Experimental newLISP/Tk
Version 0.1

P$_D$B

May 21, 2024

This text describes an experimental interface, designed to build Tcl/Tk GUIs controlled from newLISP, similar to python's tkinter. It uses the FOOP-Classes[1] to describe the widgets being used in the GUI.

The only Tcl/Tk geometry-manager available here is GRID.

It is a small subset of what Tcl/Tk offers, when building a GUI. Work is still under way.

The interface described here has been built and tested so far, using Linux (LMDE5[2] with the cinnamon desktop) as the underlying operating system.

Please keep in mind that the software is in an experimental stage. Likewise is this documentation in progress. Inconsistencies may occur. Please let me know if something is wrong or missing.

Contents

[1] See the newLISP-manual here: http://www.newlisp.org/downloads/manual_frame.html
[2] Based on Debian 11.2 bullseye

Part I.
Current state

May 19, 2024

License: GNU General Public License

State: *Experimental*

1. Requirements/Architecture

Status: May 12, 2024

1.1. Installation

This installation has been done on Linux (LMDE5).

newLISP	newLISP needs to be installed. The newLISP-version used is v. 10.7.5 64-bit on Linux.
`tcl`	Then we need to install Tcl/Tk including *`wish`*. Version is 8.6.11-2. *`wish`* (Windowing Shell) is a Tcl interpreter currently part of the Tcl/Tk programming suite. On Linux it is started by calling `/usr/bin/wish`.
`tk`	Version (8.6)
`tklib`	Version (0.7)
`iwidgets4`	(plus `itk3`) is needed, when a scrolledtext widget is to be used.

1.2. Architecture

A program using the interface described here is made up of 4 layers:

`<app>.lsp` top-level application to perform a GUI-based interaction with the user. Written in newLISP using `ts.lsp`, `Tk.lsp` and `msg.lsp`.

↓↑

`ts.lsp` library – it's an interface, a "translator"-layer, which translates newLISP-statements of our application into Tcl/Tk-language strings being sent to *`wish`* via `Tk.lsp`. This is the GUI *language* for our application, similar to python's tkinter which was kind of a template for this. It is written in newLISP.

↓↑

`Tk.lsp` library – the communication-"engine" used to send and receive information (waiting for output written from *`wish`*) between *`wish`* and the application via `ts.lsp`. It launches the process *`wish`* (`/usr/bin/wish`) in background mode using pipes:

```
(setq MAIN:PID (process "/usr/bin/wish" tcin tcout [errout]))
```

`Tk.lsp` is written in newLISP.

↓↑

`wish` process *`wish`* – it needs to be started first.

2. Libs

Status: May 20, 2024

The newLISP/Tk libs are/will be stored in the following Directory structure:

```
.                           # ~/.local/newLISP/newLISP-Tk/lib/
├── bin
│   └── build               # Built demos as standalone executables (empty)
├── doc                     # PDF and built library documentation (html) using newlispdoc
├── lib
│   ├── msg.lsp             # calls to Tcl/Tk-message-dialogs
│   ├── Tk.lsp              # interaction newlisp <--> wish-server
│   └── ts.lsp              # interface between newlisp and ‘wish‘ via Tk.lsp
├── README-newlisp-tk.md
└── src
    ├── demo.lsp            # hello world demo with messageBox
    └── tcltk.lsp           # tcltk-basics: change color of label
                              (pure tcltk, using pipes, no newLISP/Tk)
```

3. A short demo

... to receive an impression ...

