FEEL: An Implementation of EuLisp
Version 0.38

Concurrent Processing Research Group
School of Mathematical Sciences
University of Bath, United Kingdom
E-mail: `eulisp@maths.bath.ac.uk`

June 16, 1992

Abstract

This document describes an implementation of EuLisp called Feel. The primary reference for EuLisp is the EuLisp definition. In this document, the environmental operations provided in Feel, but which are not part of the EuLisp language, are described in detail.

1 Getting further information

Information about EuLisp and a copy of the Feel implementation are all available from the `eudist` mail server at the University of Bath. Here is a summary of the services provided:

Address	Subject Field	Effect
`eudist@maths.bath.ac.uk`	`feel`	Distributes the current release of Feel. Each file will be sent individually in a uuencoded, compressed format.
`eudist@maths.bath.ac.uk`	`definition`	Distributes the current release of the EuLisp definition, rationale and commentary in `.dvi` format. Each file will be sent individually in a uuencoded, compressed format.
`eubug@maths.bath.ac.uk`	something germane	Please provide body too! This mail-id is for reporting bugs in the definition or in Feel.

2 Making FEEL

The kind of Feel system you can make depends on the combined capabilities of your operating system and processor. Feel has been developed in a solely Unix environment and this has considerably warped its view of the world. So far, Feel has only been ported to different versions of Unix. A particular feature of EuLisp is support for multiple threads of control. Whether these do actually execute concurrently depends on the host system, but, in principle, it should be possible to develop a program using threads on one system—perhaps a uni-processor simulating concurrency—and later execute the same program on a multi-processor or a distributed processor to achieve the same net result.

Broadly speaking, Feel can be made in any of three main configurations ...

Generic Under the "any" machine configuration, Feel attempts to be a fully portable ANSI C program. Because there is no reliably portable method of implementing threads in C, the thread operations in

this mode are not available and only a serial version of EuLisp remains. This mode is most suitable for getting started quickly and also the most sensible place to begin for porting to new architectures or operating systems. Memory use is minimised which may benifit smaller machines such as PCs or any system where memory is at a premium.

BSD This (badly named) configuration mode requires that a stack switching operation be available for Feel to use. Given this code (typically a few lines of the local assembler), the thread operations become available with the limitation that only one thread is run at a time. This mode allows programs to be written in terms of threads which may later be run in parallel without alteration. This mode is most useful for allowing the developing multi-threaded applications under unsupportive operating systems such as BSD 4.2 or 4.3.

System V This configuration requires that stack switching code be available along with the standard System V shared memory manipulation primitives. Given these things Feel becomes a truly parallel multi-threaded system using the following model: on start-up a piece of shared memory is allocated, then Feel forks as many times as there are physical processors in the host machine (this behaviour may be modified). Each of these forked processes runs the Feel scheduler, running threads from the pool in the shared heap. Each such thread is run to conclusion—unless it yields control, in which case it will be returned to the pool. More processes may be forked than existing processors to simulate truly parallel operation on uniprocessor systems such as Suns running SunOS 4.1.

3 The FEEL environment

Feel uses the shell variable `FEEL_LOAD_PATH` to load modules. If this variable is unset, then modules are sought in the directory in which Feel was invoked and in a directory specified when the system was built. The value of `FEEL_LOAD_PATH` is read at start-up and converted to a list of strings of information in a processor-defined format concerning the filesystems or disks or directories to be searched when loading modules. Modules are stored in files with the extension `.em`.

Similarly, the shell variable `FEEL_INTF_PATH` is used to find module interface files. If the variable is unset, only the directory in which Feel was invoked will be searched. The value of `FEEL_INTF_PATH` is read at start-up and converted in a manner analogous to that for `FEEL_LOAD_PATH`. Interfaces are stored in files with the extension `.i`.

3.1 Getting in and out

Feel is started by typing `feel`, assuming correct paths and installation. To leave Feel type CNTL-D, or possibly `!exit` (see later section).

3.2 Interacting with FEEL

When Feel starts, the top-level is initially set to the `root` module. The only operations defined in the `root` module are those for loading modules and entering modules (see below). In order to get access to the usual Lisp list manipulations, function definition and so on, defined at level-0 of the language, it is necessasry to enter the `standard` module (for example, by typing `(!> standard)`). The top-level prompt provides information about which module is the current focus and the history number of the command, for example:

```
eulisp:0:root!3>
```

signifies that the top-level is executing on thread `0`, the current module focus is `root` and that the command history index of this line is `3`.

```
eulisp-handler:0:sockets!1>
```

signifies that the top-level handler is executing on thread `0`, the current module focus is `sockets` and that the command history index of this line is `1`. The following operations are defined in the `root` module:

`(reload-module` *symbol*`)` → *module* FEEL **special form**
`(!>>` *module*`)` FEEL **syntax**

Reload the specified module. This has the side-effect of resetting the exported bindings of the module, such that any importing module will reference the new values. This operation may be abbreviated to `!>>` which will cause the specified module to be reloaded and change the focus to that module. Unlike `reload-module`, `!>>` is recognized everywhere.

`(load-loudly )` → *boolean* FEEL **special form**
Switch on verbosity of monitoring during module loading.

`(load-quietly )` → *boolean* FEEL **special form**
Switch off verbosity of monitoring during module loading.

`(loaded-modules )` → *list(symbol)* FEEL **function**
Returns a list of the modules loaded so far.

`(enter-module` *symbol*`)` → *module* FEEL **special form**
`(!>` *module*`)` FEEL **syntax**

If *module-name* names a module which has been loaded, then the top-level is changed to be in that module. If a module named *module-name* is not currently loaded, then FEEL tries to load the module from a file called *module-name*`.em`. If loaded succesfully, top-level is changed to be in that module. This operation may be abbreviated to `!>`. However, it is also important to note that whilst `enter-module` is only defined in `root`, `!>` is recognized everywhere.

`(start-module` *module-name function-name* arg^*`)` → *obj* FEEL **special form**
Calls the function *function-name* in the module *module-name* with the arguments arg^*.

`(load-module` *symbol*`)` → *module* FEEL **special form**
Load the specified module.

`(load-path )` → *list(string)* FEEL **function**
`((setter load-path)` *list(string)*`)` → *list(string)* FEEL **function**
`load-path` returns a list of strings which define the paths currently searched when loading modules and accessing documentation. The setter function permits this load path to be modified dynamically.
In addition, the following features are provided for controlling the focus of the top-level:

`!root` FEEL **syntax**
Changes top-level back to the `root` module.

