escape*))
   (b (read-escaped-field strm
  (list strm *field-separator* *end-of-record*)
  *escape*)))
      (unless (equal a "abc")
(format t "~&~A: First read should have returned" 'test0021)
(format t " ~S, but it returned ~S" "abc" a))
      (unless (eq b strm)
(format t "~&~A: Second read should have returned" 'test0021)
(format t " ~S, but it returned ~S" strm b))
      (and (equal a "abc") (eq b strm)))))

(deftest test0025 ()
  "Test that READ-ESCAPED-FIELD works on two consecutive fields."
  (let ((a "abc") (b "xyz"))
    (with-input-from-string (strm (format nil "~A~A~A" a *field-separator* b))
      (let* ((terminators (list strm *field-separator* *end-of-record*))
     (xa (read-escaped-field strm terminators *escape*))
     (xseparator (read-char strm))
     (xb (read-escaped-field strm terminators *escape*))
     (xstrm (xpeek strm)))
(and (equal xa a) (eql xseparator *field-separator*) (equal xb b)
     (eq xstrm strm))))))

(deftest test0026 ()
  "Test that READ-ESCAPED-FIELD works on two records of two fields each.
The second record does not end with an end-of-record character.  It
ends with end-of-input on the stream."
  (let* ((a "abc") (b "123")            ; first record
 (c "def") (d "456")            ; second record
 (string (format nil "~A~A~A~A~A~A~A" a *field-separator* b
 *end-of-record* c *field-separator* d)))
    (with-input-from-string (strm string)
      (let* ((terminators (list strm *field-separator* *end-of-record*))
     (xa (read-escaped-field strm terminators *escape*))
     (xseparator0 (read-char strm))
```

```
     (xb (read-escaped-field strm terminators *escape*))
     (xend-of-record0 (read-char strm))
     (xc (read-escaped-field strm terminators *escape*))
     (xseparator1 (read-char strm))
     (xd (read-escaped-field strm terminators *escape*))
     (xstrm (xpeek strm)))
(and (equal xa a)
     (eql xseparator0 *field-separator*)
     (equal xb b)
     (eql xend-of-record0 *end-of-record*)
     (equal xc c)
     (eql xseparator1 *field-separator*)
     (equal xd d)
     (eq xstrm strm))))))

(deftest test0027 ()
  "Like TEST0026 except that the second record ends with an end-of-
record character."
  (let* ((a "abc") (b "123")            ; first record
 (c "def") (d "456")            ; second record
 (string (format nil "~A~A~A~A~A~A~A~A" a *field-separator* b
 *end-of-record* c *field-separator* d
 *end-of-record*)))
    (with-input-from-string (strm string)
      (let* ((terminators (list strm *field-separator* *end-of-record*))
     (xa (read-escaped-field strm terminators *escape*))
     (xseparator0 (read-char strm))
     (xb (read-escaped-field strm terminators *escape*))
     (xend-of-record0 (read-char strm))
     (xc (read-escaped-field strm terminators *escape*))
     (xseparator1 (read-char strm))
     (xd (read-escaped-field strm terminators *escape*))
     (xend-of-record1 (read-char strm))
     (xstrm (xpeek strm)))
(and (equal xa a)
     (eql xseparator0 *field-separator*)
     (equal xb b)
     (eql xend-of-record0 *end-of-record*)
     (equal xc c)
     (eql xseparator1 *field-separator*)
     (equal xd d)
     (eql xend-of-record1 *end-of-record*)
     (eq xstrm strm))))))

(deftest test0050 ()
  "Test READ-ESCAPED on an input stream containing a single record of a
single field."
  (let* ((record (list "abc"))
 (string (format nil "~A" (first record))))
    (with-input-from-string (strm string)
```

```
      (let* ((xrecord (read-escaped strm))
     (xstrm (xpeek strm)))
(and (equal xrecord record)
     (eq xstrm strm))))))

(deftest test0051 ()
  "Test READ-ESCAPED on an input stream containing a single record of two
fields."
  (let* ((record (list "abc" "123"))
 (string (format nil "~A~A~A" (first record) *field-separator*
 (second record))))
    (with-input-from-string (strm string)
      (let* ((xrecord (read-escaped strm))
     (xstrm (xpeek strm)))
(and (equal xrecord record)
     (eq xstrm strm))))))

(deftest test0052 ()
  "Test READ-ESCAPED.  After reading the single record of two fields,
the stream should be at its end.  The record is followed by several
end-of-record characters, & the stream should be at its end after
reading the record because no records follow the record terminators."
  (let* ((record (list "abc" "123"))
 (string (format nil "~A~A~A~A~A~A" (first record) *field-separator*
 (second record) *end-of-record* *end-of-record*
 *end-of-record*)))
    (with-input-from-string (strm string)
      (let* ((xrecord (read-escaped strm))
     (xstrm (xpeek strm)))
(and (equal xrecord record)
     (eq xstrm strm))))))

(deftest test0053 ()
  "Test READ-ESCAPED on an input of two, two-field records.  The second
record is followed by one end-of-record character."
  (let ((record0 '("aaa" "111"))
(record1 '("bbb" "222"))
(string (format nil "aaa~A111~Abbb~A222~A"
*field-separator* *end-of-record*
*field-separator* *end-of-record*)))
    (with-input-from-string (strm string)
      (let* ((xrecord0 (read-escaped strm))
     (xrecord1 (read-escaped strm)))
(unless (equal xrecord0 record0)
  (format t "~&First record is ~S.  Expected ~S." xrecord0 record0))
(unless (equal xrecord1 record1)
  (format t "~&Second record is ~S.  Expected~S." xrecord1 record1))
(and (equal xrecord0 record0)
     (equal xrecord1 record1))))))
```

```
;;; --- end of file ---
```

References

[1] GNU. *Gnu Lesser General Public License*, 2007. —http://www.gnu.org/copyleft/lgpl—.

[2] Eric S. Raymond. *The Art of Unix Programmer*. Addison-Wesley, 2003. `http://www.faqs.org/docs/artu/`.