read-csv

Warren Wilkinson

January 8, 2013

Contents

Overview 1

Features 2

Installation 2

Quick Lisp 2

Gentoo 2

Ubunto 2

Manual Installation 3

Running the Tests 3

Getting Support 3

Implementation 4

Collecting the Input 4

Traversing the Input 5

Parsing whole files: parse-csv 5

Test Framework 6

Tests Expressions 11

Blanks 11

Quotes and Tricky Characters 11

International Text 11

License 11

Overview

read-csv is a stream oriented CSV (comma-separated value) reader that supports excel .csv files.

```
  (with-open-file (s "/path/to/csv")
    (parse-csv s))
;; Returns a list of lists of strings.
```

Features

- Low line of code count (around 50 lines of code)
- Supports quotes, including newlines and separator characters within the quotes.
- Supports Unix style LF line endings and Dos CRLF line endings. (automatically)

Installation

Quick Lisp

Install Quick Lisp and then run:

```
(ql:quickload 'read-csv)
```

If you have problems, see the support section, and you may want to run the tests.

Gentoo

As root,

```
emerge read-csv
```

Once the emerge is finished, the package can be loaded using ASDF:

```
(asdf:operate 'asdf:load-op :read-csv)
```

If you have problems, see the support section, otherwise you may want to run the tests.

Ubunto

```
sudo apt-get install read-csv
```

Once the installation is finished, the package is loadable using ASDF:

```
(asdf:operate 'asdf:load-op :read-csv)
```

If you have problems, see the support section, otherwise you may want to run the tests.

Manual Installation

In summary: Untar the .tar package and then symlink the .asd files into a place where ASDF can find them.

1. Untar the files where you want them to be. On windows download the .zip and unzip it instead, it's the same files.
2. ASDF could be looking anywhere – it depends on your setup. Run this in your lisp repl to get a clue as to where ASDF is seeking libraries[1]:

   ```
   (mapcan #'funcall asdf:*default-source-registries*)
   ```

3. Symlink the .asd files to the source directory. If you use windows, these instructions on symlink alternatives apply to you.

[1] you might need to (require 'asdf) before running this example

Once the files are in place, the package can be loaded with ASDF by:

```
(asdf:operate 'asdf:load-op :read-csv)
```

If you have problems, see the support section. If you don't have problems you may want to run the tests anyway, because you can.

Running the Tests

Once the system is loaded, it can be tested with asdf.

```
(asdf:operate 'asdf:test-op :read-csv)
```

This should display something like the following. There should be **zero failures**, if you have failures see the support section of this document.

```
RUNNING READ-CSV TESTS...
READ-CSV TEST RESULTS:
     Tests: 519
   Success: 519
  Failures: 0
```

Getting Support

You can find support on this libraries website and/or github repository. Or you can email Warren Wilkinson.

Implementation

Most Lisp CSV readers hover around 400 lines of code. This one is 68.

The difference is this one is coded as state machine in a dense 2D grid, as described in FINITE State Machines in Forth. "If you are in this *state* and you see this *character* then you . . . "

state	white	return	linefeed	quote	separator	other
start	noop ->start	ship ->return	ship ->done!	noop ->quote	next ->start	addc ->unquote
return	noop ->start	ship ->return	noop ->done!	noop ->start	next ->start	addc ->unquote
unquote	addc ->unquote	ship ->return	ship ->done!	addc ->unquote	next ->start	addc ->unquote
quote	addc ->quote	noop ->q+ret	addl ->quote	noop ->q+quote	addc ->quote	addc ->quote
q+ret	addc ->quote	noop ->q+ret	addl ->quote	noop ->q+quote	addc ->quote	addc ->quote
q+quote	noop ->q+q&w	ship ->return	ship ->done!	addc ->quote	next ->start	addc ->unquote
q+q&w	noop ->q+q&w	ship ->return	ship ->done!	addc ->quote	next ->start	addc ->unquote

1. perform the designated function, and 2) transition to the designated new state.

For example, if we start *(e.g. state start)* and spot quote, then we perform noop and change to quote state. Then, in the quote state, if we spot '/A/', perform 'addc' (add character) and remain in the quote state.