This shows a very simple program using `ts.lsp`. It's the standard "hello world" program, which is used to show that the basic components work. It's a label and a button using the Tcl/Tk-`grid` manager here:

```
(1) #!/usr/bin/env newlisp
(2) ## Time-stamp: <2024-05-19 16:10:55 paul>
(3) (constant (global 'LIBS)
              (string (env "HOME") "/.local/newLISP/newLISP-Tk/lib/" ))
(4) (load (string LIBS "ts.lsp")) ;GUI-Server 'ts (interface)

(5) (define (MAIN:end)                      ;handler for quit-button
(6)    (let (res "no")                      ;"no"/"yes"
(7)       (while (= res "no")
(8)          (setq res
(9)                (:askyesno
(10)                (MessageBox (Title "askyesno")
(11)                            (Text
(12)                             (string "Are you sure?" ))))))
(13)      (ts:quit)))

(14) (Tk:init)

(15) (ts:setw (Window (Name "win")
                      (Title "newLISP-Tk")
                      (Minsize (Width 200) (Height 50))))
(16) (:build win)

(17) (ts:setw (Label (Name "lbl") (Text "Hello World!")))  ;define label
(18) (:build lbl)
(19) (:setgrid lbl (Row 0) (Column 0) (Padx 5) (Pady 10)) ;position

(20) (ts:setw (Button (Name "btn-end") (Text "Quit") (Command "end")))
(21) (:build btn-end)
(22) (:setgrid btn-end (Row 1) (Column 0) (Padx 5) (Pady 5))

(23) (ts:xquit)                     ;quit using the x-button of the window
(24) (Tk:mainloop )                 ;Tk: listening to in-coming events
```

In line 1[3] we tell the system where to find newLISP.

In line 4 we load `ts.lsp` from the lib-directory named in line 3. The lib `Tk.lsp` will be loaded by `ts.lsp` as well.

Lines 5...13 defines an event handler for the quit-button defined in lines 20...22. When clicking the Quit-button the is asked if he's sure. If so, `res` gets a `"yes"` and quits the program. This is here just to show a messageBox.

In line 14 Tcl/Tk (i.e. the process `wish`) is started using pipes.

In lines 15 and 16 the main window is set up (`setw` means SET Window). The `:build`-function sends it to Tcl/Tk via `Tk`.

Lines 17...19 define a label, giving it a Name and a Text. :build again sends it wish. Line 19 shows that the `grid` geometry manager is used, telling it where to place the label.

20...22: In order to quit the application we add a button in Row 1, Column 0, connecting it with the handler, defined in lines 5...13.

[3] The line numbers are not part of the program.

Line 23 makes sure that the application can be stopped by clicking the little `'x'`-button on the upper left corner of the window.

Then the event loop (infinite loop) is started in line 24 and from then on, the program will be "listening" to events done by the user (Quit-button or `'x'`) until the program is stopped.

4. A simple backup program

Status: May 10, 2024

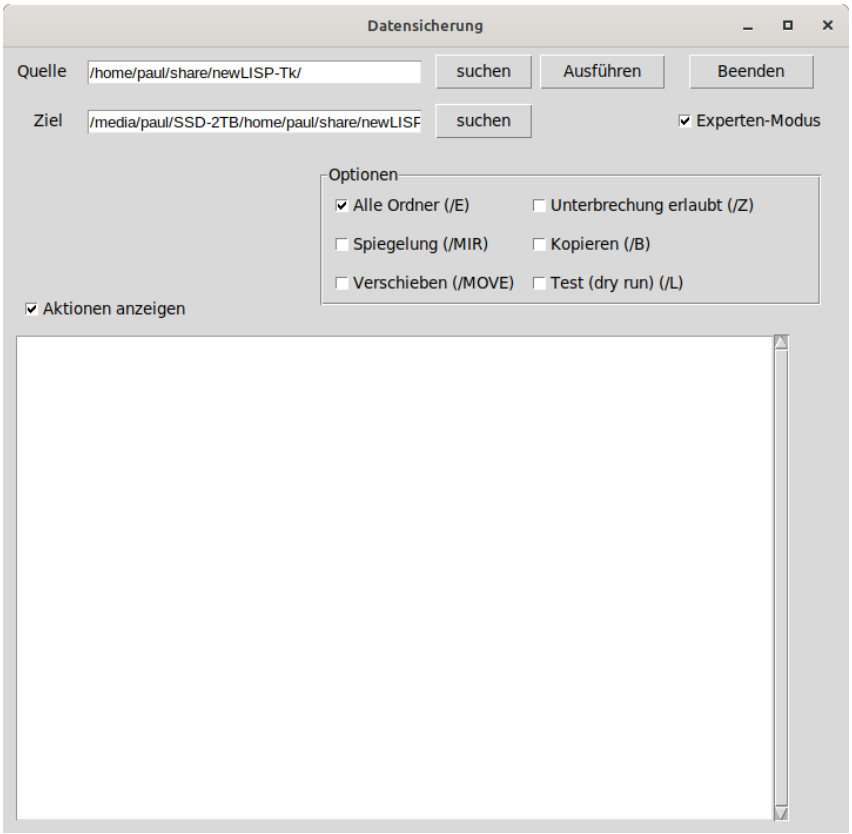
As a more extensive example, a simple backup program is described here. It makes use of the `rsync` program available on Unix-like systems. The idea for this program is to use `rsync` and part of it's options in order to perform a ***backup*** from a source-directory to a sink-directory. Building a GUI for it, this example needs a couple of widgets/elements:

- label
- entry
- button
- checkbutton
- labelframe
- frame
- scrolledtext (requires `iwidgets4`)

which, besides others, are provided by Tcl/Tk but will be indirectly used via the GUI-interface `ts.lsp` in order to make the program more readable.

The previous version was written in python, using tkinter. The GUI looked like this:

Figure 1: backup-GUI



Now we try to build it again using newLISP, `Tk.lsp, ts.lsp` and `msg.lsp`. Keep in mind that loading `ts.lsp` will also load `Tk.lsp` and `msg.lsp`:

```
(load (string LIBS "ts.lsp")) ; GUI-Server 'ts (interface)
```

Where `LIBS` is the path-string of the directory where the libs are located. Assuming `HOME` is known:

```
(constant (global 'HOME) (env "HOME"))
```

we have

```
(change-dir HOME)
(setq LIBS (real-path "./.local/newLISP/newLISP-Tk/lib/")) -->
"/home/paul/.local/newLISP/newLISP-Tk/lib/"
```

or without a **`change-dir`**:

```
(setq LIBS (real-path (string HOME "/.local/newLISP/newLISP-Tk/lib/"))) -->
"/home/paul/.local/newLISP/newLISP-Tk/lib/"
```

If **`real-path`** fails (e.g., because of a nonexistent path), **`nil`** is returned.
<<<<<<>>>>>>

5. Tk.lsp

Status: May 12, 2024

This module sets up the connection between the interface provided by `ts.lsp` and Tcl/Tk (using `/usr/bin/wish` via ***pipes***). It is a kind of an "engine" driving our car (our application) by sending and receiving statements/command-strings back and forth between `ts.lsp` and Tcl/Tk.

5.1. Communications

5.1.1. newLISP-manual

pipe: **syntax: (pipe)**
Creates an inter-process communications pipe and returns the read and write handles to it within a list.

```
(pipe) → (3 4) ; 3 for read, 4 for writing
```

The pipe handles can be passed to a child process launched via process or to fork for inter-process communications.

Note that the pipe does not block when being written to, *but it does block reading until bytes are available. A read-line blocks until a newline character is received.* A read blocks when fewer characters than specified are available from a pipe that has not had the writing end closed by all processes.[4]

[4](http://www.newlisp.org/downloads/manual_frame.html), with my italics

peek: syntax: (peek *int-handle*)
Returns the number of bytes ready to be read on a file descriptor; otherwise, it returns nil if the file descriptor is invalid. peek can also be used to check *stdin*. This function is only available on Unix-like operating systems.

read-line: syntax: (read-line [*int-file*])
Reads from the current I/O device a string delimited by a line-feed character (ASCII 10). There is no limit to the length of the string that can be read. The line-feed character is not part of the returned string. The line always breaks on a line-feed, which is then swallowed. A line breaks on a carriage return (ASCII 13) only if followed by a line-feed, in which case both characters are discarded. A carriage return alone only breaks and is swallowed if it is the last character in the stream.

By default, the current device is the keyboard (device 0). Use the built-in function device to specify a different I/O device (e.g., a file). Optionally, a file handle can be specified in the *int-file* obtained from a previous open statement.

The last buffer contents from a read-line operation can be retrieved using **current-line**.

