View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarlyCommons@Penn

University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS)

Department of Computer & Information Science

January 1994

The Jack Lisp API Version 1.1

Welton Becket
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation

Welton Becket, " The Jack Lisp API Version 1.1", . January 1994.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-94-01.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/242
For more information, please contact repository@pobox.upenn.edu.

The Jack Lisp API Version 1.1

Abstract

The Lisp interface to Jack will allow general programming of Jack internals and should simplify all forms of Jack development. It will be distributed with Jack-5.7 and will allow users without source code to extend or modify Jack, and users with source code to have a high-level, object-oriented, interactive prototyping environment. After prototyping in lisp, developers can rewrite their code in C++ if speed is crucial, otherwise, code can be left in lisp to simplify maintenance and to insure upward compatibility with future Jack versions.

Comments

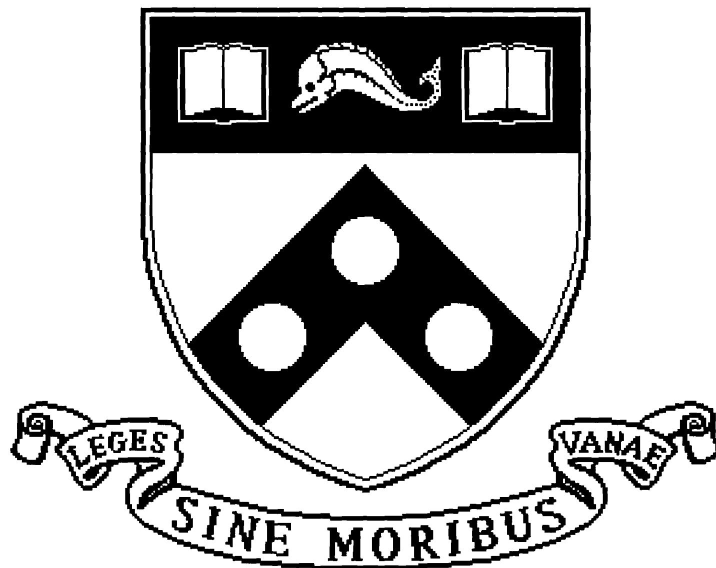
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-94-01.

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/242

The *Jack* Lisp API
Version 1.1

MS-CIS-94-01
HUMAN MODELING & SIMULATION LAB 59

Welton Becket



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

February 1994

The *Jack* Lisp API
Version 1.1

Welton Becket

February 7, 1994

1 XLISP-STAT

The Lisp interface to Jack will allow general programming of Jack internals and should simplify all forms of Jack development. It will be distributed with Jack-5.7 and will allow users without source code to extend or modify Jack, and users with source code to have a high-level, object-oriented, interactive prototyping environment. After prototyping in lisp, developers can rewrite their code in C++ if speed is crucial, otherwise, code can be left in lisp to simplify maintenance and to insure upward compatibility with future Jack versions.

The incorporation of Lisp into Jack will make development easier for a variety of reasons:

- Using an interpreted language avoids having to compile code— the Jack compile and link cycle can take up to 5 minutes. Changing or adding lisp code to a running Jack process can be done almost instantaneously and without exiting Jack.
- Lisp is a much higher level language than C or C++. It supports symbol and list processing, higher-order functions, lexical closures, and a variety of other advanced programming techniques. Data structures are also very easy to use in lisp— lisp manages memory automatically and has built-in lists, vectors, and matrices.
- Lisp will insulate code from internal Jack changes, making lisp code upward-compatible from one version of Jack to a later version.

The particular lisp we have chosen is XLISP-STAT, developed by Luke Tierney and David M. Betz. It has the following features:

1. It implements a large subset of common-lisp.
2. It has a relatively small image (unlike the public domain full common-lisps).
3. It has built in object-oriented extensions— a prototyping mechanism, which is a generalization of class hierarchies.
4. It has built-in math (including vector and matrix operations), statistical, and interactive graphing functions.
5. It provides the standard common-lisp debugger with stepping, tracing, stack printing, and variable printing or modifying.
6. It runs on under Microsoft Windows, on Macintoshes, on Amigas, and on any machine with X-windows and a C compiler. This will make it possible to do lisp development without running Jack.
7. It has no liscencing fees whatsoever: permission to "use, modify, distribute, and sell" XLISP-STAT is granted without fee or restriction.

2 The current lisp interface

The lisp interface as it will appear initially in Jack-5.7 has the following features:

1. **Unified compilation** with with Jack, with efficient, bidirectional callouts— lisp can send and receive information from Jack, and Jack can execute lisp code, get or set the value of variables, etc...
2. Set of approximately 100 **basic primitives** for accessing and modifying low level Jack internals (documentation in Section 6):
   - Direct access to most major Jack datatypes: Figures, Segments, Joints, Constraints, and Sites. Attributes, Textures, and Psurfs will be done in the 2nd Quarter 93.
   - Low level access to the motion system
3. A **JCL callout mechanism** that allows any JCL or Peabody expression to be evaluated. There are several functions for making construction of JCL strings easier. Basic documentation on this feature is in Section 4.
4. **Lisp shells around all Jack motions** (including human, figure, and timed motions) that allow higher-level construction. Each motion call takes only a few arguments and uses defaults for the rest which can be overridden with keyword arguments. There is online documentation for each of these which is also printed below in Section 5.
5. **User input functions** from within lisp, like input-string or input-figure, that call the standard, familiar Jack input procedures.
6. **Output functions** that print to the three Jack windows.
7. **Callback functions** that are executed on every Jack frame. This allows lisp code that is self-time-slicing to run in the background while Jack is running. This is described in Section 3.3
8. **Motion callback functions** similar to the standard Callback functions, but executed only while the motion system is running. These can be used to modify the animation as it progresses, pasting new motions to be executed or killing motions in progress. This is described in Section 3.3.
9. **Lisp motions** that are actual motions in the motion system. The apply, preaction, postaction, etc, are pieces of lisp code. New lisp motion types can be created and destroyed dynamically.
10. **Lisp menu functions**. There are functions that at run-time can add items to the Jack menus with lisp code attached to them. This will allow calling user-defined code interactively without having to type at the lisp-prompt (completely insulating intended users of the user-defined code from the lisp system).

I am currently adding the following to the lisp environment, all of which should be done by the end of the 2nd Quarter 93:

- An **Object-oriented Parallel Process Programming Language** in Lisp that allows general action sequencing within the Jack motion system and my new Simulation system. The actions can be *reactive*, because they can respond to unanticipated events or failures, and they can be *event-driven*, beginning, ending, or changing on various conditions. Users define subclasses of the `ActionControler` class modifying it to invoke Jack Motions or other Actions as an animation is progressing. There are facilities for:
  1. **Branching and looping** using arbitrary lisp code as tests.
  2. **Binding** persistent variables local to an ActionControler instance.

3. **Waiting for events** before proceeding. You can wait for a certain time, for another action or a jack motion to finish, or for a lisp expression to become true.
4. **Semaphores and monitors** for synchronizing actions running in parallel and for managing resources.
5. **Exceptions handlers**— whenever a lisp expression evaluates to true during the course of an action, execute a piece of action code.
6. **Jumping** to a failed or success state.

Because this process language is formulated as an operating system, we will be able to add standard deadlock detection, deadlock prevention, and resource allocation algorithms.

- **A class hierarchy encasing all Jack objects** in lisp that will allow higher-level, better protected access to the Jack internals. This hierarchy will be identical to the hierarchy currently used in the Behavioral Simulation system written in C++, which will make pushing Lisp code into C++ much easier.
- **Remote Lisp Control**— An xlisp process running either in Jack or stand-alone on any machine can have bi-directional communication with another xlisp.

