Lisp – Almost a whole Truth !

Research Report LIX RR 89 03, pp 79–106, École Polytechnique, December 1989, France.

CHRISTIAN QUEINNEC queinnec@poly.polytechnique.fr
LABORATOIRE D'INFORMATIQUE DE L'ÉCOLE POLYTECHNIQUE
92128 PALAISEAU CEDEX
FRANCE*

Abstract

Lisp is well known for its metacircular definitions. They differ by their intent (what they want to prove), their means (what linguistic features are allowed for the definition) and by their scope (what linguistic features are described). This paper provides a new metacircular definition for a complete Lisp system including traditionally neglected features such as `cons`, `read`, `print` and `error`. The programming style adopted for this interpreter is inspired both by denotational semantics and its continuation passing style (to explain continuation handling) and by the object oriented paradigm as highlighted by type-driven generic functions. The resulting interpreter lessens the number of primitives it uses to only eight: `car`, `cdr`, `rplaca`, `rplacd`, `eq`, `read-char`, `write-char` and `end`, while still providing Scheme-like essential capabilities (less arithmetic). The overall size is near 500 lines of fully encapsulated code that, if efficiency is not the main requirement, can be easily turned into a stand alone program.

Since the birth of Lisp, the *art of the interpreter* has always been considered as one of the major Lisp rites. Ranging from very simple [McCarthy 78] to more complex [Rees & Clinger 86], a Lisp interpreter is a must of many courses, many books [Abelson & Sussman 85, Dybvig 87, Queinnec 84, Winston 88, ...] and many articles [Friedman & Wand 84, Reynolds 72, ...]. The success of this cult is due to — (i) the amazing simplicity that can have such interpreters — (ii) the pedagogical benefits of either reading or writing such descriptions — (iii) the overall confidence given to the writer of such a program that s/he has now fully mastered the language. Interpreters also provide an interesting way to design variants of Lisp [Steele & Sussman 78, des Rivieres & Smith 84]. They truly are specifications of languages [Reynolds 72] from which can be derived compilers [Clinger 84] or entire systems [Brooks & Gabriel & Steele 82, Saint James 87].

Without quantifying their intentions (pedagogy [Kessler 88], proof length [Boyer & Moore 82] or even graphics [Lakin 80]) a taxonomy of these interpreters can be attempted along two main axis, see figure 1:

- the specified language (given to the user),
- the language for the specification (allowed for the implementer).

These axis are mere abstractions since power of languages is difficult to appreciate and is probably more related to trees than to straight lines. For example, allowing assignment (`setq`) or physical surgery (`rplaca` ...) increases the power of the specification language and thus may permit shorter description (in number of lines), wider description (in number of described features) or more accurate description (in terms of assembly language operations or virtual machine instruction set) of the described language.

*This work has been partly funded by Greco de Programmation.

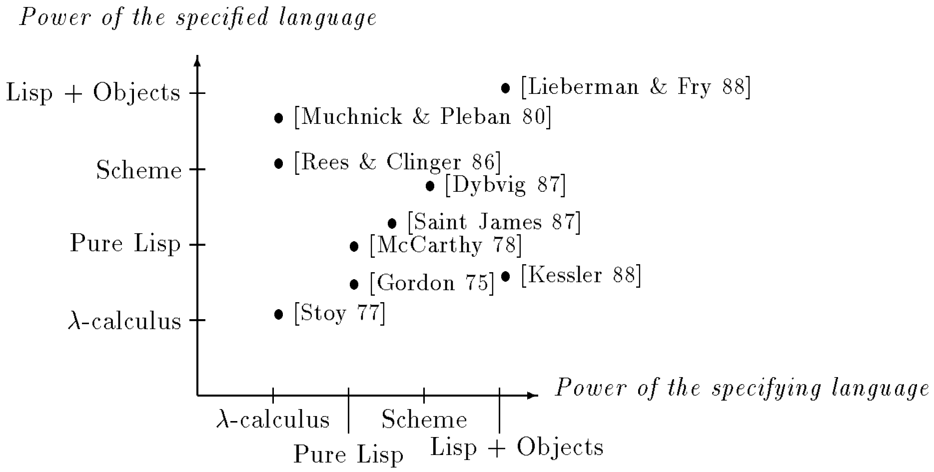


Figure 1: Highly Subjective Taxonomy of some Interpreters

Low level mechanisms like input/output, memory management and error handling have traditionally been neglected in interpreters. Furthermore the number of "primitives"[1] required to run such interpreters has not yet been investigated. Some points are missing from the figure and particularly the description of a small but complete system metacircularly described by a minimal subset. Like the XINU approach [Comer 84], our goal is to provide a thorough description of a self-sufficient system able to be translated into a stand alone program. Linguistic or implementational variants will not be covered but the interpreter offers Scheme-like capabilities excluding numbers that can easily be simulated by lists and symbols but including explicit `eval`, `cons`, `print`, `read` and `error`. The specification language is composed of exactly eight primitives which are `car`, `cdr`, `rplaca`, `rplacd`, `eq`, `read-char`, `write-char`, and `end`. Of course, three more features are used, namely alternative, functional application and variable reference. To pinpoint the extremely small number of prerequisites needed by this interpreter, we named it: FEMTO.

Although a bit longer than other descriptions, the FEMTO interpreter has some interesting virtues :

- the description language is sufficiently small and simple to be naively compiled into any reasonable assembly language. The only data structures dealt with are preallocated dotted pairs: no symbols at all are required. Nor closures nor assignments are needed to run the description which is carefully written in a tail recursive style.
- runtime structures are apparent and can be varied at length providing a lot of pedagogical exercices to improve efficiency both in time or space. Similarly, additional special forms or essential functions can also be implemented and offered to the user. The programs are carefully written to be extensible if new kinds of continuations, environment frames or functional objects are designed.
- Since the implementation is explicit, modern features such as reflection-reification can be studied. The FEMTO interpreter provides a simple testbed for these analyses.

Section 1 will present the framework of the interpreter design. Section 2 will present the overall architecture of our evaluator which will be refined for its more important parts in Section 3. Section 4 discusses some extensions like reflection. Some figures will conclude the paper.

1 Evaluator Design

Our first goal is to design a (very) small set of primitives to implement a full evaluator. This requires that the word "primitive" must be carefully defined to be able to measure the size of the primitive set.

[1] This concept will be more carefully defined later.

Languages can be specified by denotational semantics [Stoy 77, Schmidt 86], features like memory management, error handling and input/output exchanges can also be covered by this technique. Closure making, functional applications and variable reference are the three unique and sufficient features. This small number is somewhat extended in real denotations[2] by n-ary sequences and various utilities known as **`length`** ($\#$), **`append`** ($\S$), **`nth`** ($\downarrow_i$) and **`list`** ($< \ldots >$). Use of domains leads to other utilities such as equalities (`=`), membership ($\in$), projection (inD) or injection (isD) from domains to domains.

This bigger number of features may be reduced if one considers that they simply are macros above the ordinary λ-calculus. But this apparently very simple reduction to the three original features of the λ-calculus masks the complexity of different uses of the same operation. For example, the denotational expression $\varepsilon^*\downarrow_1$, if ε^* is the sequence of arguments of a function, expresses the first argument: a simple operation that may be simulated by **`first`** in COMMON LISP. Conversely the same denotational expression $\varepsilon^*\downarrow_1$, if ε^* is the list of characters to be submitted to the program, requires more implementation if the characters are to be fetched from the keyboard or by a query to the operating system. We thus require to differentiate these two cases with different primitives.