When `read-line` is reading from a file or from *stdin* in a CGI program or pipe, it will return `nil` when input is exhausted.

current-line: syntax: (current-line)
Retrieves the contents of the last **read-line** operation.

5.1.2. Own observations

- **`read-line`** breaks (stopps reading) when a `"\n"`-character is encountered. So, there might be more in the pipe.
- As a result **`read-line`** will clear the buffer where (**`current-line`**) is looking at (let's call it current-line-buffer) and put the new contents into it.
- **`read-line`** blocks until there is something in the pipe `Tk:myin`.
- it seems that a (`wish`-) pipe never gets exhausted. So read-line blocks until something shows up in our in-pipe.
- if it reads something, it will put it into the current-line-buffer from where (**`current-line`**) can read it. But (**`current-line`**) will not clear that buffer. So the content of that buffer will stay there until the next **`read-line`** occurs – even if we use an **`eval-string`**:

```
(eval-string (current-line))
```

Try: In some cases we don't read the answer from `wish` because we just don't need it. In order to clear the pipe, one or more **`read-line`**s are needed.

But: while **`read-line`** is blocking, nothing else can happen. So we need to check wether our pipe contains any more unread bytes (characters – wish always sends strings). Here **`peek`** comes into play. It's our "look-ahead" into the pipe. So, before we ask `wish`, to give us an answer, we need to clear our in-pipe `myin` thus synchronizing with `wish`. In newLISP-code:

```
(1) (while (> (peek myin) 0) (read-line myin)) ;clear pipe
```

After that, `(= (peek myin) 0)` is telling us that the pipe is "clean" – we are synced. Well, while `(> (peek myin) 0)` holds, our **`read-line`** shouldn't block so now we can send out our question-string:

```
(2) (write-line myout "<question-string>")
```

or

```
(2) (Tk "<question-string>")
```

Now we read the answer until `myin` is empty:

```
(3) (while (> (peek myin) 0)
      (read-line myin)
      (eval-string (current-line))
      ;;do something else
      )
    ;; pipe should be empty now
```

If **(current-line)** results in an empty string, **eval-string** will return **nil**.
<<<>>>

5.2. Functions

5.2.1. Tk:init

Status: May 17, 2024

Initializes the communication between `Tk:Tk` and the process *wish*. It sets up at least two pipes and starts the process *wish*.

```
(map set '(myin tcout) (pipe))
(map set '(tcin myout) (pipe))
(setq MAIN:PID (process "/usr/bin/wish" tcin tcout ))
```

5.2.2. Tk:Tk

This is the Default-Functor for the `(context 'Tk)` and the basic sending-machine. It mainly does a `write-line` into the pipe `myout`. So it's a *sender only*.

```
(write-line Tk:myout <Tcl/Tk-string>)
```

It is called like this:

```
(Tk "wm geometry . +600+100")
(Tk "wm title    . {Label-Test}")
(Tk "ttk::label .lbl -text \"Field 1\" -borderwidth 2 -relief solid")
(Tk "grid .lbl -row 0 -column 0")
```

5.2.3. Tk:mainloop

We need to get the information coming from the GUI, from *wish* using newLISP's **read-line** function and interpret it using newLISP's **eval-string**:

```
(while (read-line Tk:myin)
   (eval-string (current-line)))
```

This is an infinite loop, waiting for an event to occur. If there is a widget with a `command` attached to it, this command (a newLISP-function defined in our application) will be called, when the user triggers an event by clicking or entering the appropriate information into the GUI. Function **current-line** retrieves the contents of the last **read-line** operation.

The command-function must be defined in context **'MAIN**, because the tcl-string requires the command-function to be a **(MAIN**:<function)>.

6. ts.lsp

Module `ts.lsp` defines an interface between the users application and Tcl/Tk. It needs to load other modules:

```
(load "Tk.lsp")  ;(context 'Tk)
(load "msg.lsp") ;interface to Tcl/Tk message boxes
```

They are both loaded when `ts.lsp` is loaded. Module `Tk.lsp` is described in section Tk.lsp.

6.1. Widgets as Classes

Status: May 11, 2024
Widgets and their Options are defined as ***FOOP-Classes***[5].