`!exit` FEEL **syntax**
Within a handler loop, returns to previous top-level. At top-level, EULISP terminates.

`!`*n* FEEL **syntax**
Redo the input sequence number `n`.

`!backtrace` FEEL **syntax**
`!b` FEEL **syntax**

`!q` FEEL **syntax**

Within the handler loop, prints out a backtrace of function calls and their environments. This may be abbreiviated to `!b`. A simpler backtrace, only containing the function calls, can be printed out by typing `!q`.

4 Modules

`(dynamic-accessible-p` *module name*`)` → *boolean* FEEL **function**
Returns true if *name* is exported from *module*.

`(dynamic-access` *module name*`)` → *obj* FEEL **function**
Returns the value bound to *name* in *module*.

`(dynamic-load-module` *expression*`)` → *module* FEEL **function**
The version of `load-module` that can be used in programs. In addition, the argument is evaluated to get the name of the module to be loaded.

`(module-exports` *module*`)` → *list(symbol)* FEEL **function**
Returns a list of the names exported by *module*.

`(module-name` *module*`)` → *symbol* FEEL **function**
Returns a symbol which is the name of *module*.

`(get-module` *name*`)` → *obj* FEEL **function**
If the module named *name* is loaded, that module object is returned. Otherwise returns `()`.

5 Start-Up Configuration

FEEL's default starting behaviour may be modified in two ways ...

Command Line Arguments The interpreter recognises a number of command line arguments which are basically -heap, -stack-space and -do but you didn't want to know that. Actually:

-heap *n* The size of heap to use (in megabytes if $n < 50$, else bytes). Feel needs at least a 1.5 meg heap.

-do *cmds* A list of things to do on startup

-stack-space *n* Amount of storage to allocate for stacks and static data. This defaults to 1, but should be more for programs that use threads.

-boot Name of bytecode image file to load. See later

-map Produce bytecode map. See later.

-procs *n* Start up using n processors. Works in SystemV configuration only.

A Configuration File Having first processed its command line arguments, FEEL then looks for a file called `.feelrc` in the `$HOME` directory of the user[1]. If found, the file is read and the expressions within executed as if entered at top level.

6 Objects

The EULISP object system is called TELOS. Every data item in EULISP is part of the class hierarchy. Simple classes can be defined by `defstruct`, more complex classes with `defclass`. There is no message send primitive in EULISP, instead generic functions are used. TELOS has been designed to offer programmability, efficiency and flexibility and the next three subsections attempt to illustrate the kinds of things you can do with it by means of a few examples.

6.1 Generic Functions

You see, it's like this...[2]

[1] Likely to be elsewhere on non-UNIX systems
[2] to quote Keith

6.1.1 Univariate polynomials over the integers

We start with a polynomial structure: this is a single term, with a reductum that is the rest of the polynomial. A reductum that is an integer marks the end of the polynomial. A term consists of the leading degree and the leading coefficient.

```
(defstruct polynomial ()
((ldeg accessor ldeg initarg ldeg initform 1)
(lc accessor lc initarg lc initform 1)
(red accessor red initarg red initform 0))
constructor make-polynomial)
```

We define a method on `equal` so we can check if two polynomials are the same. Notice we do not have to check for the bottoming-out of the recursion on the reducta: the generic nature of `equal` ensures that when we get to the end of a polynomial (and we have an integer as a reductum rather than a polynomial) a different method is called. This relies on the fact that equal-methods for (int, poly) and (poly, int) do not exist: the generic function discriminator chooses the nearest applicable method on `equal`, which in this case is (`object`, `object`). This method returns `()` (as the args cannot be `eq`), which is just what we want.

```
(defmethod equal ((p polynomial) (q polynomial))
  (and (equal (ldeg p) (ldeg q))
       (equal (lc p) (lc q)) (equal (red p) (red q))))
```

Adding polynomials. We handle the cases of (`polynomial`, `integer`), (`integer`, `polynomial`) and (`polynomial`, `polynomial`) by defining a method for each.

```
  (defmethod binary-plus ((p polynomial) (q integer))
  (if (zerop q) p
    (make-polynomial 'ldeg (ldeg p) 'lc (lc p)
                     'red (binary-plus (red p) q))))

  (defmethod binary-plus ((p integer) (q polynomial))
    (binary-plus q p))
```

If we call this next method, we know we have two polynomials on our hands, and we simply add them recursively. A minor wrinkle is when the leading terms cancel: we must take care not to have a leading coefficient of 0.

```
  (defmethod binary-plus ((p polynomial) (q polynomial))
  (cond ((= (ldeg p) (ldeg q))
    (let ((sum (binary-plus (lc p) (lc q))))
      (if (zerop sum) (binary-plus (red p) (red q))
        (make-polynomial 'ldeg (ldeg p) 'lc sum 'red
        (binary-plus (red p) (red q))))))
   ((< (ldeg p) (ldeg q))
    (make-polynomial 'ldeg (ldeg q) 'lc (lc q)
                     'red (binary-plus p (red q))))
   (t (make-polynomial 'ldeg (ldeg p) 'lc (lc p)
                       'red (binary-plus (red p) q)))))

  (defmethod binary-difference ...
```

and so on for the other arithmetic operations. Also we would put new methods on `generic-prin` and `generic-write` to print out the values of polynomials using a suitable syntax.

6.2 Classes

Classes in EuLisp are not static items: they can be defined and created dynamically just as any other type in the system. The following example demonstrates this by defining a class whose instances are themselves classes, whose instances are modular numbers. The intermediate classes are parameterised by an integer, which are the bases for the modular rings. This also illustrates the use of metaclasses, which control the structure of classes.

We create a metaclass `ZmodN` which is the class of the classes `Zmod3`, `Zmod5`, `Zmod7`, etc.

```
(defclass ZmodN-class (class) ((n initarg n reader ZmodN-class-n)))
```

This will be a direct subclass of `class`, and so will inherit its methods, in particular the ability to create subclasses which are themselves classes. The instances of this class will have a slot named `n`, which will be the modular base.

Now we define a superclass for all of its instances, to place them in their own sub-hierarchy of the class graph. This class has an instance variable `z`, since the instances of its subclasses are the fully instantiated modular numbers.

```
(defclass ZmodN-object (object)
  ((z initarg z reader ZmodN-z))
  metaclass ZmodN-class)
```

The metaclass of the instances of `ZmodN-object` is defined to be the class `ZmodN-class`. Thus the structure of the instances (the classes `Zmod5`, etc.) is determined by `ZmodN-class`.