On the other hand, to be closer (in a sense) "real" implementation, we do not want to abstract the store as done in denotational semantics but we require each allocation to be explicitely performed (see `allocate` later). We also provide memory modifiers such as `rplaca` and `rplacd`, but these side-effects will always be made one at a time and in non-terminal places i.e., in the all-but-last forms of a `progn`.

Care must be taken to clearly distinguish the specified language from the specifying language. For example (as shall be seen later) user's closures do not require system closures to exist: they can be implemented by lists. The genuine altruism of Lisp imposes that features used by the system are also offered to the user. The defined language is thereafter an extension to the defining language. To enforce a visual distinction between these two levels, we will prefix entities of the defined language (the user level) by "$_u$" while the system level will be marked by a leading "$_s$".

"Primitives" will designate only these features needed to run the evaluator. But "run" itself must be further explained since some features are only needed to start the evaluator and thus are diluted into the initial environment construction. For example, the user gains access to a dotted pair via $_u$`cons` which is implemented by popping the head of the free-list: an operation which does not require $_s$`cons`. Nevertheless the initial free-list is constituted by a number of calls to $_s$`cons`: since this operation must be done initially, it may be statically compiled and thus may completely disappear from the running code. One has only to run the evaluator in a memory configuration properly initialized.

The running primitives are `car`, `cdr`, `rplaca`, `rplacd`, `eq`, `read-char`, `write-char` and `end`. The function `end` terminates the session and return to the operating system or whatever can be considered as such. The names and the meanings of the others are taken from COMMON LISP. Three more "running" features are variable reference, functional application and alternative.

This set may surprise since it does not include $_s$`cons`, $_s$`setq`, $_s$`consp`, $_s$`lambda`, $_s$`defun` ... None of them are required to run the interpreter. To present the evaluator in a so restricted language would be boring, therefore we will use some more readable syntactical abbreviations i.e., macros like `let`, `progn`, `prog1`, `or` and `and` ...

2 Principles of the Interpreter

Our goal is to provide an interesting interpreter designed along strict principles: – enforce the use of types, – adopt a uniform data representation, and – make explicit all allocations.

2.1 Types in the interpreter

First of all, the code of the interpreter is well typed. This effect is due to three properties — (i) all $_s$data structures are encapsulated by appropriate functions (constructor, predicate and field accessors), — (ii) the interpreter is written in a type-driven style, — (iii) all run-time entities are typed. $_u$Symbols and $_u$dotted

[2] We take here the notations of [Rees & Clinger 86] and [Schmidt 86].

pairs are run-time entities[3], either are continuations or environments. As usual, to have encapsulated data structures allows varying the precise implementation, provided the interface is respected. A corollary is that ${}_{s}$`car` and ${}_{s}$`cdr` are not overloaded as in common interpreters. Coercions, where they appear, are now explicit. Implementation questions such as "where lies the already computed arguments of a not yet applied function" may be answered and worked upon independently of all other design considerations i.e., in a heap frame, a special stack, etc.

The evaluation process is type-driven. The main functions of the interpreter (`evaluate`, `operate`, `resume`, `lookup` and `modify`) conform to the following scheme

```
 (defun name (arguments...)                 (defun name (arguments...)
    (eval.case (type.of one-argument)          (let ((g137 (type.of one-argument)))
      (type.cons      code1)          ==>        (cond ((eq g137 type.cons)    code1)
      (type.if.cont   code2)                           ((eq g137 type.if.cont) code2)
      ... ) )                                           ... ) ) )
```

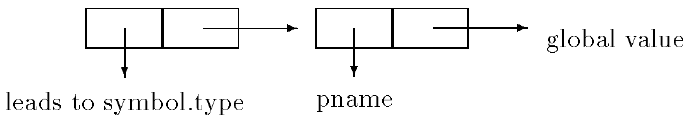
This programming style is a step towards generic functions as defined in CLOS [Bobrow et al. 88]. Each clause of the `eval.case` macro form can be viewed as a method but here methods cannot be dynamically added, nor substituted nor even removed since this is a static definition. Furthermore, instead of being macro-expanded as in the previous excerpt, `eval.case` may be better compiled in a style more reminiscent of the Object Oriented Paradigm with single inheritance. Methods can be directly invoked rather than found by a succession of `cond` clauses. This scheme works since the set of types is predefined and therefore can be indexed[4] by short numbers. A suggestive expansion might be:

```
(defun g1 (arguments...) code1)
(defun g2 (arguments...) code2)
...
(defvar *eval.case137* (make-array *number.of.types*))
(setf (aref *eval.case137* (index.of.type type.cons))    #'g1)
(setf (aref *eval.case137* (index.of.type type.if.cont)) #'g2)
...
(defun name (arguments...)
   (funcall (aref *eval.case137* (index.of.type (type.of one-argument)))
            arguments... ) )
```

This type-driven style easily allows thinking of new types and inserting their associated methods in the generic functions `operate`, `resume` and `apply`. But one has to regenerate the whole interpreter to incorporate these new methods.

2.2 Uniform Data Representation

FEMTO basically deals with three kinds of ${}_{u}$objects: ${}_{u}$symbols, ${}_{u}$dotted pairs as in Pure Lisp but also ${}_{u}$functions (Scheme's procedures). All these ${}_{u}$entities must be distinguishable in order to be not confused (for instance and in old times, functions resembled to lists which unfortunately held a `lambda` in their `car`). To lessen the complexity of ${}_{s}$`type.of`, all entities are encoded as ${}_{s}$cons-cells whose ${}_{s}$`car` allows retrieving their types. The ${}_{s}$`cdr` hold the associated fields. For instance, a ${}_{u}$symbol will be represented as a two-field entity:

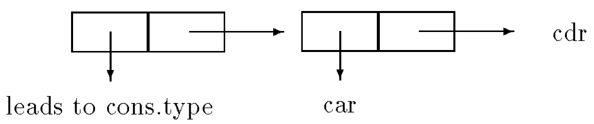


The pname of a symbol is the list of its characters; the global value is the toplevel value of the symbol. A character is represented by a symbol whose pname is composed of a single character: itself !

In the same way dotted pairs are represented by

[3] This word is coined after Steele [Steele 84, page 36], but all entities are not first-class.

[4] In the following example, the function `index.of.type` will convert a type to a short number.