> "As a convention, it is recommended to start class names in upper-case to signal that the name stands for a namespace." (manual)

The main class here is `'Window`:

```
(new Class 'Window)
```

All widgets are defined as sub-classes, like

```
(new 'Window 'Label)
(new 'Window 'Button)
(new 'Window 'Entry)
(new 'Window 'Checkbutton)
...
```

In order to enable structured widget-descriptions, all widget-options are defined as FOOP-classes themselves:

```
(new Class 'Name)
(new Class 'Width)
(new Class 'Height)
(new Class 'Title)
(new Class 'Text)
(new Class 'Padx)
(new Class 'Pady)
(new Class 'State)
...
```

A widget description for a CheckButton then might look like:

```
(Checkbutton (Name 'cbOptions)
             (Text "Use Options")
             (Variable "useOptions")
             (State "normal")        ;or: (State "disabled")
             (Width 12)              ;characters
             (Command "cbtn-show")
)
```

The option `(State "normal")` e.g. here is an anonymous object of class `'State`. There is a function **`ts:setw`** (SET Widget), which lets us define such a description:

[5] See the newLISP-manual here: http://www.newlisp.org/downloads/manual_frame.html

```
(ts:setw (Checkbutton (Name 'cbOptions) (Text "Use Options")
                      (Variable "useOptions")
                      (State "normal") ;(State "disabled")
                      (Width 12) (Command "cbtn-show") ))
```

This function not only establishes the widget-description, but also creates a new newLISP-symbol from the Name-option given. In this example a newLISP-symbol with the name `cbOptions` exists now and the widget-description shown, is assigned to it. Accessing one of it's options is done using the assoc-function:

```
(assoc Width (self)) → (Width 12)
```

giving the complete Width-object or

```
(last (assoc Width (self))) → 12
```

if we need to get the value of Width. This is done within `ts.lsp`, when building a Tcl/Tk-string being sent to `wish`.

6.2. Classes and SubClasses

6.2.1. Window – Methods

All methods defined here are inherited to all subclasses, unless they define their own methods.

`Window:build` Status: May 11, 2024.

syntax: `(:build obj)`

This method builds the Tcl/Tk-string being sent to the `wish`-server via `Tk.lsp` using the given object *`obj`*. The example describes how `:build` works with a main window object.

Example:

```
> (ts:setw (Window (Name 'win) ;main window
                   (Title "backup")
                   (Minsize (Width 720) (Height 250))))
(Window (Name "win") (Title "backup") \
        (Minsize (Width 720) (Height 250)))

> (:build win) ;will send it to Tk like this:
wm title . "backup"
wm minsize . 720 250
```

`Window:build-tk-name` Status: May 13, 2024

syntax `(:build-tk-name obj)`

Starting from *`obj`*, this method uses the Name and Parent information of the objects used in the Name-Parent-chain to build a name using the Tcl/Tk-syntax.

Example: Assuming a window, frame and a label is given, this method produces a Tcl/Tk-name:

```
(ts:setw (Window (Name "win") (Title "MyTitle")
                 (Minsize (Width 100) (Height 50))))

> frm
(Frame (Parent win) (Name "frm"))
```

```
> lbl
(Label (Parent frm) (Name "lbl") (Text "Field 1"))

> (:build-tk-name lbl)
".frm.lbl"                  ;--> Tcl/Tk-name
```

If the frame does not exist, but is named in the definition of the label, then the frame's name is used, but searching along the parent-name-chain is stopped here.

Window:setgrid Status: May 11, 2024

syntax (:setgrid *obj obj-row obj-column* [*obj-padx obj-pady*])

This method adds a grid position to (self). This stored position will be unique: the last one given, will be used.

Notice: The order of the parameters are interchangeable.

