LTK - a Lisp binding to the Tk toolkit

Peter Herth

January 30, 2005

Contents

1 Introduction

Tk is a graphics toolkit for the tcl programming language developed by John Ousterhout. Initially developed for the X-Window system, it has been ported to a wide selection of operating systems, including Windows and MacOs. Due to ist ubiquous nature, its an ideal candidate to write a portable GUI library for Lisp.

While one can find many code snippets how to set up a communication with Tk from Lisp, the use of those to create actual programs, requires tcl/tk knowledge. In fact this way the GUIs are created by tcl code put into lisp programs. But one does not become a Lisp programmer to then write the GUIs in tcl/tk. So the Ltk library was born, to create a wrapper around Tk in pure Lisp. Ideally, no tcl/tk knowledge is required to write GUIs. However the lisp code is made closely to the tcl/tk library structure, so that the man pages for the tk widgets can serve as a detailled reference. They should be readable without any tcl knowledge.

The main objective for Ltk was to create a GUI library which is portable across different operating systems *and* Common Lisp implementations. Furthermore it should be easy to set up. So with the exception of one single function, the whole code of ltk is pure ANSI Common Lisp. No external programs besides a standard installation of tcl/tk are required.

Ltk supports the following Lisp systems: Allegro, CMUCL, CLisp, ECL, LispWorks, OpenMCL, SBCL. Ltk was successfully tested using Lispworks, CLisp, CMUCL, SBCL under Linux and Lispworks, CLisp, and SBCL using Mac Os X, CLisp, Allegro and Lispworks using Windows.

2 Installation

This is the shortest section of this document. You just compile the file:

```
(compile-file "ltk")
```

and load it:

```
(load "ltk")
```

Now ltk is ready to use. For trying out the examples of this document, you might want to import the package:

```
(use-package 'ltk)
```

And to look, whether it works call the test example:

```
(ltktest)
```

or, for some fun:

```
(ltk::ltk-eyes)
```

To use Ltk you need of course tcl/tk installed. This should be default on most Linux systems, for Windows/Mac Os you need to download and install tcl/tk. Ltk has been tested against Tcl/Tk 8.4, but other versions should work also. Alternatively, you can use ASDF to load ltk. If you have a symbolic link to lkt.asd in your site-systems directory a simple (require 'ltk) compiles and loads ltk (assuming you have ASDF loaded).

3 Tutorial

3.1 First steps

Let's start with the obligatory "hello world" type of program:

```
(defun hello-1()
  (with-ltk
   (let ((b (make-instance 'button
                           :master nil
                           :text "Press Me"
                           :command (lambda ()
                                      (format t "Hello World!~&")))))
     (pack b))))
```

Figure 1: The window created by `hello-1`

Let's go through it step-by-step. The whole code of `hello-1` is wrapped into the `with-ltk` macro. It ensures that the GUI library is properly set up and the communication with the Tk toolkit is established. It runs the contained code which should initialize the GUI and after that calls `mainloop` to process the GUI events.

The next step is to create the button. This is done by creating an instance of the class button. The `:text` argument gives the text to display on the button and `:command` is the function to call, whenever the button is pressed. While the last two arguments should be obvious, the `master` needs explanation. In Tk, all GUI elements are arranged in a tree. So every GUI element has a parent node or "master" which designates its position in the tree structure. So put there the object that should be the parent for your button. Only for top level components `nil` may be given instead of an object.

For displaying any ltk object, a layout manager is used. There are two layout managers available, which can be used to arrange widgets in its parent, `pack` and `grid`. `pack` arranges widgets as a heap of boxes, which are horizontally or vertically stacked. `grid` arranges widgets in a table-like layout.

NEVER use `pack` and `grid` for the same container, unpredictable behaviour may the result (or rather, the program will very predictably crash).

