Text File Databases

Many practical application programs read plain-ascii text files consisting of records separated by newlines, each record consisting of multiple fields. Sometimes each record is processed individually, so that the output is a one-to-one mapping of the input; in other cases, the input is summarized. This note describes functions that standardize handling of input records in several formats, guide the processing of those records, and assist in output.

1. Input file readers

Although input files come in a variety of formats, a large number of those formats can be processed with just a few general-purpose readers. Each call to a reader fetches the next record from the input; as a side effect of the fetch, the port is repositioned to the next record. All the readers take an optional port as the final argument, which defaults to the current input port if no port is specified. The return value from the reader is a list of strings, one string per field; empty strings indicate null fields, and an empty record returns `'()`. Explicit white space in the input record is preserved. The reader returns the `eof-object` when the port is exhausted. Defaults are used if any arguments are invalid.

Because different systems have different conventions for the end-of-line marker, all the readers accept a variety of end-of-line markers, so that they work properly regardless of where the input file was created; the end-of-line marker may be a carriage return (`#\return`), a line feed (`#\newline`), or both characters in either order. All the readers also permit the end-of-record marker (a single end-of-line marker for most of the readers, or two successive end-of-line markers for the case where records are written one-field-per-line) to be either a separator or a terminator; in particular, the last record in a file may, but need not, end with an end-of-record marker. On operating systems where a `CTRL-Z` character (ascii `\032`) indicates end-of-file, any trailing input is truncated.

1.a. Fixed-length data fields

One common input file format has records with fields in fixed positions. These records can be read with `(read-fixed-record size defs . port)`, where `size` is the length of the record, including any line-terminator characters, and `defs` is a list of two-element lists with the field's starting position (inclusive) in the first element and the field's ending position (exclusive) in the second element, counting from zero as in `substring`. Since the format is fixed-length, each of the strings will always be the same length for each record, including leading or trailing white space.

Sometimes, a file containing fixed-length records has a fixed-length header. In that case, a header of length `n` can be discarded by calling `(read-chars n . port)`.

1.b. Character-delimited fields

Another common input file format has lines containing records with variable-length fields separated by a single-character delimiter. These records can be read with the `read-delim-record` function, which takes two optional arguments. If the first argument is a character, it is taken as the field separator. Fields consist of maximal sequences of non-delimiter characters, and there is no escape; each instance of the delimiter indicates a new field. An example of a character-delimited file is the unix password file, where colons separate fields.

 The author of this article may be reached at pbewig@gmail.com. Bugfix due to Brad Lucier on September 29, 2008: fixed incorrect `#\newline` and `#\return` in `read-csv-record`.

In the special case where the delimiter is a newline or carriage return, fields are taken to be the lines of a file, and records are separated by blank lines (two successive end-of-line markers).

In the special case where no delimiter is specified, all white space (spaces and horizontal tabs, at least, and possibly other characters depending on the implementation and character set) is treated as a field separator, and multiple instances of white space characters are treated as a single field separator. Thus, a record that consists of the string `ABC`, followed by five spaces, followed by the string `DEF`, followed by an end-of-line marker, would be treated as a record containing two fields, the strings `"ABC"` and `"DEF"`, neither field containing white space. A record with leading or trailing white space implies null fields at the beginning or end of the record, respectively; this means that a record consisting solely of white space is taken as containing two null fields. The no-delimiter case is different than the case where the delimiter is a space character, in which each space marks a new field and multiple successive spaces indicate successive null fields.

1.c. Comma-separated values

A particular type of variable-length delimited input file is known as comma-separated values, or `csv`, which originated in database and spreadsheet applications. Unfortunately, there is no standard definition of the comma-separated values format, and there are several kinky character sequences that admit multiple reasonable interpretations. The basic schema of the `csv` format provides records terminated by end-of-line markers with fields separated by commas; in European countries where the comma, rather than the period, is used as the decimal point, fields are separated by semicolons. To allow the field-separator character to appear in the data, a field may be quoted by surrounding it with double-quote marks, so that a field-separator character appearing within a quoted field loses its meaning as a field separator and becomes a regular character; a literal double-quote character may appear within a quoted field by doubling it, so that two double-quote characters appear in succession. Leading and trailing white space within fields is preserved. `Read-csv-record` permits end-of-line markers to be embedded within quoted fields; this convention differs from those `csv` parsers that treat quoted end-of-line markers as the end of the record, thus preventing a missing quote from sucking up all remaining input.

The `read-csv-record` function takes two optional arguments. If the first argument is a character, it is taken as the field separator; if no field separator is given, it defaults to a comma.

`Read-csv-record` is implemented using a state machine consisting of mutually-recursive functions. Each function calls another function in tail position, so the recursive calls consume no stack space.

1.d. Name-value data

Another frequently-used type of text-file database has self-identifying fields, where records consist of multiple fields, one field per line, separated by blank lines (two successive end-of-line markers) and each field consists of a name and a value separated by a delimiter. This format is often used for databases that have many optional fields, such as bibliographic databases where books have publishers, journals have volume-and-issue references, and there may be multiple authors. In some cases the fields may be sorted in alphabetic order by name, in order to create a canonical representation. In other cases, the order of the fields may be important; for instance, in a bibliographic data-

base, each author may be followed by an associated institution or email-address field.

