```
;;-------------------------------------------------------------
;; The Application Programmer's Interface to the Fern System.
;;
;; FERN is the Fractal Entity Relativity Node.
;;
;; creation: August 10, 1992
;; last edits: October 20, 1992
;;
;; by Geoffrey P. Coco and Colin Bricken
;; Software Engineering by Geoff Coco
;;-------------------------------------------------------------

;;-------------------------------------------------------------
;; Copyright (c) 1992, Washington Technology Center
;;
;; This program's use is restricted under the terms of the WTC LICENSE
;;-------------------------------------------------------------
```

```
------------------------------------------------------------
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
------------------------------------------------------------

     Specification of The FERN Distributed Environment

------------------------------------------------------------
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
------------------------------------------------------------

------------------------------------------------------------
- Create the fern distributed computing environment -

(fern-init <:host-pool host-list> <:display-host host-name>)
      <host-list>       - list of host-args, hosts in run pool.
      <host-name>       - STRING, host to display node windows
      returns:          - T/NIL

Once running, fern provides a homogenous, parallel
computation environment.  The host-pool defines the nodes
which make up fern's distributed multiprocessor.  In other
words, the host-pool is the maximal set of hosts that your
distributed fern program will require for this invokation.

Each item in the host-list must be either:
1) a string that names the chosen host, or
2) a list containaing the string from (1) and a
   string naming the binary executable to run as the node.

For example,
(fern-init :host-pool '("slithy"
                  ("iris2" "/home/cygnus/bin/imager")
                    ("water" "/home/cocteau/bin/swarmer")
                  "hal")

On hosts slithy and hal, the default veos binary executable
will be used.  On hosts iris2 and water, the named custom
binaries will be used.  In any case, the startup lisp file
is /home/veos/ote/veos2.2/src/tabula_rasa.lsp

A fern program should call (fern-init ... ) only once.  The
node that begins a program by calling (fern-init ... ) is
called the 'console' node.  fern-init automatically
launches and initializes nodes on all the remaining hosts
in the host-pool (the 'console' is always in the pool).

If the :host-pool argument is not specified, the default
host-pool contains only the console node.
```

If the :display-host argument is not specified, all node xterm windows automatically display on the console's host.

A running fern pool can stay running through many runs of the same program. As long as the nodes haven't crashed (bus error, etc..), programs can run again and again without making new calls to (fern-init ...).

- Takedown the entire distributed computing environment -

(fern-close)
      returns: - does not return

Takes down each node in the fern pool including itself. Other nodes must still be in (fern-go) to respond properly.

- Join with separately running node-pools.

(fern-merge-pool <terminal-node>)

      <terminal-node>   - any node (uid) in remote pool
      returns   - T/NIL

Used when fern application is composed of separate modular running node-pools. This is useful for devoting pools to significant application tasks (e.g. participant suite, entire self-sufficient worlds, etc..).

The pool that results from starting a fern-program with (fern-init ...) is the native pool for those entities within that program.

Upon merge, join with the native node pool associated with the given node. Once a merge is completed, all nodes in the remote pool become reachable from the native pool also. All nodes in the new aggregate pool are on equal terms for entity communication. Note, for accountability purposes, Fern programs can only create or destroy entities in the original pool created with (fern-init ...).

```
------------------------------------------------------------
- Detatch native pool from the aggregate pool.

(fern-detatch-pool <terminal-node>)

      <terminal-node>    - any node (uid) in remote pool
      returns            - T/NIL

Detatch from the native pool named by the remote node.
There is exactly one native pool for every entity - the
pool which was created directly with (fern-init ... ).
Relinquish access to all entities within the remote pool.
------------------------------------------------------------

------------------------------------------------------------
- Find out if the pool is still alive

(fern-ping-pool <terminal-node>)

      <terminal-node>    - any node (uid) in remote pool
      returns            - T/NIL

Send tickle packet liveness.  The timeout is the same for
(fern-seq-send ... ).
------------------------------------------------------------
```

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

Invokation of FERN Entities

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

- Run a fern program -

(fern-run <spore-ent> <:file-host filehost> <:run-host runhost>)

<spore-ent> - entity definition for initial entity
<filehost> - name of host to find init ent description
<runhost> - name of host on which to run init ent
returns - does not return

Every fern program begins with an initial 'spore' entity. The only required argument to (fern-run ...) is the entity definition of your program's spore entity. This entity can make further entities, install processes for itself, etc. This initial entity is the main() of the fern distributed program.