The constructor for the instances of `ZmodN-class` (the metaclass) could be the following:

```
(defun make-ZmodN-class (n)
   (make-instance ZmodN-class 'n n
                  'direct-superclasses (list ZmodN-object)))
```

The `make-instance` requires values for the slots in `ZmodN-class`, which include `n` (the slot we defined), and `direct-superclasses`, a slot inherited from `class`.

If you want to avoid creating duplicate `ZmodN` classes with the same N, try this definition instead:

```
(deflocal class-table (make-table))

(defun make-ZmodN-class (n)
  (or (table-ref class-table n)
      (let ((new-class (make-instance ZmodN-class 'n n
                                      'direct-superclasses (list ZmodN-object))))
        ((setter table-ref) class-table n new-class)
         new-class)))
```

The function to create the modular objects themselves could be defined as follows:

```
(defun make-modular-number (z n)
   (make-instance (make-ZmodN-class n) 'z z))
```

Getting `z` from one of these instances is already defined by the reader on `ZmodN-object`. Getting `n` involves going to the class. Making this available from instances means defining the following function:

```
(defgeneric ZmodN-n (obj))

(defmethod ZmodN-n ((ZmodN ZmodN-object))
  (ZmodN-class-n (class-of ZmodN)))
```

Next, we want to define some simple arithmetic on modular numbers, for example, addition. However, this only makes sense if we have the same modulus in both of the summands.

```
  (defun compatible-moduli (n m) (if (= (ZmodN-n n) (ZmodN-n m)) t
(error "incompatible moduli" Internal-Error)))
```

We define a method for addition on `ZmodN-object`: this will then be inherited by each instance, viz., the actual rings `Zmod3`, `Zmod5`, and so on.

```
  (defmethod binary-plus ((n1 ZmodN-object) (n2 ZmodN-object))
    (when (compatible-moduli n1 n2)
          (let ((mod (ZmodN-n n1))
                (z1 (ZmodN-z n1))
                (z2 (ZmodN-z n2)))
            (cond ((zerop z1) n2)
                  ((zerop z2) n1)
                  (t (make-modular-number (remainder (+ z1 z2) mod) mod))))))
```

We can add a method to the print function to view numbers prettily

```
  (defmethod generic-prin ((n ZmodN-object) s)
    (format s "~a<mod ~a>" (ZmodN-z n) (ZmodN-n n)))
```

Finally, some examples of numbers

```
  (deflocal zero5 (make-modular-number 0 5)) (deflocal one5
(make-modular-number 1 5)) (deflocal two5 (make-modular-number 2 5))
(deflocal three5 (make-modular-number 3 5)) (deflocal four5
(make-modular-number 4 5))

  (deflocal zero3 (make-modular-number 0 3)) (deflocal one3
(make-modular-number 1 3)) (deflocal two3 (make-modular-number 2 3))
```

Now if we try an addition:

```
> (+ two5 four5)
< 1<mod 5>
```

We didn't have to specify a plus method for each modular ring individually: the single definition on the superclass suffices.

Thanks to Harley Davis for help on this section.

6.3 Slot Descriptions

Another aspect of the programmability of ΤΕΛΟΣ is slot-descriptions. This allows the user to control how the slots of a class are accessed. Here we present an example of the use of slot-descriptions to provide a classed (typed) slot facility. The aim is to be able to define a class and, at the same time, the class of the values to be associated with a given slot. The solution is to define a new kind of slot-description to verify that only values of the correct class are stored in the slot. We start by defining a new kind of slot-description `classed-local-slot-description`.

```
(defclass classed-local-slot-description (local-slot-description)
((contents-class initform object initarg contents-class accessor
classed-local-slot-description-contents-class)) metaclass
slot-description-class)
```

The classed-local-slot-description class inherits the normal slots from `local-slot-description` and adds somewhere to keep track of the allowed class of its contents.

To police the class (type) constraint, we must check that whenever a value is written to a slot with this class—that the value is of the specified kind. A new method on (setter slot-value-using-slot-description) for classed slots will do this.

```
(defmethod (setter slot-value-using-slot-description) (obj (sd
classed-local-slot-description) val)

  (if (not (subclassp (class-of val)
(classed-local-slot-description-contents-class sd))) ;; Bad class
(error "invalid class of value for slot" some-error 'object obj
'slot-description sd 'value val) ;; OK (call-next-method)))
```

The `call-next-method` is reached only if the value satisfies the class constraint. It just means the value is acceptable—go ahead and do whatever you normally do to put the slot value inside.

All that remains is how to use one of these slots in a class. The example you give can be done as follows—but remember that `defclass` must be used instead of `defstruct` because the latter does not support user-defined slot classes.

```
(defclass person () ((age slot-class classed-local-slot-description
slot-initargs (contents-class integer) accessor age) (name slot-class
classed-local-slot-description slot-initargs (contents-class string)
accessor name) (ordinary-slot initform 'bleagh)))
```

The slots `age` and `name` are of the new class of slot with their contents class set to `integer` and `string` respectively. Of course, other slots with different classes of slot description may also be defined.

Now, we may type the following:

```
(setq i (make-instance person)) ((setter age) i 27)
```

which is fine and `(age i)` will return 27.

```
((setter age) i 'not-a-number)
```

but this signals an error. Thanks to Luis Mandel for prompting this example.

7 Differences between FEEL and EuLisp

Inevitably there are a number of minor ways in which FEELis not an accurate implementation of EuLisp. This section outlines these differences. The 0.69' version is taken as the main version.

The whole of Level 0 is implemented. Much of Level 1 is also running. The library modules (section 5 of EuLisp0.69) are much more patchy.

A major area of incompatability is in the conditions, some of which are not used when the document says they should. This is being improved daily[3]

7.1 Input and Output

The function `read` is supposed to take zero or one argument, but in FEEL it requires one argument which must be a stream. In order to make it easier to use the value `nil` is taken as an abbreviation for `(standard-input-stream)`.

The same restrictions apply to `read-char`, `read-byte`, `peek-char` and `peek-byte`.

[3]at least that is the intention!

7.2 Elementary Functions Module

All the functions are implemented except `generic-log`. The `log` function is not complete, in that it does not check its argumets correctly. It looks like an n-ary function, and takes notice of the first one or two arguments only. The constant `pi` does exist with the value 3.141592653589794. The functions `sin`, `cos` are implemented as in the definition. In addition the constant `e` is set to to 2.718281828459046.

The major difference in this module is that the elementary methematical functions are implemented as functions rather than generic functions as specified in EuLisp.

