EQUATIONEDITOR

PROGRAMMER'S GUIDE

By: Tad Hogg (Hogg.pa@Xerox.com)

DESCRIPTION

This document describes how to define new kinds of equations to be used with the equation editor in TEdit, and how to construct equation image objects by function calls.

User Switches:

The global variable *EquationDefaultSelectionFn* specifies the default function to be called when an equation is selected with the middle mouse button. The initial value, EQIO.DefaultSelectFn, allows the user to select a piece of the equation by selecting from a menu.

The global variable *UnknownEquationData* is a formated string to use for displaying equations whose types are not defined.

The global variable *EquationInfo* is used to record all currently defined equation types. The specification information can be obtained by calling

(EQIO.GetInfo *type info*) [Function]

which returns the specified info for equations of the given type. Any of the PROPS mentioned below for EQIO.AddType, or *formFn* or *numPieces*, can be used as a value for *info* to get the corresponding data for this type of equation. Example: (EQIO.GetInfo 'fraction 'numPieces) returns the number of pieces in a fraction.

Individual specification items of an existing equation type can be modified using

(EQIO.SetInfo *type info newValue*) [Function]

although the caller must be sure that any new information is consistent with the remaining properties. For example, (EQIO.SetInfo 'fraction 'menuLabel myLabel) will make the value of myLabel be used for fractions in the equation type menu.

Defining new kinds of equations:

This module allows new types of equations to be defined. An equation consists of some number of pieces of text and, perhaps, some extra symbols or lines. A method for computing the relative position of the various pieces must be specified when new equation types are used. The pieces of equations consist of "formatted strings" which allow font information to be associated with strings and can also include image objects (which thus allows equations to contain other equations). Formatted strings are described below. Additional properties can be specified to determine the particular behavior of the new equation.

Additionally, a function can be provided to allow equations of the new type to be created under program control. Typically these are named EQ.Make.xxx where xxx = atom specifying the equation type.

Specifically, a new equation type (or a new definition for an existing type) is created by calling

(EQIO.AddType *type formFn numPieces PROPS*) [Function]

where

- *type* is an atom identifying the equation type (e.g. fraction)
- *formFn* is the name of a function, described in detail below, which specifies the relative location of the equation pieces and, if requested, draws any extra lines or symbols required by the equation that are not included in any of its pieces.The formFn can also specify the selection region to be used for each piece of the equation, i.e. that region of the equation within which the left mouse button can be used to select the piece.
- *numPieces* is the number of parts the equation has (e.g. a fraction has two parts: numerator and denominator). If the equation has a variable number of pieces, then numPieces is the default initial value.
- *PROPS* is a prop list of optional properties for the equation which are described below.

The equation form function:

The form function is called with arguments (eqnObj imageStream draw?) and specifies the size of the entire equation and the location of each piece with respect to the lower left corner of the box. Furthermore, if *draw?* is non-NIL, it draws any extra lines or symbols required by the equation that are not included in any of its pieces. (If *draw?* is NIL, nothing should be drawn -- the function is being called only to determine how big the equation is and the relative location of its parts.) The formFn can also specify the selection region to be used for each part of the equation.

For example, a fraction has two parts (numerator and denominator) and a single extra line between them. In this case the formFn draws the line and specifies the location of the numerator and denominator.

Specifically, the formFn should return an equation specification created by a call to

(EQIO.MakeSpec *box dataSpecList*) [Function]

where *box* is an IMAGEBOX which specifies the size of the equation; and *dataSpecList* is a list containing a piece specification for each piece of the equation. A piece specification is created by a call to

(EQIO.MakeDataSpec *pos selectRegion*) [Function]

where *pos* is the position, relative to the lower left corner of the box, where this piece is to be displayed and *selectRegion* gives the selection region to use for this piece (relative to the l.l. corner of the equation box). If *selectRegion* is NIL, then the region within which the piece is displayed will be used as the selection region. Normally the selectRegion should include the display region inside it, and is intended to allow selection regions to be larger than just the display region. Note that *pos* specifies where the stream should be positioned before displaying the formatted string defining the contents of the piece.

The ATTACHEDBOX routines, described below, may be useful for constructing the form functions of new equations since it provides functions to position various regions so that they do not overlap.

Equation type properties:

The following properties can be specified for any equation type when it is defined with EQIO.AddType. Note that all equations of a given type have the same values for these properties.