The `read-name-value-record` function takes two optional arguments. If the first argument is a character, it is taken as the separator between name and value; otherwise, the first maximal run of white-space characters on the line is taken as the separator between name and value. Note that only the first separator on each line is special; subsequent separators are simply part of the value. The function returns the next input record on the port, or the `eof-object` when the port is exhausted. The record is returned as an association list with the name in the key field and the associated value in the value field of each association, with fields in the order they appeared in the input. Fields may be extracted from the record by the normal a-list functions `assoc`, `assq`, and `assv`.

2. The `filter-port` input combinator

Sometimes, input records must be filtered, so that only some of the records participate in the processing while others are excluded. Function `(filter-port reader pred?)` is a combinator that takes a reader and a predicate and returns a new reader that only passes those records for which the predicate is not `#f`; the predicate is a function that takes an input record and returns a boolean or other value acting as a boolean.

Speaking generally, input combinators are often useful when dealing with text-file databases, though they tend to be application-specific rather than general-case, as with the filter combinator. For instance, it is easy to write an input combinator that trims non-siginificant whitespace from a field, or an input combinator that maps all instances of a particular code in a particular field to a different value. The main value of the combinator is separation of function — it transforms the input data so the main program doesn't have to.

3. Processing the input file

Several higher-order functions encode familiar idioms that process the text file databases read by the functions given above. In all cases, if the reader function requires parameters, they must be curried into the definition of the function.

3.a. `For-each-port`

Procedure `(for-each-port reader proc . port)` performs a procedure for its side-effects, applying `proc` to each input record in turn until the port is exhausted. It returns nothing.

3.b. `Map-port`

Function `(map-port reader mapper . port)` creates a list from the records on a port, applying the `mapper` function to each input record in turn, continuing until the port is exhausted, and returning the mapped records in a list with one item per input record in the same order as the input.

3.c. `Fold-port`

Function `(fold-port reader folder base . port)` summarizes the records on a port to a single value. `Reader` specifies the function that reads input records, and `port` is the location of the input records. `Folder` is a function that takes a value and an input record and returns a new value. When the first record is read, `folder` is called as `(folder base record)`, where `record` is the first record, and returns a value. As each subsequent record is read, `folder` is called again, with the accumulating value from the prior record in place of the base value for the first record, as `(folder accum record)`. When the input is exhausted, `fold-port` returns the final value.

3.d. `Map-reduce-port`

Another way to summarize a text-file database uses the map-reduce paradigm introduced by Jeffrey Dean and Sanjay Ghemawat of Google in their paper *MapReduce: Simplified Data Processing on Large Clusters* presented at the Sixth Symposium on Operating System Design and Implementation in San Francisco in December 2004 (available at `labs.google.com/papers/mapreduce.html`). Users specify a read function that fetches records from an input port, a map function that extracts a key/value pair from the input record, and a reduce function that merges all intermediate values associated with the same intermediate key.

Function `(map-reduce-port reader mapper reducer lt? . port)` has type:

```
(port → α) ×                          reader
(α → (values β γ)) ×                  mapper
((β × γ × γ) → γ) ×                   reducer
((β × β) → boolean) ×                 lt?
port →                                input
    list (cons β γ)                   result
```

`(reader port)` is a function that takes a port and returns a list of strings which are the fields of the next input record; if the reader requires parameters, they must be curried into the reader. `(mapper item)` is a function that extracts the key and value from an input record; it takes a record and returns `(values key value)`. `(reducer key value1 value2)` is a function that combines two intermediate key/value pairs with identical keys into a single value; it takes a key, an existing value and a new value and returns a new value. `(lt? key1 key2)` is a function that takes two keys and returns `#t` if the first key is less than the second and `#f` otherwise. The return value of `map-reduce` is a list of pairs whose `car` is a key and whose `cdr` is a value, with keys ordered by the `lt?` predicate.

The `map-reduce-port` function reads a port, submitting each input record to the mapping function, then submitting the key/value pair returned by the mapping function to a dictionary that either inserts the key/value pair, if the key isn't in the dictionary, or uses the reducing function to combine the new value with the value currently associated with the key if the key already exists in the dictionary. Once the input port is exhausted, the key/value pairs are retrieved from the dictionary and inserted into the output in order.

The dictionary is implemented using red/black trees stolen from section 3.3 of Chris Okasaki's book *Purely Functional Data Structures*; those interested in the details of the trees' operation should refer to the book. Two functions are provided: `(insert tree key value)` and `(enlist tree base)`. The `insert` function recursively winds down the red/black tree until it finds the proper place for the key. If the key doesn't exist, a new node is added to the tree using the current key/value combination, and balancing rotations and re-colorings are performed as the stacked function calls unwind. However, if the key already exists, the reducing function is called, passing the key, the current value, and the new value as arguments, and the result replaces the value currently in the tree. The `enlist` function performs in-order traversal of the tree, building the list right-to-left so that it is properly ordered when the function completes.

4. Writing text-file database records

Writers are provided for all the same text-file database types for which readers are provided. In all cases, the output of a writer may be read by the corresponding reader. The first argument to each writer is a record represented as a list of fields of type string; some of the writers take additional argu-

ments, and all take an optional port as their last argument, which defaults to the current output port if no port is specified. All the writers are procedures and return no value. All the writers use `(newline)` to write the end-of-line marker, so the actual end-of-line marker may vary depending on the implementation (on some systems, `(newline)` may write more than one character), and all the writers write the end-of-line marker as a record terminator, not a record separator, so the last record in a file will always be followed by an end-of-line marker. Defaults are used if any arguments are invalid.