7.3 List Operator Module

The functions `append`, `last-pair`, `memq`, `null`, `nreverse`, `mapc`, and `mapcar` are implemented as in the specification. The functions `assoc` and `member` require three arguments, and as such the predicate is not optional. These functions are in the base code.

The functions `assq`, `list-copy`, `copy-alist`, `list-tail`, `list-ref`, `reverse` `posq` and `pos` are defined in interpreted code in the module **standard**.

The functions `maplist`, `mapcon` and `mapcan` are only implemented partially. `maplist` only exists in a two argument version, and the other two exist in two, three and four argument versions only.

`tconc`, `lconc` and `mapl` are not implemented.

7.4 String operators Module

This module is all implemented.

7.5 Formatted-IO Module

The function `format` is defined, and understands the esacapes `~a`, `~s`, `~t`, `~%` and `~~` only. It also implements the additional escape `~u` for printing lists with hexadecimal values for system debugging. The numerical formats for binary, octal decimal and hexadecimal are only implemented for fixed integers. The treatment of the field sizes in g, e and f formats are very inconsistent.

The function `scan` is not implemented.

8 Debugging

There is a simple tracing facility, which may be used by loading the module `trace` and importing it into the module where the binding to be traced exists. Note that it is only possible to trace bindings in interpreted modules—the system does not allow bindings in C modules to be reset. It is not possible to trace an imported binding, since all exports are immutable, in the importing module. Instead, change focus to the module defining the binding, import the trace module and trace the binding there.

`(trace-bindings` *symbol** `)` FEEL **macro**
The top lexical binding named by each *symbol* is updated with a function which embeds the original value of *symbol*. When *symbol* is called the arguments will be printed at an indentation corresponding to the current call depth. On exit, the result will be printed. All output is sent to `trace-output-stream`.

`(untrace-bindings` *symbol** `)` FEEL **macro**
The top lexical binding of each *symbol* is updated with the value of the top lexical bindings saved at the time `trace-bindings` was called.

9 Pretty printing

There is an elementary prettyprinter in the module `pretty`. It exports the following three functions.

(`prettyprint` *object*) → *object* FEEL **function**
The answer is the same as the argument, but as a sideffect a pretty-printed form of the object is printed. The layout is controlled by a table which can be modified. In theory (but not yet in practice) setting the variable `*symmetric*` controls whether the printed form is capable of re-entry.

(`superprintm` *object integer*) → *object* FEEL **function**
The object is pretty-printed indented by the integer value. Otherwise as prettyprint.

(`superprinm` *object integer*) → *object* FEEL **function**
A `superprintm` except that there is no final new line printed.

10 AVL tree Module

To be written The source code is a reasonable description

11 OPS5 Module

To be written

12 Graphics

To minimise our implementation effort and to maximise portability and familiarity, FEEL is interfaced to the X-Windows system. However, we decided that a direct interface to the X server such as provided by CLX in Common was too low-level for the majority of the operations we wanted (clearly this is a personal viewpoint) and so we use the YY server developed by Masayuki Ida and his resarch team at Aoyama Gakuin University. The YY server provides a higher level of abstraction than the X server at the cost of more communication and another heavyweight process. Thus, FEEL sockets to YY and YY sockets to X. Documentation for the YY system itself is provided separately. This section contains details of the Eulisp to YY interface package. The package comes in 3 parts: a server, a low level interface and an API. It is available by anonymous ftp from `csrl.aoyama.jp`. For further information contact the YYonX mailing list—`yyonx@csrl.aoyama.jp`. The program as delivered, comprises a client and a server. The client is an Applications Program Interface (API) written in Common Lisp and CLOS and is not used by the FEEL system.

12.1 The YY server

The YY server is the program which connects to an X display from an application program (the client). It acts on the commands of the client, and conveys input to the display back to the client, after a filtering stage. Because it works over internet sockets, it can run on any machine on the local net, and so need not compete for CPU cycles with FEEL. FEEL can use any X windows display for its output as long as the machine running the server has permision to use that display. Thus up to three machines can be involved in the process of producing a picture. The server currently runs on sun-3 and sun-4 machines, and may have difficulty communicating with machines which use a different byte order.

12.1.1 Starting the YY Server

The server is started from the YY-server directory with the command `serv`. There is a `-p` <*arg*> option which selects the port number. The argument should be a non-negative even number. The port number defaults to 0.

12.2 The Eulisp YY module

The YY interface requires a number of extra C level functions, which are compiled and linked to make a distinct executable called 'look' or similar in the distribution.

These C functions can be accessed via the *YY* module, together with some functions to make them simpler to use. This module should contain all the necessary functions to connect and run a YY session. Any function that is not included in the module but defined within the YY protocol specification should be in the module, although several have not yet been implemented in the server and therefore cannot be used by the module. The details of the connection to the server, including a description of the low level interface are described in a later section.

12.2.1 Connecting to the YY-Server

Before connecting to the server, the program must ensure that the server is running and able to connect (or not be over-worried about breaking at this point). The normal way is one of:

- ask the user to type `serv` on an appropriate machine
- run a remote-shell command, for instance:

```
(system "rsh <server-host> serv")
```

- hope that the server is running.

The third is the usual option, and can easily be achieved by invoking EuLisp from a shell script, which also starts the server program.

`(YY-connect` *server port*`)` → *boolean* **YY function**
This function that makes the connection from EuLisp to the YY server. *server* is the name of the machine running the server, and *port* argument should be equal to the `-p` option given to the `serv` command, or `0` if the `-p` option was not given.

`(initialise-server` *X-display Window-name Icon-name*`)` → *structure(Info)* **YY function**
After connecting, the server must initialise the YY server with the name of the X11 display to use. This is relative to the server, so specifying "unix:0.0" means the console display of the machine running the server. The name of the window, in its normal and iconified form should also be specified. An object of type *Info*, which can be used to query various constants about the display (eg. display resoloution, etc) is returned.

`(YY-initialised-p )` → *bool* **YY function**
Returns true if initialise-server has been called.

12.3 Territories

The central class of object for manipulating graphics is the *Territory*. A territory has the ability to display data, be moved and raised or lowered relative to other territories. The concept is very similar to that of windows in the X11 world, except that the programmer has more control over their movement. They provide the means by which functions reference the display, and allow input from the user. The coordinate frame used by territories is that of X-windows: X ordinates increasing left to right and Y ordinates increasing from top to bottom, both relative to the top left corner of the territory. There is no facility to rescale these at the server layer.

12.4 Input-Territories

Not yet available

12.5 Classes

In this section the function interface to the classes is decribed. The interface is not yet complete, so a selection of the functions available is given.