The arguments pertaining to the spore entity (init-expr, file-host, run-host) have the same semantics as in (fern-new-ent ...).

Once you've called (fern-run ...), EVERYTHING is done within an entity context...

1. Fern performs pattern matches from the perspective of the calling entity. Each entity sees a customized version of the grouplespace.

2. All process in fern is associated with some entity in the form of 'methods', 'persist procs', or 'react procs'.

   I. Methods are user-defined entry-points for inter-entity communication. Entities can call their own or each other's methods with (fern-send ...). A Method call is accountable to the calling entity.

   II. Persist Procs are user-defined processes associated

```
   with fern entities.  An entity can install multiple
   persist procs which execute in that entity's context.
   See fern-persist for more details.

   III. React Procs are user-defined functions associated with
   data-update events.  An entity can install multiple react
   procs which execute in that entity's context when new data
   arrives from other entities.  See fern-perceive for more details.
-------------------------------------------------------------

-------------------------------------------------------------
- Create and invoke a new fern entity -

(fern-new-ent <init-expr> <:run-host runhost> <:file-host filehost> )
      <init-expr>       - list of lisp expressions or filename
      <filehost>        - name of host to find init ent description
      <runhost>   - name of host on which to run init ent
      returns:          - new entid

The init-expression is a list of user-defined lisp
expressions that will prepare the entity for normal
execution.  See (fern-entity ... ) below for more.

Alternatively, the init-expr can be the filename of an
initial entity definition.  These files follow the same
format - expressions which when evaluated determine an
entity's character.  Fern entity definition files must end
in ".fent", and (fern-new-ent ... ) automatically appends
the ".fent" suffix onto filenames.  Always use full
pathnames for best portability.

To get emacs to edit ".fent" files as lisp, put this line in
your .emacs file:
(setq auto-mode-alist
      (cons '("\\.fent" . lisp-mode) auto-mode-alist))

The optional :file-host argument can be used to instruct
fern where to look for the named entity definition file.
If the file-host is not specified, fern looks for the file
on the console node.

The :run-host argument is also optional.  It tells fern
where to run the new entity.  If run-host is not specified,
it currently runs on a random host in the pool.

When the entity has been created and the initial expression
has been evaluated, the new entid is returned to the caller.
-------------------------------------------------------------
```

```
------------------------------------------------------------
- Takedown a fern entity and free all its resources -

(fern-dispose-ent <entid> )
      <entid>           - optional entid of entity to dispose
      returns:          - T/NIL

The entid argument is optional.  If an entid is not given, the
entity which made call has requested self-disposal.
------------------------------------------------------------

------------------------------------------------------------
- Generate an initial expression for a fern entity -

(fern-entity <expr> <expr> <expr> ... )
      <expr>            - unevaluated lisp expressions
      returns:          - an unevaluated list of the exprs

The argument expressions are quoted lisp code which a new
Fern entity will evaluate upon startup.  (fern-entity ... )
returns an expression suitable to pass (fern-new-ent ... )
for creating an entity with Fern.

These normally include:
1. Creating self attributes with (fern-put.bndry.attr ... )
2. Staking out private workspace with (fern-put.locl ... )
3. Creating methods of behavior with (fern-def-meth ... )
4. Creating entity processes with (fern-persist ... )
5. Declaring active database needs with (fern-perceive ... )
6. Initializing any peripheral connections, e.g. (sensor-init ... )
7. Defining functions behind the methods, e.g. (defun ... )
------------------------------------------------------------
```

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

Object-Oriented Aspects of Entities

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

- Define fern entity behavior -

```
(fern-def-meth <name> <lambda-expr> )
      <name>            - name of method, STRING argument
      <lambda-expr>     - lambda which defines method behavior
      returns:          - T/NIL
```

A method is a well-defined entry point to a fern entity.
Entity can communicate and pass data directly using method
calls.

