FEEL: An Implementation of EuLisp
Version 0.89

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

July 15, 1993

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, and examples on the use of some eulisp features are provided.

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[1]:

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 . . .

[1]These addresses may not work. The authors of this document are `pab,jap@maths.bath.ac.uk`

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 `user` module[2]. This imports `eulisp0`, and therefore contains most of the usual lisp functions. Most of the module manipulation functionality is defined in the `root` module. The only operations

[2]This can be changed by setting the environment variable FEEL_START_MODULE

defined in the `root` module are those for loading modules and entering modules (see below). 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` *Feel special form*

Arguments

symbol: name of the module to load.

`!>>` *Feel special form*

Arguments

symbol: name of the module to load.

Result

the name of the module.

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` *Feel*

Result

A boolean.

Switch on verbosity of monitoring during module loading.

`load-quietly` *Feel*

Result

A boolean.

Switch off verbosity of monitoring during module loading.

`loaded-modules` *Feel*

Result

A list of the names of the currently loaded modules.

`(!>` *Feel special form*

Arguments

module: Name of module to load.

`enter-module` *Feel special form*

Arguments

module: Name of module to load.

Result

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` *Feel special form*

Arguments

module-name: Name of a module

function-name: Name of a function

arg: argument

Result

Calls the function *function-name* in the module *module-name* with the arguments *arg**.

`load-module` *Feel special form*

Arguments

module-name: A symbol

Load the specified module.

`load-path` *Feelaccessor*

`load-path` *Feelupdator*

Result

`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 special form*

Changes top-level back to the `root` module.

`!exit` *Feel special form*

Within a handler loop, returns to previous top-level. At top-level, EuLisp terminates.

`!`*n* *Feel special form*

Redo the input sequence number `n`.

`!backtrace` *Feel special form*

`!b` *Feel special form*

`!q` *Feel special form*

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`. The `eulisp0` module exports a function, `!B` (not a special form) which may do a better job in some circumstances, but is more prone to infinite loops.

4 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.
- **-stack-size** The size of the interpreter stack. Default 32, max is 64. it should note be necessary to change this unless your program stops with a stack-overflow message. Beware that an infinite non-tail recursion problem may also trigger this message.

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[3]. If found, the file is read and the expressions within executed as if entered at top level.

5 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.

[3]Likely to be elsewhere on non-UNIX systems

5.1 Generic Functions

You see, it's like this. . . [4]

5.1.1 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.

```
(defclass <polynomial> (<number>)
  ((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))))
```

We now need some operations on this new type. If we are trying to add polynomials to integers, we would like some method for converting between integers and polynomials. We use the function `lift-numbers` to do this.

```
  (defmethod lift-numbers ((i <integer>) (p <polynomial>))
    <polynomial>)

  (defmethod lift-numbers ((p <polynomial>) (i <integer>))
    <polynomial>)

  (defmethod (converter <polynomial>) ((x <integer>))
    (make-polynomial 'lc x 'ldeg 0))
```

Now if we call any operation with a polynomial and an integer, the integer is lifted to class polynomial, and the operation proceeds as normal. For two polynomials, the method is easy. 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)
```

[4] to quote Keith

```
                    '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.

5.2 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))
    metaclass <class>)
```

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> (<number>)
    ((z accessor 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 <zmodn-class>
  'direct-superclasses (list <zmodn-object>)
  'name (make-symbol (format nil "<zmod-~a>" n))
  'n n))
```

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:

```
  (defconstant *zmodn-table*
     (make <table> 'comparator = 'hash-function generic-hash))
  (defun make-zmodn-class2 (n)
    (or (table-ref *zmodn-table* n)
(let ((cl (make-zmodn-class n)))
  ((setter table-ref) *zmodn-table* n cl)
  cl)))
```

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

```
  (defun make-modular-number (z n)
     (make-instance (make-zmodn-class2 n) 'z z))
```

Note that this implemenatation guarentees that the number is of the appropriate range:

```
  (defmethod initialize ((proto <zmodn-object>) lst)
    (let ((i (call-next-method)))
      ((setter zmodn-z) i
       (remainder (scan-args 'z lst required-argument)
  (zmodn-n i)))
      i))
```

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 ((z <zmodn-object>))
     (zmodn-class-n (class-of z)))
```

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)
      (make-modular-number (+ (zmodn-z i) (zmodn-z j))
   (zmodn-n i))))
```

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.

5.3 Slot Descriptions

Another aspect of the programmability of TELOS 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
     reader 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. we therefore want a new method on `compute-primitive-writer-using-slot-description`.

```
  (defmethod compute-primitive-writer-using-slot-description
    ((csd <classed-local-slot-description>) cl lst)
    (let ((std-writer (call-next-method))
      (contents-cl (classed-local-slot-description-contents-class csd)))
      (lambda (obj val)
        (if (subclassp (class-of val) contents-cl)
            (std-writer obj val)
          (error "invalid class of value for slot"
                 some-error 'object obj 'sd csd 'val val)))))
```

The call to the standard writer 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>
          initarg age
```

```
        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 <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.