Optionally there can be `padx` and `pady` pix-values given, in order to have some space around the widget. The 'sticky'-option may also be used.

example

```
> (ts:setw (Label (Name 'lbl_quelle) (Text "Quelle"))) ;define label
(Label (Name "lbl_quelle") (Text "Quelle"))

> (:setgrid lbl_quelle (Row 0) (Column 0) (Padx 5) (Pady 10))
```

This will lead to the following Tcl/Tk-string:

```
"grid .lbl_quelle -row 0 -column 0 -padx 5 -pady 10"
```

Window:delgrid Status: May 11, 2024

syntax `(:delgrid` *`obj`*`)`

Method `:delgrid` deletes the grid position from *`obj`*.

Window:import Status: May 20, 2024

syntax `(:import` *`obj pos str-file-name`*`)`

parameter *obj:* widget (entry)
parameter *pos:* position "0"... or "end"
parameter *str-file-name:* string denoting a file path

Imports the content of a text file named in *`str-file-name`* into the widget described by *`obj`*. Normally a `Scrolledtext` area.

example

```
(:import tx "end" "readme.txt")
;; --> tcltk:  ".tx import \"readme.txt\" end"
```

Window:insert Status: May 11, 2024

syntax `(:insert obj num-ind str-val)`

Insert Text in *str-val* into an Entry-widget *obj*, starting at position *num-ind*.

example

```
(:insert line_edit_z 0 "Text")
```

Window:erase Status: May 11, 2024

syntax `(:erase obj num-from [num-to | "end"])`

Delete Text from an Entry-widget, starting at *num-from* up to *num-to* or the last position denoted by "*end*".

example

```
(:erase line_edit_q 0 "end")
```

6.2.2. Label – Methods

Label:build Status: May 11, 2024

syntax `(:build obj)`

`Label:build` actually knows Name and Text options given with *obj*. This method overwrites the one defined in class `'Window`.

example

```
> (ts:setw (Label (Name 'lbl_source) (Text "source")))

> (:build lbl_source)
ttk::label .lbl_source -text "source"
```

6.2.3. Button – Methods

Button:build Status: May 11, 2024

syntax `(:build obj)`

`Button:build` actually knows Name, Text and Command options given with *obj*. This method overwrites the one defined in class `'Window`.

example

```
> (ts:setw (Button (Name 'bt) (Text "Quit") (Command "puts (exit); exit")))
(Button (Name "bt") (Text "Quit") (Command "puts (exit); exit"))

> (:build bt)
"ttk::button .bt -text \"Quit\" -command {puts (exit); exit}"
```

6.2.4. Entry – Methods

`Entry:build` Status: May 11, 2024

syntax `(:build obj)`

`Entry:build` actually knows Name and Width (number of characters) options given with *`obj`*. This method overwrites the one defined in class `'Window`.

example

```
> (ts:setw (Entry (Name 'line_edit_q) (Width 40)))
> line_edit_q
(Entry (Name "line_edit_q") (Width 40))

> (:build line_edit_q)
"ttk::entry .line_edit_q -width 40"
```

6.2.5. Checkbutton – Methods

Status: May 11, 2024

```
(new 'Window 'Checkbutton)
```

`Checkbutton:build` Status: May 11, 2024

syntax `(:build obj)`

`Checkbutton:build` actually knows Name, Text, Variable, Width, State and Command options given with *`obj`*. This method overwrites the one defined in class `'Window`.

example

```
(ts:setw (Checkbutton (Name 'cbtn) (Text "Expert Mode")
                      (Command "hide_unhide_controls")))
(:build cbtn)
ttk::checkbutton .cbtn -text "Experten-Modus" \
            -command {puts \"(MAIN:hide_unhide_controls)\"}

(ts:setVar "useOptions" "0")
(ts:setw (Checkbutton (Name 'cbOptions) (Text "Use Options")
                      (Variable "useOptions")
                      (State "normal") ;(State "disabled")
                      (Width 12)
                      (Command "cbtn-show")
                      ))
(:build cbOptions)
ttk::checkbutton .cbOptions -text "Use Options" -variable useOptions \
          -width 12 -state normal -command {puts "(MAIN:cbtn-show)"}
```

`Checkbutton:select-it`

syntax `(:select-it obj)`

Select the Checkbutton described in *`obj`*. As shown in the above example (`Checkbutton:build`), this requires a variable bound to that button using a Tk-variable – see section Tk-variables on page 18.