3 Using XLISP-STAT

XLISP can currently be accessed from within Jack at the user level from a lisp prompt, through menu items bound to lisp code, or from callback functions executed on every Jack frame or every animation frame. The current executable is `becket/bin/lispjack`, and to use it you must have the OBJECTJACK environment variable set as:

```
OBJECTJACK=/home1/becket/xlispstat/
```

On startup, the file `$(HOME)/.jack.lisp` is read and executed if it exists.

NOTE: In addition to the user-level interface to XLISP, there is also a code-level interface where lisp strings constructed in **C** or **C++** can be executed, and there is a socket interface which allows bidirectional communication between a remote process and the lisp within a running Jack process. These are currently undocumented.

3.1 The XLISP prompt

The command `xlisp` will start an xlisp prompt in the shell where Jack was invoked. Control is not returned to Jack until the user explicitly leaves xlisp by typing `CNTRL-D` at the xlisp prompt. While using the xlisp prompt, the Jack window is not updated unless a user interface function is called. The lisp function `REDRAW` can also be used to explicitly redraw all Jack windows.

The command `xlisp string` asks for a lisp string in the status window. The string is then executed in lisp and the result is printed on the Jack overlay planes.

3.2 Menu Items

New menu items can be added to Jack that execute lisp code when invoked. The basic approach is to use:

1. `(MENU-BEGIN)` — this destroys the old Jack menus including any previous lisp menu items commands defined using MENU-ADD. It then rebuilds the standard Jack menus.
2. `(MENU-END)` — rebuilds the Jack command structure and rebuilds the displayed menus.

3. **(MENU-ADD (MENU-GET)** *name-str code***)** — this adds a new command to the end of the top-level menu with the given name and bound lisp code. **MENU-GET** currently only returns the top level Jack menu, though later it will be extended to return a submenu of a menu. The code can access global variables or can have local persistent variables using lexical closures. An example without variables:

```
(menu-begin)
(menu-add (menu-get) "Fred" '(format t "Fred must have goat cheese or he will die.~%"))
(menu-end)
```

   This will add the command **Fred** to the menus and to jcl, that will print a string to the shell where Jack was invoked. An example with bound variables:

```
(menu-begin)
(menu-add (menu-get) "Sam"
    (let ((count 0))
       (list #'(lambda ()
                  (format t "Frame: ~a~%" count)
                  (setf count (1+ count))))))
(menu-end)
```

3.3 Callback Functions

Callback functions can be bound that are executed on every Jack frame (like a background simulation function), or just on every motion system frame (only while an animation is running).
Use:
  (BIND-BG-CALLBACK *namestr code***)**
to bind code executed on every frame, and use:
  (BIND-MOTION-CALLBACK *namestr code***)**
to bind code only executed during an animation. The *namestr* argument can be any lisp object that is unique for all callbacks defined (otherwise the previous one with the same object is overwritten). To unbind a callback function, use:
  (UNBIND-BG-CALLBACK *namestr***)**, and
  (UNBIND-MOTION-CALLBACK *namstr***)**
to unbind individual callbacks, or:
  (UNBIND-BG-CALLBACKS), and
  (UNBIND-MOTION-CALLBACKS)
to delete all callbacks of the appropriate type. The *code* argument for callback functions is the same as for **MENU-ADD** described in Section 3.2.

4 JCL calls

The lowest level access to JCL (and Peabody in general) is through the `rawjcl` function. This takes a string argument and executes it as peabody code. JCL commands must end with a semi-colon, and all embedded quotes must be preceded with a '\' so the parser knows where the whole jcl string ends. For example:
  `(rawjcl ''move_figure(\''cube\'',trans(0,100,0));'')`
will execute supplied JCL command assuming there is a figure named **cube**.

The `jcl` function provides slightly higher level access. The first argument to `jcl` must be the name of the JCL function as a string or a symbol, and the rest of the arguments should be the arguments of the JCL call. The `jcl` function puts parentheses around the argument list, commas between each argument, and a semi-colon at the end. Any argument passed is converted to a string before being stuffed in the jcl call. The above call would then be:

`(jcl 'move_figure ''\''cube\'''' ''trans(0,100,0)'')`

Several functions are available for building strings for common jcl constructs:

`(js` *str*`)`

Embeds the argument in quotes and returns as a string. If *str* is a symbol, converts it to a string in lower-case. For example: `(js 'cube)` returns `''\''cube\''''`.

`(jtrans` *x y z*`)`

Returns a jcl translation string from the three arguments, which can be strings or numbers. For example, `(jtrans 0 100 ''0'')` returns `''trans(0,100,0)''`.

`(jxyz` *x y z*`)`

Returns a jcl rotation string from the three arguments, which can be strings or numbers. For example, `(jxyz 0 90 ''0'')` returns `''xyz(0,90,0)''`.

`(jmult` *a b*`)`

Returns a jcl product string of *a* and *b*. For example, `(jmult (jxyz 0 90 0) (jtrans 0 100 0))` returns `''xyz(0,90,0)*trans(0,100,0)''`.

`(junits` *n unitstr*`)`

Appends *n* (which should be a string or a number) with *unitstr*, which should be a string or symbol specifying units. For example, `(junits 2.0 ''sec'')` returns `''2.0sec''`.

The above example now becomes:

`(jcl 'move_figure (js 'cube) (jtrans 0 100 0))`

This becomes most useful when using variables, for example if the name of the figure is in the variable `fig` and the height is in the variable `y`, we can execute:

`(jcl 'move_figure (js fig) (jtrans 0 y 0))`

Other useful functions for constructing jcl strings are the builtin functions `strcat` and `format` (all of the above functions are defined in terms of these).

5 Motion Shells

All of these shells construct a JCL string that is executed in Jack to paste the appropriate motion in the timeline. Any argument to any of these functions can be the special symbol `$`, to have JCL call for user input. For example, instead of executing:

`(move-figure 'cube 1.0 2.0 (jtrans 0 100 0))`

which would call for a figure motion for cube from time 1sec to 2sec to the location (0,100,0), you could pick the location and the time interactively like:

`(move-figure 'cube $ $ $)`

```
------------------------------------------------------------------------------

                              HUMAN MOTIONS

------------------------------------------------------------------------------
ARM-MOTION                                                    [function-doc]
Syntax: (arm-motion hstr start end leftp mat
                       &key (ref 'palm) (relativep nil) (weight 1.0)
                           (wf 'const) (vc 'ease))
   hstr  -- name of human (string)
```

```
   start -- start time in seconds
   end   -- ending time in seconds
   leftp -- t for left, nil for right
   mat   -- a 4x4 matrix, a jcl matrix string ("xyz(...)*trans(...)" or
            $ to generate an input-transform. See mat-to-str.
   :ref  -- one of 'forearm 'palm or 'attached, 'palm by default.
   :relativep -- nil by default, pass as t for relative
   :weight    -- weight of this motion (increase for higher priority).
   :wf   -- weight function, see wf-to-str
   :vc   -- velocity control, see vc-to-str
-------------------------------------------------------------------------------
COM-MOTION                                                       [function-doc]
Syntax: (com-motion hstr start end mat
    &key (how 'location) (weight 2) (wf 'decay) (vc 'ease))
   hstr  -- name of human (string)
   start -- start time in seconds
   end   -- ending time in seconds
   how   -- one of 'between, 'location for "between feet"
            or "location" motion types.
   :weight    -- weight of this motion (increase for higher priority).
   :wf   -- weight function, see wf-to-str
   :vc   -- velocity control, see vc-to-str
-------------------------------------------------------------------------------
EYE-MOTION                                                       [function-doc]
Syntax: (eye-motion hstr start end sitenm &key (vc 'ease))

   hstr   -- name of human (string)
   start  -- start time in seconds
   end    -- ending time in seconds
   sitenm -- name of the site to look at
   :vc    -- velocity control, see vc-to-str
-------------------------------------------------------------------------------
FOOT-MOTION                                                      [function-doc]
Syntax: (foot-motion hstr start end leftp mat
     &key (height 5.0) (pos-orient 0.5)
  (weight 2.0) (wf 'const) (vc 'ease))
   hstr  -- name of human (string)
   start -- start time in seconds
   end   -- ending time in seconds
   leftp -- t for left, nil for right
   mat   -- a 4x4 matrix, a jcl matrix string ("xyz(...)*trans(...)" or
            $ to generate an input-transform. See mat-to-str.
   :height -- height of the foot motion in cm
   :pos-orient -- position/orientation weight in [0,1] (0 for position
            only, 1 for orientation only).
   :weight    -- weight of this motion (increase for higher priority).
   :wf   -- weight function, see wf-to-str
   :vc   -- velocity control, see vc-to-str
-------------------------------------------------------------------------------
HEEL-MOTION                                                      [function-doc]
Syntax: (heel-motion hstr start end leftp height
```

```
                    &key (weight 2.0) (wf 'decay) (vc 'ease))
   hstr  -- name of human (string)
   start -- start time in seconds
   end   -- ending time in seconds
   leftp -- t for left, nil for right
   height -- height of the heel motion in cm
   :weight    -- weight of this motion (increase for higher priority).
   :wf   -- weight function, see wf-to-str
   :vc   -- velocity control, see vc-to-str
-------------------------------------------------------------------------------
PELVIS-MOTION                                                   [function-doc]
Syntax: (pelvis-motion hstr start end mat
       &key (weight 5.0) (wf 'decay) (vc 'ease))
   hstr   -- name of human (string)
   start  -- start time in seconds
   end    -- ending time in seconds
   mat    -- a 4x4 matrix, a jcl matrix string ("xyz(...)*trans(...)" or
             $ to generate an input-transform. See mat-to-str.
   :weight    -- weight of this motion (increase for higher priority).
   :wf   -- weight function, see wf-to-str
   :vc   -- velocity control, see vc-to-str
-------------------------------------------------------------------------------
TORSO-MOTION                                                    [function-doc]
Syntax: (torso-motion hstr start end angles
                     &key (tp 'bend) (vc 'const))
   hstr   -- name of human (string)
   start  -- start time in seconds
   end    -- ending time in seconds
   angles -- list of angles as (f a l) for the default human:
              f = flexion in [-51.78,84.02]
              a = axial in [-42.63,42.27]
              l = lateral in [-38.94,39.46]
   tp     -- 'bend to bend from waist or 'curl to curl from neck.
   vc     -- velocity control, see vc-to-str


-------------------------------------------------------------------------------

                              FIGURE MOTIONS
-------------------------------------------------------------------------------
CAMERA-MOTION                                                   [function-doc]
Syntax: (camera-motion wstr start end mat &key (vc 'ease))

   hstr  -- name of human (string)
   start -- start time in seconds
   end   -- ending time in seconds
   mat   -- a 4x4 matrix, a jcl matrix string ("xyz(...)*trans(...)" or
            $ to generate an input-transform. See mat-to-str.
   :vc   -- velocity control, see vc-to-str
-------------------------------------------------------------------------------
FIGURE-MOTION                                                   [function-doc]
```

```
Syntax: (figure-motion figstr start end mat &key (vc 'ease))
   figstr  -- name of a figure (string)
   start -- start time in seconds
   end   -- ending time in seconds
   mat   -- a 4x4 matrix, a jcl matrix string ("xyz(...)*trans(...)" or
            $ to generate an input-transform. See mat-to-str.
   :vc   -- velocity control, see vc-to-str
```

```
JOINT-MOTION                                                    [function-doc]
Syntax: (joint-motion fstr start end displacements &key vc)

   hstr  -- name of human (string)
   start -- start time in seconds
   end   -- ending time in seconds
   displacements -- a list of angles, one for each degree of freedom,
            or a $ for user input.
   :vc   -- velocity control, see vc-to-str
```

TIMED CONTROLS

```
TIMED-BALANCE-CONTROL                                           [function-doc]
Syntax: (timed-balance-control hstr start end
        &key (btype 'follow))
   hstr   -- name of human (string)
   start  -- start time in seconds
   end    -- ending time in seconds
   :btype -- one of:
               follow  hold-pos  hold-elev  release-elev  seated  upper
            for
              "follow feet" "hold current position"
              "hold current elevation" "release elevation"
              "seated" "use upper body"
```

```
TIMED-FOOT-CONTROL                                              [function-doc]
Syntax: (timed-foot-control hstr start end
                            &key (which 'both) (foot-type 'hold-global))
   hstr   -- name of human (string)
   start  -- start time in seconds
   end    -- ending time in seconds
   :which -- 'left, 'right, or 'both
   :foot-type -- one of:
              release  hold  hold-global  hold-local  follow-balance
            for
              "release" "pivot" "hold global location"
        "hold local location" "follow balance line"
```

```
TIMED-HAND-CONTROL                                              [function-doc]
Syntax: (timed-hand-control hstr start end
```

```
       &key (which 'both) (hand-type 'hips)
(start-joint 'shoulder))
   hstr  -- name of human (string)
   start -- start time in seconds
   end   -- ending time in seconds
   :which -- 'left, 'right, or 'both
   :hand-type -- one of:
             hips  knees  hold-global  hold-local  site  release
   :start-joint -- 'shoulder or 'waist
------------------------------------------------------------------------------
TIMED-HEAD-CONTROL                                              [function-doc]
Syntax: (timed-head-control hstr start end
                            &key (eye-type 'fixate) (goal-type '(hold)))
   hstr  -- name of human (string)
   start -- start time in seconds
   end   -- ending time in seconds
   :eye-type -- 'fixate or 'release
   :goal-type -- list of arguments defining the goal. Passes directly
            to goal-to-str.
------------------------------------------------------------------------------
TIMED-PELVIS-CONTROL                                            [function-doc]
Syntax: (timed-pelvis-control hstr start end
                      &key (pelvis-type 'hold))
   hstr  -- name of human (string)
   start -- start time in seconds
   end   -- ending time in seconds
   :pelvis-type -- one of: 'hold or 'follow, for "hold orientation"
            or "follow feet".
------------------------------------------------------------------------------
TIMED-TORSO-CONTROL                                             [function-doc]
Syntax: (timed-torso-control hstr start end
     &key (torso-type 'vertical)
          (bend-type 'bend))

   hstr  -- name of human (string)
   start -- start time in seconds
   end   -- ending time in seconds
   :torso-type -- one of: 'vertical, 'hold, or 'none. For "vertical",
            "hold orientation", or "none".
   :bend-type -- one of: 'curl or 'bend for "curl from neck" or
            "bend from waist".
```

6 Lisp Primitives

These functions are the lowest level interface to Jack from lisp. They provide direct access to Jack data structures and functions. They are very fast, but relatively unprotected.

```
-----------------------------------------------------------------------

                          UTILITY CALLOUTS
```

SUBSTR

```
USAGE: (substr string sub-str)
NOTES:  If sub-str is a sub-string of string, returns the rest
        of the string at the point where substr first appears.
        Otherwise returns nil.
```

JCL CALLS

RAWJCL

```
USAGE: (rawjcl &rest strings)
NOTES: raw jcl callout mechanism. Executes the strings, which must
       already end in semicolons.  Returns TRUE if all of the jcl calls
       succeeded.
```

EXPLICITLY REDRAWING JACK

REDRAW

```
USAGE: (REDRAW)
NOTES: no args, redraws all jack windows
```

A NOTE ON POINTERS

In these low-level access functions, all jack objects are refered to by actual pointers. When you call (figure-create "cube.pss"), a fixnum (long integer) is returned that is the address of the created figure. These pointers are currently unprotected, and there are no checks to see if the object pointed to is of the correct type (which can cause core dumps very easily). Soon there will be a whole class hierarchy mechanism over these low-level functions that will offer both pointer protection and higher-level access.

