UCLA Statistics Series #111 September 1992

Reading Formatted Input in X-LISP STAT

Jan de Leeuw
Depts. of Mathematics and Psychology, UCLA

INTRODUCTION

The XLISP-STAT package lacks a number of procedures which would make it more suitable for routine data handling and analysis. As examples we mention recoding variables, making cross tables, printing nice tables and publication-quality graphics, and the reading of formatted input (as in FORTRAN format strings and the scanf function in C. On can argue, of course, that this is not necessarily a bad thing. The UNIX philosophy tells us that tools should be highly specialized. Tools which can do everything, don't do anything right.

On the other hand there is no reason not to use XLISP-STAT as a convenient interface to other special-purpose UNIX tools, or to add LISP functions that people can either use or ignore.

Making tables and graphs will be discussed in another note. Here we discuss some XLISP-STAT functions to read formatted input. Code is listed below, and will also be available on statlib in ftp.stat.cmu.edu.

EXAMPLE

Consider the following small input file, which is called foo.dat.

```
001aap -1.235555
002mens +.154444
```

We read this into a list of lists, each list containing one record, each record having the same number of data items. For this we use the function

```
(read-file "foo.dat" ffor),
```

with ffor a format string, more precisely a list of format strings. In this case the first format string we try out is

```
("i3" "a4" "f5.2" "4i1")
```

which creates

```
((1 "aap " -1.23 5 5 5 5) (2 "mens" 0.15 4 4 4 4))
```

If we want to, we can also use "f3.0" for "i3", but generally i-formats are somewhat more efficient.

Other formats are possible as well. We can use

```
("i3" "a4" "x5" "4i1")
```

which creates

```
((1 "aap " 5 5 5 5) (2 "mens" 4 4 4 4))
```

or

```
("i3" "x4" "f5.2" "t9" "a4" "x5" "4i1")
```

which creates

```
((1 -1.23 "aap " 5 5 5 5) (2 0.15 "mens" 4 4 4 4))
```

Thus using x-formats allows us to skip characters, while the t-format allows us to back up. Combining x-formats and t-formats makes it possible to read the data on a record in arbitrary order. To conclude the example:

```
("f3.0" "3a1" "x3" "f3.2" "2i2")
```

gives
`((1 "a" "a" "p" 0.23 55 55) (2 "m" "e" "n" 0.15 44 44))`.
and
`("a8" "f2.0" "i6")`
gives `(("001aap -" 1 235555) ("002mens " 0 154444))`.
This seems to indicate a considerable level of flexibility.

SYNTAX

- The i-formats, a-formats, and f-formats can have repeat factors (as in `"4f7.5"`), the x-format (skip next n places) and the t-format (go n places back) do not have repeats (in a sense, they are repeats).
- We do not allow `"4(f10.4,x2)"` etc.
- The string read by the a-format can contain all `ascii` characters.
- The i-format can read characters 0-9, spaces, and signs, with the spaces first, then either a + or a −, and then a string of digits (which can have leading zeroes). Spaces and signs can be missing.
- The f-format can read characters 0-9, point, spaces, and signs, with the spaces first, then either a + or a −, and then a string of digits (which can have leading zeroes), then the decimal point, and then another string of digits (which can have trailing zeroes). Spaces and signs can be missing. The decimal point and the digit string following it can be missing. The decimal string before the decimal point can be missing. We even allow both decimal strings to be missing, in which case the number is interpreted as zero.