4.a. Fixed-length data fields

Text-file databases with fixed-length data fields are written by procedure `(write-fixed-record rec size defs . port)`, where `size` is the length of the output record (in characters) and `defs` defines the field positions in the same manner as read-fixed-record. The `defs` need not specify the contents of each character position within the output record; unspecified character positions are filled with blanks (`#\space`). All fields are left-justified within the allocated space.

4.b. Character-delimited fields

Text-file databases with character-delimited fields are written by procedure `write-delim-record`, which takes one required argument and two optional arguments. The first argument is the output record represented as a list of fields. If the second argument is a character, it is taken to be the delimiter, and is written to the output between fields; a missing delimiter is taken to be a single blank. `Write-delim-record` writes records with fields terminated by end-of-line markers and records separated by a blank line if the delimiter is a carriage-return or linefeed.

4.c. Comma-separated values

Text-file databases in comma-separated values format are written by procedure `write-csv-record`, which takes one required argument and two optional arguments. The first argument is the output record represented as a list of fields. If the second argument is a character, it is taken to be the delimiter between fields; if no delimiter is given, it defaults to a comma. `Write-csv-record` is conservative, quoting only those fields that contain an instance of the delimiter, double-quote, or end-of-line marker.

4.d. Name-value data

Text-file databases with name-value fields are written by `write-name-value-record`, which takes one required argument and two optional arguments. Unlike the other writers, the first argument is the output record represented as an association list of name/value pairs. If the second argument is a character, it is taken to be the delimiter between name and value; if no delimiter is given, it defaults to a single blank.

5. Utility functions

A variety of utility functions are provided, which may find use in some programs involving text-file databases.

5.1. `Read-chars`

Function `(read-chars n . port)` returns a string containing up to `n` characters from the named port or the current input port, or the `eof-object` at end of file. `Read-chars` is called by `read-fixed-record`. All strings returned by `read-char` will be of length `n` characters, except possibly the last string at the end of the file, which will be of less than length `n` if `n` characters do not remain on the file.

5.2. `Read-line`

Function `(read-line . port)` returns the next line from the named input port or the current input port, or the `eof-object` at the end of file. As with the other readers the end of a line is marked by a carriage-return, a line-feed, or the combination of both characters in either order, and the last line need not have an end-of-line marker. `Read-delim-record` calls `read-line` when the delimiter is the end-of-line marker.

5.3. `Quote-csv`

The csv writer takes care of quoting strings as they are written, but sometimes it is convenient for a programmer to quote strings directly. Function `(quote-csv delim str)` returns `str` with the appropriate quotations. `Quote-csv` is called by `write-csv-record`.

5.4. `String-trim`

When reading fixed-length text files, functions that trim leading and trailing spaces from strings may be useful. Functions `(string-trim-left str)`, `(string-trim-right str)`, and `(string-trim str)` are shown below:

```
(define (string-trim-left s)
  (let ((c (string-ref s 0)))
    (if (char-whitespace? c)
        (string-trim-left
          (substring s 1
            (string-length s)))
        s)))

(define (string-trim-right s)
  (let* ((len (- (string-length s) 1))
         (c (string-ref s len)))
    (if (char-whitespace? c)
        (string-trim-right
          (substring s 0 len))
        s)))

(define (string-trim s)
  (string-trim-left (string-trim-right s)))
```

5.5. `Lpad`

The `(lpad str size . pad)` function right-justifies `str` in a string of `size` characters by adding pad characters to the left of `str`; if `pad` is missing, it defaults to a blank.

```
(define (lpad str size . pad)
  (let ((p (if (null? pad)
               #\space (car pad)))
        (len (string-length str)))
    (if (< len size)
        (string-append
          (make-string (- size len) p) str)
        (substring str (- len size) len))))
```

`Lpad` is useful for right-justifying strings when writing fixed-format records, because `write-fixed-record` always left-justifies strings.

5.6. `A-cons` and `a-sort`

Since the name-value functions use association lists, it is convenient to have some functions that work on association lists. The `a-cons` function is useful when building an association list, inserting a new key/value pair at the head of an association list:

```
(define (a-cons key value a-list)
  (cons (cons key value) a-list))
```

When processing name-value files, it is sometimes useful to sort the fields by name, thus giving records a canonical representation. Function `a-sort` uses insertion sort to return a new association list with the car of each item ordered by the `lt?` predicate:

```
(define (a-sort lt? a-list)
  (define (foldl op base lst)
    (if (null? lst)
        base
        (foldl op (op base (car lst))
                  (cdr lst))))
  (define (insert lst x)
    (cond ((null? lst) (list x))
          ((lt? (car x) (caar lst))
            (cons x lst))
          (else (cons (car lst)
                      (insert (cdr lst) x)))))
  (foldl insert '() a-list))
```

6. Examples

Several examples show how to exploit the functions given above; all read the `emp.data` file shown below, with four space-separated fields representing employee name, hourly wage rate, hours worked, and department:

```
Beth  12.75  0 mfg
Dan    8.50 10 sales
Kathy 11.40 30 sales
Mark  12.75 40 mfg
Mary   7.50 20 mfg
Susie 10.30 25 acctg
```

Function `wages` calculates weekly wages :

```
(define (wages emp)
  (* (string->number (cadr emp))
     (string->number (caddr emp))))
```