FIGURE ACCESS FUNCTIONS

FIGURE-CREATE

```
USAGE: (figure-create filename &optional figname)
NOTES: First arg is the name of the file to read, figname
       is optional and is the name the figure will have.
       Currently works only for reading psurf files. Read
       figure files using the jcl command read_file.
----------------------------------------------------------------------
FIGURE-TRANSFORM

USAGE: (figure-transform figptr &optional matrix-4x4)
NOTES: First arg must be a figure pointer. If there is no second arg
       it gets the figure transform and returns it as a 4x4 matrix.
       If there is a second arg, it must be a 4x4 matrix that is to
       be the new transform for the figure.
----------------------------------------------------------------------
FIGURE-BBOX

USAGE: (figure-bbox figptr)
NOTES: Returns the bounding box as two 3-vecs in a list (min max).
----------------------------------------------------------------------
FIGURE-XZRADIUS

USAGE: (figure-xzradius figptr)
NOTES: Returns the radius of the objects bounding cylinder in the
       global xz plane.
----------------------------------------------------------------------
FIGURE-LOCALBBOX

USAGE: (figure-localbbox figptr)
NOTES: Returns the axis-aligned bbox of the figure in its local
       space as two 3-vecs in a list (min,max).
----------------------------------------------------------------------
FIGURE-ROOTSITE

USAGE: (figure-rootsite figptr)
NOTES: Returns a pointer to the rootsite for the figure.
----------------------------------------------------------------------
FIGURE-COM

USAGE: (figure-com figptr &optional globalOrLocal)
NOTES: If globalOrLocal is present, it must be 'global or 'local.
       Default is 'global.

----------------------------------------------------------------------
FIGURE-SEGMENT

USAGE: (figure-segment figptr &optional segname)
NOTES: if figname is given, it must be a string that is the name
       of the segment within figptr to return. Otherwise, returns
       all segments in figptr in a list.
----------------------------------------------------------------------
```

```
FIGURE-SITE

USAGE: (figure-site figptr &optional sitename)
NOTES: If sitename, it should be a string that is the name of
       the site within figptr to return. Otherwise a list of all sites
       is returned.
----------------------------------------------------------------------
FIGURE-JOINT

USAGE: (figure-joint figptr &optional jointname)
NOTES: If jointname, it should be a string that is the name of
       the joint within figptr to return. Otherwise returns a list
       of all joints. Returns nil if joint not found or no joints.
----------------------------------------------------------------------
FIGURE-FIND

USAGE: (figure-find &optional figname)
NOTES: If figname present, it should be a string that is the
       name of the figure within the env to return. Otherwise,
       returns a list of all figures in the environment.
----------------------------------------------------------------------
FIGURE-NAME

USAGE: (figure-name figptr)
NOTES: Returns the name of the figure as a string.
----------------------------------------------------------------------
FIGURE-ON

USAGE: (figure-on figptr1 figptr2)
NOTES: Takes two figures and describes the on relationship
       between them -- returns:
          'FIG1-ON-FIG2  -- if fig1 on fig2
          'FIG2-ON-FIG1  -- if fig2 on fig1
          'TOUCHING      -- if fig1 and fig2 intersect but there is no
                            'physically based' on relationship.
          'NOT-TOUCHING  -- if no intersections at all.
       Does not work on humans.
----------------------------------------------------------------------
FIGURE-TORQUE

USAGE: (figure-torque figptr &optional initp)
NOTES: Re-evaluates torque for a figure. If initp true, initializes
       torque for a figure (which MUST be done once for a figure --
       it's done by the torque window so no need to do it if
       there's a torque window on the figure).

----------------------------------------------------------------------

                         SEGMENT ACESS FUNCTIONS
----------------------------------------------------------------------
```

```
SEGMENT-TRANSFORM

USAGE: (segment-transform segptr)
NOTES: Returns the segment transform as a 4x4 matrix. The first
       three entries is the global position vector, the upper 3x3
       is the rotation matrix.
----------------------------------------------------------------------
SEGMENT-BBOX

USAGE: (segment-bbox segptr)
NOTES: Returns the segment's global bounding box as two 3-vectors
       in a list, ( #(xmin ymin zmin) #(xmax ymax zmax) ).

----------------------------------------------------------------------
SEGMENT-XZRADIUS

USAGE: (segment-xzradius segptr)
NOTES: Returns the radius of the segment in the xz plane.
----------------------------------------------------------------------
SEGMENT-LOCALBBOX

USAGE: (segment-localbbox segptr)
NOTES: Returns the segment's local bounding box as two 3-vectors
       in a list, ( #(xmin ymin zmin) #(xmax ymax zmax) ).
----------------------------------------------------------------------
SEGMENT-ROOTSITE

USAGE: (segment-rootsite segptr)
NOTES: Returns a pointer to the segment's root site.
----------------------------------------------------------------------
SEGMENT-COM

USAGE: (segment-com segptr &optional space)
NOTES: If space is present, it should be either 'global or 'local
       to say whether to get the center of mass in global or
       local coordinates (default is 'global).  The center of mass
       is returned as a 3-vector.
----------------------------------------------------------------------
SEGMENT-SITE

USAGE: (segment-site segptr &optional sitename)
NOTES: Returns the site or nil if sitename present, otherwise
       returns a list of pointers to all sites in the segment.
----------------------------------------------------------------------
SEGMENT-NAME

USAGE: (segment-name segptr)
NOTES: Returns name of the segment as a string or nil if no name.
----------------------------------------------------------------------
SEGMENT-FULLNAME
```

USAGE: (segment-fullname segptr)
NOTES: Returns full name of the segment as a string.

SEGMENT-FILENAME

USAGE: (segment-filename segptr)
NOTES: Returns the filename for a segment as a string,
       or nil if none.

SEGMENT-ATTRIBUTES

USAGE: (segment-attributes segptr &optional n newattr)
NOTES: If n and newattr not present, returns a list of all attributes
       for segptr. If n present returns attribute n for segptr. If
       n and newattr, sets attribute n for segptr to newattr.
       Returns nil if the segment has no psurf. On an attribute set,
       returns t if the set was successful or nil if not (out of bounds).
       On an attribute get, returns the attribute on success, or nil on
       out of bounds.

SEGMENT-HASGEOM

USAGE: (segment-hasgeom segptr)
NOTES: Returns t if the segment has a psurf, nil otherwise.

SEGMENT-NNODES

USAGE: (segment-nnodes segptr)
NOTES: NOT IMPLEMENTED YET.

SEGMENT-NODES

USAGE: (segment-nodes segptr)
NOTES: NOT IMPLEMENTED YET.