(fern-def-meth ...) adds the given method to the calling
entity. Other entities can use (fern-send ...) to have
the entity perform a method on behalf of the calling
entity.

```
Ex:  let's say ent1 defines a method:
      (fern-def-meth "print-plus" (lambda (x) (print (+ x 1))))
     then, ent2 can use the method:
      (fern-send ent1 "print-plus" 2)
     the screen when ent1 is running should print '3'.
```

- Undefine fern entity behavior -

```
(fern-undef-meth <name> )
      <name>            - name of method, STRING argument
```

```
------------------------------------------------------------
- Asynchronous method call -

(fern-send <entid> <method-name> <arg1> <arg2> ... )
      <entid>     - id of destination entity
      <method-name>     - STRING name of method
      <args> ...  - the arguments to the method

Entities may call their own methods by passing 'self' as the
entid.
------------------------------------------------------------

------------------------------------------------------------
- Synchronous method call -

(fern-seq-send <entid> <method-name> <arg1> <arg2> ... )
      <entid>     - id of destination entity
      <method-name>     - STRING name of method
      <args> ...  - the arguments to the method


The calling semantics are the same as (fern-send ... )
except that (fern-seq-send ... ) dispatches the destination
entity's named method immediately.  (fern-seq-send ... )
returns only when the method has completed and has returned
a result.  (fern-seq-send ... ) returns the result of the
method call.
------------------------------------------------------------

------------------------------------------------------------
- Asynchronous method call -
- using streams for flow control -

(fern-str-send <entid> <method-name> <arg1> <arg2> ... )
      <entid>     - id of destination entity
      <method-name>     - STRING name of method
      <args> ...  - the arguments to the method
      returns:          - T/NIL

The calling semantics are the same as (fern-send ... )
except that (fern-str-send ... ) only performs the send if
all outstanding sends to this destination have been serviced.

Fern uses a message pacing algorithm called streams.  A
stream is an logical connection from one entid to another
that has a maximum carrying capacity.  In other words,
streams ensure that sender entities only send messages as
fast as their receiver entities can digest them.
```

(fern-str-send ...) sends the given method call only if the stream to that entity is clear. A return value of true means that the method call was sent. A return value of NIL may mean that the stream was full and the caller should try again later. A return value of NIL could suggets others errors as well.

Fern users can save much overhead and ambiguity in using this pacing mechanism by testing the stream _before_ calling (ferr-str-send ...). To test a stream for clearness, use (fern-str-clrp entid) as described below.

- Test a stream -

(fern-str-clrp <entid>)
      <entid>   - stream destination
      returns: - T/NIL

This function quickly checks whether the logical channel between the calling entity and the destination entity is sufficiently clear for passing messages.

- Asynchronous informal entity process call -

(fern-as <entid> <expr>)
      <entid>   - id of destination entity
      <expr>   - quoted evaluable expression
      returns: - T/NIL

Same semantics as (fern-send ...), but without the method formality. The given expression is evaluated in the context of the named entity.

- Synchonous informal entity process call -

(fern-seq-as <entid> <expr>)
      <entid>   - id of destination entity
      <expr>   - quoted evaluable expression
      returns: - T/NIL

Same semantics as (fern-seq-send ...) revised as above.

```
------------------------------------------------------------
- Asynchronous informal entity process call -
- using streams for flow control -

(fern-str-as <entid> <expr> )
      <entid>     - id of destination entity
      <expr>      - quoted evaluable expression
      returns:          - T/NIL

Same semantics as (fern-str-send ... ) revised as above.
------------------------------------------------------------
```

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

Specification of Entity Process

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

- Lightweight task creation -

```
(fern-persist <expr> <:name procname> )
      <expr>            - repeatable lisp expression.
      <procname>        - optional STRING name of proc
      returns:          - new proc name
```

Installs a new persist proc in the calling entity's proc table. This is similar to forking a process.

The persist concept implements a form of cooperative multitasking. To ensure proper multitasking and to approach the effect of parallelism, your persist expressions should:

1. Be fast-evaluating.

If the proc represents an ongoing process, try to break the task into discrete tasks which each take a small slice of time. Use quick state checks to exit the proc early in case there is no work to do. Watch out for loops where the number of iterations can become large at times.

2. Be atomic.

The job of one proc should be conceptually neat. This is so that when procs are evaluated in different orders, their relative behaviors remain consistent. If there's a job that two procs collectively perform, they should be one proc.