Collecting the Input

The functions used by the above table are:

noop Do no action

addc Add a character to the current CSV record.

addl Add a newline character to the current CSV record.

next Finish the current CSV record and start the next.

ship Cleanup the current set of CSV records for returning them to the end user.

Internally, these methods play with three dynamic variables: × **record** and **records** and **white-char-count**. The first, during run time, holds a list of characters – in reverse order. The second holds the list of parsed records – also in reverse order (but each record in proper order). The **next** method reverses the order and coerces the csv data to a string. The **ship** method reverses the **records** list so it's in proper order.

The last variable, white-char-count* keeps a count of how many characters we've seen since after the quote. It's used to let us to remove whitespace characters after the closing quote without removing whitespace characters within the quotes.

```
(defun noop (c) (declare (ignore c)))
(defun addc (c) (push c *record*))
(defun addl (c) (declare (ignore c)) (push #\Newline *record*))
(flet ((white (c) (or (char= c #\Space) (char= c #\Tab))))
  (defun next (c)
    (declare (ignore c))
    (let ((eow (or (position-if-not #'white *record*) (length *record*))))
      (push (coerce (nreverse (nthcdr (max 0 (min eow (1- *white-char-count*))) *record*)) 'string) *recor
    (setf *record* nil)))
(defun ship (c) (next c) (setf *records* (nreverse *records*)))
```

Traversing the Input

We read CSV by running our state machine until the *done* state is reached.

```
(defun char-class (sep char)
  (case char (#\Space 0) (#\Return 1) (#\Linefeed 2) (#\" 3) (otherwise (if (char= sep char) 4 5))))

(defun read-csv (stream &optional (sep #\,) (eof-error-p t) eof-value)
  "Return CSV data and a second value that's true if row ended by EOF."
  (let ((*records* nil)
        (*record* nil)
        (*white-char-count* 0))
    (declare (special *record* *records* *white-char-count*))
    (loop with state = start
          for char = (read-char stream (and (null *records*) eof-error-p) :eof)
          when (eq char :eof)
          do (return-from read-csv (values (if *records* (ship :eof) eof-value) t))
          do (incf *white-char-count*)
          do (let ((class (char-class sep char)))
               (when (= class quote) (setf *white-char-count* 0))
               (funcall (aref +csv-table+ state class 0) char)
               (setf state (aref +csv-table+ state class 1)))
       until (eq state done!))
    (values *records* (eq :eof (peek-char nil stream nil :eof)))))
```

Parsing whole files: parse-csv

To parse a whole file, the utility parse-csv calls read-csv until the end-of-file.

```
(defun parse-csv (stream  &optional (sep #\,))
  (loop for (line end-p) = (multiple-value-list (read-csv stream sep nil :eof))
        unless (eq line :eof) collect line
        until end-p))
```

Test Framework

The test framework combines jumbles of CSV statements and calls read-csv on them. In practice it means I take (upto) 5 patterns, sequentially, from the a predefined list of hard patterns. Then I take all permutations (120, if 5 patterns) and combine them as 1 record per row (and 5 rows), 2 per row, 3 per row, 4 per row and 5 per row. Then I test that I can parse it back correctly.

The jumbler is shown below, but I also test parse-csv by parsing larger examples but that is not shown here.

```
(defun concat-with (strings item)
  (if (null strings)
      ""
      (apply #'concatenate 'string (first strings) (mapcan #'(lambda (a) (list item a)) (rest strings)))))

(defun build-answers (i strings)
  (loop while strings
        collect (loop for n upto (1- i)
                      while strings
                      collect (car strings)
                      do (setf strings  (cdr strings)))))

(defun build-string (i strings)
  (concatenate 'string
     (concat-with (mapcar #'(lambda (s) (concat-with s ",")) (build-answers i strings)) (list #\Newline))
     '(#\Newline)))

(defun all-combinations (patterns)
   (if (null (cdr patterns))
       (list patterns)
       (loop for i in patterns
             nconc (mapcar #'(lambda (p) (cons i p)) (all-combinations (remove i patterns))))))

(defun make-test (description)
  #'(lambda ()
      (block test
        ;(loop for pattern in (all-combinations description)
        (format t "~%Pattern: ~s" (substitute #\' #\" (remove #\Newline (build-string (length description)
```

```
        (dotimes (i (length description) t)
          (format t "~%  @~d" i)
          (let ((string (build-string (1+ i) (mapcar #'car description)))
                (answers (build-answers (1+ i) (mapcar #'cdr description))))
            (with-input-from-string (s string)
              (loop for answer in answers
                    for (got end) = (multiple-value-list (read-csv s))
                    unless (equalp answer got)
                    do (format t "~%Expected ~a, got ~a" answer got)
                    and do (return-from test nil)
                    if (eq answer (car (last answers)))
                    unless end
                    do (format t "~%Expected EOF, but didn't see it!")
                    unless (not end)
                    do (format t "~%Did not expect EOF, but saw it!"))
              (let ((read-more (read-csv s #\, nil :eof)))
              (unless (eq read-more :eof)
                (format t "~%Could read past end: ~s" read-more)
                (return-from test nil)))))))))
```

