Some Utilities for Lisp-Stat

Hani J. Doss
Department of Statistics
The Ohio State University
Columbus, OH 43210

B. Narasimhan
Department of Statistics
Stanford University
Stanford, CA 94305

Revision : 1.1 of *Date* : 1998/05/0723 : 57 : 07

Contents

Abstract

We present a set of utilities for easier programming in Lisp-Stat.

1 Copyright

We begin with our usual copyright.

3 ⟨*Copyright* 3⟩≡ (4)

```
;;;
;;;  $Revision: 1.1 $ of $Date: 1998/05/07 23:57:07 $
;;;
;;;  Copyright (C) 1994, 1995, 1998. Doss and Narasimhan
;;;
;;;  Hani J. Doss (doss@stat.ohio-state.edu) and
;;;  B. Narasimhan (naras@stat.stanford.edu)
;;;
;;;  This program is free software; you can redistribute it and/or modify
;;;  it under the terms of the GNU 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 General Public License for more details.
;;;
;;;  You should have received a copy of the GNU General Public License
;;;  along with this program; if not, write to the Free Software
;;;  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
;;;
```

Defines:
`copyright`, never used.

2 Introduction

This section deals with code that we have found to be generally useful in writing `Lisp-Stat` software. This suite of utility functions is implemented as a stand-alone `Lisp-Stat` package.

4 ⟨*Utility Package* 4⟩≡

```
⟨Copyright 3⟩
(defpackage "UTILITY" (:use "XLISP"))
(in-package "UTILITY")
(import '(user::combine))
⟨Modifications to System 5a⟩
⟨The arrange function 9⟩
⟨The ok-or-abort-dialog function 10a⟩
⟨The get-a-string-dialog function 10b⟩
⟨The get-nonempty-string-dialog function 11a⟩
⟨The get-a-value-dialog function 11b⟩
⟨The get-nonnil-value-dialog function 12a⟩
⟨The get-tested-value-function 12b⟩
⟨The get-yes-no-list function 13a⟩
⟨The convert-to-numbers function 13b⟩
⟨The some-files-dont-exist function 14a⟩
⟨The coerce-to-layout-of function 14b⟩
⟨The strcat function 15a⟩
⟨The definedp function 15b⟩
⟨The ask-user-for-value function 15c⟩
⟨The make-sliders function 16⟩
(export
        '(value-item-proto
          arrange
          ok-or-abort-dialog
          get-a-string-dialog
          get-nonempty-string-dialog
          get-a-value-dialog
          get-nonnil-value-dialog
          get-tested-value-dialog
          get-yes-no-list
          convert-to-numbers
          some-files-dont-exist
          coerce-to-layout-of
          strcat
          definedp
          ask-user-for-value
          modified-boxplot
          make-sliders))
(provide "utility")
```

Uses `arrange` 9, `ask-user-for-value` 15c, `coerce-to-layout-of` 14b, `convert-to-numbers` 13b, `definedp` 15b, `get-a-string-dialog` 10b, `get-a-value-dialog` 11b, `get-nonempty-string-dialog` 11a, `get-nonnil-value-dialog` 12a, `get-tested-value-dialog` 12b, `get-yes-no-list` 13a, `make-sliders` 16, `ok-or-abort-dialog` 10a, `some-files-dont-exist` 14a, `strcat` 15a, and `value-item-proto` 6c.

To use some functions, changes have to be made to the `Lisp-Stat` system objects. So we begin with those changes.

3 Modifications to the `Lisp-Stat` System Objects

In this section, we deal with some modifications to the `Lisp-Stat` system. The modifications are mostly to the dialog items provided in the `Lisp-Stat` system. For example, a `:width` method for `text-item-proto` makes it very easy to produce better looking dialogs. So here we go.

5a ⟨*Modifications to System* 5a⟩≡ (4)
⟨*The :width method for text-item-proto* 5b⟩
⟨*The :width method for button-item-proto* 5c⟩
⟨*The :width method for interval-scroll-item-proto* 5d⟩
⟨*The :value method for interval-scroll-item-proto* 6a⟩
⟨*The value-item object* 6b⟩
⟨*The :display-value method for interval-scroll-item-proto* 7c⟩
⟨*The :value-item method for interval-scroll-item-proto* 8⟩

3.1 The `:width` Method for `text-item-proto`

5b ⟨*The :width method for text-item-proto* 5b⟩≡ (5a)
```
(defmeth text-item-proto :width (&optional width)
  (if width
      (let ((sz (slot-value 'size)))
        (setf (slot-value 'size) (list width (select sz 1))))
    (select (slot-value 'size) 0)))
```
Defines:
`:width`, used in chunk 16.