3. Never block.

A proc that blocks will starve other procs of valuable time slices. Remember, since frames rates in veos aren't enforced, please be a 'good citizen'.

There are indeed situations where it makes no sense to continue without a certain resource (like data from a remote query, or data from a hardware driver). But instead of freezing up the persist cycle while waiting for your resource, use an asynchronous method.

For example, when polling hardware, use a non-blocking scheme - if there's no data, return and check again next cycle. Also, when waiting for a remote query reply, send a request and check for the reply each cycle until it arrives.

- Lightweight task disposal -

(fern-desist <procname>)

	<procname>	- optional name of persist proc to kill
	returns	- the deleted proc-name

Terminate the given persist proc. If no proc is specified, terminate the calling proc.

If a procname is not specified, fern removes the currently running persist proc from the system proc list. That running proc is allowed to complete it's last evaluation normally.

(fern-desist ...) returns the name of the proc that was deleted from the run queue.

```
------------------------------------------------------------
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
------------------------------------------------------------

   Configuration of Trademark FERN Virtual Grouplespace

------------------------------------------------------------
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
------------------------------------------------------------

------------------------------------------------------------
- Enter a fern space -

(fern-enter <space-entid> )
      <space-entid>     - the unsuspecting entity
      returns:          - T/NIL

Request to becoming a 'subling' of the named entity.  You
would do this to instigate an implicit and automatic
awareness of other entities that may also be 'entered' in
the same space.  An entity entered in another entity (a
space by virtue of this relationship), will 'see' other
entities also in that space.

Entities that are entered in the same space are 'siblings'.
Siblings 'see' each other through fern's automatic database
propogation.  When one sibling changes its local database
(its boundary), these changes are automatically propogated
to others siblings' databases (their externals).

Entities in spaces can express particular interests or
filters, to limit the automatic database propogation to the
necessary data streams.  These data interests are called
'in-senses'.  In-senses are hints to fern about what kind
of data a subling wishes to receive (see fern-perceive)

When any entity enters a space, the entity will appear in
its own external as a sibling.  This virtual representation
is filtered however through the entity's in-senses (see
fern-perceive).

A space entity need not have any special behavior to
function properly as a space.  All fern entities are equally
equipped to serve as spaces.  Space entities perform very
little computational legwork.  Instead, they provide a
conceptual meeting ground for entities having particular
relationships.
------------------------------------------------------------
```

```
------------------------------------------------------------
- Exit a fern space -

(fern-exit <space-entid> )
      <space-entid>     - the unsuspecting entity
      returns:          - T/NIL

Relinquish all services of the named space entity.  The
calling entity will no longer 'see' siblings that were
entered in this space.  Those siblings will no longer 'see'
the calling entity in their external.
------------------------------------------------------------


------------------------------------------------------------
- Subscribe to data flows -

(fern-perceive <attr-name> <:react (lambda (entid attr-val) ...)> )
      <attr-name>       - attribute of interest
      <lambda>          - optional react proc
      returns:          - T/NIL

Declare an in-sense.  By calling this function once, the
calling entity declares that whenever it is entered in a
space, it would like to 'see' entities in that space which
have the named attribute.  You can perceive many attributes
with multiple calls to (fern-perceive ... ).

Likewise, if another entity sharing a space with your
entity has declared an in-sense as above, then it will
'see' data in your entity's boundary.

Exactly how the caller 'sees' other entities is determined
by the optional :react argument.  In the regular case where
no :react argument is given, the calling entity's
'siblings' partition will be automatically updated by
fern when sibling entities make changes to their boundaries.