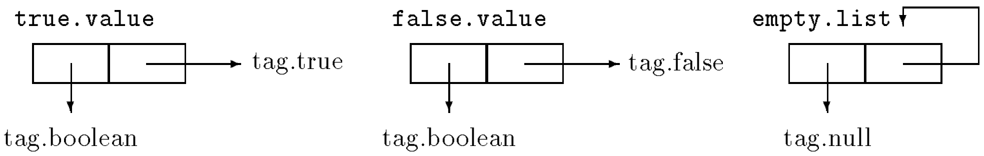


More compact representations can be adopted, for instance: mapping ${}_u$cons-cells on ${}_s$cons-cells. Since types are first-class entities in this interpreter, they may appear in ${}_u$`car` of ${}_u$dotted pairs. In order to avoid confusing such pairs with other entities, the ${}_s$`car` of any entity is a "tag". A tag lies behind the scene i.e., it is not a first-class entity but permits a type to be derived thanks to the ${}_s$`type.of` function. ${}_s$`type.of` may then be defined as:

```
(defun type.of (e)
  (cond ((eq (car e) tag.cons)    type.cons)
        ((eq (car e) tag.if.cont) type.if.cont)
        ...
        (t (error "Untyped entity !?")) ) )
```

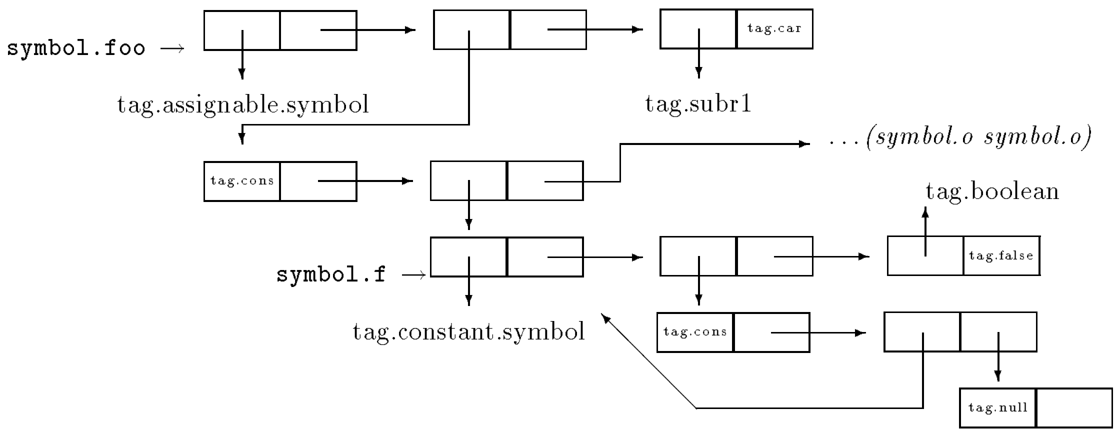
The default clause can be turned into `(t type.cons)` if one wants to spare cells.

The booleans and the empty list are not confused (at least in the interpreter, although one can regenerate it with such confusion). According to our representation scheme, truth values belong to `type.boolean` while the empty list belongs to `type.null`.



The ${}_s$`cdr` of `empty.list` is not used and therefore can be anything and, why not, `empty.list` itself. These three entities are accessible from the user as the predefined values of the ${}_u$symbols `T`, `F` and `NIL`.

The following figure partially summarizes the exact representation of the symbol `foo` just after the assignment `(setq foo car)`, to make a "subr" appear:



Two kinds of ${}_u$symbols exist: assignable or constant symbols. Either may be bound by `lambda` and `setq`'ed in the scope of this `lambda`. But only assignable symbols may be `setq`'ed at the toplevel. Constant symbols are predefined such as ${}_u$`car`, ${}_u$`cdr`, ${}_u$`cons` . . . or ${}_u$`F`, ${}_u$`T`, ${}_u$`NIL`. A compiler is therefore free to open-code or constant-fold the value of such a constant symbol if not shadowed by a lexical binding.

In the sequel we will use the following terminology. For each type, say `rib.env` (the "rib cage" structure used to record bindings from names to values as in [Steele 84, page 125]) we will take for granted that

- `(is.a.rib.env?` *expression*`)` is the predicate only satisfied by instances of `rib.env`.

- `(a.rib.env` *names values env*`)` returns a new rib cage environment associating *names* to *values* as an extension of *env*.
- `(names.of.rib.env` *rib.env*`)` selects the `names` component of a rib cage instance.
- `(set.env.of.rib.env` *rib.env env*`)` writes the `env` field of a rib cage instance. The value of this form is left unspecified.
- `tag.rib.env` is the tag found in the $_s$`car` of instances of `rib.env`.
- `type.rib.env` is the $_s$variable which value is the $_u$entity representing the type of `rib.env` entities. It is also the value of $_u$`rib.env.type` and of $_s$`(type.of (a.rib.env` *names values env*`))`.

2.3 Explicit Allocation

Allocations are completely explicit in our interpreter. The allocation interface is as follows:

```
(allocate entity-specification
          e r k ss s1 s2 fl
          #'(lambda (new.entity e r k ss new.s1 new.s2 new.fl)
               ... ) )
```

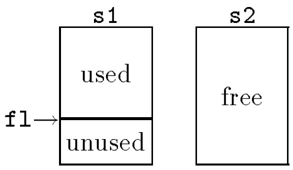
The first parameter specifies which entity (or entities) is (resp. are) to be allocated, for instance[5]:

```
(allocate (a.cons (an.assignable.symbol pnametag.undefined) ss)
          e r k ss s1 s2 fl
          #'(lambda (new.ss e r k ss new.s1 new.s2 new.fl) ...) )
```

An assignable $_u$symbol and a $_u$dotted pair are allocated and their fields are initialized as shown. The $_u$dotted pair is then given to the continuation variable `new.ss`. The allocation is done with respect to the current free-list `fl` and the new free-list after allocation will be bound to `new.fl`.

Since the first parameter is a static description of the entities to be allocated, it is not possible to allocate a varying number of entities. All our entities have a fixed number of fields and therefore varisized entities cannot be allocated as in [Queinnec & Cointe 88]. This is why the rib cage environment had been preferred to the usual association list since, given a list of actual arguments, a `rib.env` results from a single call to `allocate` where an alist environment would be obtained by as much calls to `allocate` as they are variables to be bound.

The `allocate` form looks like a function but nevertheless is a macro. If the free-list is not long enough, a garbage collection must be performed and the intended allocation must be retried. Another allocation failure would then `end` the whole interpreter. The current GC technique is "stop and copy" [Fenichel & Yochelson 69]. The from-space is `s1`, `s2` is the to-space and `fl` is the free-list of `s1`.



¿From an implementational point of view, `s1`, `s2` and `fl` are $_s$lists of $_s$cons-cells. The `s1` and `s2` lists are never altered and act as a memory "spine": $_s$`cdr` on this spine permits to access the next location of the memory. As usual these data types are encapsulated. An approximate expansion of `allocate` is therefore:

```
(if (can-allocate entity-specification fl)
    ((lambda (new.entity e r k ss new.s1 new.s2 new.fl) ...)
     (initialize the new entity) e r k ss s1 s2 (rest of fl) )
    (let ((fl (garbage.collect e r k ss s1 s2 fl)))
       (if (can-allocate entity-specification fl)
           ((lambda (new.entity e r k ss new.s1 new.s2 new.fl) ...)
```

[5] This excerpt is taken from the definition of $_u$`implode` which may create a new $_u$symbol and pushes it onto the oblist: `ss`.

```
            (initialize the new entity)
            (new.location.of.entity e)
            (new.location.of.entity r)
            (new.location.of.entity k)
            (new.location.of.entity ss)
            s2 s1 (rest of fl) ) ;exchange s1 and s2
           (end) ) ) ) ;abort !
```

Where, in the case of the oblist allocation previous example, (*can-allocate entity-specification* `fl`) would be:

```
(and (consp fl) (consp (cdr fl)) (consp (cddr fl)) (consp (cdddr fl)))
```

in order to allocate the four ${}_s$cons-cells needed and where (*initialize the new entity*) would be

```
(let ((new.ss (cell.of.store fl))
      (two (next.cell.of.store fl))
      (new.symbol (next.next.cell.of.store fl))
      (four (next.next.next.cell.of.store fl)))
   (set.tag.of.cell new.ss tag.cons)
   (set.field.of.cell new.ss two)
   (set.tag.of.cell new.symbol tag.assignable.symbol)
   (set.field.of.cell new.symbol four)
   (set.car.of.cons one new.symbol)
   (set.cdr.of.cons one ss)
   (set.pname.of.assignable.symbol new.symbol pname)
   (set.global.value.of.assignable.symbol new.symbol tag.undefined)
   new.ss )
```

And finally (*rest of* `fl`) would just be `cddddr`.