The first example shows a simple calculation on each record in the file:

```
(with-input-from-file "emp.data"
  (lambda ()
    (map-port
      read-delim-record
      (lambda (emp)
        (list (car emp) (wages emp))))))
```

Output from this command is the weekly wages for each employee:

```
(("Beth" 0)
 ("Dan" 85)
 ("Kathy" 342)
 ("Mark" 510)
 ("Mary" 150)
 ("Susie" 257.5))
```

The same output can be produced using `fold-port`:

```
(reverse
  (with-input-from-file "emp.data"
    (lambda ()
      (fold-port
        read-delim-record
        (lambda (base emp)
          (cons
            (list (car emp) (wages emp))
            base))
        '()))))
```

The second example shows how records can be processed selectively:

```
(with-input-from-file "emp.data"
    (lambda ()
      (for-each-port
        (filter-port
          read-delim-record
          (lambda (emp)
            (string=? (caddr emp) "sales")))
        (lambda (emp)
          (display (car emp))
          (display #\tab)
          (display (wages emp))
          (newline)))))
```

Output from this command is the weekly wages for employees in the `sales` department:

```
Dan     85.00
Kathy   342.00
```

The third example shows the calculation of total weekly wages using the `fold-port` function:

```
(with-input-from-file "emp.data"
  (lambda ()
    (fold-port
      read-delim-record
      (lambda (base emp)
        (+ base (wages emp)))
      0)))
```

Output from this command is `1344.5`.

The fourth example shows how data can be summarized by department:

```
(with-input-from-file "emp.data"
  (lambda ()
    (map-reduce-port
      read-delim-record
      (lambda (emp)
        (values (caddr emp) (wages emp)))
      (lambda (k v1 v2) (+ v1 v2))
      string<?)))
```

Output from this command is total wages by department, sorted on department name:

```
(("acctg" . 257.5)
 ("mfg" . 660)
 ("sales" . 427))
```

The fifth example shows the use of `read-fixed-record` and produces the same output as the previous example. It assumes a modified `emp.data` with 22-byte fixed-length records terminated by `CR/LF` and a modified `wages` that calls `string-trim` on its arguments. Note that the parameters to the reader function are curried (for those with impaired fonts, a quasi-quote introduces the second argument to `apply`):

```
(with-input-from-file "emp.data"
  (lambda ()
    (map-reduce-port
      (lambda (port)
        (apply read-fixed-record `(22
           ((0 5) (6 11) (12 14) (15 20))
           ,port)))
      (lambda (emp)
        (values
          (string-trim (caddr emp))
          (wages emp)))
      (lambda (k v1 v2) (+ v1 v2))
      string<?)))
```

It is somewhat harder to process name-value records than the other record types, because fields are referenced by name rather than position. For instance, consider the follow-

ing version of the `emp.data` file, stored in the string `emp-data-tab`, shown folded into three columns to conserve space:

```
name Beth       name Kathy      name Mary
rate 12.75      rate 11.40      rate 7.50
hrs  0          hrs  30         hrs  20
dept mfg        dept sales      dept mfg

name Dan        name Mark       name Susie
rate 8.5        rate 12.75      rate 10.30
hrs  10         hrs  40         hrs  25
dept sales      dept mfg        dept acctg
```

The sixth example shows the calculation of total weekly wages using the `fold-port` function with a name-value record:

```
(let ((in (open-input-string emp-data-tab)))
  (fold-port
    (lambda (port)
      (read-name-value-record port))
    (lambda (base emp)
        (+ base
           (* (string->number
                (cdr (assoc "rate" emp)))
              (string->number
                (cdr (assoc "hrs" emp))))))
    0
    in))
```

Output from this calculation is `1344.5`.

The seventh and final example adds a wages field and converts `emp.data` to comma-separated values format:

```
(with-input-from-file "emp.data"
  (lambda ()
    (for-each-port
      read-delim-record
      (lambda (emp)
        (write-csv-record
          (append emp
            (list
              (number->string
                (wages emp)))))))))
```

Output from this example is:

```
Beth,12.75,0,mfg,0
Dan,8.50,10,sales,85
Kathy,11.40,30,sales,342
Mark,12.75,40,mfg,510
Mary,7.50,20,mfg,150
Susie,10.30,25,acctg,257.5
```

7. Testing

Testing of the readers and writers was done by performing round-trips through corresponding write/read functions. To illustrate, testing of character-delimited databases was done by writing `emp.data` to a string port using a pipe-character as a delimiter, then reading it back and calculating total wages using `fold-port`. Here's the code to convert `emp.data` to a pipe-delimited string, stored in the string variable `emp-data-pipe`:

```
(define emp-data-pipe
  (let ((out (open-output-string)))
    (with-input-from-file "emp.data"
      (lambda ()
        (for-each-port
          read-delim-record
          (lambda (emp)
            (write-delim-record
              emp #\| out)))))
    (get-output-string out)))
```

Given the `emp-data-pipe` string, the following code calculates total weekly wages using the `fold-port` function, with a reader that handles pipe-delimited records, and compares it to the expected value, returning `#t` for success and `#f` if the calculated amount is in error:

```
(= 1344.5
  (let ((in (open-input-string
              emp-data-pipe)))
    (fold-port
      (lambda (port)
        (read-delim-record #\| port))
      (lambda (base emp)
        (+ base (wages emp)))
      0
      in)))
```

Similar tests were conducted for the following reader/writer combinations: fixed, pipe-delimited, newline-delimited, null-delimited, csv, and tab-delimited name-value. The filter combinator and the various processing functions and procedures were tested using the example code shown above.