If a :react lambda argument is given, fern calls this
lambda function when sibling changes occur.  The lambda
function must take two arguments: the entid of the sibling
that's changing its boundary and the new attribute.

In this mode, fern does not automatically update the siblings
partition.  To achieve the normal fern auto-grouplspace behavior
and _also_ use this :react feature, your react proc should call
(fern-put.sib.attr entid attr) before it returns.
------------------------------------------------------------
```

```
------------------------------------------------------------
- Unsubscribe to data flows -

(fern-unperceive <attr-name> )
      <attr-name>       - attribute of interest
      returns:          - T/NIL

Eradicate an in-sense.  The calling entity's external will
be rid of all references to this attribute.
------------------------------------------------------------


------------------------------------------------------------
- Produce data flows (explcitly) -

(fern-exude <attr> )
      attr        - ("attr-name" attr-val)
      returns:          - T/NIL

Create inter-entity data flows manually.  The same behavior
automatically occurs when an entity calls fern-put.attr.

Normally, after an entity calls fern-put.attr, fern passes
these boundary changes to the entity's siblings at some
later time.  With fern-exude, the changes bypass the
grouplespace and flow directly to perceiving siblings.

NOTE: entities that use this function must call
fern-put.attr to initialize the sibling-to-sibling
connections.  For example, during entity initialization,
call fern-put.attr once for each attribute you plan to
exude.
------------------------------------------------------------
```

```
------------------------------------------------------------
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
------------------------------------------------------------

        Accessing The FERN Perception Partition

------------------------------------------------------------
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
------------------------------------------------------------



------------------------------------------------------------
- Put, Get, Copy to the entity groueplspace -

Here are the most useful fern grouplespace pattern matches.
To see what the grouplespace looks like in FernII, load up
an entity or two with the works (boundary-attrs, in-senses,
procs and methods) and then call (dump).
------------------------------------------------------------

------------------------------------------------------------
;; E X T E R N A L
------------------------------------------------------------
- Source -

(fern-copy.src )
      returns:          - source entid

      if you are an entity, the entity that created you is your
      'source'.  All entities in a program have a source except
------------------------------------------------------------
- Space -

(fern-copy.sps <:test-time test-time> )
      test-time         - optional nancy timestamp
      returns:          - list of current space entids
------------------------------------------------------------
- Siblings -

(fern-copy.sibs <:test-time test-time> )
      test-time         - optional nancy timestamp
      returns:          - list of perceived entities

      Each sib is of the form: (entid (attr-list)).
      (fern-copy.sibs) returns a list of these structures, one
      for each sibling which shares a fern space with the calling
      entity.  These attr-lists are comprized of only attributes that
      the calling entity has perceived via (fern-perceive... ).
```

```
      This function is the bread & butter of reactive
      programming.  It is common practice during a persist proc
      to perform a (fern-copy.sibs ... ) with a timestamp, and
      parse the resulting sparse structure of sibling changes to
      compute new reactive behavior.


(fern-copy.sibs.entids <:test-time test-time> )
      test-time         - optional nancy timestamp
      returns:          - list of entids

        Returns the entids of all the calling entity's siblings.

(fern-copy.sib <entid> <:test-time test-time> )
      test-time         - optional nancy timestamp
      returns:          - the named entity's attr-list

(fern-copy.sib.attr <entid> <attr-name> )
      entid       - entid of sibling
      attr-name         - name of sibling's attribute
      returns:          - given attr value

(fern-copy.sib.attr-names <entid> <:test-time test-time> )
      entid       - entid of sibling
      test-time         - optional nancy timestamp
      returns:          - list of named entity's attribute names

(fern-copy.sib.attr-if-attr-name <must-name> <gimme-name> )