- *changeFn* A function with argument (eqnObj) called when the number of pieces in a variable piece equation is changed. It is meant to allow any specific properties such as saved menus to be adjusted to reflect the change.
- *initialData* A way to specify the text to be used when equations of this type are created. It can be either the name of a function or a list. If it is a list, then it should contain a single integer for each piece of the equation. This number specifies how the font size for that piece should be changed relative to the initial font size set in TEdit. For example, 0 means use the default font, +1 means use the next bigger font, -2 means use a font two sizes smaller, etc. Missing values default to zero, i.e. the pieces are initially set in the normal size font. The actual fonts used are specified by the array EquationFontSpecs and any request for a font larger (smaller) than the largest (smallest) available in the array defaults to using the largest (smallest). In this case, the initial text will be a single blank.

If initialData is a function, it is called with arguments (initialFontSpec type numPieces dataList) when a new equation of this type is created. It should return a list of formatted strings, with each item in the list to be used for the corresponding piece of the equation. The argument initialFontSpec will specify the current font when the equation is added; and numPieces will be the number of pieces in the equation. dataList, if non-NIL, is a list of items to use for each of the pieces of the equation. The items can be either format strings, or just a string in which case the initialData function should attach an appropriate font.

- *initialPropFn* A function with argument (type) called when a new equation of this type is created. It returns a prop list to be used as properties for this equation. These values are added to (and override) any props given in objectProps. For example, this allows the user to specify the number of rows and columns in a new matrix. If the number of pieces is to be specified, it should be returned as the value of the numPieces prop, and the equation type should allow a variable number of pieces.
- *makeFn* A function which constructs an image obj for this type of equation from its arguments. Generally this will just provide a convenient way of calling EQN.Make.
- *menuLabel* A label to use for this type in the equation type menu displayed after selecting Equation in the main TEdit menu. If not given, the type atom is used as the label.
- *objectProps* A prop list of properties and their initial values that are needed for this equation. These properties can be used by the formFn to lay out the equation and will typically be modified by the wholeEditFn.
- *pieceNames* A list of names of the pieces of the equation. This is used by the default middle button selection function to provide a menu of choices. E.g. ("numerator" "denominator") for a fraction. If this property is not specified, then the default menu will just contain the numbers of the pieces. This is generally meant for equations with a fixed number of pieces.
- *specialSelectFn* A function with argument (eqnObj) to be called when the equation is selected with the middle button instead of the default action. It should return the number of the piece selected, or NIL if no piece of the equation is selected. The selected piece, if any, will then be edited.
- *wholeEditFn* A function with arguments (eqnObj window button) called when the entire equation, rather than a single piece, is selected. This can be used to change global properties of

the equation and should return non-NIL if the object is modified, NIL otherwise. *window* is the window which contains the equation and *button* is the mouse button used to select it.

- *variable?* Non-NIL to indicate this equation type allows a variable number of pieces.

Equation data functions:

The functions used with new equation types should make use of the routines provided for formatted strings as well as the following functions when using the various equation data:

- (**EQIO.EqnType** eqnObj) returns the atom specifying the kind of equation *eqnObj* is.
- (**EQIO.EqnDataList** eqnObj) returns the list of formatted strings which specify the pieces of the equation.
- (**EQIO.SetDataList** eqnObj newDataList) replaces the data list (i.e. the list of formatted strings corresponding to each piece of the equation) with newDataList. The caller must update the number of pieces in the equation appropriately. This is useful, for instance, when the wholeEditFn has made major changes in the equation.
- (**EQIO.EqnData** eqnObj piece#) returns the formatted string corresponding the the piece of *eqnObj* specified by piece#.
- (**EQIO.EqnProperty** eqnObj prop {newValue}) returns the current value of the specified property of *eqnObj* if *newValue* is not present, otherwise sets the specified property to the value of *newValue* even if it is NIL. This can be used to associate arbitrary properties with individual equations. Currently, the following properties are used by the equation editor and should not be used for other purposes:
  - *fontSpec* a specification of the current font at the time the equation was created
  - *numPieces* the current number of pieces in a variable-piece equation
  - *selectionMenu* the current default middle-button selection menu for a variable-piece equation

When equations are copied, any data items that are not atoms, strings or lists are set to NIL. These items are also PRIN2'ed on files. Thus other data types should only be used to cache values (e.g. menus) that can be recomputed if necessary from the other properties.