SEGMENT-NATTRS

USAGE: (segment-nattrs segptr)
NOTES: NOT IMPLEMENTED YET.

SEGMENT-ATTRS

USAGE: (segment-attrs segptr)
NOTES: NOT IMPLEMENTED YET.

SEGMENT-NEDGES

USAGE: (segment-nedges segptr)
NOTES: NOT IMPLEMENTED YET.

SEGMENT-EDGES

```
USAGE: (segment-edges segptr)
NOTES: NOT IMPLEMENTED YET.
------------------------------------------------------------------------
SEGMENT-NFACES

USAGE: (segment-nfaces segptr)
NOTES: NOT IMPLEMENTED YET.
------------------------------------------------------------------------
SEGMENT-FACES

USAGE: (segment-faces segptr)
NOTES: NOT IMPLEMENTED YET.
------------------------------------------------------------------------
SEGMENT-ADDSITE

USAGE: (segment-addsite segptr name location)
NOTES: Creates a new site in segptr with the given name and local
       location. If name not unique for segptr it will make a new name
       based on the given name. Returns a pointer to the new site.
------------------------------------------------------------------------
SEGMENT-FIGURE

USAGE: (segment-figure segptr)
NOTES: Returns a pointer to the segment's figure.
------------------------------------------------------------------------
SEGMENT-MASS

USAGE: (segment-mass segptr &optional newmass)
NOTES:  If newmass, sets the segment's mass (in grams). Otherwise, returns
        the current mass of the segment in grams.
------------------------------------------------------------------------
SEGMENT-HIGHLIGHT

USAGE: (segment-highlight segptr onP)
NOTES: Turns segment highlighting on and off (pass onP as t for
       highlight on, nil for highlight off).


------------------------------------------------------------------------

                       JOINT ACCESS FUNCTIONS
------------------------------------------------------------------------

JOINT-NAME

USAGE: (joint-name jptr)
NOTES: Returns the name of the joint as a string.
------------------------------------------------------------------------
JOINT-FULLNAME
```

```
USAGE: (joint-fullname jptr)
NOTES: Returns the full name of the joint as a string.
```

JOINT-FIGURE

```
USAGE: (joint-figure jptr)
NOTES: Returns a pointer to a joint's figure.
```

JOINT-FROM

```
USAGE: (joint-from jptr)
NOTES: Returns a pointer to the 'from' site of a joint.
```

JOINT-TO

```
USAGE: (joint-to jptr)
NOTES: Returns a pointer to the 'to' site of a joint.
```

JOINT-ROOTSITE

```
USAGE: (joint-rootsite jptr)
NOTES: Returns a pointer to the root site of a joint.
```

JOINT-NDOFS

```
USAGE: (joint-ndofs jptr)
NOTES: Returns the number of degrees of freedom of a joint.
```

JOINT-AXIS

```
USAGE: (joint-axis jptr &optional n)
NOTES: Returns the axis for deggree of freedom n for a joint
       as a 3-vec, or nil if a bound error on n.
       Returns a list of all axes if n not present.
```

JOINT-TYPE

```
USAGE: (joint-type jptr &optional n)
NOTES: Returns #\r or #\t for translational or rotational joint
       for degree of freedom n (nil if bound error). If n not
       present, returns a list of all types.
```