5.4 Mixins

Feel supplies a mixin module[5] to allow the use mixin classes à la flavors. A mixin class is a class that can be used in a multiple inheritance network, but has certain restrictions to enable the creation of more efficient accessors—multiple inheritance is restricted to non-instantiable classes and these classes, *mixins* are then used for specialisation of instantiable objects, *base-objects.* Mixins tend to be used to describe attributes of objects, and then these are "mixed in" with base classes to create specialized classes. The mixin implementation has two metaclasses

- `<mixin-class>` The class of a mixin class
- `<mixin-base-class>` The class of a base-object class

Instances of `<mixin-class>` are not instantiable, but allow full MI. Only instances of `<mixin-base-class>` may inherit from mixin-classes, and the list of direct superclasses of a `<mixin-base-class>` must have all mixin-classes before a single non-mixin class (In Feel, it may inherit from any other class in the system, including `<class>`).

Note on the implementation: `<mixin-class>` has a different default slot type, `<mixin-slot-description>`. When this slot is inherited *directly* by a `<mixin-base-class>` its the accessor is computed. If the slot is not newly created, however, no new access method is computed, therefore reducing the number of such methods for a given accessor.

6 eql methods

These are supplied bye the eql module. Effectively it defines new generic function and method classes, and allows eql methods to be defined. See `eql.em` for full details.

[5]called `mixins`

```
(defclass <point> ()
  ((x initform 0 accessor point-x initarg x)
   (y initform 0 accessor point-y initarg y))
  )

(defclass <colored> ()
  ((color initform 'black initarg color
  reader color))
  metaclass <mixin-class>)

(defgeneric color-of (obj)
  method (((obj <object>)) 'gray)
  method (((obj <colored>))
  (color obj)))

(defclass <colored-point> (<colored> <point>)
  ()
  metaclass <mixin-base-class>)

(setq p1 (make <point>))
(color-of p1)

(setq p2 (make <colored-point> 'x 1 'y 1 'color 'red))
(color-of p2)
```

Figure 1: Usage of mixin inhertance

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 0.98 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[6]

7.1 Input and Output

The EuLispdefinition does not yet have a powerful input/output mechanism. The one in FEELis intended to be a starting point, and no more. It provides input and output primitives, plus scan and format.

7.2 Elementary Functions Module

None of the elementary functions are implemented on integers.

7.3 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

[6]at least that is the intention!

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.

8 Pretty printing

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

`prettyprint` *Feel*

Arguments

object: Object to print

Result

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. The code also shows how one adds new stream-type objects to the system.

9 AVL tree Module

To be written The source code is a reasonable description

10 OPS5 Module

To be written

11 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.

11.1 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` *Future macro*

Syntax

(future (*expression**))

Remarks

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` *Future generic*

Arguments

obj: The object to be tested

Result

nil if obj is not a future, otherwise, non-nil.

`future-value` *Future*

Arguments

future: The value to be evaluated

Result

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 a *future*.

`future-select` *Future*

Arguments

list: a list of futures

Result

Return the first of future-list to complete

11.2 Linda

To be written. See `eulinda.em`

11.3 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.

11.3.1 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*

11.3.2 Bindings

`make-Channel` *CSP*

Result

Returns an unconnected simple channel

`make-Chan-Pair` *CSP*

Result

Returns an unconnected channel pair

`connect-channel-input` *CSP*

Arguments

channel: a channel

Result

A channel. Connects the thread executing the statement to the input end of the channel.

`connect-channel-output` *CSP*

Arguments

channel: A channel

Result

A channel. Connects the thread executng the statement to the output end of the channel.

`connect-chan-pair` *CSP*

Arguments

channel-pair: A channel pair

Result

Returns one end of the specified channel as a connected channel.

`IN` *CSP macro*

Syntax

`(IN` *channel* `[`*variable*`])`

Remarks

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` *CSP macro*

Syntax

`(OUT` *channel obj*`)`

Remarks

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` *CSP macro*

Syntax

`(PAR` *expression**`)`

Remarks

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` *CSP macro*

Syntax

`(MAPPAR` *function list*`)`

Remarks

Apply *function*, which should take one argument to each element of the list, as a parallel operation.

`FOR` *CSP macro*

Remarks

PAR as iteration over a sequence of values.

`ALT` *CSP macro*

Syntax

`(ALT` *alternative**`)`

Remarks

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` *CSP macro*

(channel-var value-var) channels . expressions

Syntax

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

Remarks

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` *CSP macro*

Syntax

`SEQ` *expression** The same as progn, ie execute the following expressions in the order given.

11.4 Time Warp

To be written

12 Distributed Processing

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

`socketp` FEEL

Arguments

obj: An object

Result

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

`make-listener` FEEL

Result

Allocates a fresh `listener` object.

`make-socket` FEEL

Result

Allocates a fresh `socket` object. Note that this function is almost never called.