Appendix: source code

```
; READ-CHARS N [PORT]
(define (read-chars n . port)
  (let ((p (if (null? port) (current-input-port) (car port))))
    (if (eof-object? (peek-char p))
        (peek-char p)
        (let loop ((n n) (c (peek-char p)) (s '()))
          (cond ((and (eof-object? c) (pair? s)) (list->string (reverse s)))
                ((eof-object? c) c)
                ((zero? n) (list->string (reverse s)))
                (else (let ((c (read-char p)))
                        (loop (sub1 n) (peek-char p) (cons c s)))))))))

; READ-LINE [PORT]
(define (read-line . port)
  (define (eat p c)
    (if (and (not (eof-object? (peek-char p)))
             (char=? (peek-char p) c))
        (read-char p)))
  (let ((p (if (null? port) (current-input-port) (car port))))
    (let loop ((c (read-char p)) (line '()))
      (cond ((eof-object? c) (if (null? line) c (list->string (reverse line))))
            ((char=? #\newline c) (eat p #\return) (list->string (reverse line)))
            ((char=? #\return c) (eat p #\newline) (list->string (reverse line)))
            (else (loop (read-char p) (cons c line)))))))

; READ-FIXED-RECORD SIZE DEF-LIST [PORT]
(define (read-fixed-record size defs . port)
  (let ((p (if (null? port) (current-input-port) (car port))))
    (let ((fix-rec (read-chars size p)))
      (if (eof-object? fix-rec)
          fix-rec
          (let loop ((defs defs) (result '()))
            (if (null? defs)
                (reverse result)
                (loop (cdr defs)
                      (cons (substring fix-rec (caar defs) (cadar defs)) result))))))))

; READ-DELIM-RECORD [DELIM] [PORT]
(define (read-delim-record . args)
  (define (eat p c)
    (if (and (not (eof-object? (peek-char p)))
             (char=? (peek-char p) c))
        (read-char p)))
  (define (read-delim delim port)
    (cond ((eof-object? (peek-char port)) (peek-char port))
          ((and delim (or (char=? delim #\return) (char=? delim #\newline)))
            (let loop ((f (read-line port)) (fields '()))
              (if (or (eof-object? f) (string=? f ""))
                  (reverse fields)
                  (loop (read-line port) (cons f fields)))))
          (else
            (let loop ((c (read-char port)) (field '()) (fields '()))
              (cond ((eof-object? c) (reverse (cons (list->string (reverse field)) fields)))
                    ((char=? #\return c) (eat port #\newline)
                      (reverse (cons (list->string (reverse field)) fields)))
                    ((char=? #\newline c) (eat port #\return)
                      (reverse (cons (list->string (reverse field)) fields)))
                    ((and delim (char=? delim c))
                      (loop (read-char port) '() (cons (list->string (reverse field)) fields)))
                    ((char-whitespace? c)
                      (if (char-whitespace? (peek-char port))
                          (loop (read-char port) field fields)
                          (loop (read-char port) '()
                                (cons (list->string (reverse field)) fields))))
                    (else (loop (read-char port) (cons c field) fields)))))))
  (cond ((null? args) (read-delim #f (current-input-port)))
        ((and (null? (cdr args)) (char? (car args)))
          (read-delim (car args) (current-input-port)))
        ((and (null? (cdr args)) (port? (car args)))
```

```
          (read-delim #f (car args)))
        ((and (pair? (cdr args)) (null? (cddr args)) (char? (car args)) (port? (cadr args)))
          (read-delim (car args) (cadr args)))
        (else (read-delim #f (current-input-port)))))

; READ-CSV-RECORD [DELIM] [PORT]
(define (read-csv-record . args)
  (define (read-csv delim port)
    (define (add-field field fields)
      (cons (list->string (reverse field)) fields))
    (define (start field fields)
      (let ((c (read-char port)))
        (cond ((eof-object? c) (reverse fields))
              ((char=? #\return c) (carriage-return field fields))
              ((char=? #\newline c) (line-feed field fields))
              ((char=? #\" c) (quoted-field field fields))
              ((char=? delim c) (not-field '() (add-field field fields)))
              (else (unquoted-field (cons c field) fields)))))
    (define (not-field field fields)
      (let ((c (read-char port)))
        (cond ((eof-object? c) (cons "" fields))
              ((char=? #\return c) (carriage-return '() (add-field field fields)))
              ((char=? #\newline c) (line-feed '() (add-field field fields)))
              ((char=? #\" c) (quoted-field field fields))
              ((char=? delim c) (not-field '() (add-field field fields)))
              (else (unquoted-field (cons c field) fields)))))
    (define (quoted-field field fields)
      (let ((c (read-char port)))
        (cond ((eof-object? c) (add-field field fields))
              ((char=? #\" c) (may-be-doubled-quotes field fields))
              (else (quoted-field (cons c field) fields)))))
    (define (may-be-doubled-quotes field fields)
      (let ((c (read-char port)))
        (cond ((eof-object? c) (add-field field fields))
              ((char=? #\return c) (carriage-return '() (add-field field fields)))
              ((char=? #\newline c) (line-feed '() (add-field field fields)))
              ((char=? #\" c) (quoted-field (cons #\" field) fields))
              ((char=? delim c) (not-field '() (add-field field fields)))
              (else (unquoted-field (cons c field) fields)))))
    (define (unquoted-field field fields)
      (let ((c (read-char port)))
        (cond ((eof-object? c) (add-field field fields))