JOINT-ANGLE

```
USAGE: (joint-angle jptr &optional (n newangle) | newangles)
NOTES: If no optional args returns list of angles for each dof
       in radians. If n present returns just the angle for that dof
       or nil on a bound error. If n and newangle, sets the
       dof angle returning t on success or nil on failure.
       If the optional arg is a list, it is assumed to be a list
       of new angles for the dofs for the joint and returns
```

```
       t or nil on success or failure for the set.
----------------------------------------------------------------------
JOINT-ULIMIT

USAGE: (joint-ulimit jptr &optional (n newulimit) | ulimits)
NOTES: If no optional args, returns upper limits for each dof
       for joint jptr as a list. If n present as a fixnum,
       returns upper limit for dof n or nil on a bound error.
       If n and newulimit, sets upper limit n to newulimit
       returning t on success, or nil on failure.
       If optional arg is a list, it is assumed to be a list
       of numbers that will be the new upper limits for each dof
       (the list must be of the correct size).
----------------------------------------------------------------------
JOINT-LLIMIT

USAGE: (joint-llimit jptr &optional (n newllimit) | llimits)
NOTES: See j_jointulimit().
----------------------------------------------------------------------
JOINT-TORQUE

USAGE: (joint-torque jptr &optional n)
NOTES: Gets the torque for degree of freedom n for jptr. Value
       is undefined if torque computation has not been setup
       (though probably 0.0). Returns a list of torques
       if n not present.
----------------------------------------------------------------------
JOINT-PREVANGLE

USAGE: (joint-prevangle jptr &optional n)
NOTES: Returns previous angle for dof n or a list of previous
       angles if n not present.
----------------------------------------------------------------------

                      SITE ACCESS FUNCTIONS
----------------------------------------------------------------------

SITE-NAME

USAGE: (site-name sptr)
NOTES: Returns the name of the site as a string.
----------------------------------------------------------------------
SITE-FULLNAME

USAGE: (site-fullname sptr)
NOTES: Returns the full name of the site as a string.
----------------------------------------------------------------------
SITE-SEGMENT

USAGE: (site-segment sptr)
NOTES: Returns a pointer to the segment that owns the site.
```

```
-----------------------------------------------------------------------
SITE-JOINTS

USAGE: (site-joints sptr)
NOTES: Returns a list of pointers to joints attached to the site.
-----------------------------------------------------------------------
SITE-CONSTRAINTS

USAGE: (site-constraints sptr)
NOTES: Returns a list of pointers to constraints attached to the site.
-----------------------------------------------------------------------
SITE-ROOTJOINT

USAGE: (site-rootjoint sptr)
NOTES: Returns a pointero to the root joint for the site.
-----------------------------------------------------------------------
SITE-DISPLAY

USAGE: (site-display sptr &optional onP)
NOTES: If onP, it should be nil to turn the display of the site
       off, or anything else to turn it on.
       If onP not present, returns the current state of the
       site display (t or nil).
-----------------------------------------------------------------------
SITE-LOCATION

USAGE: (site-location sptr &optional space)
NOTES: Returns global site location by default as a 3-vector.
       If space present, it must be 'global or 'local and the return
       value will be global or local to the segment.
-----------------------------------------------------------------------

                        JACK INPUT FUNCTIONS
-----------------------------------------------------------------------

INPUT-STRING

USAGE: (input-string prompt &optional default)
NOTES: Prompt must be a string. Default, if present, must be
       nil for no default or a string. Returns 'cancel if the user
       ^c'd or a string on success.
-----------------------------------------------------------------------
INPUT-INFILE

USAGE: (input-infile prompt &optional default)
NOTES: Prompt must be a string.  Default, if present, must be nil
       for no default or a string. Returns 'cancel if the user ^c'd,
       or a string on sucess.
-----------------------------------------------------------------------
INPUT-OUTFILE
```

```
USAGE: (input-outfile prompt &optional default extension)
NOTES: Prompt must be a string. Default, if present, must be nil
       for no default or a string. Extension, if present, must be
       a string that is the extension for the file. If the file
       already exists, asks the user if they want to overwrite.
       Returns 'cancel if the user ^c'd, or a string on sucess.
------------------------------------------------------------------------
INPUT-BOOLEAN

USAGE: (input-boolean prompt &optional default)
NOTES: Prompt must be a string. Default, if present, should
       be nil for default of FALSE or anything else for a
       default of TRUE. Returns t or nil on success, or 'cancel
       if the user ^c'd.
------------------------------------------------------------------------
INPUT-FLOAT

USAGE: (input-float prompt &optional default)
NOTES: Prompt must be a string. Default, if present, should be
       a number (if not present it's set to 0.0). Returns
       A flonum on success, or 'cancel if the user ^c'd.
------------------------------------------------------------------------
INPUT-INT

USAGE: (input-int prompt &optional default)
NOTES: Prompt must be a string. Default, if present, should be
       a number (if not present it's set to 0.0). Returns
       A flonum on success, or 'cancel if the user ^c'd.
------------------------------------------------------------------------
INPUT-VECTOR

USAGE: (input-vector prompt (default|size))
NOTES: Prompt must be a string. The second arg must be a vector
       of floats for a default vec, or a size if there is no particular
       default. Returns a vector on success, or 'cancel if the user ^c'd.
------------------------------------------------------------------------
INPUT-TRANSFORM

USAGE: (input-transform prompt &optional default)
NOTES: Prompt must be a string. Default, if present, must
       be a matrix. Returns the matrix on success, or 'cancel
       if the user ^c'd.
------------------------------------------------------------------------
INPUT-NAMEDTYPE

USAGE: (input-namedtype prompt choices &optional default)
NOTES: Prompt must be a string. Choices must be a list of alternative
       strings. Default, if present, must be a string or nil for no
       no default. Returns the chosen string on success, or
       'cancel if the user ^c'd.
------------------------------------------------------------------------
```

```
INPUT-LISP

USAGE: (input-lisp prompt &optional default)
NOTES: gets a lisp expression and returns the unevaluated result.
       Prints the lisp expr in 'default' as the default value,
       if it's present.
       Returns 'cancel if user ^c'd and 'error on a read error.
-----------------------------------------------------------------------
INPUT-FIGURE

USAGE: (input-figure prompt)
NOTES: Prompt must be a string.  Asks the user to select a figure
       and returns a pointer to the figure.
-----------------------------------------------------------------------
INPUT-SEGMENT

USAGE: (input-segment prompt)
NOTES: Prompt must be a string.  Asks the user to select a segment
       and returns a pointer to the segment.
-----------------------------------------------------------------------
INPUT-SITE

USAGE: (input-site prompt)
NOTES: Prompt must be a string.  Asks the user to select a site
       and returns a pointer to the site.
-----------------------------------------------------------------------
INPUT-JOINT

USAGE: (input-joint prompt)
NOTES: Prompt must be a string.  Asks the user to select a joint
       and returns a pointer to the joint.
-----------------------------------------------------------------------
INPUT-ATTRIBUTE

USAGE: (input-attribute prompt)
NOTES: Prompt must be a string.  Asks the user to select an attribute
       and returns a pointer to the attribute.
-----------------------------------------------------------------------
INPUT-NODE

USAGE: (input-node prompt)
NOTES: Prompt must be a string. Asks the user to select a node.
       Returns a list whose first elem is the segment ptr and whose
       second is the node index.
-----------------------------------------------------------------------
INPUT-EDGE

USAGE: (input-edge prompt)
NOTES: Prompt must be a string. Asks the user to select an edge.
       Returns a list whose first elem is the segment ptr and whose
       second is the edge index.
```

```
----------------------------------------------------------------------
INPUT-FACE

USAGE: (input-face prompt)
NOTES: Prompt must be a string. Asks the user to select a face.
       Returns a list whose first elem is the segment ptr and whose
       second is the face index.
----------------------------------------------------------------------

                    OUTPUT MESSAGES TO JACK
----------------------------------------------------------------------

OUTPUT_LOGMSG

USAGE: (output-logmsg str type)
NOTES: Calls LogMsg(). Prints the string to the log window. The
       String can have newlines. Type must be one of:
          'normal
          'highlight
          'reverse
          'underline
          'highlight-reverse
----------------------------------------------------------------------
OUTPUT-STATMSG

USAGE: (output-statmsg str)
NOTES: Prints a status message to the status line.
----------------------------------------------------------------------
OUTPUT-STATERROR

USAGE: (output-staterror str)
NOTES: Prints an error message to the status line and beeps.
----------------------------------------------------------------------
OUTPUT-STATPAUSE

USAGE: (output-statpause str)
NOTES: Output a message to the status line and pauses.
----------------------------------------------------------------------
OUTPUT-INFOMSG

USAGE: (output-infomsg str)
NOTES: Print a message in the overlay planes.
----------------------------------------------------------------------

                          MENU FUNCTIONS
----------------------------------------------------------------------


MENU-BEGIN

USAGE: (menu-begin)
```

```
NOTES: Initiates rebuilding of the menus. Any menu changes should
       take place between (menu-begin) and (menu-end).
----------------------------------------------------------------------
MENU-END

USAGE: (menu-end)
NOTES: Called after changes to menus are finished. Rebuilds the
       menus and the command list.
----------------------------------------------------------------------
MENU-GET

USAGE: (menu-get)
NOTES: Currently this function takes no args and only returns
       a pointer to the main jack menu. Later it may be extended
       to find a menu inside another menu...
----------------------------------------------------------------------
MENU-NEW

USAGE: (menu-new name)
NOTES: Returns a new, empty menu with the given name.
----------------------------------------------------------------------
MENU-ADD

USAGE: (menu-add into (menu | (name action)))
NOTES: 'into' must be a pointer to a menu.
       If adding a menu to 'into', pass a pointer to the menu to add.
       If adding a command to 'into', pass a string that will be
       the name of the command, and an action, which must be a
       lisp expression to evaluate when the command is selected.
----------------------------------------------------------------------

                          MOTION-FUNCTIONS
----------------------------------------------------------------------
MOTION-LAST

USAGE: (motion-last)
NOTES: Returns a pointer to the last motion created (by whatever
       means...lisp, jcl, interactive).
----------------------------------------------------------------------
MOTION-KILL

USAGE: (motion-kill motionptr)
NOTES: Frees space for a motion and removes it from the list
       of active motions. Executes the motion's postaction
       if it is currently running.
----------------------------------------------------------------------
MOTION-STATUS

USAGE: (motion-status motionptr)
NOTES: Returns the status of the motion as:
          'active
```

```
          'executed
          'waiting
```

MOTION-TIME

```
USAGE: (motion-time &optional newtime)
NOTES: Returns current time in seconds if no args. Sets current
       time to 'newtime', which must be a flonum if it is present.
```

MOTION-ON

```
USAGE: (motion-on &optional onP)
NOTES: If no args, returns t if the motion system is advancing
       and nil if not. If there is an argument, if it is nil,
       the motion system is stopped, otherwise it is started
       if it is not already running.
```

MOTION-CURRENT

```
USAGE: (motion-current &optional motionptr)
NOTES: If no arg, returns a pointer to the current motion.
       Otherwise, sets the current motion (the current motion
       in the animation windows -- needed for changing
       a motion using jcl).
```

MOTION-START

```
USAGE: (motion-start motionptr &optional starttime)
NOTES: If starttime present, sets start time of motion, otherwise
       returns start time (all in seconds).
```

MOTION-DUR

```
USAGE: (motion-dur motionptr &optional duration)
NOTES: If starttime present, sets start time of motion, otherwise
       returns start time (all in seconds).
```

MOTION-LISP

```
USAGE: (motion-lisp name start dur apply & optional data preaction postaction)
NOTES: Creates a new lisp motion with the given start time and
       duration. Name is a documentation string that will appear
       in the motion window.
       The preaction is called when the motion starts,
       postaction when it finishes, and apply is called on every
       frame. Pass nil to have no effect for that function.
       When apply, preaction, and postaction are
       executed, (MOTION-DATA) will return the data for the
       motion, and (MOTION-PTR) will return the motion ptr
       for the motion.  Returns a pointer to the motion.
```