12.5.1 Territory

`(make-Territory` *initlist*`)` → *Territory* **YY function**

A new territory is created by `make-Territory` (via `make-instance`). This also informs the server of the territory's existence. The following initargs are used:

parent this argument is an object of class `Territory`, or `nil`. `nil` specifies that the object is a root territory, and therefore has no parent. It is not an error to have more than one root territory

area this must be an object of class `Area`, and specifies the size and position (relative to the parent) of the territory

top left bottom right Specify the area as above. If the `area` initarg is given, then it takes precedence

visible specifies whether the window can be displayed by the X-server. If this is `nil`, then the territory is can only be used as a temporary store for pixel data.

drawable some territories can be used for reading input information holding only, or as a place holder in the tree of territories. The root territory should not be specified as drawable and therefore cannot be used for drawing.

`(clear-territory` *colour*`)` → *null* **YY function**

Clears the territory to the specified colour.

`(move-territory` *territory point*`)` → *null* **YY function**

Moves the territory to the specified point

`(destroy-territory` *territory*`)` → *null* **YY function**

Kills the specified territory.

`(display-territory` *territory*`)` → *null* **YY function**

If flag is true then map the territory, if nil, hide it.

`(Territory-width` *territory*`)` → *number* **YY function**
`(Territory-height` *territory*`)` → *number* **YY function**

`(Territory-top` *territory*`)` → *number* **YY function**

`(Territory-left` *territory*`)` → *number* **YY function**

Give the approporiate dimension of the territory.

12.5.2 Colours

`(make-Colour` *arg-list*`)` → *Colour* **YY function**

makes an instance of class `Colour`. It takes the following initargs:

red an integer in the range [0,66535]

green an integer in the range [0,66535]

blue an integer in the range [0,66535]

These specify the amount of red, green and blue in the colour. The actual colour displayed on screen is defined by the X display that the server is connected to. The initargs may be abbreviated to **r**, **g**, and **b** respectively.

`(Black-Colour )` → *Colour* **YY function**
`(White-Colour )` → *Colour* **YY function**

Returns the appropriate colour, or nil if the YY server is not yet connected.

12.5.3 Fonts

`(load-font` *fontname*`)` → *Font* **YY function**
This function loads a font into the server. The font must be specified in the `yyfontinfo` file in the YY-server directory.

`(Std-Font )` → *Font* **YY function**
Returns the standard font (called "fixed" in this implementation, defined by the constant *Std-Font* in the YY module.

12.6 Drawing functions

The drawing functions are direct calls on the YY server, and, in consequence, take a large number of arguments, most of which are not used. For an example of a higher level interface see the `turtle` module described later in this document.

The `op` parameter is a flag describing the bitblt operation needed to render the object on the screen. It may take the following values: `GCLEAR, GAND, GANDREVERSE, GCOPY, GANDINTEVERTED, GNOOP, GXOR, GOR, GNOR, GEQIV, GINVERT, GORREVERSE, GCOPYINVERTED, GORINVERTED, GNAND or GSET`. What these operations achieve is specified in the Xlib manuals. The `edge` parameter may take the following values: `SQUARE-LINE-EDGE, SQUARE-LINE-EDGE-WITHOUT-END, or ROUND-LINE-EDGE`, again specified in the X-manuals.

`(draw-point-xy` *territory x y op colour*`)` → *null* **YY function**
Draws a point at the specified (x, y) point using the specified operation and colour.

`(draw-line-xy` *territory x1 y1 x2 y2 width op edge colour dash*`)` → *null* **YY function**
Draws a line form $(x1, y1)$ to $(x2, y2)$ using the specified width, operation, edge type and colour. The dash parameter is currently ignored.

`(draw-string-xy` *territory x y op colour font string*`)` → *null* **YY function**
Writes the string onto the territory at (x, y) using the specified operation and colour. The y coordinate is the baseline of the text.

`(draw-string-centred` *territory x y op colour font string*`)` → *null* **YY function**
Draws the string with its centre at (x, y).

`(draw-filled-rectangle-xy` *territory x y w h op colour source-territory*`)` → *null* **YY function**
Draws the specified rectangle filled using the colour, and using the source territory as a mask. If this is null, then the territory is simply filled according to the given operation.

`(draw-circle-xy` *territory x y r width op colour dash*`)` → *null* **YY function**
Draws a circle of radius r centred at (x, y). The other arguments are as `draw-rectangle-xy`.

`(draw-filled-circle-xy` *territory x r y op colour source-territory*`)` → *null* **YY function**
Similar to draw-filled-rectangle-xy.

`(bitblt-xy` *src-territory s-left s-top dest-territory dest-left dest-top width height op*`)` → *null* **YY function**
General-purpose block copier. Copies bits from the source location to the destination an area of the specified width and height. NB: This function is at present unimplemented.

12.7 Input Functions

Listed here are the low level functions for getting events from the server. A higher level, widget based module is contained in the modules `dispatcher`,`widgets` and button. This is currently under development, so the exact interface is not described here.

`(combine-input-mask` *event-type-1 . events*`)` → *event-mask* **YY function**
Returns an event mask suitable for use by `set-event-mask`. The event-type can be any of: `BUTTON-1-DOWN`, `BUTTON-2-DOWN`, `BUTTON-3-DOWN`, `BUTTON-1-RELEASE`, `BUTTON-2-RELEASE`, `BUTTON-3-RELEASE`, `MOVE-BUTTON`, `ENTER-TERRITORY`, `EXIT-TERRITORY`, `STAY-TERRITORY`, `BUTTON-1-DOUBLE-CLICK`, `BUTTON-2-DOUBLE-CLICK`, `BUTTON-3-DOUBLE-CLICK`, `INTERRUPT-KEY-DOWN`, `META-KEY-DOWN`, `CONTROL-KEY-DOWN`, `SHIFT-KEY-DOWN`.

`(set-event-mask` *territory event-mask*`)` → *event-mask* **YY function**
Sets the event mask on the specified territory. Events that occur in the specified territory which are included in the event-mask will then be placed on the input queue.

`(read-next-event )` → *event-list* **YY function**
Waits (blocks if Feel is single-threaded) for an event on the input queue. The retuned value is a vector consisting of the territory that received the event, a bit-vector representing the event and the location of the cursor when the event occured. If an event occured,

`(bit-vector-ref` *vect event*`)`

returns 1, otherwise it returns 0.

`(sample-next-event )` → *event-list / null* **YY function**
If an event is on the input queue, return it as above, otherwise, return `nil`.

12.8 Input Classes

Will be coverered later.