3.2 A more complex example

```
(defun hello-2()
  (with-ltk
   (let* ((f (make-instance 'frame))
          (b1 (make-instance 'button
                             :master f
                             :text "Button 1"
                             :command (lambda () (format t "Button1~&"))))
          (b2 (make-instance 'button
                             :master f
                             :text "Button 2"
                             :command (lambda () (format t "Button2~&")))))
     (pack f)
     (pack b1 :side :left)
     (pack b2 :side :left)
     (configure f :borderwidth 3)
     (configure f :relief :sunken)
     )))
```

Figure 2: The window created by `hello-2`

The example `hello-2` shows how you group 2 buttons within a frame and configure widgets in general. The created frame is given as the master parameter to the button creations. This automatically ensures that the buttons are packed within the frame. To change the appeareance of the Frame `f`, the `configure` function is used. This is a very generic function, which can be used upon any tk object. It takes two arguments, the name of the configuration option and the value to set into it. The value can be any tk object or any properly printable Lisp object.

In this case, we set the width of the border of the frame to 3 and make it a sunken border. Other styles would be raised, ridge, groove, flat and solid. For a comprehensive list of configuration options look in the manpage of the tk widgets as well as man options for options shared by all tk widgets.

3.3 Running it manually

While the `with-ltk` macro is the most convenient way to run Ltk, you can do it manuall, especially if you want to play with the Ltk objects in the REPL. To start Ltk you just need to call:

```
(start-wish)
```

which starts the Tk sub process and initializes the stream to communicate with it. Now you can create and use any Ltk objects. To enable event handling call

```
(mainloop)
```

which is responsible for event handling. You can interrupt it any time you like, call any lisp function and restart it again.

3.4 Special variables

The following special variables are defined:

`*debug-tk*` When `t`, the communication with `wish` is echoed to the standard output. Default value: `t`

`*wish-pathname*` The path to the executable to wish.

`*wish-args*` The arguments passed to the call to wish. Default value: `("-name" "LTK")`

`*ltk-version*` The version of the Ltk library.

3.5 Generic functions

The following generic functions are defined on widgets:

`(value` *widget*`)` Reads/sets the value of the widget. Applicable for: `check-button`, `radio-button`, `menucheckbutton`, `menuradiobutton`, `scale`.

`(text` *widget*`)` Reads/sets the text of the widget. Depending on the widget, this can be text displayed on the widget (`button`) or contained as data (`entry`). Applicable for `button`, `check-button`, `entry`, `label`, `labelframe`, `spinbox`, `text`.

3.6 The pack geometry manager

The pack geometry manager treats widges as boxes to be piled into one direction. This direction can be either horizontally or vertically. Complex layouts can be created by using frames to pack piles together.

The behaviour of the pack geometry manager is controlled by the keyword parameters to the pack function. The keywords and their effects are:

`:side` The direction in which the widgets are packed. Possible values are `:left` , `:right`, `:top` (default), `:bottom`.

`:expand` If `t`, then the packed widget may take more place than

`:fill` Allows the packed widget to grow in the given direction, if it gets expanded. Possible values are `:none` (default), `:x`, `:y` or `:both` needed.

`:after` *widget* Pack it after the widget.

`:before` *widget* Pack it before the widget

`:padx` *n* Leave *n* pixel space in x direction around the widget.

`:pady` *n* Leave *n* pixel space in y direction around the widget.

`:ipadx` *n* Grow the widget *n* pixel in x direction.

`:ipady` *n* Grow the widget *n* pixel in y direction.

`:anchor` *direction* Specify which point of the widget to use for anchoring it, for example `:ne` for the upper right corner.

3.7 The grid geometry manager

The grid geometry manager creates a table-like layout. So to arrange a widget with the grid manager, use the `grid` function with the parameters of row and column number (starting from zero). There is one keyword parameter `:sticky` which governs the widget alignment within its table cell. Its a string containing any combination of "n" "e" "s" and "w".

The behaviour of the single rows and columns of the grid are controlled by the `grid-rowconfigure` and `grid-columnconfigure` functions. Its most common use is to set the weight of a column between 0 and 1 to control the resizing behaviour.