MOTION-DATA

```
USAGE: (motion-data)
NOTES: Gets data for a lisp motion whose apply,pre,or post function
       is currently active. Returns nil if no data or no
       lisp motion currently active.
```

MOTION-PTR

```
USAGE: (motion-ptr)
NOTES: Gets the motion pointer for the currently active lisp motion
       if one is running (otherwise returns nil).
```

MOTION-FIND

```
USAGE: (motion-find &optional namestr)
NOTES:  If namestr present, looks for a motion with the given
        full-name and returns a pointer to it, otherwise returns
        a list of pointers to all motions. Returns nil if name not
        found or no motions.
```

MOTION-NAME

```
USAGE: (motion-name mptr &optional local)
NOTES:  Returns the full name of the motion as a string as
        "motionsetname.motionname". If local present and non-nil,
        returns only the local name -- "motionname".
```

4X4 MATRIX PRIMITIVES

MATRIX-TOJCL

```
USAGE: (matrix-tojcl matrix-4x4)
NOTES: Returns jcl representation of the matrix as a string.
```

MATRIX-FROMJCL

```
USAGE: (matrix-fromjcl jclstr)
NOTES: Returns lisp representation of a jcl matrix. (Given a jcl
       matrix as "xyz(90,0,90)*trans(0,100,0)", returns the
       lisp matrix #2A((...)(...)(...)(...)).
```

MATRIX-TRANS

```
USAGE: (matrix-trans (x y z) | matrix-4x4 &optional newtrans)
NOTES: If first arg is a matrix:
          Sets or gets translation component of the 4x4 matrix.
          If newtrans is present it must be a 3-vec, and the matrix's
```

translation component will be set to it. If newtrans is not
present, returns the current translation component of the matrix.
If first arg is a number:
Returns a new translation matrix using x y and z, which
must all be numbers.

MATRIX-ROT

USAGE: (matrix-rot axis theta)
NOTES: Returns a rotation matrix of angle 'theta' (float in radians)
around 'axis' (a 3-vector).

MATRIX-XYZ

USAGE: (matrix-xyz rx ry rz)
NOTES: Returns a rotation matrix of rx, ry, and rz (all floats in
radians) around the x, y, and z axes.

MATRIX-ID

USAGE: (matrix-id)
NOTES: Returns a new copy of the 4x4 id matrix.

MATRIX-INV

USAGE: (matrix-inv matrix-4x4)
NOTES: Returns the inverse of the given matrix, which must be
homogeneous (no shear or scaling).

MATRIX-MULT

USAGE: (matrix-mult base delta &optional (localp t))
NOTES: Transforms the 4x4 matrix 'base' by the 4x4 matrix delta.
Default is a local transformation of base by delta. If
localp set to nil then does a global transformation.

MATRIX-HOMOGENEOUS

USAGE: (matrix-homogeneous matrix-4x4)
NOTES: Returns t if the matrix is homogeneous, otherwise nil.

MATRIX-INTERP

USAGE: (matrix-interp from to tval)
NOTES: Returns a linearly interpolated matrix between 4x4 matrices
'from' and 'to' using tval. Tval=0 is 'from' and tval=1 is 'to'.

MOTION GROUP PRIMITIVES

```
------------------------------------------------------------------------
MGROUP-EXPANDED

USAGE: (mgroup-expanded mgptr expandedp)
NOTES: If expandedp true, then sets the display status to expanded,
       otherwise to contracted.
------------------------------------------------------------------------
MGROUP-NAME

USAGE: (mgroup-name mgptr)
NOTES: Returns the name for the motion group.
------------------------------------------------------------------------
MGROUP-START

USAGE: (mgroup-start mgptr &optional starttime)
NOTES: If starttime present, sets the start time for the motion group
       to the given value, which must be in seconds. Otherwise
       returns the current value.
------------------------------------------------------------------------
MGROUP-DUR

USAGE: (mgroup-dur mgptr &optional dur)
NOTES: If duration present, sets the duration for the motion group
       to the given value, which must be in seconds. Otherwise
       returns the current value.
------------------------------------------------------------------------
MGROUP-FIND

USAGE: (mgroup-find &optional name)
NOTES: If name, looks for a motion group with the given name (which
       must be a string), and returns a pointer to it, or nil if
       not found. If name not present, returns list of pointers to
       motiongroups.
------------------------------------------------------------------------
MGROUP-MOTIONS

USAGE: (mgroup-motions mgptr)
NOTES: Returns a list of pointers to motions in the motion group.
------------------------------------------------------------------------
MGROUP-ADD

USAGE: (mgroup-add mgptr mptr)
NOTES: Adds the motion pointed to by mptr to the motion group mgptr.
------------------------------------------------------------------------
```

7 Changes in Revision 1.0

1. Changed:
   - FIGURE-TRANSLATION should have been FIGURE-TRANSFORM

2. New lisp primitives:
   - FIGURE-TORQUE
   - SEGMENT-MASS
   - SEGMENT-HIGHLIGHT
   - MOTION-FIND
   - MOTION-NAME
   - MATRIX-FROMJCL
   - MGROUP-*

8 Parallel Transition Nets (PaT-Nets)

Parallel Transition Nets (PaT-Nets) are the interface to a parallel programming environment within *Jack*. They allow the user to write parallel automata embedded in a *Jack* simulation. Some features of PaT-Nets:

- Nets in XLISP, time-sliced into *Jack*
- Net classes: nodes, actions, conditions, monitors
- Instances have parameters and local variables
- Actions/Conditions can call/query Jack
- Actions can spawn new nets, execute waits
- Object Oriented: Subclasses, multiple inheritance
- Probabilistic transitions
- Possible extension to Petri Nets

Below is the preliminary documentation for PaT-Nets:

```
1) CREATING FSMS

   Create an FSM with DEFNET:

        (DEFNET fsm-name &optional :locals '(local1 local2...)
                                   :class-vars '(cvar1 cvar2...)
                                   :parents (list parent1 parent2)

             spec1
             spec2
               .
               .
               .)

    DEFNET defines an FSM class. Each instance of the class will
    have local1...localn as local variables available to all methods
    (all actions and conditions).  Class-vars are shared among all
    instances of fsm-name.  Parents should be the names of other
    FSM classes.  The FSM class will inherit all methods (actions and
```

```
 conditions) and all nodes of the parents. Note that every
 FSM gets the FSM class as a parent automatically -- don't put
 it in the parent list.

Specs can be:

   A) (DEFINIT (arg1 arg2 ... argn)
         code)

      Define initializer function to be called when a new FSM
      is instantiated from the FSM class.  It takes the given
      arguments, which can be accessed DIRECTLY, as opposed
      to needing the VAL function like with instance and class
      vars (see below).

   A) (DEFACTION :action-name
         code)

      Create an action with the given name (which must begin with a ':').
      Code can be any arbitrary lisp code.  The return value from
      an action is ignored.

   B) (DEFCOND :condition-name
         code)

      Create a condition with the given name (which must begin with
      a ':').  A condition should return true if the arc containing
      it is to be taken and nil if it is not to be taken.

   C) (DEFNODE node-name :action-name
         (:cond1 state1) (:cond2 state2) ... (:condn staten))

      Create a node with the given name (which doesn't have to
      start with a colon, though it can). When the node is entered
      it executes the :action-name action.  Then it blocks
      on any waits executed in the action (see below). Then it
      looks for an arc with a true condition to follow, doing
      nothing if no arc is true (but not executing the action
      again -- there's an implicit wait-for-arc).  Note that
      the first arc with a true condition is taken, so order
      is important (unless conditions are mutually exclusive).
      Each (:cond state) is an arc -- :cond is the name of a
      condition to evaluate, state is the name of a node to go to.

      Special actions, conditions, and nodes:

          :no-op   =  No action. This can go wherever an :action
                      can go.

          :default =  Default condition.  This ALWAYS evalutates
                      to true -- any arcs appearing after an arc
```

```
                    with the :default condition will never be
                    considered.

        exit     =  Exit state.  This is how to exit an FSM,
                    use exit as the name of the state to go to
                    and the fsm will be exited on the current
                    cycle.
```

