Using Euscheme

Russell Bradford

0. Introduction

Euscheme is a simple EuLisp Level 0 interpreter, and is used as the language for several of the assignments in final year courses. EuLisp Level 0 is a small and compact Lisp, but nevertheless has many interesting features, such as modules, an object system, and multithreading. EuLisp Level 1 has extra features, the most notable being a full metaobject system. These are a few notes on the use of Euscheme. We assume you are already familiar with Lisp, so we concentrate on those things unique to Eulisp.

There is a basic "teach yourself Lisp" course, you might like to look at: it even contains a few self assessment exercises.

1. Running Euscheme

Just type `euscheme` and that will do it. You get something like

```
EuScheme - Version 0.30s
```

```
user>
```

where the final line is the prompt. Euscheme uses the usual read-eval-print cycle of interactive Lisps: type something, it will be evaluated, and the result printed. To exit, use

```
(exit)
```

and (usually) `^D` will work, too.

2. Constants

There are the usual self-evaluating bits and pieces:

- strings: in double quotes `"hello"`
- numbers: integers and floating point `1234` and `3.1415`
- characters: preceded by `#\` as in `#\c` for the character 'c'
- vectors: delimited by `#(` and `)` as in `#(1 a (3))` which is a vector of length 3, containing an integer, a symbol, and a list.

3. Lists and Vectors

Lists are created with the usual `cons`, `list` and quoted forms `'(1 2 3)`. Use `car` and `cdr` to access the elements.

A vector is created by the function `make-vector` as in

```
(make-vector 4)
```

which creates a vector of length 4, indexed from 0 to 3, all elements initialised to be `()`s. In fact, `make-vector` can take a second argument

```
(make-vector 4 0)
```

which creates a vector as before, with all elements initialised to 0.

To access a vector element use `(vector-ref vec index)`; to update use `((setter vector-ref) vec index newval)`. See below for details about the `setter` function.

Take care with the creation of vectors: `(make-vector 3 #(0 0 0))` will create a vector of three slots, all initialised to the *same* (`eq`) value.

4. Expressions

As is usual, anything that is not a constant is an expression to be evaluated, and those things marked by a quote are deemed to be constant. Thus

```
(+ 1 2)
```

is an expression to be evaluated, while

```
'(+ 1 2)
```

is a constant list of 3 elements (modifying constant lists has an undefined effect, so it's best not to do so). EuLisp has both `progn` to collect together several expressions into a single expression, and `let` for the declaration of local variables.

```
(progn
  expr
  expr
  ...
)
```

and

```
(let ((var val)
      (var val)
      ...)
  expr
  expr
  ...
)
```

(Semantics: evaluate all the `val`s first, then make the bindings to the corresponding `val`s. Thus the `val`s cannot refer to the `var`s. Use `let*`

```
(let* ((var1 val1)
       (var2 val2)
       ...)
  ...)
```

with semantics of evaluate `val1`, bind to `var1`, evaluate `val2`, bind to `var2`, etc., if you need to refer back to previous values.)

The values of these expressions are the values of their last `expr`s. Named `let`s and `let*`s are also supported.

Numbers have the usual syntaxes: `123`, `1.23` and `1.2e4`. Additionally, you can enter integers in base 2: `#b101010`, base 8: `#o7654`, base 16: `#x12ab`, and any other base up to 36: `#23r12gd` for a base 23 integer.

The full syntax of symbols is somewhat tricky, but "alphanumerics, starting with a letter" is certainly OK. Dodgy characters, such as space, or a leading digit can be escaped with a `\`. A multiple character escape is introduced and ended by `|`. Within the confines of these delimeters any character is fine, except that `\|` is interpreted as a literal `|`, and `\\` as a literal `\`.

All the following are the same symbol:

```
\1\ 23
|1 |23
|1 23|
|1 |2|3|
|1 |2\3
\1| |2||3
```

Their canonical form is `|1 23|`.

5. Conditionals

EuLisp has the usual `(if boolexpr trueexpr falseexpr)` (always with both `trueexpr` and `falseexpr`), and the `cond` form. The single false value in EuLisp is `()`: anything else is deemed to be true. The symbol `nil` is bound to `()`, while `t` is bound to the symbol `t`, providing a convenient canonical true value. Additional conditional forms include

```
(when boolexpr
   expr
   expr
   ...
)
```

where the `expr`s are evaluated when the condition is true; and

```
(unless boolexpr
   expr
   expr
   ...
)
```

where the `expr`s are evaluated when the condition is false.

6. Assignment

You've got `setq`. It's also good to define your global variables

```
(deflocal foo 2)
```

somewhere, too. You can omit the initial value if you want. The `deflocal` form should only be used at the top level (i.e., never inside a function definition or a `let`).