12.9 Utility Classes

12.9.1 Area

`(make-Area` *initargs*`)` → *Area* **YY function**
`(Area-top` *Area*`)` → *number* **YY function**

`(Area-left` *Area*`)` → *number* **YY function**

`(Area-bottom` *Area*`)` → *number* **YY function**

`(Area-right` *Area*`)` → *number* **YY function**

`(Area-width` *Area*`)` → *number* **YY function**

`(Area-height` *Area*`)` → *number* **YY function**

An area is specified by its top left coordinate and its bottom right coordinate.

12.10 Any Other Business

An example of a YY application is given in the appendix.

13 Thread Abstractions

EuLisp provides a set of primitive operations for thread creation and manipulation, but for most work these are too low-level and require the user to be overly concerned with their management. It is also true that one of the design goals of EuLisp was to provide an experimentation environment for parallel processing, so it should not be surprising that several thread abstractions have been built on the EuLisp thread primitives. So far these abstractions comprise: *futures*, *linda* and *timewarp*. The next three subsections describe them in detail.

13.1 Futures

The nature of the EuLisp thread mechanism means that it lends itself quite naturally to providing a base for the implementation of a simple future abstraction. The acts of creating futures and of eventually interrogating them for their values map almost directly onto starting threads and accessing thread results.

The code for basic future manipulation is given below. A couple of examples of replacements for "strict" functions that allow for future objects are shown. The extensibility of generic functions and module renaming can be used to make these necessary changes transparent for users.

`(future` *expression*`)` **Future macro**
Constructs a future object and spawns a thread to calculate the value of *expression*. An object of class *future* is returned by the expression resulting from the macro expansion. The implementation of `future` in Feel is:

```
(defmacro future exp
  `(let
     ((future (make-future-object))
      (task (make-thread
              (lambda (future fun)
                ((setter future-object-value) future (fun))
                ((setter future-object-done) future t)
                t))))
       ((setter future-object-thread) future task)
       ((setter future-object-function) future (lambda () ,@exp))
(thread-start task future (lambda () ,@exp))
future))
```

`(futurep` *obj*`)` → *boolean* **Future generic**
`(futurep` *future*`)` → *t* **Future `futurep` method**
`(futurep` *obj*`)` → *null* **Future `futurep` method**
Returns `t` if *obj* is a *future*, otherwise `()`.

`(make-future-object` *function thread value done*`)` → *future* **Future constructor**
`(future-object-function` *future*`)` → *function* **Future function**
`(future-object-thread` *future*`)` → *thread* **Future function**
`(future-object-value` *future*`)` → *obj* **Future function**
`(future-object-done` *future*`)` → *boolean* **Future function**
A *future* is constructed by `make-future-object` and the four fields of a *future* are accessed by: `future-object-function`, which returns the function being evaluated, `future-object-thread`, which returns the thread on which the function is being evaluated, `future-object-value`, which returns the value computed by the future and `future-object-done` which is flag indicating whether the future has completed or not.

`(future-value` *future*`)` → *obj* **Future function**
Forces the evaluation of a *future* and if the result of the evaluation is also a *future* that too is forced until the result is not *future*.

`(future-select` *future-list*`)` → *obj* **Future function**
Return the first of future-list to complete

13.2 Linda

To be written

13.3 Communicating Sequential Processes

The CSP module provides a new class of thread, called a `CSP-thread`, the channel class and several syntactic extensions. These extension are detailed below.

13.3.1 Channels

Channels are the basic form of interprocess communication in CSP. They provide a means for 2 processes to communicate via a synchronous link. In this version of CSP there area two types of channel: the simple channel and the channel pair. A simple channel is specified at dynamically by the functions `connect-channel-input` and `connect-channel-output`. Only `IN` operations are permitted on the channel returned by the former, and only *OUT* operations on the latter. These operations are only alowed on the thread that connected the channel or one that it created. A channel pair is made up of two channels and is connected to a thead by *connect-chan-pair*

13.3.2 Bindings

`(make-Channel )` → *Channel* **CSP function**
Returns an unconnected simple channel

`(make-Chan-Pair )` → *Channel-Pair* **CSP function**
Returns an unconnected channel pair

`(connect-channel-input` *channel*`)` → *channel'* **CSP function**
Connects the thread executing the statement to the input end of the channel.

`(connect-channel-output` *channel*`)` → *channel'* **CSP function**
Connects the thread executng the statement to the output end of the channel.

`(connect-chan-pair` *channel-pair*`)` → *connected-channel-pair* **CSP function**
Returns one end of the specified channel as a connected channel.

`(IN` *channel . variable*`)` **CSP macro**
Waits until transmitter (process on the other end of *channel*) is ready to send data (signaled by doing `OUT` on *channel*) and then reads the object from *channel* assigning it to *variable*. If the *variable* is ommited, IN simply returns the value on the channel.

`(OUT` *channel obj*`)` **CSP macro**
Waits until reciever (process on the other end of channel) is ready to receive data (by doing `IN`) and then outputs *obj* to the channel *channel*.

`(PAR` *expression**`)` **CSP macro**
The expression can be any lisp expression. Execute each *expression* as a seperate thread. The construct waits until all its subexpressions have completed, and returns a list of values returned by the expressions.

`(MAPPAR` *function list*`)` **CSP macro**
Apply *function*, which should take one argument to each element of the list, as a parallel operation.

`(FOR )` **CSP macro**

PAR as iteration over a sequence of values.

`(ALT` *alternative**`)` **CSP macro**

Each alternative has the following form:

((IN *channel variable*) . expression*)

When one one the listed channels is known to be ready, this executes the code associated with the IN statement with the specified variable bound to the next value on the channel. This construct should be viewed as non-deterministic – ie. it does not necessarily return the first ready channel.

`(IN-FROM` *(channel-var value-var) channels . expressions*`)` **CSP macro**

When one of the channels is known to be ready, the expressions are evaluated with *channel-var* bound to the channel, and *value-var* to the value on that channel.

`(SEQ` *expression**`)` **CSP macro**

The same as progn, ie execute the following expressions in the order given.

13.4 Time Warp

To be written

14 Distributed Processing

The basis for distributed processing in FEEL under Unix is supplied by the `sockets` module, which exports the functions defined below.

`(socketp` *obj*`)` → *boolean* FEEL **function**

If *obj* is a `socket` returns `t`, otherwise `()`.

`(make-listener )` → *listener* FEEL **function**

Allocates a fresh `listener` object.

`(make-socket )` → *socket* FEEL **function**

Allocates a fresh `socket` object.