3.2 The `:width` Method for `button-item-proto`

5c ⟨*The :width method for button-item-proto* 5c⟩≡ (5a)
```
(defmeth button-item-proto :width (&optional width)
  (if width
      (let ((sz (slot-value 'size)))
        (setf (slot-value 'size) (list width (select sz 1))))
    (select (slot-value 'size) 0)))
```
Defines:
`:width`, used in chunk 16.

3.3 The `:width` Method for `interval-scroll-item-proto`

5d ⟨*The :width method for interval-scroll-item-proto* 5d⟩≡ (5a)
```
(defmeth interval-scroll-item-proto :width (&optional width)
  (if width
      (let ((sz (slot-value 'size)))
        (setf (slot-value 'size) (list width (select sz 1))))
    (select (slot-value 'size) 0)))
```
Defines:
`:width`, used in chunk 16.

3.4 The `:value` Method for `interval-scroll-item-proto`

6a ⟨*The :value method for interval-scroll-item-proto* 6a⟩≡ (5a)

```
(defmeth interval-scroll-item-proto :value (&optional (val nil set))
  (let ((interval (slot-value 'xlisp::interval))
        (num-points (slot-value 'xlisp::num-points)))
    (if set
        (let* ((min (elt interval 0))
               (max (elt interval 1))
               (val (if (= min max)
                        0
                    (floor (* (1- num-points) (/ (- val min) (- max min)))))))
          (call-next-method val)
          (send self :display-value)
          (send self :user-action)))
    (let ((min (elt interval 0))
          (max (elt interval 1)))
      (+ min (* (/ (call-next-method) (max 1 (1- num-points)))
         (- max min))))))
```

Defines:
`:value`, used in chunks 7c, 8, and 13a.
Uses `:display-value` 7c.

3.5 The Value Item Object

This object is a descendant of `text-item-proto`. Its main purpose is to allow for formatted printing of slider values in a `text-item`. It can be used in exactly the same manner as a `text-item` and behaves like a `text-item` when a format is not specified.

6b ⟨*The value-item object* 6b⟩≡ (5a)
⟨*The value-item object prototype* 6c⟩
⟨*The :isnew method for value-item* 7a⟩
⟨*The :format method for value-item* 7b⟩

3.5.0.1 The Value Item Object Prototype

6c ⟨*The value-item object prototype* 6c⟩≡ (6b)

```
(defproto value-item-proto '(format)
  () text-item-proto
  "A dialog item for formatted displaying of values of slider stops.")
(export 'value-item-proto)
```

Defines:
`value-item-proto`, used in chunks 4, 7, and 16.

3.5.0.1.1 The `:isnew` Method for Value Item

7a ⟨*The :isnew method for value-item* 7a⟩≡ (6b)

```
(defmeth value-item-proto :isnew (string &rest args &key (format "~g"))
  "Method args: (string &rest args &key (format ~g))
Creates an instance of value-item-proto; format is used for formatted
printing. It can be a format string or a list of two numbers. In the
latter case, variable format printing is assumed using ~v,vf."
  (setf (slot-value 'format) format)
  (apply #'call-next-method string args))
```

Defines:
`:isnew`, never used.
Uses `value-item-proto` 6c.

3.5.0.1.2 The `:format` Method for Value Item

7b ⟨*The :format method for value-item* 7b⟩≡ (6b)

```
(defmeth value-item-proto :format (&optional format)
  "Method args: (&optional format)
Sets or retrieves the format, which should be a format string or a
list of 2 numbers for ~v,~vf printing."
  (if format
      (setf (slot-value 'format) format)
    (slot-value 'format)))
```

Defines:
`:format`, used in chunks 7c and 16.
Uses `value-item-proto` 6c.

4 The `:display-value` method for `interval-scroll-item-proto`

We must also modify the `:display-value` method of `interval-scroll-item-proto` to use formatted printing. Note how we have to access the internal symbols of the `xlisp` package.