`garbage.collect` computes a new free-list after copying all used entities from `s1` to `s2`. It becomes clear from the model that ${}_u$`cons` is nothing but `(allocate (a.cons ...) ...)`. That allocation is performed on the spaces of cons-cells `s1` or `s2` and no ${}_s$`cons` is ever required at run-time: ${}_s$`cons` is not a run-time primitive.

To have explicit allocation imposes the adoption of a continuation passing style. That style is already necessary to express the meaning of escapes i.e., `call/cc`. Had we compared denotational semantics and our interpreter on a style level, there are not much differences. On the other hand, our interpreter is complete and can be run as a stand alone program. Explicit allocation af all entities necessary for the execution is uniform compared to the two allocation processes present in classical denotational semantics (see, for example [Rees & Clinger 86]): — (i) the "implementation dependent" *newlocation*(σ) computing new locations for dotted pairs or functions creation stamps (for `eq?`) — (ii) λ-abstractions or $\lambda[./.]$-extensions closing environments or continuations and representing stack- or heap- frames. Our interpreter deals with these details, may probably be proved with respect to denotational semantics and nevertheless keeps a reasonable size.

3 Architecture of the Interpreter

The interpreter is based on five mutually recursive procedures:

`evaluate`	handles variables references and special forms (i.e., `eval`).
`operate`	applies entities having a functional meaning (i.e., `apply`). Contains the definition of the ${}_u$primitives.
`resume`	resumes continuations.
`lookup`	looks up variable value.
`modify`	modifies variable value.

They all have impressive lists of variables following the same terminology:

`e`	the expression to be evaluated or to be returned.
`r`	the lexical environment.
`or`	the original current lexical environment (only used in `lookup` and `modify`).
`k`	the continuation.
`ss`	the list of `symbols`. This list (also known as the `oblist`) is used by `read`, or more precisely by `implode`, to intern symbols. It can also be seen as the "current package".
`s1`, `s2`	from- and to- spaces.
`fl`	the free-list.
`v`	a variable.

The interpreter takes care of ill formed expressions. It also uses some utility auxiliary functions whose names have been chosen to be easily understandable.

3.1 Evaluation of Expressions

The evaluator is naturally driven by the type of the expression to be evaluated. Symbols are "parsed" as variables, lists as special forms or applications. All other entities are self-evaluating.

```
(defun evaluate (e r k ss s1 s2 fl)
  (eval.case (type.of e)
    ((type.constant.symbol type.assignable.symbol)
     (lookup r e r k ss s1 s2 fl) )
    (type.cons
     (eval.case (car.of.cons e)
      (symbol.quote
       (if (has.1.parameter? (cdr.of.cons e))
         (resume k (cadr.of.cons e) r ss s1 s2 fl)
         (wrong msg.ill.formed.quotation e e r k ss s1 s2 fl) ))
      (symbol.if
       (if (has.3.parameters? (cdr.of.cons e))
         (allocate (an.if.cont e r k)
                   e r k ss s1 s2 fl
                   #'(lambda (newk e r k ss s1 s2 fl)
                       (evaluate (cadr.of.cons e) r newk ss s1 s2 fl) ) )
         (wrong msg.ill.formed.alternative e e r k ss s1 s2 fl) ))
      (symbol.setq
       (if (has.2.parameters? (cdr.of.cons e))
         (eval.case (type.of (cadr.of.cons e))
             ((type.constant.symbol type.assignable.symbol)
                 (allocate (a.setq.cont e r k)
                           e r k ss s1 s2 fl
                           #'(lambda (newk e r k ss s1 s2 fl)
                               (evaluate (caddr.of.cons e) r newk ss s1 s2 fl) ) ) )
             (else (wrong msg.requires.a.symbol e e r k ss s1 s2 fl)) )
         (wrong msg.ill.formed.assignment e e r k ss s1 s2 fl) ))
      (symbol.progn
       (if (consp (cdr.of.cons e))
         (evaluate.progn (cdr.of.cons e) r k ss s1 s2 fl)
         (resume k empty.list r ss s1 s2 fl) ))
      (symbol.lambda
       (if (is.a.cons? (cdr.of.cons e))
         (if (is.a.variable.list? (cadr.of.cons e))
           (allocate (a.closure (cadr.of.cons e) (cddr.of.cons e) r)
                     e r k ss s1 s2 fl
                     #'(lambda (o e r k ss s1 s2 fl)
```

```
                    (resume k o r ss s1 s2 fl) ) )
          (wrong msg.ill.variables e e r k ss s1 s2 fl) )
        (wrong msg.ill.formed.function e e r k ss s1 s2 fl) ) )
      (else
       (allocate (an.eval.function.cont e r k)
                 e r k ss s1 s2 fl
                 #'(lambda (newk e r k ss s1 s2 fl)
                     (evaluate (car.of.cons e) r newk ss s1 s2 fl)) )) ) )
    (else (resume k e r ss s1 s2 fl)) ) )
```

As usual progn handling deserves a special function:

```
(defun evaluate.progn (forms r k ss s1 s2 fl)
  (if (has.1.parameter? forms)
    (evaluate (car.of.cons forms) r k ss s1 s2 fl)
    (allocate (a.progn.cont forms r k)
              forms r k ss s1 s2 fl
              #'(lambda (newk forms r k ss s1 s2 fl)
                  (evaluate (car.of.cons forms) r newk ss s1 s2 fl) ) ) ) ) )
```