7. Defining Functions

Here we use `defun`.

```
(defun len (l)
  (if (null l)
      0
      (+ 1 (len (cdr l)))))
```

EuLisp is fully tail-recursive, so a function written in a tail-recursive way uses no stack:

```
(defun foo (n)
  (print n)
  (foo (+ n 1)))
```

will run forever.

Variable arity functions are available, too:

```
(defun bar (a b . c)
  (list a b c))
```

can take 2 or more arguments. The first two arguments are bounds to `a` and `b` as usual, the rest are made into a list which is bound to `c`. Thus `(bar 1 2 3 4 5)` prints `(1 2 (3 4 5))`, and `(bar 99 100)` prints `(99 100 ())`

8. Arithmetic

All the usual stuff here. Functions `+`, `-`, `*` and `/`, `abs`, `sin`, `exp` and so on. Use `(pow a b)` to raise `a` to power `b`. Additionally, the basic arithmetic ops have variable arity:

```
(+)          -> 0
(+ 1)        -> 1
(+ 1 2)      -> 3
(+ 1 2 3)    -> 6
(- 1)        -> -1
(* 1 2 3 4)  -> 24
```

and so on.

9. Modules

Now for something a little different. The basic unit of a program in EuLisp is the *module*. Modules provide a way of fixing the global namespace pollution problem: each module has its very own namespace. A module can import names from other modules, and can export names too.

Here is a simple module:

```
(defmodule one
  (import (level0))

  (defun foo ...)
  (defun bar ...)
  (deflocal baz ...)
  ...

  (export foo baz)
)
```

The module `one` *imports* from the system module named `level0`. This module contains all the useful stuff like `cons`, `car`, `defun`, `+` and so on. In fact, it's generally a good idea to import the `level0` module, otherwise you can't actually do anything.

In module `one` we define a few name, like `foo`, `bar` and `baz`, and *export* `foo` and `baz`. Now any module that imports `one` can access `foo` and `baz`, but `bar` is completely hidden from everyone.

If now, we have

```
(defmodule two
  (import (level0 one))

  ...
)
```

the module `two` imports `one` (and `level0`), so `two` can refer to `foo` and `baz` from `one`. If `two` uses a name `bar`, it is its own `bar`, and has nothing to do with the `bar` in `one`.

Modules in Euscheme

Euscheme requires each module to be in a file of its own: thus `one` should be in a file named `one.em` (for *E*uLisp *m*odule), and `two` in `two.em`. To enter a module, use

```
(!> one)
```

which will load `one` if it is not already loaded, and will set the current module to be `one`. This is indicated by the prompt

```
user> (!> one)
<reading one.em>
<read one.em>
<one...done>
#t
one>
```

Now the read-eval-print loop acts on bindings in the `one` module. Use `(!> user)` to switch back to the original module.

To re-load a module (after, say, changing the file) use `(!>> one)`.

10. Errors and the Debug Loop

When you make an error, EuLisp will call an *error handler*. The full use of error handlers is too tricky for an introductory set of notes, so we shall rely on the default (built-in) handler. In Euscheme an error puts the system into a simple debugging loop:

```
user> qwerty
Continuable error---calling default handler:
Condition class is #<class unbound-error>
message:        "variable unbound in module 'user'"
value:          qwerty

Debug loop.  Type help: for help
Broken at #<Code #1008a768>

DEBUG>
```

There is a lot of information here, and you should look carefully at what Euscheme is telling you.

In this case, the call of error is an 'unbound-error', i.e., reference to an undefined variable. The `message` gives an English description of the error, while the `value` fills in some details, so it is the variable named `qwerty` that is at fault.

Another error:

```
user> (car 5)
Continuable error---calling default handler:
Condition class is #<class bad-type>
message:        "incorrect type in car"
value:          5
expected-type:  #<class cons>

Debug loop.  Type help: for help
Broken at #<Code #100820a8>

DEBUG>
```

This is a 'bad-type' error, where the function `car` was expecting a different type of argument; it got a `5`, where it was expecting something of class `cons`, i.e., some sort of list.

The prompt becomes `DEBUG>` to indicate we are in the debug loop. In this loop things act as normal, except we have some additional functions to play with. Type `help:` to get

```
Debug loop.
top:                                return to top level
resume:  or  (resume: val)          resume from error
bt:                                 backtrace
locals:                             local variables
cond:                               current condition
up:  or  (up: n)                    up one or n frames
down:  or  (down: n)                down one or n frames
where:                              current function
```

The most useful of these is `top:`, which clears up the error and returns us to the top-level read-eval-print loop; and `bt:` which gives us a backtrace, i.e., a list of the function calls and their arguments that took us to where we are now. (Note that, as EuLisp is tail recursive, Euscheme does not save all the return addresses of the functions that it travels through, so the backtrace may omit certain intermediate function calls.)