              ((char=? #\return c) (carriage-return '() (add-field field fields)))
              ((char=? #\newline c) (line-feed '() (add-field field fields)))
              ((char=? delim c) (not-field '() (add-field field fields)))
              (else (unquoted-field (cons c field) fields)))))
    (define (carriage-return field fields)
      (let ((c (peek-char port)))
        (cond ((eof-object? c) fields)
              ((char=? #\newline c) (read-char port) fields)
              (else fields))))
    (define (line-feed field fields)
      (let ((c (peek-char port)))
        (cond ((eof-object? c) fields)
              ((char=? #\return c) (read-char port) fields)
              (else fields))))
    (if (eof-object? (peek-char port)) (peek-char port) (reverse (start '() '()))))
  (cond ((null? args) (read-csv #\, (current-input-port)))
        ((and (null? (cdr args)) (char? (car args)))
          (read-csv (car args) (current-input-port)))
        ((and (null? (cdr args)) (port? (car args)))
          (read-csv #\, (car args)))
        ((and (pair? (cdr args)) (null? (cddr args)) (char? (car args)) (port? (cadr args)))
          (read-csv (car args) (cadr args)))
        (else (read-csv #\, (current-input-port)))))

; READ-NAME-VALUE-RECORD [DELIM] [PORT]
(define (read-name-value-record . args)
  (define (eat p c)
    (if (and (not (eof-object? (peek-char p)))
             (char=? (peek-char p) c))
```

```
        (read-char p)))
  (define (read-name-value delim port)
    (if (eof-object? (peek-char port))
        (peek-char port)
        (let loop ((c (read-char port)) (key '()) (value '()) (fields '()))
          (if (string? key)
              (cond ((eof-object? c)
                     (reverse (cons (cons key (list->string (reverse value))) fields)))
                    ((char=? #\return c) (eat port #\newline)
                     (loop (read-char port) '() '()
                           (cons (cons key (list->string (reverse value))) fields)))
                    ((char=? #\newline c) (eat port #\return)
                     (loop (read-char port) '() '()
                           (cons (cons key (list->string (reverse value))) fields)))
                    (else (loop (read-char port) key (cons c value) fields)))
              (cond ((eof-object? c)
                     (reverse (cons (cons (list->string (reverse key)) "") fields)))
                    ((char=? #\return c) (eat port #\newline)
                     (reverse (cons (cons (list->string (reverse key)) "") fields)))
                    ((char=? #\newline c) (eat port #\return)
                     (reverse (cons (cons (list->string (reverse key)) "") fields)))
                    ((and delim (char=? delim c))
                     (loop (read-char port) (list->string (reverse key)) value fields))
                    ((and (not delim) (char-whitespace? c))
                     (if (char-whitespace? (peek-char port))
                         (loop (read-char port) key value fields)
                         (loop (read-char port) (list->string (reverse key)) value fields)))
                    (else (loop (read-char port) (cons c key) value fields)))))))
  (cond ((null? args) (read-name-value #f (current-input-port)))
        ((and (null? (cdr args)) (char? (car args)))
          (read-name-value (car args) (current-input-port)))
        ((and (null? (cdr args)) (port? (car args)))
          (read-name-value #f (car args)))
        ((and (pair? (cdr args)) (null? (cddr args)) (char? (car args)) (port? (cadr args)))
          (read-name-value (car args) (cadr args)))
        (else (read-name-value #f (current-input-port)))))

; FILTER-PORT READER PRED?
(define (filter-port reader pred?)
  (lambda args
    (let loop ((x (apply reader args)))
      (cond ((eof-object? x) x)
            ((pred? x) x)
            (else (loop (apply reader args)))))))

; FOR-EACH-PORT READER PROC [PORT]
(define (for-each-port reader proc . port)
  (let ((p (if (null? port) (current-input-port) (car port))))
    (let loop ((item (reader p)))
      (if (not (eof-object? item))
          (begin (proc item) (loop (reader p)))))))

; MAP-PORT READER MAPPER [PORT]
(define (map-port reader mapper . port)
  (let ((p (if (null? port) (current-input-port) (car port))))
    (let loop ((item (reader p)) (result '()))
      (if (eof-object? item)
          (reverse result)
          (loop (reader p) (cons (mapper item) result))))))

; FOLD-PORT READER FOLDER BASE [PORT]
(define (fold-port reader folder base . port)
  (let ((p (if (null? port) (current-input-port) (car port))))
    (let loop ((item (reader p)) (result base))
      (if (eof-object? item)
          result
          (loop (reader p) (folder result item))))))

; MAP-REDUCE-PORT READER MAPPER REDUCER LT? [PORT]
(define (map-reduce-port reader mapper reducer lt? . port)
  (define (tree c k v l r) (vector c k v l r))
```

```
  (define empty (tree 'black 'nil 'nil 'nil 'nil))
  (define (empty? t) (eqv? t empty))
  (define (color t) (vector-ref t 0))
  (define (key t) (vector-ref t 1))
  (define (value t) (vector-ref t 2))
  (define (lkid t) (vector-ref t 3))
  (define (rkid t) (vector-ref t 4))
  (define (red? c) (eqv? c 'red))
  (define (black? c) (eqv? c 'black))
  (define (balance c k v l r)
    (cond ((and (black? c) (red? (color l)) (red? (color (lkid l))))
           (tree 'red (key l) (value l)
             (tree 'black (key (lkid l)) (value (lkid l))
               (lkid (lkid l)) (rkid (lkid l)))
             (tree 'black k v (rkid l) r)))
          ((and (black? c) (red? (color l)) (red? (color (rkid l))))