`(listener-id` *listener*`)` → *pair* FEEL **function**

Returns a pair, whose car field contains a symbol naming the local host and whose cdr field is a port number on that host.

`(listen` *listener*`)` → *socket* FEEL **function**

Listens on the port number returned by `listener-id` applied to *listener* and returns `socket` when a connection is established.

`(connect` *pair*`)` → *socket* FEEL **function**

The pair contains the information returned by `listener-id` and makes a connection to the named machine on the specified port, returning *socket* which is the handle on the established connection bewteen the two processes.

`(close-listener` *listener*`)` → *null* FEEL **function**

Changes internal state of *listener* so that it can no longer be used for listening.

`(close-socket` *socket*`)` → *null* FEEL **function**

Flushes all pending data related to *socket* and changes the internal state of *socket* so that it is no longer readable or writable.

`(socket-write` *socket obj*`)` → *obj* FEEL **function**

Write *obj* to *socket*, returning *obj*.

`(socket-read` *socket*`)` → *obj* FEEL **function**
Read from *socket* returning the constructed LISP expression *obj*.

`(socket-readable-p` *socket*`)` → *boolean* FEEL **function**
If there is data available for reading from *socket*, returns `t`, otherwise `()`.

`(socket-writable-p` *socket*`)` → *boolean* FEEL **function**
If data can be written to *socket*, returns `t`, otherwise `()`.

Figure 1 is two scripts of a simple example of establishing a socket connection and a dialogue across the connection

machine-1	machine-2

`eulisp:0:root!0> (!> standard)`
`Loading module 'standard'`
`Loading module 'extras'`
`Loaded 'extras'`
`Loaded 'standard'`
`eulisp:0:standard!0< standard`
 	
`eulisp:0:standard!1> (import sockets)`
`eulisp:0:standard!1< ()`
 	
`eulisp:0:standard!2> (setq s`
`  (connect '(machine-2 . 1236)))`
 	
`eulisp:0:standard!2< #socket(3,3)`
 	
`eulisp:0:standard!3> (socket-read s)`
`eulisp:0:standard!3< 1`
 	
`eulisp:0:standard!4> (socket-write s 2)`
`eulisp:0:standard!4< 2`
	`eulisp:0:root!0> (!> standard)`
	`Loading module 'standard'`
	`Loading module 'extras'`
	`Loaded 'extras'`
	`Loaded 'standard'`
	`eulisp:0:standard!0< standard`
	
	`eulisp:0:standard!1> (import sockets)`
	`eulisp:0:standard!1< ()`
	
	`eulisp:0:standard!2> (setq l`
	`  (make-listener))`
	
	`eulisp:0:standard!2< #listener(3,1)`
	
	`eulisp:0:standard!3> (listener-id l)`
	`eulisp:0:standard!3< (machine-2 . 1236)`
	
	`eulisp:0:standard!4> (setq s (listen l))`
	`eulisp:0:standard!4< #socket(4,3)`
	
	`eulisp:0:standard!5> (socket-write s 1)`
	`eulisp:0:standard!5< 1`
	
	`eulisp:0:standard!6> (socket-read s )`
	`eulisp:0:standard!6< 2`

Figure 1: Example socket based communication

14.1 Linda

To be written

14.2 Time Warp

To be written

14.3 PVM

14.4 The PVM module

The pvm module provides an interface to the pvm library. This section assumes that the reader has read at least some of the pvm documentation.

The module differs from the pvm library in the following ways:

- Arbitrary lisp expressions (including circular structures) may be sent from machine to machine
- The format in which objects are sent is not the XDR format used by pvm, but an internal format. It is hopefully machine (and byte order) independent. See the section on the reader module for more details.
- Several reads may occur simultaneously on separate threads [4]. In other words, it is possible to call thread-suspend during a read.

The module exports the following functions:

`(make-pvm-id` *string*`)` → *id* **PVM function**
creates a pvm-id which can be used to broadcast to a group of remote processes.

`(pvm-status` *id*`)` → *bool* **PVM function**
Query the status of the process with id *id*.

(function-body`(pvm-send` *dest type msg . reader*`)` → *any* **PVM fun**

Send a message of type *type* to the process specified by the id *dest* containing the value *msg*. If a reader is specified it is used to handle any complex lisp types inside the message.

`(pvm-recv` *type info? . reader*`)` → *msg* **PVM function**
Block until a message of type *type* is recieved. If *info?* is nil, then the message is returned. If *info?* is non-nil, a list is returned in the following format: `(`*msg type from*`)` where msg is the message, type is the type and from is the process-id of the sending processes.

`(pvm-recv-multi` *type-list info? . reader*`)` → *msg* **PVM function**
As pvm-recv, but blocks until a message which has a type in the type-list.

`(pvm-initiate-by-type` *type name*`)` → *id* **PVM function**
Start a process on a host of the specified type with the name *name*. It returns the pvm identifier of the process.

`(pvm-initiate-by-hostname` *hostname name*`)` → *id* **PVM function**
Start a process on the host with name *hostname* with the name *name*

`(pvm-enroll` *name*`)` → *id* **PVM function**
Enroll into pvm under the given name. Must be called before any other pvm function.

`(pvm-leave )` → *any* **PVM function**
Exit from pvm-control. After this is called, all pvm functions return an error message (except pvm-enroll).

`(pvm-probe` *type*`)` → *bool* **PVM function**
Test for messages of a given type. Returns the type, or nil if no message of that type is in the input queue.

`(pvm-probe-multi` *...*`)` → *bool* **PVM function**
Test for messages from a list of types. Not yet implemented. Can be easily simulated with probe.

`(pvm-whoami )` → *pvm-id* **PVM function**
Return the pvm-id (the value returned by enroll) of the process.

[4] note that pvm is not yet interfaced to the System V version.

`(pvm-make-id-from-pair` *pair*`)` → *id* **PVM function**
construct a pvm-identifier from a cons cell. Mostly used when passing addresses around — when a pvm-id is sent, it is read as a cons-cell.

`pvm-id:`*class* **PVM constant**
The class of pvm-identifiers. It is a subclass of pair.

The other functions provided are

- pvm-barrier
- pvm-ready
- pvm-waituntil
- pvm-terminate

The Feel versions are untested, but ought to work. See the PVM documentation for details about their functionality.

15 The Reader Module

The reader module provides functions to read and write lisp forms as bytevectors. It is intended to be reasonably machine independent, although at the current time it falls a little short. The module currently deals with reading and writing lisp forms for the pvm, socket and dbm modules.