In a debug loop `^D` will act as `resume:`, which is to try to carry on from the point of error. Debug loops can be nested.

11. Classes and Generic Functions

EuLisp has a full object system. At Level 0, it is a simple, non-reflective system, comparable to C++'s class system. Every object in EuLisp has a class, which is itself a first-class object: this means that classes are supported at the same level as any other object in the system, and can be created, passed to functions, returned from functions, and so on. For example, the integer `1` has class `<integer>` (or rather, has a class with *name* `<integer>`).

In fact, Euscheme has `(class-of 1)` to be `<fpi>` (for *f*ixed *p*oint *i*nteger), which is a subclass of `<integer>`.

Classes are fully-fledged objects, so they have a class, too

```
(class-of <integer>) -> #<class class>
```

the print representation of the class `<class>`. Finally, `(class-of <class>)` is `<class>` itself, or else we would need an infinite tower of classes.

To make an instance of a class, use `make`

```
(make <cons> car: 1 cdr: 2) -> (1 . 2)
```

The keywords (symbols whose names end with colons) indicate how to fill in the various slots of the instance of the class. The keywords can be in any order, and can be omitted if not necessary: though some classes have slots with *required* keywords. This means that instances of such classes *must* have certain information passed to `make` in order to succeed. Some classes are *abstract*, and you cannot make instances of them. They are there purely for other classes to inherit from. The class `<list>` is abstract, while its subclass `<cons>` is *concrete*.

It is simple to create new classes by the use of `defclass`.

```
(defclass <rat> ()
   ((num keyword: num:
         default: 0
         accessor: num)
    (den keyword: den:
         default: 1
         accessor: den))
    predicate: rat?
    constructor: (rat num: den:))
```

There are many parts to explain.

This form defines a new class named `<rat>`. Classes in EuLisp are conventionally noted by the use of angle brackets `<>`, but they are just normal names. The `()` next is the list of classes for `<rat>` to inherit from. In EuLisp Level 0, there is only single inheritance, so this should be a list of at most one class. Any empty list indicates some suitable default superclass.

Next is a list of *slot descriptions*. Each has a slot name first, then a list of *slot options*. The slot options are identified by keywords which can come in any order, and can be omitted it you don't want them.

The slot options are

- `keyword:` a keyword to use in a `make` of the class instance.
- `default:` a default value to put in the slot if a value is not passed via the keyword.
- `accessor:` a name that will be bound to functions to read and write the slot. In the above example, `num` will name a function to read the `num` slot in an instance of `<rat>`. Similarly, `(setter num)` will be a function to write to such a slot. See setters.
- `reader:` a name for a slot reader.
- `writer:` a name for a slot writer.

- `requiredp:` use `requiredp: t` to indicate a required slot. This slot must have a `keyword:` keyword!

The `accessor:`, `reader:` and `writer:` options can be repeated as many times as you wish with different names.

Next come the class options. Again, in any order or omitted.

- `predicate:` a symbol to name a function that will return true on an instance of the class, and false on all other objects.
- `constructor:` a way to name a function to make an instance of the class. In this case, `rat` will name a function of two arguments that makes an instance of `<rat>`. The first argument will be given to the `num:` keyword, the second to the `den:`. This is equivalent to defining

  ```
  (defun rat (n d)
    (make <rat> num: n den: d))
  ```

  As usual, you can reorder or leave out bits as you feel.
- `abstractp: t` to indicate that this class is abstract, and no direct instances can be made.

The class options `predicate:` and `constructor:` can be repeated.

To see all the currently defined classes in Euscheme use `(class-hierarchy)`. Other useful functions include `class-superclasses`, `class-subclasses` and `class-slots`.

Generic Functions

Generic functions are (again) first-class objects in EuLisp, constructed by `defgeneric`. Methods are added to them by `defmethod` (unlike some other systems, a generic function must be created by `defgeneric` *before* `defmethod` will work.)

```
(defgeneric foo (a b))

(defmethod foo ((a <integer>) (b <integer>))
  (list 'int 'int))

(defmethod foo ((x <float>) (y <float>))
  (list 'float 'float))
```

This defines a generic of two arguments, and two methods. So

```
(foo 4 5)     -> (int int)
(foo 1.0 2.0) -> (float float)
(foo 2 2.0)   -> error, "no applicable methods"
```

The methods discriminate off all the arguments, working left to right. Adding another method

```
(defmethod foo ((n <number>) (m <number>))
  (list 'num 'num))
```

we get `(foo 2 2.0) -> (num num)`. Generally the most specific method for a given set of arguments is the method that is executed in a generic call. The next most specific method can be invoked by using `(call-next-method)` in the body of the current method.

12. Threads

EuLisp supports multiple threaded programming by supplying some basic thread primitives.