- (**EQIO.NumPieces** eqnObj {newValue}) returns the current number of pieces in *eqnObj* if *newValue* is not present. Otherwise if *eqnObj* is a variable-piece equation, it sets the number of pieces to *newValue* and adjusts any necessary properties by calling the equation's changeFn.

Additionally, properties can be associated with the equation type itself by use of

- (**EQIO.TypeProp** type prop {newValue}) which gets or sets the property *prop* for equation *type*. This can be used to save properties that are the same for all equations of a given type (e.g. selection menus for equations with a fixed number of pieces).

Equation image objects can be created by a program (and then, for example, inserted into TEdit) by use of

- (**EQN.Make** type dataList fontSpec PROPS) which makes a *type* equation whose arguments are format strings contained in *dataList*. *fontSpec* is an initial font specification and *PROPS* is a prop list of equation properties which should include *numPieces* for equations with a variable number of pieces.

FORMATSTRINGS

The basic components of equations are represented as formatstrings which allow fonts and super/subscripting to be associated with strings and can also include image objects. A format string is a list of items, each of which is either an imageobject or a list giving a font specification, a string and an optional shift specifying the number of points to move up when displaying the string. The following functions can be used to create and display format strings as well as insert and extract them from TEXTSTREAMs. All formatstrings must be on a single line.

Functions:

For creating and accessing format strings:

(FS.MakeItem *fontSpec string shift*) [Function]

creates a formatstring item from *fontSpec*, a font specification such as (Gacha 10), *string* and *shift*. The string can be null.

(FS.ItemFont *item*) [Function]

returns the font associated with *item*, or NIL if *item* is an imageobject.

(FS.ItemValue *item*) [Function]

returns *item* if it is an imageobject, otherwise its associated string.

(FS.ItemShift *item*) [Function]

returns the shift associated with *item*.

For inserting and extracting from TEXTSTREAMs:

(FS.Extract *stream*) [Function]

returns a format string created from the text in the TEXTSTREAM stream. Any unallowed characters (as determined by FS.AllowedChar) in the stream are ignored. Note that any format information other than character fonts, super/subscripting and image objects is discarded. The file pointer associated with the stream is modified.

(FS.Insert *data stream*) [Function]

inserts the format string *data* at the current location in TEXTSTREAM stream.

For displaying and manipulating the format strings:

(FS.Box *data imageStream*) [Function]

returns an IMAGEBOX specifying the size of the format string *data* on *imageStream*.

(FS.Copy *data*) [Function]

returns a copy of the format string *data*.

(FS.Display *data imageStream invert?*) [Function]

displays the format string *data* on *imageStream*. If *invert?* is non-NIL, the display is inverted.

(FS.Get *fileStream*) [Function]

reads a formatstring, or list of formatstrings, from the current location on *fileStream*.

(FS.Put *data fileStream*) [Function]

prints the formatstring (or list of formatstrings) *data* to *fileStream*.

Additional functions:

(FS.AllowedChar *charcode*) [Function]

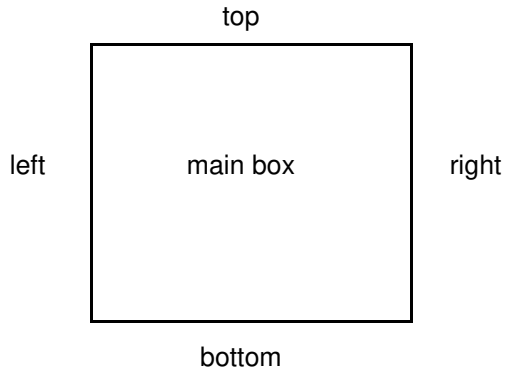
returns non-NIL if *charcode* is allowed in formatstrings.

(FS.RealStringP *item nullOK*) [Function]

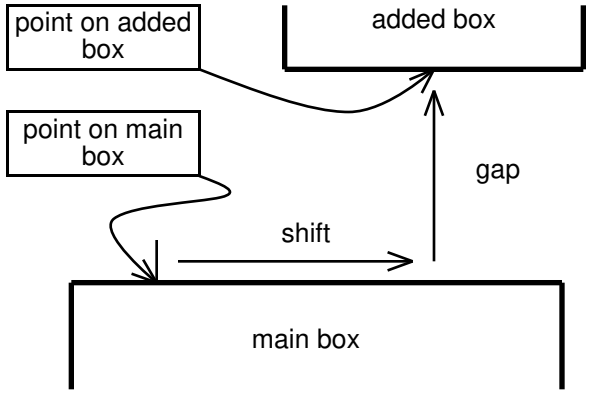
returns non-NIL if *item*'s value is a string (rather than an imageobject) and either *nullOK* is non-NIL or the string is not the nul string.