3.2 Resumption of a Continuation

Many continuation types appear in evaluate namely if.cont, setq.cont, progn.cont, etc. These entities may be seen as frames pushed on the top of the "stack". It is not a true stack since the control link [Bobrow & Wegbreit 73] is an explicit field. The meanings of these elementary continuation frames may be found in resume which is naturally driven by the type of the continuation. resume returns a value to the continuation.

```
(defun resume (k e r ss s1 s2 fl)
  (eval.case (type.of k)
      (type.eval.function.cont
         (let ((form (form.of.eval.function.cont k))
               (r    (env.of.eval.function.cont k))
               (k    (cont.of.eval.function.cont k)) )
           (allocate (an.apply.cont e k)
                     form r e ss s1 s2 fl
                     #'(lambda (newk form r e ss s1 s2 fl)
                         (evaluate.parameters (cdr.of.cons form)
                               r newk ss s1 s2 fl ) ) ) ) ) )
      (type.apply.cont
         (let ((func (func.of.apply.cont k))
               (k    (cont.of.apply.cont k)) )
           (operate func e r k ss s1 s2 fl) ) )
      (type.parameters.cont
         (let ((parameters (parameters.of.parameters.cont k))
               (r          (env.of.parameters.cont k))
               (k          (cont.of.parameters.cont k)) )
           (allocate (a.build.arguments.frame.cont e k)
                     parameters r k ss s1 s2 fl
                     #'(lambda (newk parameters r k ss s1 s2 fl)
                         (evaluate.parameters (cdr.of.cons parameters)
                              r newk ss s1 s2 fl ) ) ) ) ) )
      (type.build.arguments.frame.cont
         (let ((argument (argument.of.build.arguments.frame.cont k))
               (k        (cont.of.build.arguments.frame.cont k)) )
           (allocate (an.arguments.frame argument e)
                     e r k ss s1 s2 fl
```

```
                    #'(lambda (o e r k ss s1 s2 fl)
                           (resume k o r ss s1 s2 fl ) ) ) ) ) )
      (type.build.arguments.list.cont
         (let ((argument (argument.of.build.arguments.list.cont k))
               (k        (cont.of.build.arguments.list.cont k)) )
           (allocate (a.cons argument e)
                    e r k ss s1 s2 fl
                    #'(lambda (o e r k ss s1 s2 fl)
                           (resume k o r ss s1 s2 fl ) ) ) ) ) )
      (type.if.cont
         (let ((form (form.of.if.cont k))
               (r    (env.of.if.cont k))
               (k    (cont.of.if.cont k)) )
           (evaluate (if (eq e false.value)
                      (cadddr.of.cons form) (caddr.of.cons form) )
                    r k ss s1 s2 fl ) ) ) )
      (type.progn.cont
         (let ((forms (forms.of.progn.cont k))
               (r     (env.of.progn.cont k))
               (k     (cont.of.progn.cont k)) )
           (if (is.a.cons? (cddr.of.cons forms))
             (allocate (a.progn.cont (cdr.of.cons forms) r k)
                      forms r k ss s1 s2 fl
                      #'(lambda (newk forms r k ss s1 s2 fl)
                               (evaluate (cadr.of.cons forms) r
                                         newk ss s1 s2 fl ) ) ) )
             (evaluate (cadr.of.cons forms) r k ss s1 s2 fl) ) ) ) )
      (type.setq.cont
         (let ((form (form.of.setq.cont k))
               (r    (env.of.setq.cont k))
               (k    (cont.of.setq.cont k)) )
           (modify r (cadr.of.cons form) e r k ss s1 s2 fl) ) ) )
      (type.no.cont
         (femto.end) )
      (else (wrong msg.ill.formed.continuation k e r k ss s1 s2 fl)) ) ) )
```

These definitions are rather classic and natural. In particular if "tail-recursive"-ness means no unnecessary stack growth, the interpreter is properly tail-recursive. Note that since continuations are made of cons-cells, there is a neat consumption of resources even in the case of tail recursion. Continuation frames are not updated in place [Hudak 86]. This wastes memory but allows chronologically linking the frames, obtaining a reversible stepper as in [Lieberman 87]: a definite improvement when "debugging" semantics.

The evaluation of a form deserves special explanation. First of all, the functional part is evaluated from `evaluate` and its value is given back to `eval.function.cont` which stacks it (into an `apply.cont`) and starts to evaluate the parameters, thanks to:

```
(defun evaluate.parameters (parameters r k ss s1 s2 fl)
  (if (is.a.cons? parameters)
    (allocate (a.parameters.cont parameters r k)
             parameters r k ss s1 s2 fl
             #'(lambda (newk parameters r k ss s1 s2 fl)
                    (evaluate (car.of.cons parameters) r
                              newk ss s1 s2 fl ) ) ) )
    (resume k no.argument r ss s1 s2 fl) ) )
```

The parameters are evaluated from left to right and stacked into a `build.arguments.frame.cont` frame, once computed. When all parameters are evaluated, all `build.arguments.frame.cont` continuations contributes to building the arguments frame to be given to `apply.cont`. Other constructions may be chosen such as initially allocating an empty arguments frame of the right size (the number of parameters) and writing each computed argument in successive slots of it. This unfortunately requires allocating a varisized frame which is out of the scope of our current `allocate`. Should handling n-ary functions be done, it would complicate the normal process since the extra multiple arguments must be gathered (as Scheme specifies it) by $_u$`cons` rather than being stacked into the arguments frame.

3.3 Functional Application

All the functional behaviours are contained in `operate` which is naturally driven by the type of the entity to apply. The function `operate` is prudent and checks arguments number.

```
(defun operate (f e r k ss s1 s2 fl)
  (eval.case (type.of f)
      (type.closure
       (if (same.length e (variables.of.closure f))
         (allocate (a.rib.env (variables.of.closure f)
                              e
                              (environment.of.closure f) )
                   f r k ss s1 s2 fl
                   #'(lambda (newr f r k ss s1 s2 fl)
                       (evaluate.progn (body.of.closure f) newr k ss s1 s2 fl) ) )
         (wrong msg.incorrect.number.of.arguments f e r k ss s1 s2 fl) ) )
      (type.subr.zero
       (if (has.0.argument? e)
         (eval.case (tag.of.subr.zero f)
           (tag.end (femto.end))
           (tag.readch
            (resume k (read.ch) r ss s1 s2 fl) )
           (else (wrong msg.unknown.niladic.primitive f e r k ss s1 s2 fl)) )
         (wrong msg.requires.no.argument f e r k ss s1 s2 fl) ) )
      (type.subr.one
       (if (has.1.argument? e)
         (eval.case (tag.of.subr.one f)
           (tag.car (if (is.a.cons? (first.argument.of.arguments.frame e))
                        (resume k (car.of.cons (first.argument.of.arguments.frame e))
                                r ss s1 s2 fl )
                        (wrong msg.requires.a.cons f e r k ss s1 s2 fl) ))
           (tag.cdr (if (is.a.cons? (first.argument.of.arguments.frame e))
                        (resume k (cdr.of.cons (first.argument.of.arguments.frame e))
                                r ss s1 s2 fl )
                        (wrong msg.requires.a.cons f e r k ss s1 s2 fl) ))
           (tag.consp (resume k (if (is.a.cons? (first.argument.of.arguments.frame e))
                                    true.value false.value )
                              r ss s1 s2 fl ))
           (tag.symbolp (resume k (if (is.a.symbol? (first.argument.of.arguments.frame e))
                                      true.value false.value )
                                r ss s1 s2 fl ))
           (tag.princh
            (if (is.a.character? (first.argument.of.arguments.frame e))
              (progn (prin.ch (first.argument.of.arguments.frame e))
                     (resume k (first.argument.of.arguments.frame e) r ss s1 s2 fl) )
              (wrong msg.requires.a.character f e r k ss s1 s2 fl) ) )
           (tag.eval/ce
```

```
     (evaluate (first.argument.of.arguments.frame e) r k ss s1 s2 fl) )
    (tag.explode
     (if (is.a.symbol? (first.argument.of.arguments.frame e))
       (resume k (pname.of.symbol (first.argument.of.arguments.frame e))
               r ss s1 s2 fl )
       (wrong msg.requires.a.symbol f e r k ss s1 s2 fl) ) )
    (tag.implode
     (if (all.character? (first.argument.of.arguments.frame e))
       (let ((s (find.symbol (first.argument.of.arguments.frame e) ss)))
         (if s (resume k s r ss s1 s2 fl)
             (allocate (an.oblist
                         (an.assignable.symbol
                          (first.argument.of.arguments.frame e)
                          tag.undefined )
                         ss )
                        e r k ss s1 s2 fl
                        #'(lambda (newss e r k ss s1 s2 fl)
                            (progn
                              (resume k (car.of.cons newss)
                                      r newss s1 s2 fl ) ) ) ) ) ) )
       (wrong msg.requires.a.character.list e e r k ss s1 s2 fl) ) )
    (tag.type.of
     (resume k (type.of (first.argument.of.arguments.frame e))
             r ss s1 s2 fl ) )
    (else (wrong msg.unknown.monadic.primitive f e r k ss s1 s2 fl)) )
  (wrong msg.requires.one.argument f e r k ss s1 s2 fl) ) )
(type.subr.two
  (if (has.2.arguments? e)
    (eval.case (tag.of.subr.two f)
     (tag.rplaca
      (if (is.a.cons? (first.argument.of.arguments.frame e))
        (progn
          (set.car.of.cons (first.argument.of.arguments.frame e)
                           (second.argument.of.arguments.frame e) )
          (resume k (first.argument.of.arguments.frame e) r ss s1 s2 fl) )
        (wrong msg.requires.a.cons f e r k ss s1 s2 fl) ))
     (tag.rplacd
      (if (is.a.cons? (first.argument.of.arguments.frame e))
        (progn
          (set.cdr.of.cons (first.argument.of.arguments.frame e)
                           (second.argument.of.arguments.frame e) )
          (resume k (first.argument.of.arguments.frame e) r ss s1 s2 fl) )
        (wrong msg.requires.a.cons f e r k ss s1 s2 fl) ))
     (tag.cons
      (allocate (a.cons (first.argument.of.arguments.frame e)
                        (second.argument.of.arguments.frame e) )
                e r k ss s1 s2 fl
                #'(lambda (o e r k ss s1 s2 fl)
                    (resume k o r ss s1 s2 fl) ) ))
     (tag.eq
      (resume k (if (eq (first.argument.of.arguments.frame e)
                        (second.argument.of.arguments.frame e) )
                  true.value false.value )
```

```
                r ss s1 s2 fl ))
          (else (wrong msg.unknown.dyadic.primitive f e r k ss s1 s2 fl)) )
        (wrong msg.requires.two.arguments f e r k ss s1 s2 fl) ) )
  (else (wrong msg.not.applyable f e r k ss s1 s2 fl)) ) )
```