To make a thread use

```
(make-thread fn)
```

which returns a thread object (another first-class object). The `fn` is the function that the thread will start executing when it and when starts running.

A thread will not run until it is *started*

```
(thread-start thr arg arg ...)
```

This function takes a thread `thr` and starts executing the function `fn` (from `make-thread`) on the arguments `arg`. That is, it starts executing `(fn arg arg ...)`.

Or it would start executing the thread if there were enough processors to do so. As is most likely, the thread is simply marked as *ready to run* whenever the resource is available. The EuLisp model requires the programmer to write in such a manner that does not presume any particular number of processors are available. Even if there is just one processor, the program should be written to work. To aid this, there is the function

```
(thread-reschedule)
```

which will suspend the current thread, and allow another to run in its place. If there are enough processors so that all threads are running, then `thread-reschedule` could have no effect at all.

An single-threaded implementation such as Euscheme requires a sprinkling of `thread-reschedule`s for a parallel program to work.

Threads are often used for their effect, but they can also return a value.

```
(thread-value thr)
```

will suspend the calling thread (and allow another to run in its place) until the thread `thr` returns a value (and returns what the `thr` returned). A thread can return a value simply by returning from its initial function (`fn`, above).

Semaphores

EuLisp provides simple binary semaphores, named *locks*, with functions `make-lock` to make one, `lock` to gain a semaphore, and `unlock` to release.

Locking a locked lock will suspend the calling thread (and allow another to run) until some other thread releases the lock.

13. Input and Output

EuLisp is still a little undecided as to how i/o is going to turn out, so for the meantime Euscheme uses Scheme's functions.

- `read` to read a Lisp expression.
- `write` output in a way that can be re-read if possible. Thus, for example, strings are quoted.
- `prin` output in a human-friendly manner. Strings and such are not quoted. Compare

```
(print "asd")   prints: asd
(write "asd")   prints: "asd"
```

- `print` as `prin`, with a newline.
- `newline` output a newline.

All of the above take an optional extra argument, which is a stream to print on. This defaults to the standard output.

For stream manipulation:

- `open-input-file` takes a string, and opens and returns a corresponding stream for input. Returns `()` if not such file exists.
- `open-output-file` creates a file if it didn't already exist.
- `open-update-file` opens for append.
- `get-file-position` and `(setter get-file-position)` move the file pointer in a file opened for update.
- `close-port` closes an open stream.

Format

A more complicated printing function is `format`, which is somewhat akin to C's `printf`.

```
(format stream format-string arg arg ...)
```

If `stream` is `t`, `format` prints to the standard output. If `stream` is `()`, `format` returns the formatted output as a string. Otherwise `stream` is a file stream.

The format string is copied to the output, except that `~` marks an escape (like C's `%`):

- `~a` output the next arg using `prin`
- `~s` output the next arg using `write`
- `~%` output a newline
- `~~` output a `~`
- `~c` output a character
- `~d` output an integer
- `~e` `~f`, `~g` floating point formats
- `~t` output a tab

There are other escapes to write integers in other bases, output new pages, and so on.

14. Macros

EuLisp employs the usual backquoted template style of macros.

```
(defmacro first (x)
  `(car ,x))
```

Note that a macro cannot be used in the module where it is defined: a module must be fully macroexpanded before it can be compiled. If you don't know what is and what isn't a macro beforehand, it is very difficult to do this. Thus a module containing

```
(defmacro second (x)
   `(cadr ,x))

(defun foo (x) (+ 1 (second x)))
```

is doomed to failure by this restriction.

There is a wrinkle in the way that macros interact with modules: suppose a macro expands into something that refers to bindings that are not imported into the current module?

```
(defmodule one
  (import (level0))

  (defmacro foo (x)
     `(car (bar ,x)))

  (defun bar (a) ...)

  (export foo)
)
```

Here the module `one` exports `foo` only, but `foo` expands into a reference to `bar`.

```
(defmodule two
  (import (level0))

  ...
  (foo 4)
  ...
)
```

In the macroexpansion of module `two`, a reference to `bar` would appear, but `bar` is not defined in `two`. Worse, maybe `bar` *was* defined in `two`: which `bar` does the macroexpanded form refer to? The `bar` from `one` or the `bar` from `two`?

The answer is "the right `bar`", that is that `bar` in the module of macro definition, not the `bar` in the module of macro use. Euscheme takes care of all of this transparently for you: essentially every symbol remembers which module it was defined in, and always refers back to that module for its value.

This provides a simple solution to the "macro hygene" problem that has always plagued Lisp macros.

Sometimes you *do* want a symbol to be captured in the module of use: Euscheme provides a facility to allow you to do this.

```
(defmacro while (test . body)
  `(let/cc {break}
     (labels
```

```
    ((loop ()
       (when ,test
             ,@body
             (loop))))
    (loop))))
```

The symbol `loop` cannot be captured by the code in `body`, while the symbol `break` is intended to be captured. The curly braces about the symbol indicates that it is to be interpreted as coming from the module of use, *not* the module of definition. Thus, a reference to `break` in the `body` will refer to the binding in the `let/cc`.