3.8 Configuring widgets

Almost all aspects of widgets can be configured after creation via the `configure` function. It has the form: `(configure` *widget option value*`)` where widget is the widget to be configured, option the name of the option to configure (on the Lisp side a string or a keyword) and value any printable value that should be set for the option or a `tkobject`. Options used by all widgets are (not complete):

`anchor` *position* n, ne, e, se, s, sw, w, nw, center

`background` *color* Background color of the widget

`bitmap` *bitmap* Specifies a bitmap to display in the widget.

`borderwidth` *width* borderwidth in pixels

`cursor` *cursorname* Set the icon for the mouse cursor. A list of portable names is in the variable `*cursors*`.

`foreground` *color* Foreground color.

`image` *image* Photo image to be displayed on the widget.

`justify` *value* Justification of text displayed on the widget, may be left, center, or right.

`padx` *pixels* Extra padding around the widget.

`pady` *pixels* Extra padding around the widget.

`relief` *value* Effect for border display. May be raised, sunken, flat, ridge, solid, or groove.

`orient` *orientation* The orientation of the widget (e.g. for scrollbars). May be horizontal or vertical.

`takefocus` *takeit* 0 or 1, determines whether the widget accepts the focus.

`text` *string* The text to be displayed on the widget.

`underline` *index* The index of the character to underline in the text of the widget for keyboard traversal.

Example: `(configure txt :background :blue)`

4 Event handling

There are two ways to get notified by Tk events: `command` and `bind`. Widgets, which have a default event type, like pressing buttons, define a `command` initarg. With it, a function can be bound to this default event type. This is a function, that will be called with zero or one parameter, depending on the widget type. For those that use it, this parameter will be the value of the widget (example value of the scale widget).

4.1 `command`

With the `command` property a function for handling the default event type of widgets can be specified. This can be done with the `:command` initarg or the `command` accessor (settable) for those widgets. The widgets that support the `command` property are listed in table 1. The first column lists the widgets, the second which arguments the function is passed (if any) and the third one gives a brief description about when the event happens and what the arguments contain.

widget	argument	description
button	-	called when the button is clicked
check-button	value	report the value when the button is clicked
listbox	selection	a list of the selected indices (0 for first) whenever the listbox is clicked
scale	value	whenever the value is changed, called with the new value
spinbox	value	whenever the value of the spinbox is changed by the buttons, the new one is returned

Table 1: Classes with a command property and their descriptions

4.2 `bind`

A more generic and complex event type can be created via the `bind` function. With it for any widget type events can be defined, the function bound to it always needs to accept one parameter an `event` structure. Its usage is: `(bind widget event function)`[1]

A scribble example:

```
(defun scribble ()
  (with-ltk
   (let* ((canvas (make-instance 'canvas))
          (down nil))
     (pack canvas)
     (bind canvas "<ButtonPress-1>"
           (lambda (evt)
             (setf down t)
             (create-oval canvas
                          (- (event-x evt) 10) (- (event-y evt) 10)
                          (+ (event-x evt) 10) (+ (event-y evt) 10))))
     (bind canvas "<ButtonRelease-1>" (lambda (evt)
                                        (declare (ignore evt))
                                        (setf down nil)))
     (bind canvas "<Motion>"
```

[1]Currently the event has to be specified as a string as with Tk. In future releases perhaps a more Lispy style might be used.

```
(lambda (evt)
  (when down
    (create-oval canvas
         (- (event-x evt) 10) (- (event-y evt) 10)
         (+ (event-x evt) 10) (+ (event-y evt) 10)))))))
```

5 Widgets

In this section the available widgets are listed and described.

5.1 Button

Make-instance accepts the following standard keyword arguments:
`activebackground activeforeground anchor background bitmap borderwidth cursor disabledforeground font foreground highlightbackground highlightcolor highlightthickness image justify padx pady relief repeatdelay repeatinterval takefocus underline wraplength`

And the following widget-specific keyword arguments, explained in detail here:

`command`

`compound`

`default`

`height`

`overrelief`

`state`

`width`

5.2 Check-button

5.3 Radio-button

5.4 Canvas

The canvas widget is used to display all kind of graphics output. Graphic components are defined as objects like line, circle and photoimage which are displayed on the canvas. These objects can be modified through methods to change their appearance. The display and redrawing is handled by the canvas widget automatically, so that the user does not need to care for that. For convenience, ltk adds a scrolled-canvas widget which contains a canvas widget and adds automatically scrollbars to it. You gain access to the contained canvas with the `canvas` method.