All the offered $_u$primitives appear in `operate`. These are:

subr0 `end readch`
subr1 `car cdr princh eval/ce explode implode type.of`
subr2 `rplaca rplacd cons eq`

Some remarks can be made:

- $_u$`end` ends the interpreter. It just calls the implementation dependent function $_s$`femto.end` which returns a definite value thus ending the interpreter (thanks to the continuation passing style). The function `femto.end` only exists to ease the translation of the interpreter into a language different from Lisp. For instance, a C translation may be:

  ```
  (def.C.translation femto.end ()
      "exit(0);" )
  ```

  Since $_u$`end` takes no arguments and never returns a value, it might well be a special form and appear in `evaluate` rather than `operate`. The reason to leave it here is that a special form is not a first class entity as this niladic function $_u$`end` is.

- Functions $_u$`car`, $_u$`cdr`, $_u$`rplaca` and $_u$`rplacd` just call appropriately the associated function $_s$`car.of.cons`, $_s$`cdr.of.cons`, $_s$`set.car.of.cons` and $_s$`set.cdr.of.cons`. Nonetheless they check that their first argument is a dotted pair, with $_s$`is.a.cons?`. The constructor $_u$`cons` is similarly mapped onto $_s$`a.cons` but $_u$`eq` remains $_s$`eq`. It is even possible to omit the $_u$`consp` check. Naturally this very test must be done elsewhere to offer the same safety. If $_u$`unsafe.car` were defined, in `operate`, as

  ```
  (tag.unsafe.car (resume k (car.of.cons (first.argument.of.arguments.list e))
                            r ss s1 s2 fl ))
  ```

  then the system can be booted with

  ```
  (setq car ((lambda (unsafe.car)
               (lambda (e) (if (consp e) (unsafe.car e)
                                 (error) )) )
             car ))
  ```

  This expression rebuilds our original $_u$`car` but the test is now performed at the user's level. This trick reduces the size of the interpreter. It can be used in many places: before `implode`, `explode`, `rplaca` and `rplacd` ...

- $_u$`readch` reads a $_u$character from the input stream. The implementation dependent function $_s$`read.ch` reads a $_s$character and converts it to the $_u$symbol having this single character as pname. Similarly, $_u$`princh` takes a $_u$character and outputs the corresponding $_s$character on the output stream. The real work is done by the implementation dependent function $_s$`prin.ch`.

  The functions $_s$`read.ch` and $_s$`prin.ch` are generated with the interpreter but this is explained in section 3.7.

- $_u$`type.of` returns the type of its argument. It has two consequences – it economizes primitives, but – types must be first class entities. At least three types of entities can be handled by the user: symbols, dotted pairs and functions. Three predicates must be offered to discriminate among them: `symbolp`, `consp` and `functionp`. As the implementor can extend the interpreter (see section 4) s/he needs more and more types i.e., type predicates. We thus decide to offer first class type entities that may be obtained by $_u$`type.of`. A predefined set of constant $_u$symbols (`symbol.type`, `cons.type` and `function.type`) is predefined and has these types as values. Therefore type predicates can be defined by the user, for instance:

```
(setq consp (lambda (e) (eq (type.of e) cons.type)))
```

- `eval/ce` is `eval` with current environment. In usual interpreters and standard denotational semantics, `eval/ce` is a special form since it uses the current lexical environment which is not given to functions. In our interpreter, the variables `e`, `r`, `k` ... act as the registers of a virtual machine and are accessible from `operate`. Since `eval/ce` has the interface of a function i.e., takes an evaluated argument, it appears in `operate`. Strictly speaking, `eval/ce` is not really necessary. Its single use is in the `toplevel` loop that the user can write in $_u$Lisp. Moreover, `eval` (which evaluates its argument in the global lexical environment) would have sufficed for this use since the lexical environment of the `toplevel` function is not really important. Conversely interpretative debbugging is easier with `eval/ce` since it allows accessing to lexical environments.

  Both cases are simple to add to the interpreter since `evaluate` does the whole job. `eval/ce` (or `eval`) is just a costless addendum (two lines) to the interpreter. This addendum only breaks compilation, selective linking and the structural denotation principle [Muchnick & Pleban 80].

3.4 Environment Lookup and Modification

Referencing variables leads to consulting the lexical environment. `lookup` is naturally driven by the type of the environment. Two types exist: rib cage and global environments. The latter is an empty structure marking the end of the lexical environment while the former record associations between names and values as bound by function application.

```
(defun lookup (r v or k ss s1 s2 fl)
  (eval.case (type.of r)
    (type.rib.env
       (let ((bool (.appear v (names.of.rib.env r))))
         (if bool
           (resume k (.extract v (names.of.rib.env r) (values.of.rib.env r))
                   or ss s1 s2 fl )
           (lookup (env.of.rib.env r) v or k ss s1 s2 fl) ) ) )
    (type.global.env
       (eval.case (type.of v)
           (type.constant.symbol
             (let ((value (global.value.of.constant.symbol v)))
               (resume k value or ss s1 s2 fl) ) )
           (type.assignable.symbol
             (let ((value (global.value.of.assignable.symbol v)))
               (if (eq value tag.undefined)
                 (wrong msg.undefined.variable v v or k ss s1 s2 fl)
                 (resume k value or ss s1 s2 fl) ) ) ) ) )
    (else (wrong msg.undefined.environment r v or k ss s1 s2 fl)) ) )
```

The function `lookup` takes all the registers of the virtual machine and therefore is able to call `wrong` directly if a variable is unbound. Note that this test is only necessary while in the global environment. An instance of `rib.env` cannot lead to a variable bound to `tag.undefined` since there is no means to compute this value (remember that a tag is not a first class entity).