ATTACHED BOXES

The following functions place image boxes in specific locations with respect to a main box so that the added boxes won't overlap. The desired position of a new box is specified by the side of the main box to place it next to and the position of a point on the side of the new box with respect to a point on the side of the main box. The placed regions are specified with respect to the lower left corner of the main box. Sides are specified by one of the atoms *top*, *bottom*, *left* or *right* and are with respect to the main box.



The position of the added box is specified relative to some side of the main box. Specifically, the location of a reference point on the near side of the added box, the addPt, is given with respect to a reference point on the side of the main box, the mainPt. The possible points along the side are specified by one of the atoms *low*, *high*, *center* or *display* corresponding to the corner nearest the lower left corner of the box, the corner farthest from the l.l. corner of the box, the center of the side, and the display point of the image box respectively. The location of the addPt with respect to the mainPt is specified by a distance along the side, the shift, and a distance perpendicular to the side, the gap. These distances can be positive or negative. A negative value for the gap will cause the added box to overlap the main box. All boxes are assumed to have no kerning (i.e. XKERN field is zero).



Functions:

For creating and accessing format strings:

(AB.PositionRegion *mainBox addedRegions side mainPt addBox addPt gap shift clear*) [Function]

Positions *addBox* with respect to *mainBox* avoiding overlap with previously added regions. The parameters are: *mainBox* is an image box specifying the main box; *addedRegions* is a list of regions (measured with respect to the lower left corner of *mainBox*) that have already been placed next to the main box; *side* is the side (one of *top*, *bottom*, *left* or *right*) of the main box next to which the new box should be placed; *mainPt* is the reference point along the side of the main box (one of *low*, *high*, *center* or *display*); *addBox* is an image box specifying the box to be placed; *addPt* is the reference point along the side of the added box; *gap* and *shift* specify the relative positions of the reference points; and *clear* is the minimum (nonnegative) distance that the new box is allowed to be from any of the previously added regions (NIL defaults to zero which prevents any overlap of the added regions).

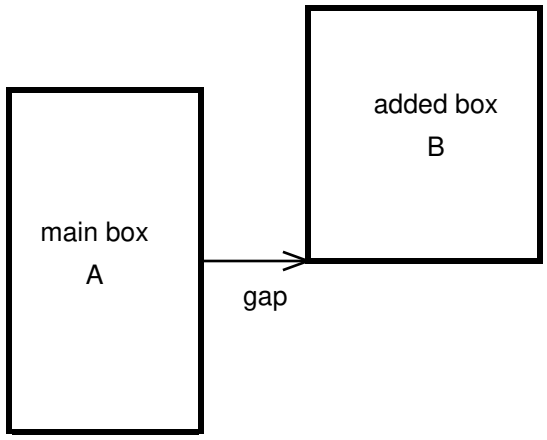
The function returns a list of the form (*region newAddedRegions*) where *region* is the region, w.r.t. the lower left corner of *mainBox*, where *addBox* was placed and *newAddedRegions* is the list of added regions updated to include this newly placed region. If the specified location of *addBox* causes it to be within a distance *clear* of any of the regions in *addedRegions*, the box is moved away from the main box in a direction perpendicular to the side (i.e. the gap is increased) until it is far enough from the previous regions.

(AB.Position2Regions *mainBox addedRegions side highBox highPt lowBox lowPt highGap lowGap highShift lowShift clear*) [Function]

This function places two boxes next to the same side of *mainBox*. If the two new boxes are within a distance *clear* of each other, they are moved apart in a direction parallel to the side next to which they are placed so that the distance each box moves is proportional to its size. Then these regions are individually checked for being too close to previously added regions and, if necessary, are moved away from the main box (i.e. perpendicular to the side). The function returns a list of the form (*highRegion lowRegion newAddedRegions*).

Example:

To place box B to the right of box A such that the low point of the near side of box B is a distance *gap* from the center of the right side of A, i.e.



use (AB.PositionRegion *A addedRegions* '*right* '*center B* '*low gap* 0) where *addedRegions* is a list of previously added regions which should not overlap *B*.