A canvas widget is created by the `make-canvas` function. It has the optional arguments `width` and `height` for the width and height used to display the canvas widget. The drawing region itself can be bigger, its size is set by the `scrollregion` method, which has the canvas and the dimension as the coordinates x0 y0 and x1 y1 as parameters.

Objects to be displayed in a canvas are created via the create-xxx methods, with xxx the type of object to be created. They take the canvas as first argument and return an index (integer) which is used as handle for the modifying functions. A list of currently supported objects and the create method parameters:

`(create-arc` *canvas x0 y0 x1 y1* `:start` *a1* `:extent` *a2* `:style` *style*`)`

Creates an arc item. The arc angles are specified in starting angle and extend of the arc. So a quater circle would have an extent of 90. Style determines how the arc is rendered. Available styles are:

`pieslice` (default) Draw the ark as the slice of a pie, that is an arc with 2 lines to the center of the circle.

`chord` Draw the arc as an arc and a line connecting the end points of the arc.

`arc` Draw only the arc.

`(create-bitmap` *canvas x y* `&key` *bitmap*`)`

Creates an bitmap on the canvas, if *bitmap* is given, its displayed in this item. Special configuration options are:

`anchor` *anchorPos*

`bitmap` *bitmap*

`foreground` *color*

`background` *color*

`(create-image` *canvas x y* `&key` *image*`)`

Creates an image on the canvas, if *image* is given, its displayed in this image. Special configuration options are:

`anchor` *anchorPos*

`image` *image*

`activeimage` *image*

`disabledimage` *image*

`(create-line` *canvas coords*`)`

Coords is a list of the coordinates of the line in the form (*x0 y0 x1 y1 x2 y2 ...*). Lines with any number of segments can be created this way. Special configuration options for line items (see itemconfigure) are:

`arrow` *where* one of none (no arrow), first (arrow on first point of line), last and both.

`arrowshape` *shape*

`capstyle` *style* butt, projecting, or round.

`joinstyle` *style* bevel, miter, or round.

`smooth` *method* 0 or bezier

`splinesteps` *number* Degree of smoothness if smooth is specified.

`(create-line*` *canvas* `&rest` *coords*`)`

Like `create-line`, but the coordinates are directly given in the form *x0 y0 x1 y1 x2 y2* .

`(create-oval` *canvas x0 y0 x1 y1*`)`

Creates an oval fitting within the rectangular of the given coordinates.

`(create-polygon` *canvas coords*`)`

Similiar to create-line, creates a closed polygon.

Special configuration options for polygon items (see itemconfigure) are:

`joinstyle` *style* bevel, miter, or round.

`smooth` *method* 0 or bezier

`splinesteps` *number* Degree of smoothness if smooth is specified.

`(create-rectangle` *canvas x0 y0 x1 y1*`)`

Creates an rectangle with the given coordinates.

`(create-text` *canvas x y text*`)`

Creates a text displaying object at the position *x,y*. *Text* is the string to be displayed. Special configuration options are:

`anchor` *anchorpos*

`font` *fontname*

`justify` *justification* left,right, or center.

`text` *string*

`width` *length* Line length for wrapping the text, if 0, no wrapping happens.

`(create-window` *canvas x y widget*`)`

Embeds a widget into the canvas at the position (*x y*). The widget has to be created before with *canvas* or one of the masters of *canvas* as its master.

5.4.1 Managing graphical objects

`(set-coords` *canvas item coords*`)`

Changes the coordinate settings for any canvas item. *item* is the handle returned by the create function, *coords* is a list with the coordinates. With this function, objects can be moved or reshaped.

`(scrollregion` *canvas x0 y0 x1 y1*`)`

Set the scroll region of the canvas. *x0 y0* are the coordinates of the upper left, *x1 y1* of the lower right corner of the scroll region.