Similarly, modifications to variables as expressed by `setq` are done via `modify` whose structure mimics that of `lookup`.

```
(defun modify (r v new or k ss s1 s2 fl)
  (eval.case (type.of r)
    (type.rib.env
       (let ((bool (.appear v (names.of.rib.env r))))
         (if bool
           (progn
             (.intract v (names.of.rib.env r) (values.of.rib.env r) new)
             (resume k new or ss s1 s2 fl) )
           (modify (env.of.rib.env r) v new or k ss s1 s2 fl) ) ) )
```

```
    (type.global.env
       (eval.case (type.of v)
          (type.assignable.symbol
             (progn
               (set.global.value.of.assignable.symbol v new)
               (resume k new or ss s1 s2 fl) ) )
          (type.constant.symbol
             (let ((value (global.value.of.constant.symbol v)))
               (if (eq value tag.undefined)
                 (progn
                   (set.global.value.of.constant.symbol v new)
                   (resume k new or ss s1 s2 fl) )
                 (wrong msg.constant.assignment v new or k ss s1 s2 fl) ) ) ) ) )
    (else (wrong msg.undefined.environment v new or k ss s1 s2 fl)) ) )
```

It is up to `modify` to check if the binding of a variable is mutable or not. In this interpreter only global bindings can be immutable and this quality is out of the control of the user. The global bindings for $_u$primitives like `consp`, types like `boolean.type` and constants like `T`, `F` or `NIL` are predefined to be immutable. The mutability information is vital for compilers which can open-code or constant-fold the values associated to these bindings. This mutability information is associated with the precise nature of the symbol naming the binding.

3.5 Error Handling

It is important in a Lisp system to provide the user a means to handle the various errors that can occur. As far as we have seen, all erroneous situations invoke `wrong` with a message, the main culprit and all the registers of the virtual machine. To stick to a simple interface we propose that the function currently bound to $_u$`error` be called with two arguments: the message and the main culprit. Then it is up to the user to do what s/he wants or what s/he can.

```
(defun wrong (msg culprit e r k ss s1 s2 fl)
  (allocate (an.arguments.frame msg
              (an.arguments.frame e no.argument) )
            e r k ss s1 s2 fl
            #'(lambda (arguments e r k ss s1 s2 fl)
                (operate (global.value.of.assignable.symbol symbol.error)
                         arguments r k ss s1 s2 fl ) ) ) )
```

To give an example of this error handling, consider the default definition of `error` as it appears in the bootstrap.

```
(setq error
      (lambda (msg culprit)
        (prin msg)
        (print culprit)
        (return.to.toplevel (quote Return\ from\ error)) ))
(setq toplevel
      ((lambda (f)
         (setq f (lambda (it)
                   (progn (prin prompt.out)
                          (print it)
                          (prin prompt.in)
                          (f (eval/ce (read))) ) ))
         f )
       nil ) )
(toplevel (call/cc (lambda (k)
                     (setq return.to.toplevel k)
                     (quote Hello\ Femto) )))
```

The `call/cc` function is explained in section 4.

3.6 Memory Management

The last parts to be exposed are the innards of the Garbage Collector. Our interpreter uses a simple stop and copy collector. Memory is represented by three $_s$lists of dotted $_s$pairs[6]. Two of them represent the two spaces needed by the collector. The third space contains static data which is never moved. This allows the evaluator global variables such as **symbol.setq**, **type.cons** ... to be located at constant addresses. User data are only coded by the $_s$**car** of these lists as shown in figure 3.6.

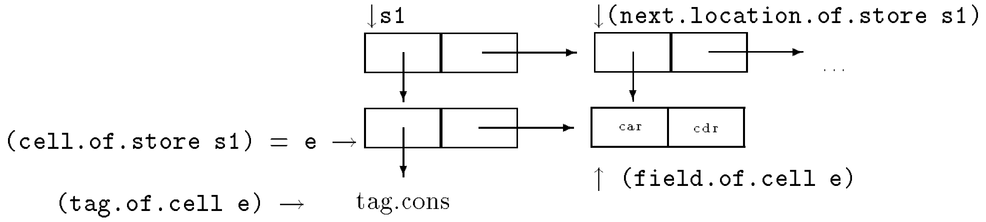


Figure 2: Store Representation

The functions **next.location.of.store** and **cell.of.store** encapsulate the store in order to be redefined if one wants a more efficient implementation. The functions involved in the GC are simple but perform many pointer redirections. Apart **garbage.collect**, the two other functions **move.entity** and **scavenge** are generated from the type descriptions. This technique is worth presenting. Types are described in the bootstrap process of FEMTO(see section 3.7). Knowing all their slots allows generating the precise methods for all types. **move.entity** displaces an entity from the from-space to the to-space, leaving a "broken heart" in the from space leading from the old to the new locaiton of the entity. **scavenge** recursively marks all reachable entities by **move**-ing them. We only give the generated excerpts of **scavenge** and move.entity on **type.cons**.

```
(defun garbage.collect (e r k ss s1 s2 fl)
  (scavenge s2
            (move.entity e
             (move.entity r
              (move.entity k
               (move.entity ss
                (scavenge static.store s2))))) ) )
(defun scavenge (tbml fl)
  ;;scan all entities lying between tbml and fl, returns fl.
  ;;(before < tbml, all sons are in the to-space).
  (if (or (is.empty.store? tbml) (eq fl tbml))
    fl  ;returns the final free-list
    (let ((e (cell.of.store tbml)))
      (eval.case (type.of e)
        (type.cons
           (scavenge (next.location.of.store
                        (next.location.of.store tbml) )
                     (prog1 (move.entity (cdr.of.cons e)
                               (move.entity (car.of.cons e) fl) )
                            (set.cdr.of.cons e
                               (new.location.of.entity (cdr.of.cons e)) )
                            (set.car.of.cons e
```

[6] It now appears that no atoms at all are present in the memory provided tags are represented by $_s$dotted pairs (for example in a way reminiscent of $_u$characters).

```
                              (new.location.of.entity (car.of.cons e)) ) ) )
        ... ) )
(defun move.entity (e fl)
   ;;returns the rest of the free-list
   (if (already.moved? e) e
     (if (is.unmovable? e) e
       (eval.case (type.of e)
         (type.cons
            (let ((one (cell.of.store fl))
                  (two (next.location.of.store (cell.of.store fl)))
                  (new.fl (next.location.of.store (next.location.of.store fl))) )
              (set.tag.of.cell one tag.type.cons)
              (set.field.of.cell one two)
              (set.car.of.cons two (car.of.cons e))
              (set.cdr.of.cons two (cdr.of.cons e))
              (set.tag.of.cell e tag.marked)
              (set.field.of.cell e one)
              new.fl ) )
          ... ) ) ) ) )
(defun already.moved? (e)
  (eq (tag.of.cell e) tag.marked) )
(defun is.unmovable? (e)
  (member e static.store) )
(defun new.location.of.entity (e)
  ;;assume (already.moved? e)
  (if (is.unmovable? e) e
      (field.of.cell e) ) )
```

3.7 Bootstrap

The bootstrap of this interpreter is not a simple problem since many inter-related data have to be set up together. A huge macro, named `def.resources`, takes care of that and generates all the code which, when evaluated, will construct the initial memory of the interpreter. We will explain the various fields of `def.resources` and indicate their meanings.