`listener-id` FEEL

Arguments

Listener: a listener

Result

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

`listen` FEEL

Arguments

listener: a listener

Result

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

`connect` FEEL

Arguments

pair: A pair identifying a listening port on a remote machine

Result

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` FEEL

Arguments

listener: a listener

Result

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

`close-socket` FEEL

Arguments

socket: a socket

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

`socket-readable-p` FEEL

Arguments

socket: a socket.

Result

If there is data available for reading from *socket*, returns `t`, otherwise `()`.

`socket-writable-p` FEEL

Arguments

socket: A socket.

Result

If data can be written to *socket*, returns `t`, otherwise `()`.

Figure 2 is two scripts of a simple example of establishing a socket connection and a dialogue across the connection

Currently `socket-read` and `socket-write` are used for input and output. If you are comminicating with non-EuLispprocesses then you should use format/print and scan/input. `Read`, for various reasons cannot be used.

12.1 Linda

To be written

12.2 Time Warp

To be written

12.3 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 [7]. In other words, it is possible to call thread-suspend during a read.

The module exports the following functions:

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

machine-1

```
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
```

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 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 2: Example socket based communication

`make-pvm-id` *PVM*

Arguments

string: A string

id: An identifier

Result

creates a pvm-id which can be used to broadcast to a group of remote processes. Note that this is simply a cons cell.

`pvm-status` *PVM*

Arguments

id: Identifier of a pvm-process

Result

Query the status of the process with id *id*.

`pvm-send` *PVM*

Arguments

dest: A pvm process identifier

type: The numeric type of the message

msg: The message (can be anything)

`[`*reader*`]`: A reader which is used to write the message.

Result

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` *PVM*

Arguments

type: The type of message to be recieved

info?: Is Information on message wanted

`[`*reader*`]`: A reader which is used to read the message.

Result

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` *PVM*

Arguments

type-list: A list of possible message types

info?: Information on message wanted flag

`[`*reader*`]`: A reader which is used to read the message.

Result

As pvm-recv, but blocks until a message which has a type in the type-list.

`pvm-initiate-by-type` *PVM*

Arguments

type: Type of machine (string)

name: Name of the new process (string)

Result

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` *PVM*

Arguments

hostname: Hostname in which to start process

name: Name of the new process

Result

Start a process on the host with name *hostname* with the name *name*

`pvm-enroll` *PVM*

Arguments

name: A string

Result

Enroll into pvm under the given name. Must be called before any other pvm function.

`pvm-leave` *PVM*

Result

Exit from pvm-control. After this is called, all pvm functions return an error message (except pvm-enroll).

`pvm-probe` *PVM*

Arguments

type: A message type

Result

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` *PVM*

Result

Test for messages from a list of types. Not yet implemented (as of PVM 2.4). Can be simulated via probe.

`pvm-whoami` *PVM*

Result

Return the pvm-id (the value returned by enroll) of the process.

`pvm-make-id-from-pair` *PVM*

Arguments

Pair: A pair

Result

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. This operation is now a null operation.

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.

13 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[8], 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*

Result

Makes a new reader object. The class and internals of this object are left unspecified.

[8]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*

Arguments

reader: A reader

class: A class

type-ident: An identifier (> 16)

function: A function to be called when an object of class `class` is encountered.

Result

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*

Arguments

reader: A reader

type-ident: An identifier

function: A function.

Result

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[9].

`read-next` *Reader*

Arguments

ptr: A pointer value

reader: A reader

Result

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.

[9]Feel goes kaboom

Arguments

object: the object to be written

ptr: A pointer value

reader: A reader

Result

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

13.1 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
  (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*)
```

14 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, and recompile.

14.1 How to run it

14.1.1 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.

Fast load files These have .fm extensions, and contain raw bytecodes for a single module.

Documentation files These have a .doc extension (probably a bad choice), and contain bytefunction names, and their documentation. Note that this is a very recent addition to the compiler, but seems simple enough to work reliably.

14.1.2 Compiling

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

`comp2sc` *Compiler*

Arguments

module-name: A symbol

Result

Compile a `.em` file into a `.sc` file, plus a `.i` file.

`(setter optimize-code)` *Compiler*

Arguments

value: New value

Result

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.

14.1.3 linking

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

`load-module` *Compiler*

Arguments

module name: A symbol

Result

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` *Compiler*

Arguments

module-list: List of symbols

Load all of the modules in mod-list in the given order.

`combine-user-modules` *Compiler*

Arguments

image-name: A symbol

module-list: List of modules

Result

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` *Compiler*

Arguments

image-name: A symbol

module-list: List of modules

filename: Module-description file

Result

As above, but use the description file provided to locate external bindings, rather than the system itself. This is primarily for cross-linking.

14.1.4 running

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

```
feel -boot bootimage
```

14.2 Bootstrapping

The directory Feel/Boot contains a Makefile. Modify this to fit in with your system, and do `make map` followed by `make init map`. This should produce a map of the system, and an initialization module. Then, in the directory Feel/Boot/CBoot, edit the Makefile, and do `make install`. After half an hour or so, an image file should be produced. To test do ../.../Src/you -boot image. This should produce a running image. `feel -boot image` should then work on your system.