`(itemconfigure` *canvas item option value*`)`

Configure one configuration option for *item* displayed on *canvas*. Options are given as strings, value is any tkobject or printable value. Options possible for all items are:

`dash` *pattern*

`activedash` *pattern*

`disableddash` *pattern*

`dashoffset` *offset*

`fill` *color* name of a color to fill the item, or empty string for none.

`activefill` *color*

`disabledfill` *color*

`outline` *color*

`activeoutline` *color*

`disabledoutline` *color*

`offset` *offset*

`outlinestipple` *bitmap*

`activeoutlinestipple` *bitmap*

`stipple` *bitmap*

`activestipple` *bitmap*

`disabledstipple` *bitmap*

`state` *state* One of normal, disabled or hidden.

`tags` *taglist*

`width` *outlinewidth*

`activewidth` *outlinewidth*

`disabledwidth` *outlinewidth*

5.4.2 Example

The function canvastest demonstrates the basic canvas usage:

```
(defun canvastest()
  (with-ltk
   (let* ((sc (make-instance 'scrolled-canvas))
          (c (canvas sc))
          (line (create-line c  (list 100 100 400 50 700 150)))
          (polygon (create-polygon c (list 50 150  250 160 250
                                           300 50 330 )))
          (text (create-text c 260 250 "Canvas test")))
     (pack sc :expand 1 :fill :both)
     (scrollregion c 0 0 800 800)
     )))
```

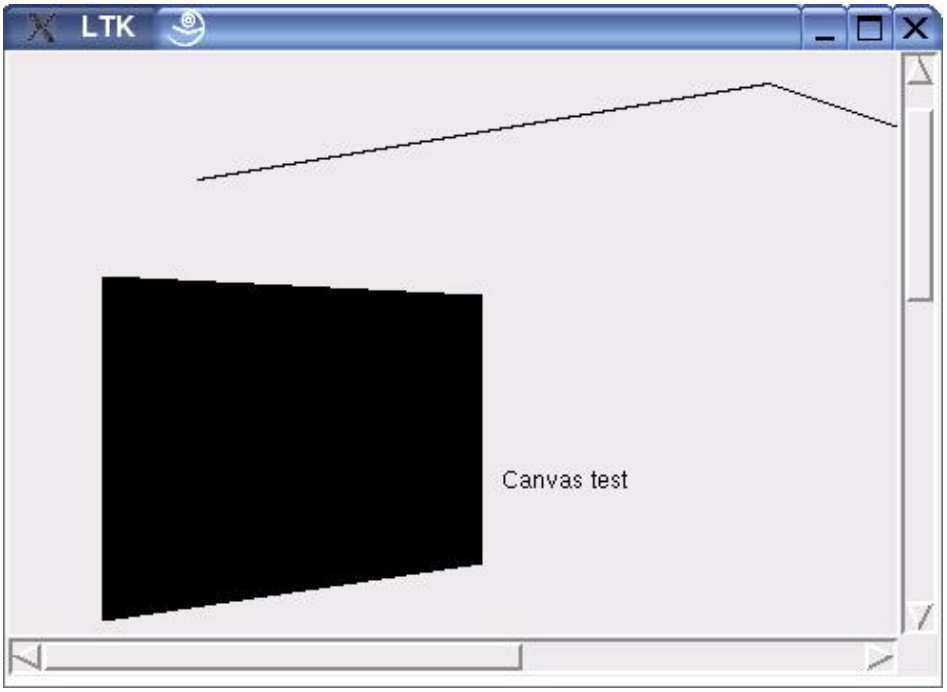

Figure 3: The window created by `canvastest`

5.5 Entry

5.6 Label

5.7 Labelframe

5.8 Listbox

5.9 Menu

5.10 Message

5.11 Paned-window

5.12 Photo-image

5.13 Scale

5.14 Scrollbar

5.15 Spinbox

5.16 Text

Methods:

`(append-text` *text txt [tag]*`)`

Appends txt to the widgets content. If given, tag is the tag to be associated with the appended text.