Notice that `(eq 'break '{break}) -> t`. As symbols they are `eq`, but as identifiers they are quite different.

15. Miscellany

Comparisons

EuLisp has the usual tests for equality:

- `eq` for identity
- `eql` for identity, but will also work for integers and characters
- `equal` for recursive equality
- `=` for numbers

Note that

```
(equal 1 1.0)  -> ()
(= 1 1.0)      -> t
```

There is also the usual `<`, `<=`, `>`, `>=`, which are n-ary:

```
(< a b c ...)
```

returns `t` when `a`, `b`, `c`, etc., form a strictly increasing sequence. Similarly `<=` for a non-decreasing sequence, and so on.

Generic Arithmetic

The arithmetic operators `+` and so on are all n-ary, i.e., take a variable number of arguments. Each operator is defined in terms of a binary generic function: `binary+` for `+`, `binary*` for `*`, etc. The n-ary form is just a repeated application of the binary form

```
(+ a b c ...) = ((..(binary+ (binary+ a b) c) ...
```

Methods can be added to the binary operators

```
(defmethod binary+ ((a <symbol>) (b <symbol>))
   ...)
```

and then you can use `+` to add symbols: `(+ 'a 'b 'c)`.

There are also generic functions `unary-` and `unary/` for the unary `(- x)` and `(/ x)` (reciprocal).

Similarly, the comparators `<`, `>`, `<=` etc., are all defined in terms of the two generic functions `binary<` and `binary=`.

Local Functions

Just like `let` introduces local variables, the `labels` form can introduce local functions.

```
(labels
  ((foo (a b)
     ... (bar a) ... )
   (bar (x)
     ... (foo x (bar x)) ... ))
  ...
  (foo 3 4)
  ...
)
```

The `labels` takes a list of function definitions. They may be self and mutually recursive. These functions may be used within the body of the `labels` just like global functions. Iterating functions are often most conveniently written in terms of `labels` as the bodies of the function definitions can refer to local variables:

```
(let ((a 1))
  (labels
    ((addit (x)
       (if (null x)
           ()
           (cons (+ a (car x)) (addit (cdr x))))))
    (addit '(1 2 3))))
->
(2 3 4)
```

Mapping and Collections

There are several functions supplied to iterate along collections. Collections include lists, vectors, strings, and [tables](#).

The generic function `map` takes a function and a collection

```
(map list '(1 2 3))   -> ((1) (2) (3))
(map - #(4 5 6))      -> #(-4 -5 -6)
```

or more than one collection

```
(map cons '(a b c) '(A B C))     -> ((a . A) (b . B) (c . C))
(map + #(1 2 3) #(10 10 10 10))  -> #(11 12 13)
```

The mapping stops when any collection runs out. Even a mixture will work

```
(map * '(2 4 6) #(1 -1 1))  -> (2 -4 6)
(map * #(2 4 6) '(1 -1 1))  -> #(2 -4 6)
```

The type of collection returned is the same as the first collection argument.

If you don't need a return value, but are iterating purely for effect, use `do`

```
(do print '(1 2 3))
```

Other iterators include `accumulate`

```
(accumulate list () #(a b c))      -> (((() a) b) c)
```

```
(accumulate * 1 '(1 2 3 4 5 6 7))  -> 5040
```

which takes a function, an initial value, an a collection to iterate over.

You can find the size of any collection using the function `size`. This returns the length of a list of string, number of elements of a vector, and so on. It can be reversed by `reverse`; an element removed by `remove` (non-destructive) or by `delete` (destructive); find an element by `(member elt collection)`. The last three (`remove`, `delete` and `member`) take an optional last argument that is a test for equality: it is this test that is used when looking for an element in the collection. It defaults to `eql`.

The function `concatenate` can be used to join collections:

```
(concatenate '(1 2 3) '(4 5 6))  -> (1 2 3 4 5 6)
(concatenate "abc" "def")        -> "abcdef"
(concatenate '(1 2 3) #(4 5 6))  -> (1 2 3 4 5 6)
```

Loops

EuLisp doesn't really need loops, as everything can be written easily in terms of tail recursive functions. However, Euscheme sneaks in a `while` loop:

```
(while bool
   expr
   expr
   ...
)
```

which loops while the `bool` returns true.

Tables

EuLisp uses tables for a general association mechanism. Euscheme implements tables as hash tables, but in general they could be implemented differently.