7c ⟨*The :display-value method for interval-scroll-item-proto* 7c⟩≡ (5a)

```
(defmeth interval-scroll-item-proto :display-value ()
  (when (slot-value 'xlisp::value-text-item)
        (let* ((v-item (slot-value 'xlisp::value-text-item))
               (format (send v-item :format))
               (value (send self :value)))
          (if (listp format)
              (send v-item :text
                    (format nil "~v,vf"
                            (select format 0) (select format 1) value))
            (send v-item :text (format nil format value))))))
```

Defines:
`:display-value`, used in chunk 6a.
Uses `:format` 7b and `:value` 6a.

5 The `:value-item` method for `interval-scroll-item`

Again, for same reasons as above, we have to modify the `:value-item]` `method for [[interval-scroll-item`.

8 ⟨*The :value-item method for interval-scroll-item-proto* 8⟩≡ (5a)

```
(defmeth interval-scroll-item-proto :value-item ()
  (slot-value 'xlisp::value-text-item))
```

Defines:
`:value-item`, never used.
Uses `:value` 6a.

6 Utility Functions

And here are the utility functions.

6.1 The `arrange` function

The function `arrange` is similar to `split-list`. It takes a list and chops it up into sublists. If `rows` is specified, returns a list of length `rows`. If `cols` is specified, returns a list of lists, each of which is of length `cols`. If neither is specified, it assumes `cols` = $\lfloor\sqrt{n}\rfloor$ where n is the length of the list.

9 ⟨*The arrange function* 9⟩≡ (4)

```
(defun arrange (list &key rows cols)
  "Method args: (list &key rows cols)
Chops up a list into sublists of rows items each and
returns a list of lists. If rows is not given, it is
floor of sqrt of length of list. Rows need not divide list length!"
  (let ((n (length list)))
    (cond
     (rows
      (unless (>= n rows)
              (error "number of items less than number of rows"))
      (multiple-value-bind (q rem) (floor n rows)
        (if (= rem 0)
            (mapcar #'(lambda(x) (select list (+ x (* rows (iseq q)))))
                    (iseq rows))
          (append
           (mapcar #'(lambda(x) (select list (+ x (* rows (iseq (1+ q))))))
                   (iseq rem))
           (mapcar #'(lambda(x) (select list (+ x (* rows (iseq q)))))
                   (iseq rem (1- rows)))))))
     (t
      (unless cols (setf cols (floor (sqrt n))))
      (unless (>= n cols)
        (error "number of items less than number of cols"))
      (let* ((q (floor n cols))
             (product (* cols q)))
        (if (= product n)
            (split-list list cols)
          (append
           (split-list (select list (iseq product)) cols)
           (list (select list (+ product (iseq (- n product))))))))))))
```

Defines:
`arrange`, used in chunks 4 and 13a.

6.2 The `ok-or-abort-dialog` function

This function prompts the user with a string in a dialog. Both an `OK` button and a `Abort` button are displayed so that the user can get back to the top level.

10a ⟨*The ok-or-abort-dialog function* 10a⟩≡ (4)

```
(defun ok-or-abort-dialog (string)
  "Args: string.
Prompts the user with a string in a dialog."
  (let ((text (send text-item-proto :new string))
        (ok (send modal-button-proto :new "OK"))
        (abort (send modal-button-proto :new "Abort"
                     :action #'top-level)))
    (send (send modal-dialog-proto :new (list text (list ok abort)))
          :modal-dialog)))
```

Defines:
`ok-or-abort-dialog`, used in chunk 4.

6.3 The `get-a-string-dialog` function

The `Lisp-Stat` `get-string-dialog` function produces a `Cancel` button, which we don't want when we loop for a correct input. We want to allow an `Abort` button that will take us to the top level. So here is our modified version of the dialog.