`(clear-text` *text*`)`

Clear the content of the widget.

`(text` *text*`)`

Get the content of the widget.

`(setf (text` *text*`)` *content*`)`

Set the content of the widget.

`(see` *text pos*`)`

Ensure that pos is within the displayed area.

`(tag-configure` *text tag option value*`)`

Configure a tag of the text widget.

`(tag-bind` *text tag event fun*`)`

Bind event to the specified tag of the widget, calling fun when the event occurs.

`(save-text` *text filename*`)`

Write the content of the widget to the file named by filename. Note: filename is a string to be interpreted by tcl/tk on the client computer.

`(load-text` *text filename*`)`

Loads the content of the widget from the file named by filename. Note: filename is a string to be interpreted by tcl/tk on the client computer.

5.17 Toplevel

6 Screen functions

`(screen-width (&optional (w nil)))`

Give the width of the screen in pixels (if w is given, of the screen the widget w is displayed on)

`(screen-height (&optional (w nil)))`

Give the height of the screen in pixels (if w is given, of the screen the widget w is displayed on).

`(screen-width-mm (&optional (w nil)))`

Give the width of the screen in mm (if w is given, of the screen the widget w is displayed on)

`(screen-heigth-mm (&optional (w nil)))`

Give the height of the screen in mm (if w is given, of the screen the widget w is displayed on)

`(screen-mouse-x (&optional (w nil)))`

Give x position of the mouse on screen (if w is given, of the screen the widget w is displayed on)

`(screen-mouse-y (&optional (w nil)))`

Give y position of the mouse on screen (if w is given, of the screen the widget w is displayed on)

`(screen-mouse (&optional (w nil)))`

Give the position of the mouse on screen as `(values x y)` (if w is given, of the screen the widget w is displayed on)

`(window-width (tl))`

Give the width of the widget in pixels. This function can be called on widgets as well as toplevel windows.

`(window-height (tl))`

Give the height of the widget in pixels. This function can be called on widgets as well as toplevel windows.

`(window-x (tl))`

Give the x position of the widget in pixels.

`(window-y (tl))`

Give the y position of the widget in pixels.

7 Window manager functions

`(wm-title toplevel title)`

Set the title of the window.

`(minsize toplevel width height)`

Set the minsize of the window in pixels. (send-w (format nil "wm minsize a a a" (path w) x y)))

`(maxsize toplevel width height)`

Set the maximum size of the window in pixels.

`(withdraw toplevel)`

Withdraw the window from display.

`(normalize toplevel)`

Set the state of the window to normal display.

`(iconify toplevel)`

Iconify the window.

`(deiconify toplevel)`

De-iconify the window.

`(geometry toplevel)`

Read the geometry string for the window.

`(set-geometry toplevel width height x y)`

Set the geometry for the window.

`(on-close toplevel fun)`

Set fun to be called whenever the close button of the window is pressed.

`(on-focus toplevel fun)`

Call fun whenever the window gets the focus.

8 Under the hood

In this section, the technical detaills of the implementation and workings of ltk are explained. Reading this section should not be neccessary to use ltk, but helps understanding it and serves as a documentation for those, who want to extend ltk.

The Tk library is a GUI library for the tcl programming language. It is used via the program `wish`. Commonly, it is used as the shell to execute tcl/tk programs. But when no script name to execute is being given, it starts in an interactive mode, using `stdin` to read commands and `stdout` to print the results. This can be used to enter the tcl commands manually in an interactive session or, as used by ltk to access `wish` from another program. Every Lisp I know of, offers a function to run a program in a subprocess and to communicate to its stdin/stdout streams. The ltk function `do-execute` wraps these platform-dependant functions in a generic one. Its parameter is the name of the program to start as a string, a list with the parameters for the program. It starts the program as a subprocess of the Lisp process and returns a two-way stream to communicate with the program. To send some text to the program, its just written into the stream, and likewise output from the program can be read from the string.

All ltk widget creation functions actually create two objects: the CLOS object to represent the widged on the Lisp side, and the corresponding Tk object.