- `make-table` returns a table.
- `(table-ref table key)` to retrieve a value, `((setter table-ref) table key value)` to update.
- `(table-delete key)` to remove a value.
- `table-keys` to get a list of current keys.
- `table-values` to get a list of current values.
- `table-clear` to completely empty a table.

When looking for a match to a key in a table, the system defaults to `eql`. You can change this by using `(make-table comparator)`, where `comparator` is `eq` or `eql` or `equal` or `=`.

If a value is not found for a particular key in the table `()` is returned. This can be changed by `(make-table comparator fill-value)`. Now `fill-value` will be returned on failure.

The mapping functions above work on tables, too.

Non-local exits

EuLisp supports a limited form of continuation capture via `let/cc`. This form captures its continuation, and allows its use as a non-local exit.

```
(let/cc out
   ...
   (out)
   ...
)
;; after
```

This stores the continuation (i.e., from 'after') in the variable `out`. This can be called as a function, whereupon control passes immediately to that continuation. The value of `out` can only be used in this way in the dynamic scope of the `let/cc` form: outside the value is 'dead' and no longer usable.

The continuation function can take a single optional argument which is a value to pass to the continuation: the default is `()`.

The forms `block` and `return-from` are simply `let/cc` and a call to a continuation:

```
(block foo
  ...
  (return-from foo)
  ...
)
```

The `unwind-protect` form ensures things are executed even if there is a non-local exit

```
(unwind-protect
  protected-form
  after-form
  after-form
  ...)
```

This starts by executing the `protected-form`. If there is no unusual exit from the `protected-form`, this will then execute the `after-form`s and will return whatever value the `protected-form` returned. If there is a non-local exit from the `protected-form` to a continuation outside the `unwind-protect`, the `after-form`s will *still* be executed before the control passes to the continuation.

Setters

Structures, like lists, vectors and class instances have elements that can be accessed. The elements of a vector can be read by `vector-ref`. To write to an element use the function `(setter vector-ref)`,

```
((setter vector-ref) vec index val)
```

Similarly, the accessor `car` has an updater `(setter car)` (often called `rplaca` in other Lisps), and so on. In general a reader function `r` will have an associated updater `(setter r)`.

The function `setter` is a simple association mechanism: `setter` is a function that takes a reader and returns the associated writer. To make such an association between functions `r` and `w` just use `setter` again

```
((setter setter) r w)
```

In fact, no particular properties of `r` and `w`, are used, so this can be used as a general facility. Further, `setter` functions, generic functions and methods can be defined directly:

```
(defun (setter foo) (a b)
  ...)
```

Convert

The function `convert` is used to change an object of one type into an object of another type. Thus to convert an integer to a float

```
(convert 1 <float>) -> 1.0
```

or the other way

```
(convert 2.6 <integer>) -> 2
```

Many other conversions are available: integer to string; character to string; string to number; symbol to string; list to vector; and so on.

Copying

There are two functions that copy structures: `deep-copy` and `shallow-copy`. The second recursively descends a structure making copies of all the elements in the structure; the first makes a single copy of the top-level structure, and fills its slots will the existing elements:

```
(setq a '((1 2) (3 4)))
(setq d (deep-copy a))
(eq a d)                  -> ()
(equal a d)               -> t
(eq (car a) (car d))      -> ()

(setq s (shallow-copy a))
(eq a s)                  -> ()
(equal a s)               -> t
(eq (car a) (car s))      -> t
```

Other Tools

Other tools that Euscheme provides:

- `describe` gives a little information about an object, e.g., `(describe <integer>)` or `(describe 4)`
- `trace` can be used to print a message every time a function is entered or exited. Thus

  ```
  (trace foo)
  ```

  will describe the ins and outs of the function `foo`. To untrace, use `(untrace foo)`.

Use `(import "trace")` to load `trace`.

16. Euscheme Modules

Euscheme provides a few sample modules.

Trace

```
(import "trace")
```

The `trace` module has been mentioned above.

Linda

```
(import "eulinda")
```

The `eulinda` module implements the Linda pool mechanism.

- `make-linda-pool` returns a new pool
- `(linda-out pool tag val val ...)` writes the tuple `(val val ...)` under the tag to the pool
- `(linda-in pool tag pat pat ...)` attempts to read a tuple matching the pattern `(pat pat ...)` from the pool. If no matching tuple exists in the pool, the call will block until such a tuple appears. When found, the tuple is removed from the pool. A pattern is
  - a literal value, to be matched exactly
  - `(? var)` to match any value, and assign the matched value to the variable
  - `?` to match any value, and to discard the result.
- `linda-read` as `linda-in` but does not remove the tuple from the pool
- `(linda-eval fun arg arg ...)` starts a new thread, running the function with the arguments.

Debugging tools are `print-linda-pool` to print the curent values in a pool, and `(tril t)` to print some trace information as the system is running.

The `tag` must be a symbol or number.