10b ⟨*The get-a-string-dialog function* 10b⟩≡ (4)

```
(defun get-a-string-dialog (s &key initial)
  "Args: (s &key initial)
Opens a modal dialog with prompt S, a text field and OK and Abort buttons.
INITIAL is converted to a string with ~A format directive. Returns
string of text field content on OK, returns to top-level on Abort."
  (let* ((text (send text-item-proto :new s))
         (edit-item (send edit-text-item-proto :new
                          (if initial (format nil "~a" initial) "")
                          :text-length 40))
         (abort (send modal-button-proto :new "Abort"
                      :action #'top-level))
         (ok (send modal-button-proto :new "OK"
                   :action #'(lambda() (send edit-item :text)))))
    (send
     (send modal-dialog-proto :new (list text edit-item (list ok abort)))
     :modal-dialog)))
```

Defines:
`get-a-string-dialog`, used in chunks 4 and 11.

6.4 The `get-nonempty-string-dialog` function

11a ⟨*The get-nonempty-string-dialog function* 11a⟩≡ (4)

```
(defun get-nonempty-string-dialog (s &key initial)
  "Args: (s &key initial)
Opens a modal dialog with prompt S, a text field and OK button.
INITIAL is converted to a string with ~A format directive. Returns
string of text field content on OK. Will return a nonempty string or
take user back to top-level."
(loop
 (let ((input (get-a-string-dialog s :initial initial)))
   (if (= (length input) 0)
       (message-dialog
        "Error!!\n\nNeed a non-empty string!")
     (return input)))))
```

Defines:
`get-nonempty-string-dialog`, used in chunk 4.
Uses `get-a-string-dialog` 10b.

6.5 The `get-a-value-dialog` function

11b ⟨*The get-a-value-dialog function* 11b⟩≡ (4)

```
(defun get-a-value-dialog (s &key (initial "" suppliedp))
  "Args: (s &key (initial \"\" suppliedp)
Opens a modal dialog with prompt S, a text field and OK button.
INITIAL is converted to a string with ~S format directive. Returns
string of text field content on OK."
  (let* ((initial (if suppliedp
                      (format nil "~s" initial)))
         (result (get-a-string-dialog s :initial initial)))
    (if result (list (read (make-string-input-stream result) nil)))))
```

Defines:
`get-a-value-dialog`, used in chunks 4 and 12.
Uses `get-a-string-dialog` 10b.

6.6 The `get-nonnil-value-dialog` function

12a ⟨*The get-nonnil-value-dialog function* 12a⟩≡ (4)

```
(defun get-nonnil-value-dialog (s &key (initial "" suppliedp))
  "Args: (s &key initial)
Opens a modal dialog with prompt S, a text field and OK button.
INITIAL is converted to a string with ~A format directive. Returns
string of text field content on OK. Will return a nonnil value or
take user back to top-level."
(loop
 (let* ((input (if suppliedp
                   (get-a-value-dialog s :initial initial)
                 (get-a-value-dialog s))))
   (if (some #'not input)
       (message-dialog
        "Error!!\n\nNeed a non-nil value!")
     (return input)))))
```

Defines:
`get-nonnil-value-dialog`, used in chunk 4.
Uses `get-a-value-dialog` 11b.

6.7 The `get-tested-value-dialog` function

12b ⟨*The get-tested-value-function* 12b⟩≡ (4)

```
(defun get-tested-value-dialog (s &key (initial "" suppliedp)
                                       (test #'(lambda(x) t))
                                       (error-message "Please retry"))
  "Args: (s &key initial (test #'(lambda(x) t))
                         (error-message \"Please Retry\"))
Opens a modal dialog with prompt S, a text field and OK button.
INITIAL is converted to a string with ~A format directive. Returns
string of text field content on OK. Will return a value or
take user back to top-level. "
(loop
 (let* ((input (if suppliedp
                   (get-a-value-dialog s :initial initial)
                 (get-a-value-dialog s))))
   (if (some #'not (mapcar test input))
       (message-dialog error-message)
     (return input)))))
```

Defines:
`get-tested-value-dialog`, used in chunk 4.
Uses `get-a-value-dialog` 11b.

6.8 The `get-yes-no-list` function

The function `get-yes-no-list` prompts the user for a Yes or No answer in a dialog box using a list of strings that go against the check boxes. It returns a list of `t` of `nil`'s.