The root class of the ltk class hierarchy is the `tkobject` class. It has only one slot: the name of the object. In tcl objects are tracked by their names, very similiarly like symbols in Lisp. To represent all widgets the `widget` class is derived from `tkobject`. It adds the slots for the object being the master of the widget and the path string for the widget. As mentioned before, all tcl objects are referenced by their name, and all tk widgets have to be put in an hierarchy. This is represented by a path-like naming system. The name of the root object is just ".". Creating a frame named `frame1` below it would lead to a path name `.frame1`. A button called `button1` placed into this frame gets the pathname `.frame1.button1`. Both the naming and the path creation is automatically handled by ltk. To create both only the reference to the master is needed. In an after-method to the `initialize-instance` method of `widget`, the name is created as an unique string and the pathname is created by appending this name to the pathname of the master widget, or ".", if the widget has no master specified. The unique name is created by

appending an upcounting number to the letter "w". Finally the method calls the `create` methode upon the new widget. This create method is, where the code interfacing with tk takes place. So to support new tk widgets, only a subclass of `widget` has to be made and a `create` method to be written.

Internally used special variables are:

`*wish*` The stream used to communicate with `wish`.

`*callbacks*` The hashtable associating widget names

`*counter*` The counter variable used to give widgets unique names (`w`n, where n is the counter variable, that gets incremented upon use).

`*event-queue*` If event messages are read while waiting for a data message they are buffered in that list.

8.1 Communication

At the startup of the wish process, some tcl helper functions are defined and then the functions in the list `*INIT-WISH-HOOK*`. These purpose of these functions is to perform initialisations, e.g. loading Tk extensions.

All communication from Tk to Lisp takes place in form of lists, which are `read`-able. The first element of the list is a keyword, which determines what kind of information is following. `:data` is the answer to a call to a function like reading out the content of a widget. `:callback` is sent upon a callback event and `:event` for an event created by the `bind` function. This design is neccessary, because events can be generated (and thus messages to Lisp sent), while Lisp is waiting for a data answer. So the function `read-data` can buffer those events until the requested data arrives. Only after the data request has been fufilled, all pending events are processed.

8.2 Writing Ltk extensions

It is difficult to give a fully generic set of instructions how to write Ltk extensions, as some part of it depends on the package that is to be wrapped, but at the example of the tix extension set, a very common case can be shown. For sake of brevity, here only the creation of a partial implementation of the tixBalloon widget is demonstrated.

The first step is to create a Lisp package to host the extension library:

```
(defpackage "LTK-TIX"
  (:use "COMMON-LISP"
        "LTK")
  (:export
        "BALLOON"
        "BALLOON-BIND"))

(in-package ltk-tix)
```

It creates a package called `ltk-tix`, based on `common-lisp`, and of course `ltk`. It exports two symbols `balloon` for the widget class to create and `balloon-bind` a function defined on this widget.

As the usage of the Tix extension requires a tcl statement to be run before any widget is used, the proper way for this would be to put it onto the `*init-wish-hook*` which is run after the startup of wish:

```
(eval-when (:load-toplevel)
  (setf *init-wish-hook* (append *init-wish-hook*
                                 (list (lambda ()
                                   (send-wish "package require Tix"))
                                 ))))
```

Now we need to create the Lisp class that wraps the balloon widget. First we need a class definition:

```
(defclass balloon (widget)
  ())
```

Unless there are some special storage needs, an empty sub-class of `widget` is all one needs. What is still missing, is the Tk code to actually create the

widget. It is put in the `initialize-instance` after-method for the widget class. This is easy to do when we look how the widget is created on the Tk side:

`tixBalloon` *pathname*

where *pathname* is the path string describing the widget to be created. This translates into Lisp code as:

```
(defmethod initialize-instance :after ((b balloon) &key)
  (format-wish "tixBalloon ~a" (path b)))
```

`path` is an accessor function created for the widget class. The corresponding slot is automatically filled in the `initialize-instance` method for the widget class. Now we can create instances of the balloon widget, what is left to do is to define the methods upon it.