      must-name         - name of attribute which must appear
      gimme-name        - name of attribute to retrieve
      returns:          - value of gimme-name

      If there is a sibling to the calling entity which has the
      must-name attribute and the gimme-name attribute,
      return the value for the gimme-name attribute.

(fern-copy.sib.attr-if-attr <must-attr> <gimme-name> )
      must-attr         - attribute which must match
      gimme-name        - name of attribute to retrieve
      returns:          - value of gimme-name

      If there is a sibling to the calling entity which
      matches the must-attr (name and value) and the
      has gimme-name attribute, return the value for the
      gimme-name attribute.
```

```
(fern-copy.sib.entid-if-attr <must-attr> )
      must-attr         - attribute which must match
      returns:          - entid of matching entity

      If there is a sibling to the calling entity which
      matches the must-attr (name and value) ,return its entid.
-------------------------------------------------------------



-------------------------------------------------------------
;; B O U N D A R Y
-------------------------------------------------------------
- Entire boundary -

(fern-copy.bndry <:test-time test-time> )
      test-time         - optional nancy timestamp
      returns:          - list of calling entity's attributes
-------------------------------------------------------------
- Boundary attributes -

(fern-put.attr <attr> )
      <attr>            - ("attr-name" attr-val)
      returns:          - old attr, if any; or T/NIL

        This function inserts and replaces.  No need to
      get-then-put.

(fern-copy.attr <attr-name> )
      <attr-name>       - name of desired attribute
      returns:          - attribute value

(fern-get.attr <attr-name> )
      <attr-name>       - name of desired attribute
      returns:          - old attr val, if any

      Instead of deleting the attribute from the boundary completely,
      this function actually replaces the current value of the
      attribute with "%", the 'skull-and-crossbones' symbol.

      The reason for keeping 'dead' data in this way is fairly
      obscure.  The "%" is like an ordinary attribute value and
      so fern will propogate it to siblings as such.  It has
      become customary to reserve the "%" attribute value to
      signify data that was but is no more.

(fern-copy.attr.names )
      returns:          - list of the entity's attr-names
-------------------------------------------------------------
```

```
------------------------------------------------------------
;; I N T E R N A L
------------------------------------------------------------
- Subs -

Sublings are the entities for whom your entity is acting as
a space.

(fern-copy.sub.entids <:test-time test-time> )
      test-time         - optional nancy timestamp
      returns:          - list of subling entids

------------------------------------------------------------
- Entire Local -

The local partition is an entity's private grouplspace
area.  It is only accessed by the user's code.  The local
is the user defined grouplespace partition.

This is your chance to write your own nancy grouplespace
patterns.  Do some experimenting with these functions to
make sure you understand how to use the local partition.

(fern-put.locl <data> <pat> <:freq frequency> )
      data        - any veos compatible expr
      pat         - any single element pattern
      frequency         - "one"/"all" default is "one"
      returns:          - old data; or T/NIL

(fern-copy.locl <pat> <:test-time test-time> <:freq frequency> )
      pat         - any single element pattern
      test-time         - optional nancy timestamp
      frequency         - "one"/"all" default is "one"
      returns:          - matched data

(fern-get.locl <pat> <:freq frequency> )
      pat         - any single element pattern
      frequency         - "one"/"all" default is "one"
      returns:          - removed data
------------------------------------------------------------
- Local Attributes -

Although the local partition is user-defined, user's may
use the standard 'attribute' regime for organizing their
entitys' local partitions.
```

```
(fern-put.locl.attr <attr> )
      attr        - ("attr-name" attr-val)
      returns:          - old attr, if any; or T/NIL

(fern-copy.locl.attr <attr-name> )
      attr-name         - name of desired attribute
      returns:          - attribute value, if any

(fern-get.locl.attr <attr-name> )
      attr-name         - name of desired attribute
      returns:          - old attr, if any
```