Parse-csv tests

These tests were yanked from CL-CSV, another Lisp CSV parser.

```
(defmacro deftest (name code result)
  `(defun ,name ()
     (format t "~%~a" ',name)
     (let ((expect ,result)
           (got ,code))
       (if (equalp expect got)
           t
           (progn (format t "~%Expected~% ~s~%but got~%~s" expect got)
                  nil)))))

(defvar *a-tough-example-answer* '(("very tough" "easier to do")))
(defun a-tough-example ()
  (with-input-from-string (s "  \"very tough\"   ,    easier to do
")
    (parse-csv s)))

(deftest test-tough (a-tough-example) *a-tough-example-answer*)

(defvar *big-example-answer*
  '(("first name" "last name"   "job \"title\""                      "number of hours" "id")
    ("Russ"       "Tyndall"     "Software Developer's, \"Position\"" "26.2"            "1")
    ("Adam"       "Smith"       "Economist"                          "37.5"            "2")
```

```
    ("John"       "Doe"         "Anonymous Human"                    "42.1"           "3")
    ("Chuck"      "Darwin"      "Natural Philosopher"                "17.68"          "4")
    ("Bill"       "Shakespeare" "Bard"                               "12.2"           "5")
    ("James"      "Kirk"        "Starship Captain"                   "13.1"           "6")
    ("Bob"        "Anon"        ""                                   "13.1"           "6")
    ("Mr"         "Iñtërnâtiônàlizætiøn" ""                          "1.1"            "0")))

(defun big-example ()
  (with-input-from-string (s "first name,last name,\"job \"\"title\"\"\",number of hours,id
Russ,Tyndall,\"Software Developer's, \"\"Position\"\"\",26.2,1
Adam,Smith,Economist,37.5,2
John,Doe,Anonymous Human,42.1,3
Chuck,Darwin,Natural Philosopher,17.68,4
Bill,Shakespeare,Bard,12.2,5
James,Kirk,Starship Captain,13.1,6
Bob,Anon,,13.1,6
Mr,Iñtërnâtiônàlizætiøn,,1.1,0")
    (parse-csv s)))


(defun quoted-big-example ()
  (with-input-from-string
      (s "\"first name\",\"last name\",\"job \"\"title\"\"\",\"number of hours\",\"id\"
\"Russ\",\"Tyndall\",\"Software Developer's, \"\"Position\"\"\",\"26.2\",\"1\"
\"Adam\",\"Smith\",\"Economist\",\"37.5\",\"2\"
\"John\",\"Doe\",\"Anonymous Human\",\"42.1\",\"3\"
\"Chuck\",\"Darwin\",\"Natural Philosopher\",\"17.68\",\"4\"
\"Bill\",\"Shakespeare\",\"Bard\",\"12.2\",\"5\"
\"James\",\"Kirk\",\"Starship Captain\",\"13.1\",\"6\"
\"Bob\",\"Anon\",\"\",\"13.1\",\"6\"
\"Mr\",\"Iñtërnâtiônàlizætiøn\",\"\",\"1.1\",\"0\"")
    (parse-csv s)))

(deftest test-big        (big-example)        *big-example-answer*)
(deftest test-quoted-big (quoted-big-example) *big-example-answer*)

(defvar *multiline-answer*
  '(("this" "is" "a" "test
of
multiline" "data")
    ("row2" "of" "the" "test
of
multiline" "data")))
```

```
(defun multiline-unix-example ()
  (with-input-from-string (s "this,is,a,\"test
of
multiline\", data
row2,of,the,\"test
of
multiline\", data")
    (parse-csv s)))

(defun multiline-dos-example ()
  (with-input-from-string
      (s (concatenate 'string "this,is,a,\"test" (list #\Return #\Linefeed)
                      "of" (list #\Return #\Linefeed)
                      "multiline\", data" (list #\Return #\Linefeed)
                      "row2,of,the,\"test" (list #\Return #\Linefeed)
                      "of" (list #\Return #\Linefeed)
                      "multiline\", data" (list #\Return #\Linefeed)))
    (parse-csv s)))

(defun multiline-mixed-example ()
  (with-input-from-string
      (s (concatenate 'string "this,is,a,\"test" (list #\Linefeed)
                      "of" (list #\Return #\Linefeed)
                      "multiline\", data" (list #\Return #\Linefeed)
                      "row2,of,the,\"test" (list #\Linefeed)
                      "of" (list #\Return #\Linefeed)
                      "multiline\", data" (list #\Linefeed)))
    (parse-csv s)))

(deftest test-multiline-unix  (multiline-unix-example)  *multiline-answer*)
(deftest test-multiline-dos   (multiline-dos-example)   *multiline-answer*)
(deftest test-multiline-mixed (multiline-mixed-example) *multiline-answer*)
```