We want to implement the bind command upon the balloon widget. First lets again look at the Tk side of it:

pathname `bind` *widget options*

pathname is the path of the balloon widget, *widget* is another widget for which the balloon help should be displayed and options are additional command options. The following options should be implemented:

`-msg` *text* Sets the displayed message to `text`.

`-balloonmsg` *text* Sets the balloon message to `text`.

`-statusmsg` *text* Sets the statusbar message to `text`.

To implement it, we need to define a generic function: [2]

```
(defgeneric balloon-bind (b w &key msg balloonmsg statusmsg))
```

We call this `balloon-bind` to avoid name conflicts with the function `bind` defined by the Ltk package. It is a generic function of two parameters, the balloon widget and the widget the message should be bound to. The message is to be specified by the keyword parameters. The actual implementation of the generic function is very straight forward and looks like this:

[2] It is not required to have a `defgeneric` definition for each generic function, as to the standard, `defmethod` implicitly generates the definition if they do not exist, but as SBCL issues a warning in this case and shipped code preferrably should not issue warnings on compilation, I add the defgeneric statements for all generic functions I create.

```
(defmethod balloon-bind ((balloon balloon) (widget widget)
                         &key msg balloonmsg statusmsg)
  (format-wish "~a bind ~a~@[ -msg {~a}~]~
                        ~@[ -balloonmsg {~a}~]~
                        ~@[ -statusmsg {~a}~]"
      (path balloon) (path widget) msg balloonmsg statusmsg))
```

Format wish is a wrapper around the `format` function, that sends the output to wish and automaticalls flushes the output buffer, so that the statement is directly executed by wish. It is worth noting, that the Lisp `format` function has some very nice options, allowing us to elegantly implement the optional keyword arguments. The `~@[ ~]` format directive peeks at the next argument in the list and only when it is non-nil, the format code inside is executed, otherwise, this argument will be consumed. As unspecified keyword argumends are set to nil, if no argument is specified this nicely fits to this format directive. So `~@[ -msg {~a}~]` will output nothing, if the argument `msg` is not given at the invokation of `balloon-bind`, or print " `-msg xxx`", where xxx is the content specified for the `msg` argument.

9 ltk-remote

As the connection between Lisp and tcl/tk is done via a stream, it is obvious that this connection can easily be run over a tcp socket. This allows the gui to be displayed on computers different to the one running the Lisp program. So ltk applications are not only network transparent accross different operating systems, they are actually very efficiently network transparent, since the creation of a button requires only in the magnitude of 100 bytes of network transfer. Likewise, only the generated events are transmitted back to the Lisp server.

The only difference for the lisp application to enable remote access is using the `with-remote-ltk` *`port`* macro instead of the `with-ltk` macro. As sockets are not part of the ANSI Common Lisp standard, currently only CMUCL, SBCL and Lispworks are supported by ltk-remote.

The only thing required on the client computer is tcl/tk installed and the remote.tcl script (which has less than 30 lines of code in it). Connection to the lisp process is established by

```
wish remote.tcl hostname port
```

Where hostname is the name of the computer running the lisp process and port the port on which the lisp process is listening.

10 ltk-mw

Ltk-mw is a "megawidgets" package inspired by PMW (Python Mega Widgets). It contains extension widgets for Ltk written in Lisp. Besides serving as an example, how to extend Ltk, it provides usefull new widgets listed below.

10.1 progress

A progress-bar widget. It displays a bar which covers the width of the widget in the given percentage. The widget has two settable accessor functions: `percentage` and `color`.

10.2 history-entry

History-entry is an entry widget, that provides a history of all input. The history can be browsed with the cursor-up and down keys. If the `:command` initarg is passed to `make-instance` when creating an instance of the widget, the specified function is called whenever the user pressed the `return` key. The function gets passed the text in the widget and the input field of the widget is cleared.

10.3 menu-entry

A combo-box style entry widget, that displays a menu of input content, when the user clicks the widget. The initial content for the menu is passed as a list to the `:content` initarg. To modify the menu, the generic functions `(append-item` *entry string*`)` and `(delete-item` *entry index*`)` can be used.

Index