`Checkbutton:deselect-it` Status: May 11, 2024

syntax `(:deselect-it` *`obj`*`)`

Deselect the Checkbutton described in *`obj`*. This requires a variable bound to that button using a Tk-variable – see section Tk-variables on page 18.

example

```
(ts:setw (Checkbutton (Name 'cbOptions) (Text "Use Options")
                      (Variable "useOptions")
...
))
(:deselect-it cbOptions)
"set useOptions 0"
```

6.2.6. Labelframe – Methods

Status: May 11, 2024

```
(new 'Window 'Labelframe)
```

Labelframe:build Status: May 11, 2024
Builds a frame with a label attached to it, to hold other widgets.

syntax `(:build obj)`

example

```
> (ts:setw (Labelframe (Name 'lf) (Text "Optionen")
                       (Width 20) (Height 100)
                       (Labelanchor "nw")))
(Labelframe (Name "lf") (Text "Optionen") \
            (Width 20) (Height 100) \
            (Labelanchor "nw"))

> (:build lf)
"ttk::labelframe .lf -text \"Optionen\" \\
                     -width 20 -height 100 -labelanchor nw"
```

6.2.7. Not yet done

- (new 'Window 'Frame)
- (new 'Window 'Radiobutton)
- (new 'Window 'Menubutton)
- (new 'Window 'Listbox)
- (new 'Window 'Combobox)
- (new 'Window 'Notebook)
- ...

6.3. Functions

6.3.1. ts:ask-directory

Use the Tcl/Tk-`tk_chooseDirectory`-dialog to ask for a directory-path. Returns the path name of the directory chosen:

```
> (setq source-dir (ts:ask-directory)) ;--> "/home/paul/share/newLISP-Tk"
```

6.3.2. ts:quit

It just sends a

```
(Tk "puts \"(exit)\"; exit")
```

to `wish`, stopping the application.

6.3.3. ts:setw

syntax `(ts:setw obj-widget-definition)`

The *obj-widget-definition* contains a list of nested objects. Method `ts:setw` "set widget" creates a newLISP symbol derived from the parameter given as `(Name name)`, holding the *obj-widget--definition.*

example

```
;; define a label with name "lb1"
> (ts:setw (Label (Name "lb1") (Text "Feld 1")))
(Label (Name "lb1") (Text "Feld 1"))

> lb1 ; a new newLISP-variable
(Label (Name "lb1") (Text "Feld 1"))
```

6.3.4. ts:setVar & ts:getVar

– see section Tk-variables–
<<<<<<<<<<<May 21, 2024>>>>>>>>>>>

6.4. Tk-variables

Status: May 10, 2024

Certain Widgets use ***Tk-variables*** to hold their state. A `checkbutton` for instance has two states, `selected` (represented as an integer=1) and `deselected` (represented as an integer=0). In order to store, retrieve and manage such Tk-variables we need to be able to read/write these variables. Once they get connected to widgets, the connection should work both ways: if the variable changes, the value of the widget should also get updated with the new value. Using a variable which holds the state of the widget, enables us to change it from anywhere in our program.

The module `ts.lsp` creates a hash-table `TkVar` to hold those variables:

```
(define TkVar:TkVar)        ;create a hash-table, holding all Tk-variables
```

6.4.1. ts:setVar

Status: May 17, 2024

```
(define (ts:setVar name val)
   "send name and set it's value to 'val' "
   (if (not (and (string? name) (string? val)))
       (throw-error "ts:setVar arguments must both be of type string"))
   (TkVar name val)                    ;now we know it's value
   (Tk "set " name " " val)            ;send it to Tk
   );ts:setVar
```

6.4.2. ts:getVar

Status: May 19, 2024 – o.k.

syntax `(ts:getVar` *`str-name`*`)`

Send a variable name and get it's actual value from Tk. ts:getVar: *str-name* must be a string.

example

```
(setq checked (ts:getVar "cbtn_expert_var"))
```

7. msg.lsp

8. next

- complete `msg.lsp`
- add more widgets and options
- options:
  - justify
- try to make it run on the Windows10 platform

.

Index