           (tree 'red (key (rkid l)) (value (rkid l))
             (tree 'black (key l) (value l) (lkid l) (lkid (rkid l)))
             (tree 'black k v (rkid (rkid l)) r)))
          ((and (black? c) (red? (color r)) (red? (color (lkid r))))
           (tree 'red (key (lkid r)) (value (lkid r))
             (tree 'black k v l (lkid (lkid r)))
             (tree 'black (key r) (value r) (rkid (lkid r)) (rkid r))))
          ((and (black? c) (red? (color r)) (red? (color (rkid r))))
           (tree 'red (key r) (value r)
             (tree 'black k v l (lkid r))
             (tree 'black (key (rkid r)) (value (rkid r))
               (lkid (rkid r)) (rkid (rkid r)))))
          (else (tree c k v l r))))
  (define (insert t k v)
    (define (ins t)
      (let ((tc (color t)) (tk (key t)) (tv (value t)) (tl (lkid t)) (tr (rkid t)))
        (cond ((empty? t) (tree 'red k v empty empty))
              ((lt? k tk) (balance tc tk tv (ins tl) tr))
              ((lt? tk k) (balance tc tk tv tl (ins tr)))
              (else (tree tc tk (reducer k tv v) tl tr)))))
    (let* ((z (ins t)) (zk (key z)) (zv (value z)) (zl (lkid z)) (zr (rkid z)))
      (tree 'black zk zv zl zr)))
  (define (enlist t base)
    (cond ((empty? t) base)
          ((and (empty? (lkid t)) (empty? (rkid t)))
            (cons (cons (key t) (value t)) base))
          (else (enlist (lkid t)
                        (cons (cons (key t) (value t))
                              (enlist (rkid t) base))))))
  (let ((p (if (null? port) (current-input-port) (car port))))
    (let loop ((item (reader p)) (t empty))
      (if (eof-object? item)
          (enlist t '())
          (call-with-values
            (lambda () (mapper item))
            (lambda (k v) (loop (reader p) (insert t k v)))))))))

; QUOTE-CSV DELIM STR
(define (quote-csv delim str)
  (define (string-find str pat)
    (let loop ((i 0))
      (cond ((<= (string-length str) i) #f)
            ((string=? (substring str i (+ i (string-length pat))) pat) i)
            (else (loop (+ i 1))))))
  (define (string-replace-all str pat repl)
    (let ((len-str (string-length str))
          (len-pat (string-length pat))
          (spot (string-find str pat)))
      (if spot
          (string-append
            (substring str 0 spot)
            repl
            (string-replace-all (substring str (+ spot len-pat) len-str) pat repl))
          str)))
  (let ((new-str (string-replace-all str "\"" "\"\"")))
```

```
    (if (or (string-find str (string delim))
            (not (string=? str new-str))
            (string-find str (string #\return))
            (string-find str (string #\newline)))
        (string-append "\"" new-str "\"")
        str)))

; WRITE-FIXED-RECORD REC SIZE DEFS [PORT]
(define (write-fixed-record rec size defs . port)
  (let ((p (if (null? port) (current-output-port) (car port)))
        (out (make-string size #\space)))
    (do ((rec rec (cdr rec))
         (defs defs (cdr defs)))
        ((or (null? rec) (null? defs)) (display out p))
      (do ((s 0 (+ s 1))
           (t (caar defs) (+ t 1)))
          ((or (= s (string-length (car rec))) (= (cadar defs) t)))
        (string-set! out t (string-ref (car rec) s))))))

; WRITE-DELIM-RECORD REC [DELIM] [PORT]
(define (write-delim-record rec . args)
  (define (write-delim delim port)
    (do ((rec rec (cdr rec)))
        ((null? rec) (newline port))
      (display (car rec) port)
      (if (pair? (cdr rec)) (display delim port)))
    (if (or (char=? delim #\return) (char=? delim #\newline)) (newline port)))
  (cond ((null? args) (write-delim #\space (current-output-port)))
        ((and (null? (cdr args)) (char? (car args)))
          (write-delim (car args) (current-output-port)))
        ((and (null? (cdr args)) (port? (car args)))
          (write-delim #\space (car args)))
        ((and (pair? (cdr args)) (null? (cddr args)) (char? (car args)) (port? (cadr args)))
          (write-delim (car args) (cadr args)))
        (else (write-delim #\space (current-output-port)))))

; WRITE-CSV-RECORD REC [DELIM] [PORT]
(define (write-csv-record rec . args)
  (define (write-csv delim port)
    (do ((rec rec (cdr rec)))
        ((null? rec) (newline port))
      (display (quote-csv delim (car rec)) port)
      (if (pair? (cdr rec)) (display delim port))))
  (cond ((null? args) (write-csv #\, (current-output-port)))
        ((and (null? (cdr args)) (char? (car args)))
          (write-csv (car args) (current-output-port)))
        ((and (null? (cdr args)) (port? (car args)))
          (write-csv #\, (car args)))
        ((and (pair? (cdr args)) (null? (cddr args)) (char? (car args)) (port? (cadr args)))
          (write-csv (car args) (cadr args)))
        (else (write-csv #\, (current-output-port)))))

; WRITE-NAME-VALUE-RECORD REC [DELIM] [PORT]
(define (write-name-value-record rec . args)
  (define (write-name-value delim port)
    (do ((rec rec (cdr rec)))
        ((null? rec) (newline port))
      (display (caar rec) port)
      (display delim port)
      (display (cdar rec) port)
      (newline port)))
  (cond ((null? args) (write-name-value #\space (current-output-port)))
        ((and (null? (cdr args)) (char? (car args)))
          (write-name-value (car args) (current-output-port)))
        ((and (null? (cdr args)) (port? (car args)))
          (write-name-value #\space (car args)))
        ((and (pair? (cdr args)) (null? (cddr args)) (char? (car args)) (port? (cadr args)))
          (write-name-value (car args) (cadr args)))
        (else (write-name-value #\space (current-output-port)))))
```