LISTING

```
(defun read-file (fname fstring)
"Args: (string list)
Reads file STRING into a list of lists of characters.
Then reads the list of strings containing fortran
type formats, and uses these formats to write the
characters into LIST, a list of lists of data items."
  (let* (
        (contents (read-record fname))
        (flist (handle-format-list fstring))
        (n (length contents))
        (m (length flist))
        (y (coerce (make-array (list n (count-vars fstring))) 'list))
        )
(dotimes (i n)
      (let  (
            (rec (elt contents i))
            (counter 0)
            (mm 0)
            )
 dotimes (j m)
       let*  (
             (fr (elt flist j))
             ch (first fr))
```

```
            (rp 1)
            )
(cond ((eq ch #\t) (setf counter (- counter (elt fr 1))))
      ((eq ch #\x) (setf counter (+ counter (elt fr 1))))
      (t (setf rp (elt fr 1))
         (setf fw (elt fr 2))
         (dotimes (k rp)
            (setf sub (select rec (+ counter (iseq fw))))
            (setf counter (+ counter fw))
            (cond ((eq ch #\i) (setf (elt (elt y i) mm) (dconvert sub)))
                  ((eq ch #\a) (setf (elt (elt y i) mm) (coerce sub 'string)))
                  ((eq ch #\f) (setf (elt (elt y i) mm) (fconvert sub))))
            (setf mm (+ 1 mm)))
         ))
))
y))
```

```
(defun read-record (fname)
"Args: (string)
Reads characters from file STRING until it runs into a newline.
Puts what it reads into a list of characters. Then continues
to read the next line into the next list. And so on. Returns
the list of lists. Empty lines read into nil list. Lines
which do not end in newline are not returned (i.e. maybe the
last one is not read)."
   (let (
        (f (open fname))
        (m nil)
        (s nil)
        )
(loop
(setf c (read-char f))
(cond ((equal c nil) (return m))
      ((equal c #\Newline)
         (cond ((equal m nil) (setf m (list s)) (setf s nil))
               (t (setf m (append m (list s))) (setf s nil))))
      (t (cond ((equal s nil) (setf s (list c)))
               (t (setf s (combine s c)))))
))))
```

```
(defun handle-format-list (flist)
"Args: (list)
LIST is a format list, i.e. a list of strings of the
fortran format type, with f, a, i, t, and x formats allowed.
These strings are translated to short lists, which
are easier to handle."
(let* (
      (n (length flist))
      (u (repeat 0 n))
      )
(dotimes (i n)
     (let (
          (w (coerce (elt flist i) 'list))
          )
(setf (elt u i)
     (cond ((position #\f w) (f-format w))
           ((position #\x w) (x-format w))
           ((position #\t w) (t-format w))
           ((position #\a w) (a-format w))
           ((position #\i w) (i-format w))
           (t (error "no f, x, t, a, i format"))))))
u))
```

```
(defun count-vars (flist)
"Args: list
LIST is a format list, i.e. a list of strings of the
fortran format type, with f, a, i, t, and x formats allowed.
This function counts the number of variables described
by the list."
(let (
     (n (length flist))
     (s 0)
     )
(dotimes (i n)
     (let (
          (w (coerce (elt flist i) 'list))
          )
     (cond ((position #\f w) (setf s (+ s (elt (f-format w) 1))))
           ((position #\a w) (setf s (+ s (elt (a-format w) 1))))
           ((position #\i w) (setf s (+ s (elt (i-format w) 1))))
     )))
s))
```

```
(defun f-format (fl)
"Args: list
This function converts a LIST of characters, representing a
fortran-type f-format, to a string containing the
character #\f, the repeat, the width of the field,
and the number of decimals."
(let (
     (times '(#\1))
     (decimal 0)
     (field 0)
     (lpos (length fl))
     (fpos (position #\f fl))
     (dpos (position #\. fl))
     )
(if (not dpos) (error "f-format has no decimal point"))
(if (> fpos 0) (setf times (select fl (iseq fpos))))
(setf field (select fl (+ 1 fpos (iseq (- dpos fpos 1)))))
(setf decimal (select fl (+ 1 dpos (iseq (- lpos dpos 1)))))
(list #\f (dconvert times) (dconvert field) (dconvert decimal))
))
```

```
(defun x-format (fl)
"Args: list
This converts a LIST of characters, representing a
fortran-type x-format, to a number: the width of the field."
(let (
     (lpos (length fl))
     (xpos (position #\x fl))
     )
(if (> xpos 0) (error "x not in first place in x-format"))
(setf field (select fl (+ 1 xpos (iseq (- lpos xpos 1)))))
(list #\x (dconvert field))
))
```

```
(defun t-format (fl)
"Args: list
This converts a LIST of characters, representing a
fortran-type t-format, to a number: the number of places to go back."
(let (
     (lpos (length fl))
     (tpos (position #\t fl))
     )
(if (> tpos 0) (error "t not in first place in t-format"))
(setf field (select fl (+ 1 tpos (iseq (- lpos tpos 1)))))
(list #\t (dconvert field))
))
```

```
(defun a-format (fl)
"Args: list
This converts a LIST of characters, representing a
fortran-type a-format, to two numbers: the repeat and
the width of the field."
(let (
     (times '(#\1))
     (field 0)
     (lpos (length fl))
     (apos (position #\a fl))
     )
(if (> apos 0) (setf times (select fl (iseq apos))))
(setf field (select fl (+ 1 apos (iseq (- lpos apos 1)))))
(list #\a (dconvert times) (dconvert field))
))
```

```
(defun i-format (fl)
"Args: list
This converts a LIST of characters, representing a
fortran-type i-format, to two numbers: the repeat and
the width of the field."
(let (
     (times '(#\1))
     (field 0)
     (lpos (length fl))
     (ipos (position #\i fl))
     )
(if (> ipos 0) (setf times (select fl (iseq ipos))))
(setf field (select fl (+ 1 ipos (iseq (- lpos ipos 1)))))
(list #\i (dconvert times) (dconvert field))
))
```

```
(defun fconvert (cs)
"Args: list
Converts a LIST of characters to a decimal number. Only
characters -, +, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 and the
decimal point are allowed. Leading zeroes or spaces are no problem."
(let* (
      (sgn 1)
      (c1 (reverse (rest (member #\. (reverse cs)))))
      (c2 (rest (member #\. cs)))
      )
(if (and (not c1) (not c2)) (setf c1 cs))
(if (not c2) (setf c2 (list #\0)))
(if (not c1) (setf c1 (list #\0)))
(cond ((member #\+ c1) (setf c1 (rest (member #\+ c1))))
      ((member #\- c1) (setf c1 (rest (member #\- c1)))
                       (setf sgn -1))
      )
(if (not c1) (setf c1 (list #\0)))
(let* (
      (b1 (reverse (** 10 (iseq (length c1)))))
      (b2 (** .1 (+ 1 (iseq (length c2)))))
      (t1 (- (mapcar #'char-code c1) 48))
      (t2 (- (mapcar #'char-code c2) 48))
      )
```

```
(* sgn (+ (sum (* b1 t1)) (sum (* b2 t2))))
)))
```

```
(defun dconvert (cs)
"Args: list
Converts a LIST of characters to an integer. Only
characters -, +, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
are allowed. Leading zeroes or spaces are no problem."
(let  (
      (sgn 1)
      )
(cond ((member #\+ cs) (setf cs (rest (member #\+ cs))))
      ((member #\- cs) (setf cs (rest (member #\- cs)))
                       (setf sgn -1))
      )
(let  (
      (b (reverse (** 10 (iseq (length cs)))))
      (t (- (mapcar #'char-code cs) 48))
      )
(* sgn (sum (* b t)))
)))
```