Modular Numbers

```
(import "modular")
```

The module `modular` is a simple implementation of modular integers. The function `mod` constructs a modular number

```
(setq a (mod 3 5))  -> #<3 mod 5>
(setq b (+ a a))    -> #<1 mod 5>
(/ a)               -> #<2 mod 5>
```

Scheme

```
(import "scheme")
```

This module, `scheme`, provides a mostly-conformant Scheme environment. It is probably not wise to mix Scheme constructs, such as `call/cc`, with EuLisp constructs, such as threads.

Paralation Lisp

```
(import "tpl")
```

This emulates a paralation system. The module `tpl` (for *t*iny *p*aralation *l*isp) exports

- `(make-paralation n)` to make a new paralation of size `n`. This returns a index field of the new paralation.
- `elwise` is the element-wise operator:

  ```
  (elwise (a b) (+ a b))
  ```

  where `a` and `b` are fields on the same paralation.
- `(match field field)` to create a map between fields, and
- `(move field map combine default)` to move a `field` down a `map`, using `combine`, (a function taking an appropriate number of arguments) to combine elements that end up at the same element of the target field, and `default` as the default value for a field element that is not in the image of the map.

Values

```
(import "values")
```

This is an emulation of Scheme and Common Lisp's multiple values. The module `values` exports

- `(values val val ...)` as the basic multiple value return
- `call-with-values` for the Scheme-like values:

  ```
  (call-with-values
    (lambda () ...)             ; a thunk returning values
    (lambda (a b c ...) ...))   ; that are passed here, bound
                                ; to a, b, etc.
  ```

- `multiple-value-setq`; `multiple-value-list`; `multiple-value-call`; `values-list`; `multiple-value-bind` are all as in Common Lisp.

If you pass multiple values to a continuation that only expects a single value you will probably get strange results.

Sort

```
(import "sort")
```

A fast stable merge sort. The module `sort` exports `sort` (non-destructive) and `sort!` (destructive). They are called as `(sort l)`, where `l` is a list of values to be sorted. The comparison operator used is `<`. Alternatively, you can use `(sort l comp)`, where `comp` is a comparator function.