Appendix: test code

```
(define (wages emp)
  (* (string->number (cadr emp))
     (string->number (caddr emp))))

(define (string-trim-left s)
  (let ((c (string-ref s 0)))
    (if (char-whitespace? c)
        (string-trim-left
          (substring s 1
            (string-length s)))
        s)))

(define (string-trim-right s)
  (let* ((len (- (string-length s) 1))
         (c (string-ref s len)))
    (if (char-whitespace? c)
        (string-trim-right
          (substring s 0 len))
        s)))

(define (string-trim s)
  (string-trim-left (string-trim-right s)))

(define (a-cons key value a-list)
  (cons (cons key value) a-list))

(define emp-data-fixed
  (let ((out (open-output-string)))
    (with-input-from-file "emp.data"
      (lambda ()
        (for-each-port
          read-delim-record
          (lambda (emp)
            (write-fixed-record emp 22
              '((0 5) (6 11) (12 14) (15 20)) out)))))
    (get-output-string out)))

(define emp-data-pipe
  (let ((out (open-output-string)))
    (with-input-from-file "emp.data"
      (lambda ()
        (for-each-port
          read-delim-record
          (lambda (emp)
            (write-delim-record
              emp #\| out)))))
    (get-output-string out)))

(define emp-data-newline
  (let ((out (open-output-string)))
    (with-input-from-file "emp.data"
      (lambda ()
        (for-each-port
          read-delim-record
          (lambda (emp)
            (write-delim-record
              emp #\newline out)))))
    (get-output-string out)))

(define emp-data-null
  (let ((out (open-output-string)))
    (with-input-from-file "emp.data"
      (lambda ()
        (for-each-port
          read-delim-record
          (lambda (emp)
            (write-delim-record
              emp out)))))
    (get-output-string out)))
```

```
(define emp-data-csv
  (let ((out (open-output-string)))
    (with-input-from-file "emp.data"
      (lambda ()
        (for-each-port
          read-delim-record
          (lambda (emp)
            (write-csv-record emp out)))))
    (get-output-string out)))

(define emp-data-tab
  (let ((out (open-output-string)))
    (with-input-from-file "emp.data"
      (lambda ()
        (for-each-port
          read-delim-record
          (lambda (emp)
            (write-name-value-record
              (a-cons "name" (list-ref emp 0)
              (a-cons "rate" (list-ref emp 1)
              (a-cons "hrs"  (list-ref emp 2)
              (a-cons "dept" (list-ref emp 3) '())))))
             #\tab out)))))
    (get-output-string out)))

(and ; should return #t
  (= 1344.5
    (let ((in (open-input-string emp-data-fixed)))
      (fold-port
        (lambda (port) (read-fixed-record 22 '((0 5) (6 11) (12 14) (15 20)) port))
        (lambda (base emp) (+ base (* (string->number (string-trim (list-ref emp 1)))
                                      (string->number (string-trim (list-ref emp 2))))))
        0
        in)))
  (= 1344.5
    (let ((in (open-input-string emp-data-pipe)))
      (fold-port
        (lambda (port) (read-delim-record #\| port))
        (lambda (base emp) (+ base (wages emp)))
        0
        in)))
  (= 1344.5
    (let ((in (open-input-string emp-data-newline)))
      (fold-port
        (lambda (port) (read-delim-record #\newline port))
        (lambda (base emp) (+ base (wages emp)))
        0
        in)))
  (= 1344.5
    (let ((in (open-input-string emp-data-null)))
      (fold-port
        (lambda (port) (read-delim-record port))
        (lambda (base emp) (+ base (wages emp)))
        0
        in)))
  (= 1344.5
    (let ((in (open-input-string emp-data-csv)))
      (fold-port
        (lambda (port) (read-csv-record port))
        (lambda (base emp) (+ base (wages emp)))
        0
        in)))
  (= 1344.5
    (let ((in (open-input-string emp-data-tab)))
      (fold-port
        (lambda (port) (read-name-value-record #\tab port))
        (lambda (base emp) (+ base (* (string->number (cdr (assoc "rate" emp)))
                                      (string->number (cdr (assoc "hrs" emp))))))
        0
        in))))
```