The module exports the following interface (for use in user modules):

```
/*
  * obread.h
  * interface for obread
  */

/* class of the reader */

extern LispObject object_reader;

/* functions */

extern void write_obj(LispObject *,LispObject, unsigned char **,
                      LispObject);
extern LispObject read_obj(LispObject *,unsigned char  ast*, LispObject);

#define EUBUG(x)
```

The reader in its default form can read any 'simple' lisp expression that is: integers, floats, strings, symbols[5], lists and vectors. The extensibility is provided via an extra argument which may be supplied to control the reader's behaviour on complex lisp types. A type here means a group of classes which can be read in the same way. The type of an object is given by the integer identifier passed to `add-writer` and `add-reader`.

`(make-obj-reader )` → *reader* **Reader function**
Makes a new reader object. The class and internals of this object are left unspecified.

[5]Support for symbols may be removed in future versions because they may require some caching, which will be provided by a lisp level

`(add-writer` *reader class type-ident function*`)` → *any* **Reader function**

This function adds a new writer function, *function* to the given reader. The function is called when an object of class *class* (or one of its subclasses) is encountered by a write process. It is called with three arguments: the object to be written, a value representing write buffer and the reader which called the function. The function should call `write-next` with any data associated with the object.

`(add-reader` *reader type-ident function*`)` → *any* **Reader function**

This function adds a new reader function `function` to the given reader. The function is called whenever an object of type `type-ident` is encountered by a read process. It is called with two arguments: a value representing the read buffer plus the reader supplied by the caller of the read. The function then calls `read-next` to obtain any data associated with the object. If the function fails to consume all the data written by its corresponding write, an unhandled error condition results[6].

`(read-next` *ptr reader*`)` → *obj* **Reader function**

This function returns the next object in the read-buffer specified by *ptr*, using *reader* as the reader object. It can only be called inside the dynamic scope of a read function.

`(write-next` *object ptr reader*`)` → *any* **Reader function**

This function writes the object *object* onto the write-buffer specified by ptr, using *reader* as the reader object.

15.1 Example

```
;; define a structure which we want to pass around}

(defstruct silly-cons-pair ()
  ((car initarg car reader silly-car)
   (cdr initarg cdr reader silly-cdr))
  constructor (silly-cons car cdr))

;; invent a number --- this *must* be more than 16
(defconstant *silly-type-id* 18)

;; make a reader

(defconstant *the-reader* (make-obj-reader))

;; define readers and writers for silly-cons

;; note that both these functions *can* side effect, so circular
;; structures and caching can be handled (using tables or similar), also that the
;; particular reader can be changed for the recursive call
;; to the reader (although I do neither here).

(defun write-silly-cons (obj ptr rdr)
  ;; easy really. Just write whats inside.
  (write-next (silly-car obj) ptr rdr)
  (write-next (silly-cdr obj) ptr rdr))

(defun read-silly-cons (ptr rdr)
  ;; read the internals
```

[6] Feel goes kaboom

```
  (let* ((a-car (read-next ptr rdr))
         (a-cdr (read-next ptr rdr)))
    ;; construct the appropriate object
    (silly-cons a-car a-cdr)))

;; add them to the reader structure

(add-reader *the-reader*
            *silly-type-id*
            read-silly-cons)
(add-writer *the-reader*
            silly-cons-pair
            *silly-type-id*
            write-silly-cons)

;; we can add more types later...

;; should make
;; (pvm-send (pvm-whoami) 102
;;      (silly-cons (silly-cons 1 2)
;;                  (silly-cons 3 4))
;;            *the-reader*)
;; work ok.

;; to receive, (pvm-recv 102 nil *the-reader*)
```

16 Bytecode Compiler

The Feel bytecode interpreter is implemented as an add-on to feel, rather than the integral part of the system that, in an ideal world, it would be. The code produced is quite respectable, and should give significant improvements over interpreted code.

The compiled code does not do any error checking on car, cdr, vector-ref and similar functions. A later extension will define these functions as generic so that type errors can be detected. In fact, all one needs to do is let extras0.em redefine the relevant functions.

16.1 How to run it

16.1.1 File Types

eulisp module files These have a `.em` suffix and contain eulisp source code.

Standard compiled modules These have a `.sc` suffix, and contain position and byte-order independent compiled code.

Interface files These have a `.i` suffix, and contain the interface exported by their module, and information on dependencies, etc.

Bytecode files These have `.ebc` and `.est` sufficies. They hold the raw bytecodes and statics for a group of modules.

16.1.2 Compiling

The compiler is invoked from inside the compile module. This module exports the following bindings:

`(comp2sc` *module-name*`)` → *compiled-object* **Compiler function**
Compile a `.em` file into a `.sc` file, plus a `.i` file.

`((setter optimize-code)` *value*`)` → *old-value* **Compiler function**
If value is non-nil, then the peephole optimiser will be invoked at the end of compilations. This reduces code size by an average 10%, and also makes code execute a little faster. The optimiser handles most obvious optimisations, but does not attempt any cross procedure-call/branch optimisations.

16.1.3 linking

The linker is invoked from the combine module. It exports the following bindings:

`(load-module` *mod-name*`)` → *unspecified* **Compiler function**
Load a module into the current feel image. It will also load any submodules that the module needs. For this to work correctly, there should be no implicit dependencies between the initialisation of modules — if (the initialisation of) module A depends on module B, then A should use a binding from B. For the majority of cases, it should work OK.

`(load-modules` *mod-list*`)` → *unspecified* **Compiler function**
Load all of the modules in mod-list in the given order.

`(combine-user-modules` *image-name mod-list*`)` → *unspecified* **Compiler function**
Link the modules in mod-list with the system module standard0, producing an file which can be used with the -boot option.

`(combine-user-modules-with-desc` *name desc-file mod-list*`)` → *unspecified* **Compiler function**
As above, but use the description file provided to locate external bindings, rather than the system itself. This is primarily for cross-linking.

16.1.4 running

To have feel load an image, `bootimage`, produced by combine-modules, do:

```
feel -boot bootimage
```

16.2 Bootstrapping

`(compile-boot-modules )` → *unspecified* **Compiler function**
This function (in the compile module) compiles the base modules (in Compiler/BootCode) that are needed to produce a bootable image. These modules cannot be compiled using comp2sc, instead, comp2rawsc is used. Note that care should be taken to ensure that the standard0 modules in the Modules directory, rather than the ones in the BootCode directory.

`(make-boot-code` *desc-file name*`)` → *unspecified* **Compiler function**
This function (in the combine module) produces enough of a boot image from a description file such that combine-user-modules can work correctly. The description-file is produced by `feel -map`.

Index