Running Tests

```
(defstruct results
  (tests 0)
  (failures nil))
(defun results-failure-count (results)
  (length (results-failures results)))
(defun results-successes (results)
  (- (results-tests results)
     (results-failure-count results)))

(defun runtest (fun results)
```

```
  (let* ((success t)
         (output (with-output-to-string (*standard-output*)
                   (setf success (handler-case (funcall fun)
                                   (error (e) (princ e) nil))))))
    (make-results
     :tests (1+ (results-tests results))
     :failures (if success
                   (results-failures results)
                   (acons fun output (results-failures results))))))

(defun present-failures (results)
  (format t "~%READ-CSV FAILURES:~%")
  (loop for (fn . problems) in (results-failures results)
        do (format t "~%~a~a~%" fn problems)))
(defun present-results (results)
  (format t "~%READ-CSV TEST RESULTS:")
  (format t "~%     Tests: ~a~%   Success: ~a~%  Failures: ~a"
          (results-tests results)
          (results-successes results)
          (results-failure-count results))
  (when (results-failures results)
    (present-failures results)))

(defun run-combination-tests (starting-results)
  (reduce #'(lambda (description results) (runtest (make-test description) results))
          (nreverse (mapcan #'(lambda (thing) (all-combinations (subseq thing 0 (min (length thing) 5))))
                    (loop for i on (reverse *all-statements*) collecting i)))
          :from-end t
          :initial-value starting-results))

(defun run-explicit-tests (starting-results)
  (reduce #'(lambda (results function) (runtest function results))
          (list #'test-tough
                #'test-big
                #'test-quoted-big
                #'test-multiline-unix
                #'test-multiline-dos
                #'test-multiline-mixed)
          :initial-value starting-results))

(defun run-tests ()
  (format t "~%RUNNING READ-CSV TESTS...")
  (present-results (run-explicit-tests (run-combination-tests (make-results)))))
```

The bulk of the test code just has to do with collecting results and making pretty output.

Tests Expressions

This package is tested by combining tricky CSV parts in numerous ways, and then ensuring the parser can parse them correctly.

Blanks

CSV	Should Parse	Note
><	><	blank input
> ̂<	><	whitespace input, **should be ignored.**
>""<	><	blank input, but quoted, should be empty
>" "<	> ̂<	blank input, but quoted, **should keep the whitespace.**

Quotes and Tricky Characters

CSV	Should Parse	Note
>"multi"<	>multi<	Multiline input should work
>",,"<	>,<	Should be able to have seperator characters
>""""<	>"<	Double quotes should become a single quote.

International Text

CSV	Should Parse	Note
>"êve,yng""once"<	>êve,yng"once<	Everything at once

License

Read-csv is distributed under the LGPL2 License.