17. Euscheme functions

Here is a summary of the functions available in Euscheme. Not all of these correspond to EuLisp.

```
;; specials
   quote
   lambda
   delay
   let
   let*
   setq
   if
   cond
   progn
   and
   or
   let/cc
   while
   block
   return-from
   labels
   when
   unless
   export
   expose
   enter-module
   !>
   reenter-module
   !>>
   call-next-method
   next-method-p
   import
   generic-lambda
   method-lambda

;; defining forms
   defun
   defgeneric
   defmethod
   deflocal
   defconstant
   defmodule
   defmacro

;; list functions
   cons
   car
   cdr
   caar
   cadr
   cdar
   cddr
   caaar
   caadr
   cadar
   caddr
   cdaar
   cdadr
   cddar
   cdddr
   caaaar
   caaadr
   caadar
   caaddr
   cadaar
   cadadr
   caddar
   cadddr
   cdaaar
```

```
    cdaadr
    cdadar
    cdaddr
    cddaar
    cddadr
    cdddar
    cddddr
    list
    list*
    append
    last-pair
    length
    memv
    memq
    assv
    assq
    list-ref
    list-tail

 ;; symbol functions
    bound?
    symbol-value
    symbol-plist
    gensym
    get
    put

 ;; vector functions
    vector
    make-vector
    vector-length
    vector-ref

 ;; predicates
    null
    atom
    listp
    numberp
    booleanp
    consp
    symbolp
    keywordp
    complexp
    floatp
    double-float-p
    rationalp
    integerp
    charp
    stringp
    vectorp
    functionp
    portp
    input-port-p
    output-port-p
    objectp
    eof-object-p
    default-object-p
    eq
    eql
    equal

 ;; arithmetic functions
    +
    -
    *
```

```
     /
     %
     zerop
     positivep
     negativep
     oddp
     evenp
     truncate
     floor
     ceiling
     round
     random
     quotient
     remainder
     sin
     cos
     tan
     asin
     acos
     atan
     exp
     sqrt
     log

  ;; bitwise logical functions
     logand
     logior
     logxor
     lognot

  ;; string functions
     make-string
     string-length
     string-null?
     string-append
     string-ref
     substring

  ;; i/o functions
     read
     read-char
     read-byte
     read-short
     read-long
     write
     write-char
     write-byte
     write-short
     write-long
     prin
     print
     newline
     char-ready-p
     peek-char
     format

  ;; print control functions
     print-breadth
     print-depth

  ;; file i/o functions
     open-input-file
     open-output-file
     open-append-file
     open-update-file
```

```
   close-port
   close-input-port
   close-output-port
   get-file-position
   unlink
   current-input-port
   current-output-port

;; utility functions
   transcript-on
   transcript-off
   getarg
   prompt?
   exit
   compile
   decompile
   gc
   save
   restore

;; debugging functions
   trace-on
   trace-off

;; module functions
   module-symbols
   module-exports
   symbol-module
   current-module
   module-list
   unintern

;; telos
   allocate
   describe
   classp
   subclassp

;; tables
   make-table
   table-ref
   table-comparator
   table-delete
   table-length
   table-keys
   table-values
   table-fill
   table-clear

;; plus some others
   binary
   text
   not
   prin1
   princ
   t
   nil
   eval                ; no guarantees this one will work
   else
   system
   getenv
   putenv
   tmpfile
   current-time
   ticks-per-second
```

```
    backtrace
    backtracep

 ;; threads
    <thread>
    <simple-thread>
    make-thread
    threadp
    thread-reschedule
    current-thread
    thread-kill
    thread-queue
    current-thread
    thread-start
    thread-value
    thread-state
    <thread-condition>
    <thread-error>
    <thread-already-started>

    <lock>
    <simple-lock>
    make-lock
    lockp
    lock
    unlock
    <lock-condition>
    <lock-error>

    wait
    <wait-condition>
    <wait-error>

 ;; errors and handlers
    with-handler
    unwind-protect
    <wrong-condition-class>
    signal
    error
    cerror

 ;; classes
    <object>
    <class>
    <simple-class>
    <list>
    <cons>
    <null>
    <number>
    <integer>
    <fpi>
    <float>
    <double-float>
    <symbol>
    <keyword>
    <string>
    <simple-string>
    <port>
    <input-port>
    <output-port>
    <i/o-port>
    <vector>
    <simple-vector>
    <char>
    <simple-char>
```

```
   <promise>
   <table>
   <hash-table>
   <function>
   <simple-function>
   <subr>
   <continuation>
   <generic>
   <simple-generic>
   <method>
   <simple-method>
   <slot>
   <local-slot>
   <structure>

   generic-prin
   generic-write
   wait

   make
   initialize
   class-hierarchy

;; setter
   setter

;; converter
   converter
   convert
   <conversion-condition>
   <no-converter>

;; condition classes
   defcondition
   conditionp
   condition-message
   condition-value
   <condition>
   <telos-condition>
   <telos-error>
   <telos-general-error>
   <telos-bad-ref>
   <no-applicable-method>
   <no-next-method>
   <incompatible-method-domain>
   <arithmetic-condition>
   <arithmetic-error>
   <error>
   <general-error>
   <bad-type>
   <unbound-error>
   <compilation-error>
   <macro-error>
   <syntax-error>
   <user-interrupt>

;; generic arithmetic
   binary+
   binary-
   unary-
   binary*
   binary/
   unary/
   binary%
   binary-gcd
```

```
    gcd
    abs
    pow

 ;; comparisons
    equal
    binary<
    binary=
    <
    =
    >
    <=
    >=
    max
    min
    assoc

 ;; macros
    defmacro
    quasiquote
    unquote
    unquote-splicing
    symbol-macro
    macroexpand
    macroexpand1
    syntax

 ;; collections and sequences
    <collection-condition>
    <collection-error>
    collectionp
    sequencep
    accumulate
    accumulate1
    allp
    anyp
    concatenate
    delete
    do
    element
    emptyp
    fill
    map
    member
    remove
    reverse
    size

 ;; copying
    deep-copy
    shallow-copy

 ;; telos introspection
    class-of
    class-name
    class-superclasses
    class-precedence-list
    class-slots
    class-keywords
    class-subclasses
    class-instance-size
    class-abstract-p
    generic-name
    generic-args
    generic-optargs?
```

```
   generic-methods
   generic-cache1
   generic-cache2
   method-generic
   method-function
   method-domain
   add-method
   slot-name
   slot-keyword
   slot-default
   slot-required-p

;; other functions
   apply
   map-list
   load
   load-noisily
   force
```

18. Command Line Arguments

The Euscheme interpreter accepts a few arguments:

- `-t` enable trace debugging
- `-n` do not load an image
- `-q` do not print results in the read-eval-print loop
- `-f` see below
- `-s` disable the `system` function

Other arguments are passed to the intepreter and are available as `(getarg 0)` (the name of the program), `(getarg 1)` (first argument), `(getarg 2)` (second argument), and so on. The function `getarg` returns `()` for a non-existent argument.

Shell Scripts

Euscheme can be used in a shell script by means of the `-f` flag:

```
#!/usr/local/bin/euscheme -f

(print "hello world")
```

It is usual to use the `-q` flag to prevent the echo from the read-eval-print loop, and the `-s` flag to prevent the use of the `system` function.

```
#!/usr/local/bin/euscheme -fq

(print "hello world")
```

Here is the latest version of these notes.