To understand the purpose of 'local attributes', think of each entity as a program which may need its own global variables. Since many entities may inhabit the same lisp environment (i.e. node), lisp global variables are not suitable for this purpose.

Imagine what happens when two entities in the same lisp environment use a global named cur_pos. Each entity will use the cur_pos variable in its own way, thus altering each other's state unpredictably. Using local attributes ensures that entities have exclusive use of their memories.

```
------------------------------------------------------------
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
------------------------------------------------------------

  Additional Features of the FERN Distributed Envionment.

------------------------------------------------------------
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
------------------------------------------------------------

------------------------------------------------------------
- Read a file, from anywhere in the fern pool -

(fern-read-file <file-name> <:host hostname> )
      <file-name>       - name of file to load
      <hostname>        - name of host from which to load file
      returns:          - unevaluated list of all exprs in file

Load the given file from somewhere in the pool.  If no hostname
is given, fern attempts to load the file from the console node.
------------------------------------------------------------

------------------------------------------------------------
- Read and evaluate a file, from anywhere in the fern pool -

(fern-eval-file <file-name> <:host hostname> )
      <file-name>       - name of file to load
      <hostname>        - name of host from which to load file
      returns:          - T/NIL

Retrieve the named file and evaluate each expression contained
within.  Uses (fern-read-file ... ) to offer extension of the
standard lisp (load ... ) function.  NOTE: Always use full pathnames
so that the calling code will behave the same on different hosts.
------------------------------------------------------------

------------------------------------------------------------
- Write a file, to anywhere in the fern pool -

(fern-write-file <file-name> <expr-list> <:host hostname> )
      <file-name>       - name of file to write
      <expr-list>       - list of data
      <hostname>        - name of host on which to write file
      returns:          - T/NIL

Write the given expressions to the named file.  This file
can later be accessed though (fern-read-file ... ).
------------------------------------------------------------
```

```
------------------------------------------------------------
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
------------------------------------------------------------

            Attention to Node Operation

------------------------------------------------------------
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
------------------------------------------------------------

- Fern Node Global Variables:

------------------------------------------------------------
home        <read-only>

The host name where symbol is evaluated.
------------------------------------------------------------
self        <read-only>

The entity-id of the evaluating entity.
------------------------------------------------------------
fern-display      <read-only>

The hostname where all node xterms display.
------------------------------------------------------------
fern-debug        <set-at-will>

Enables verbose general fern operations.
------------------------------------------------------------
as-debug          <set-as-will>

Enables tracing of entity context switching.
------------------------------------------------------------
frame-debug       <set-at-will>

Enables display of frame rate statistics.
------------------------------------------------------------
flow-debug  <set-at-will>

Enables tracing of automatic data flow.
------------------------------------------------------------
str-debug         <set-at-will>

Enables tracing of stream message operations.
------------------------------------------------------------
file-debug        <set-at-will>

Enables tracing of file operations.
```

```
------------------------------------------------------------
fmaxclog          <set-with-caution>

The maximum msg width of inter-host streams from this node.
Use these heuristics to tune for specific application:

high values (4-50) yield
    best thruput (best processor utilization and parallism)
    worst latency (lots of msgs are getting queued)
    very clumpy flow (msgs tend to caravan through system)

low values (2-3) yield
    better thruput (some parallism, good processor utilization)
    not-so-good latency (some msgs are getting queued)
    clumpy flow (msgs tend to caravan through system)

minimum value (1) yields
    worst thruput (little parallism, lots of idle waiting)
    best latency (msgs get serviced right away)
    most consistent transmission (msgs flow at even pace)

the default setting is: 1
------------------------------------------------------------
```

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

Major Confusions

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
---

1. Idea of entities and methods.

Entities are smart little optimizers. They do their jobs, you do yours. Share the responsibility, and the abstraction. Let the entity model encourage a high degree of modularity, mobility and correctness in your programs.

Everything happens from within _some_ entity's context, except the first call to (fern-run ...). I mean everything!!

When matches don't work, take the perspective of the entity. What does _it_ see? A quick (dump) will reveal each entity's grouplespace perspective. It helps to become familiar with object-oriented thinking.

---

2. The entity concept has new meaning.

Users of FernI will find that the old notion of an entity has been rightly reshaped in FernII into the concepts of the entity and node.

The FernI entity was a heavyweight unix process usually programmed for a discrete task. A FernI program could run several entities on a host machine calling on unix to simulate parallelism by context switching between them. Entities could pass messages to each other via heavyweight unix network sockets.

In FernII, each participating host machine runs exactly one unix process called a FERN node. The need for costly unix context switching is diminished because each node works within one unix process.

The FernII entity is still programmed for a discrete task; but the entity has become lightweight. Many entities run

on a single node sharing the unix process. FernII's nodes simulates paralellism by performing lightweight context switching between entities.

---

3. The entity definition.

An entity definition is an unevaluated list of lisp expressions which when evaluated by fern-new-ent will define a fern entity. The lisp expressions will normally consist of calls to: (defun ..) (fern-put.attr ..) (fern-perceive ..) (fern-persist ..) (fern-def-meth) (fern-put.locl ..) and any other setup your entity needs to do.

---

4. Fern spaces are an entity grouping facility.

The FERN virtual grouplspace features are designed to define a crude world 'perspective' for an entity. Furthermore, spaces provide a mechanism for grouping entities into relational sets. Consider the following canonical example:

EntityA is earmarked as the 'gravity' space. Currently entered in EntityA are Entity1, Entity2, Entity3, and EntityGrav. EntityGrav is entered in this space only (by design), and so can easily perceive all entities in the gravity space. The EntityGrav entity is thus equipped to affect its sibling entities with gravity.

Note that in this example, Entity1, Entity2 and Entity3 would need to exude the proper attributes (like "mass" and "6D") to be acted upon by EntityGrav. However, they would not need to perceive anything special to enjoy gravity space influence - they are acted upon by virtue of their membership in that space.