13a ⟨*The get-yes-no-list function* 13a⟩≡ (4)

```
(defun get-yes-no-list (info-string list &key (default nil))
  "Args: (info-string list &key (default nil))
Prompts the user for a Yes/No answer in a dialog box. Info-string
is a informative string. List is a list of strings that go against
the check-boxes. Default is all boxes not checked."
  (let* ((info (send text-item-proto :new info-string))
         (check-items
          (if (and default (listp default))
              (mapcar #'(lambda(x)
                          (send toggle-item-proto :new x :value y))
                      list default)
            (mapcar #'(lambda(x)
                        (send toggle-item-proto :new x :value
                              default))
                    list)))
         (ok (send modal-button-proto :new "OK"
                   :action
                   #'(lambda()
                       (mapcar #'(lambda(x) (send x :value)) check-items))))
         (d (send modal-dialog-proto :new
                  (list (list info) (arrange check-items) (list ok)))))
    (send d :modal-dialog)))
```

Defines:
`get-yes-no-list`, used in chunk 4.
Uses `arrange` 9 and `:value` 6a.

6.9 The `convert-to-numbers` function

The `convert-to-numbers-function` converts a list of strings into numbers. In case of error, returns `nil`.

13b ⟨*The convert-to-numbers function* 13b⟩≡ (4)

```
(defun convert-to-numbers (list-of-strings)
  "Args: list-of-strings
Converts a list of strings into numeric values. Returns nil if stymied."
  (let ((values
         (ignore-errors (mapcar #'read-from-string list-of-strings))))
    (if (notevery #'numberp values)
        nil
      values)))
```

Defines:
`convert-to-numbers`, used in chunk 4.

6.10 The `some-files-dont-exist` function

The `some-files-dont-exist` function takes a list of file names and returns `t` if any file doesn't exist, `nil` otherwise.

14a ⟨*The some-files-dont-exist function* 14a⟩≡ (4)

```
(defun some-files-dont-exist (list-of-file-names)
  "Args: list-of-files
Returns true unless all files exist."
  (some #'not (mapcar #'probe-file list-of-file-names)))
```

Defines:
`some-files-dont-exist`, used in chunk 4.

6.11 The `coerce-to-layout-of` function

The `coerce-to-layout-of` function accepts a layout list `y` and an input list `x`, which should be a list of simple items, and not a list of lists. It returns a list of the items of `x` laid out like `y`.

14b ⟨*The coerce-to-layout-of function* 14b⟩≡ (4)

```
(defun coerce-to-layout-of (y x)
  "Args: (x y)
Returns a list of contents of x, laid out like y. Y is merely
referenced and should be either a list or a list of lists."
  (let* ((items x)
         (n (length items))
         (m (length (combine y)))
         (reversed-result ())
         (count 0))
    (unless (= m n)
            (error "list lengths are unequal."))
    (dolist (element y)
            (let ((l (if (listp element)
                         (length element)
                       1)))
              (cond
               ((listp element)
                (setf reversed-result
                      (cons
                       (select items (+ count (iseq l))) reversed-result)))
               (t
                (setf reversed-result
                      (cons (select items count) reversed-result))))
              (setf count (+ count l))))
    (reverse reversed-result)))
```

Defines:
`coerce-to-layout-of`, used in chunks 4 and 16.

6.12 The `strcat` function

Although `Lisp-Stat` documentation shows a `strcat` function, it doesn't seem to be available. Here is a trivial one that will break if too many arguments are given to it.

15a ⟨*The strcat function* 15a⟩≡ (4)

```
(defun strcat (&rest args)
  "Args: (&rest args)
Concatenates a bunch of strings and returns the result."
  (apply #'concatenate (append (list 'string) args)))
```

Defines:
`strcat`, used in chunk 4.

6.13 The `definedp` function

Obvious!

15b ⟨*The definedp function* 15b⟩≡ (4)

```
(defun definedp (symbol)
  "Method args: symbol
Returns t if symbol is defined and bound."
  (and (boundp symbol) symbol))
```

Defines:
`definedp`, used in chunk 4.

6.14 The `ask-user-for-value` function

The `ask-user-for-value` function is just for convenience. It accepts a bunch of strings and concatenates them for use as a prompt string for a dialog.

15c ⟨*The ask-user-for-value function* 15c⟩≡ (4)

```
(defun ask-user-for-value (&rest args)
  "Method args: &rest args
Asks the user to enter value in a dialog with a prompt string which
is all of args concatenated."
  (get-value-dialog (apply #'strcat args)))
```

Defines:
`ask-user-for-value`, used in chunk 4.