C) (DEF-PROBNODE node-name :action
       (prob1 state1) (prob2 state2) ... (probn staten))

Take each of the states with the given probabilities.
Each prob-i should be a number in [0,1]. The prob-i
should sum to less that or equal to 1.0. If no arc
is taken (which happens only if prob-i sum to less that 1.0),
then there's an implicit wait.

An example:

```
      (DEF-PROBNODE state1 :action1
          (0.1 state1) (0.7 state2) (0.2 state3))
```

which has a 10% chance of going to state1, a 70% chance
of going to state2, and a 20% chance of going to state 3.

D) (DEF-WEIGHTNODE node-name :action
      (weight1 state1) (weight2 state2) ... (weightn staten))

Similar to a DEF-PROBNODE, except the weight-i are scaled
to sum to 1.0.

E) (DEF-RANDNODE node-name :action
      state1 state2 state3 ... staten)

Takes one of the states randomly. Translates to a DEF-PROBNODE
with each probability as 1/number-of-states.

F) (DEFMONITOR monitor-name :condition :action-name)

A monitor is evaulated on every cycle by an FSM before
the state is advanced -- monitors are even evaluated when
the FSM is in a wait. Monitor-name is the name of the
monitor (which needn't start with a colon). :Condition
is a condition which should evaluate to true when the
monitor is to be executed. :Action-name is the name of
an action that is executed when the monitor is true.

An Example:

```
    (DEFNET samp-fsm  :locals '(a b)
```

```
        (DEFINIT (a b)
           (val a a)
           (val b b))

        (DEFACTION :inca
           (val a (1+ (val a))))

        (DEFACTION :incb
           (val b (1+ (val b))))

        (DEFCOND :c1 (< (val a) 5))

        (DEFCOND :c5 (< (val b) 5))

        (DEFNODE state1 :inca (:default state2))

        (DEFNODE state2 :no-op (:c1 state1) (:default state3))

        (DEFNODE state3 :incb (:default state4))

        (DEFNODE state4 :no-op (:c5 state3) (:default exit)))
```

----

2) CREATING FSM INSTANCES

Create an instance by sending and FSM class the message :new
with any arguments required by the FSM initializer function.
Like:

```
        (send fsm-class :new arg1 arg2 ... argn)
```

which returns the new FSM object. The object is automatically
inserted in the active list of FSMs, and it will begin executing on
the next cycle. Using the example above, do something like:

```
        (setf fred (send samp-fsm :new 1 2))
        (setf sam  (send samp-fsm :new 3 3))
```

To create a new instances of samp-fsm with different arguments
bound to fred and sam. The two FSMs will begin executing in
parallel in the next cycle.

----

3) ACTIONS AND CONDITIONS

The actions and conditions are actually methods in the FSM class
in which they are defined (this is the reason they need to begin
with a colon -- it's required by the class mechanism...).

A) LOCAL VARIABLES

Actions and conditions can access an FSMs local variables or class variables using VAR:

```
(VAR a)  -- returns the value of FSM variable a

(VAR a newval) -- sets the value of FSM variable to newval.
```

Note that a is not evaluated, but newval, if present, is evaluated.

B) SPAWNING NEW FSMS

An FSM can spawn a new fsm in an action just by using the standard FSM instantiation mechanism described above. An example action that does this (and uses a wait, described below):

```
(DEFACTION :action1
   (val waiting-for
        (send another-fsm :new 5 6))
   (send self :wait-fsm (val waiting-for)))
```

C) WAITS

An action can post a set of waits that the fsm will wait for on exiting the action. The FSM will wait for the conjunction of all posted waits before continuing. Note that the FSM does not block until it exits the action. Each wait is signaled by the FSM sending a message to itself:

A) (SEND self :wait-fsm fsm-instance &key state)

Wait for an fsm to exit before continuing. If state is present it waits for FSM to pass through the given state (a node name) before continuing. Note that fsm-instance must be an instance of an FSM class and not the class itself. An example action:

```
(DEFACTION :action1
    (send self :wait-fsm (send myfsm :new 3 4)))
```

and to wait for a state:

```
(DEFACTION :action1
    (send self :wait-fsm (send myfsm :new 3 4) :state 'state2))
```

B) (SEND self :wait-time time)

Wait for a specific time in the motion system (in seconds). To wait 3 seconds before proceeding,

```
(send self :wait-time (+ (motion-time) 3.0))
```

C) (SEND self :wait-condition condition-fn)

Wait for an arbitrary lisp expression to evaluate to true. The condition-fn should be a fucntion closure taking no arguments, like:

```
(send self :wait-condition #'(lambda () (< x 3.0)))
```

D) (SEND self :wait-motion motion-ptr)

Wait for a motion in the motion system to finish before proceeding. Motion-ptr must be a pointer to a motion, which can be found by name with the MOTION-FIND function. Example:

```
(send self :wait-motion (motion-find "default.ccube"))
```

E) (SEND self :wait-semaphore (s &key (priority 1.0)))

Wait on a semaphore s. Semaphores are instances of the semaphore class created like:

```
(setf s1 (send semaphore :new))
```

The FSM is placed on the semaphore's wait queue with the given priority (defaulting to 1 -- higher numbers for higher priority).

A semaphore is signaled (released) with:

```
(send self :signal-semaphore s)
```

Note that since waits don't block until after the action exits, no code dependent on having the semaphore should appear after the wait.

An example fsm using semaphores which waits on two semaphores then releases them in the next state:

```
(setf s1 (send semaphore :new))
(setf s2 (send semaphore :new))

(DEFNET semwait1

   (DEFACTION :action1
      (format t "in state 1~%")
      (send self :wait-semaphore s1)
      (send self :wait-semaphore s2))

   (DEFACTION :action2
```

```
        (format t "in state 2~%")
        (send self :signal-semaphore s1)
        (send self :signal-semaphore s2))

     (DEFNODE state1 :action1 (:default state2))
     (DEFNODE state2 :action2 (:default exit)))
```

D) OTHER MESSAGES AVAILABLE IN ACTIONS AND CONDITIONS

1) `(send fsm-instance :exit &optional (exitcode 'finished))`

   Force an fsm instance (which can be self) to exit with a given exit code, one of 'finished or 'failed. The fsm kills any dependent processes and kills any fsms waiting on a particular state in itself.

2) `(send fsm-instance :status)`

   Get the status of the fsm, one of:

```
'running  -- in active list
'idle     -- not running yet
'failed   -- exited with failed status
'finished -- exited with successful status
'waiting  -- waiting on conditions
```

3) `(send fsm-instance :add-dependent dep-fsm-instance)`

   Tell the fsm-instance that dep-fsm-instance is a dependent process, so that when fsm-instance exits, it kills dep-fsm-instance.