6.15 The `make-sliders` function

The `make-sliders` function is quite neat, in our opinion. It accepts a list of `triples` consisting of a label string, an interval as a list, and an action function that accepts one argument. It returns a multiple value result of a list of dialog items that can be passed on to `dialog-proto` and a list of `scroll-items` representing the sliders which can be used as necessary. The keyword arguments shown below allow for some fine tuning.

16 ⟨*The make-sliders function* 16⟩≡ (4)

```
(defun make-sliders (triples &key layout formats value-text-lengths
                             no-of-slider-stops)
  "Args: (triples &key layout formats value-text-lengths slider-stops)
Triples is a list consisting of triples: a label, an interval, and an
action function taking a single argument. Layout is used if
provided. Format is a list of formats suitable for value-item-proto.
Value-text-lengths defaults to a list of 10's if formats is not
specified or a list and no-of-slider-stops to a list of 51's."
  (let ((n (length triples))
        (slider-items ())
        (sliders-alone ()))
    (when (not formats)
          (setf formats (repeat "~g" n)))
    (when (not value-text-lengths)
          (setf value-text-lengths (repeat 10 n)))
    (when (not no-of-slider-stops)
          (setf no-of-slider-stops (repeat 51 n)))
    (flet ((make-item (a b c d)
                      (let* ((ti (send text-item-proto :new (select a 0)))
                             (vi (send value-item-proto :new ""
                                       :format b
                                       :text-length
                                       (if (listp b)
                                           (select b 0)
                                         c)))
                             (si (send interval-scroll-item-proto :new
                                       (select a 1)
                                       :points d
                                       :text-item vi
                                       :action (select a 2))))
                        (send si :width (+ (send ti :width)
                                           (send vi :width)))
                        (if layout
                            (list (list ti vi) si)
                          (list (list (list ti vi) si)))))))
          (setf slider-items (mapcar #'make-item triples formats
                                     value-text-lengths no-of-slider-stops)))
    (setf sliders-alone
          (if layout
              (map-elements #'select slider-items 1)
            (mapcar #'(lambda(x) (select (select x 0) 1))
                    slider-items)))
    (when layout
          (setf slider-items (coerce-to-layout-of layout slider-items)))
```

```
      (values slider-items sliders-alone)))
```

Defines:
  `make-sliders`, used in chunk 4.
Uses `coerce-to-layout-of` 14b, `:format` 7b, `value-item-proto` 6c, and `:width` 5b 5c 5d.

7 Index of Code Chunks

This list is generated automatically. The numeral is that of the first definition of the chunk.

⟨*Copyright* 3⟩
⟨*Modifications to System* 5a⟩
⟨*The arrange function* 9⟩
⟨*The ask-user-for-value function* 15c⟩
⟨*The coerce-to-layout-of function* 14b⟩
⟨*The convert-to-numbers function* 13b⟩
⟨*The definedp function* 15b⟩
⟨*The :display-value method for interval-scroll-item-proto* 7c⟩
⟨*The :format method for value-item* 7b⟩
⟨*The get-a-string-dialog function* 10b⟩
⟨*The get-a-value-dialog function* 11b⟩
⟨*The get-nonempty-string-dialog function* 11a⟩
⟨*The get-nonnil-value-dialog function* 12a⟩
⟨*The get-tested-value-function* 12b⟩
⟨*The get-yes-no-list function* 13a⟩
⟨*The :isnew method for value-item* 7a⟩
⟨*The make-sliders function* 16⟩
⟨*The ok-or-abort-dialog function* 10a⟩
⟨*The some-files-dont-exist function* 14a⟩
⟨*The strcat function* 15a⟩
⟨*The :value method for interval-scroll-item-proto* 6a⟩
⟨*The :value-item method for interval-scroll-item-proto* 8⟩
⟨*The value-item object* 6b⟩
⟨*The value-item object prototype* 6c⟩
⟨*The :width method for button-item-proto* 5c⟩
⟨*The :width method for interval-scroll-item-proto* 5d⟩
⟨*The :width method for text-item-proto* 5b⟩
⟨*Utility Package* 4⟩

8 Index of Identifiers

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.]