First of all we have to define all the data structures used for the interpreter. This is the role of the `types` field.

```
(def.resources (types
  (if.cont (form env . cont))
  (eval.function.cont (form env . cont))
  (build.arguments.list.cont (argument . cont))
  ...
  (boolean tag)
  (null ())
  (closure (variables body . environment))
  (cons (car . cdr))
  ... )
```

For each type we generate a constructor, a predicate, the read and write accessors and the move and mark methods for GC. This functionality is similar to `defstruct` where instances would be built by $_s$cons-cells rather than vectors. A first class entity is also generated for each type (with a special case for `type` which has itself as type):

```
(setq type.cons (a.type))
```

The `type.of` function converting tags to types is then generated straightforwardly:

```
(defun type.of (exp)
   (let ((type (car exp)))
     (cond ((eq type tag.cons) type.cons)
           ... ) ) )
```

For each legal character, appearing in the `characters` clause, an associated character is built.

```
(def.resources ...
    (characters ...
                #\N #\O #\P #\Q #\R #\S #\T #\U #\V #\W #\X #\Y #\Z
                #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9 #\0 (star #\*)
                (dash #\-) (quote.mark #\') (back.slash #\) (sharp #\#)
                ... ) )
```

The generated code is simply

```
(setq |symbol.N| (a.symbol 'ignore tag.undefined))
(set.pname.of.symbol |symbol.N| (a.cons |symbol.N| empty.list))
```

Of course `tag.undefined` and `empty.list` were defined, by hand, just before. The `prin.ch` and `read.ch` converters can now be defined as

```
(defun prin.ch (c)
  (write-char (cond ((eq c symbol.A) #\A)
                    ... )) )
(defun read.ch ()
  (let ((ch (read-char)))
    (cond ((eq ch #\A) symbol.A)
          ... ) ) )
```

Symbols can now be composed by their pnames. Other fields of `def.resources` are

```
(def.resources ...
    (special.form.keywords quote if setq progn lambda)
    (subr.zero   end readch )
    (subr.one    car cdr consp princh implode explode symbolp eval/ce type.of)
    (subr.two    cons rplaca rplacd eq)
    (messages (msg.ill.formed.quotation      "Ill formed quotation")
              (msg.ill.formed.continuation   "Ill formed continuation")
              (msg.ill.formed.alternative    "Ill formed alternative")
              ... )
    (boot (progn ...)) )
```

Characters are already built. Type names, special form names, subr names, messages, constant names and all symbol names appearing in the boot expressions are collected and defined as $_u$symbols. For example:

```
(setq symbol.CONS
   (a.symbol (a.cons symbol.C (a.cons symbol.O
               (a.cons symbol.N (a.cons symbol.S empty.list)) ))
             tag.undefined ) )
```

It is therefore possible to create the initial oblist

```
(setq ss.init
      (an.oblist symbol.A (an.oblist symbol.B (an.oblist symbol.C
       (an.oblist symbol.CAR
        ... (an.oblist symbol.Z (an.empty.oblist)) ...)))))
```

Now values of these symbols can be defined where they must be. For all type names :

```
(set.global.value.of.constant.symbol symbol.CONS.TYPE type.cons) ...
```

For all subrs :

```
(set.global.value.of.constant.symbol symbol.cons (a.subr tag.cons)) ...
```

For all constants :

```
(setq true.value (a.boolean tag.true))
(setq false.value (a.boolean tag.false))
(set.global.value.of.constant.symbol symbol.T true.value)
(set.global.value.of.constant.symbol symbol.F false.value)
(set.global.value.of.constant.symbol symbol.NIL empty.list)
```

The initial content of the `static.store` and the `s1` space[7] can now be built from the `boot` clause of `def.resources`.

```
(def.resources ...
    (boot (progn
            (setq prompt.out (quote \
\=\= ))
            (setq prompt.in (quote \?\?\ ))
            ((lambda (quote.char dot.char space.char newline.char
                       bra.char ket.char back.slash.char peekch
                       readch eq cons consp )
               (setq quote.char (quote \'))
               (setq dot.char (quote \.))
               ...  )))))
```

In particular and since ${}_u$`readch` and ${}_u$`princh` are all we need to ${}_u$`read` or ${}_u$`print` expressions, the definitions of the reader and the printer are simply done in ${}_u$Lisp and appear in the boot expression (see the appendix for more details). In fact only reader must be, at the beginning, in the memory. It can thus read the rest of the boot expressions and particularly the printer. Since the mechanism exists to put some expressions within the initial memory, let us put all the necessary functions i.e., `read`, `print`, `toplevel` and `error`. Therefore, from the boot expression, is generated the code which constructs the initial memory, something like

```
(setq e.init
   (a.cons symbol.PROGN                              ;(progn
           (a.cons (a.cons symbol.SETQ               ;(setq prompt.out '...
                           (a.cons symbol.PROMPT.OUT
                                   (a.cons (a.cons symbol.QUOTE
                        ...
```

The interpreter is now almost set up. All the previously performed allocations have been recorded in a list: `s1`. We just have to adjoin a sufficient free-list and to provide the same number of dotted pairs in `s2`.

```
(setq fl.init (mapcar #'identity
                      (make-list *memory.size*) ))
(setq s1 (nconc fl.init s1))
(setq s2 (mapcar #'(lambda (cell) (cons nil nil))
                 s1 ))
(setq r.init (a.global.env))
(setq k.init (a.no.cont))
```

The interpreter can now be started by the `femto` function

```
(defun femto ()
  (evaluate e.init r.init k.init ss.init s1 s2 fl.init) )
```

4 Extensions to Reflection

Great care has been exercised that many implementation techniques can be done by simple redefinitions of encapsulations. For example, various compilations of `eval.case` or `allocate` were tested and the GC was

[7] These data will disappear after some collections and particularly the boot expression.

originally done with arrays. A C version of this interpreter is under progress. It is obtained by code-walking the Lisp code and translating it to C.

The extension towards reflection is worth presenting.

Reflection has been introduced in [des Rivieres & Smith 84], discussed and refined in [Friedman & Wand 84], [Wand & Friedman 86], [Danvy & Malmkjær 88], [Bawden 88]. Adding some reflective capabilities to this interpreter is straightforward. At any time, the content of the `e`, `r`, `k` or `ss` registers of the virtual machine are always first-class entities and therefore do not need to be reified[8]. We just provide the hooks so the user may be given these entities and return them back. Let us introduce `reify` which gives the exact content of `e`, `r`, `k` and `ss` to its argument:

```
(reify (lambda (e r k ss) ...))
```

The value of the body of the argument of `reify` will become the value of the `reify` form. We just have to add a few lines in `operate`

```
(tag.reify
  (if (is.a.function? (first.argument.of.arguments.list e))
    (allocate (an.arguments.list e
               (an.arguments.list r
                (an.arguments.list k
                 (an.arguments.list ss no.argument) ) ) )
              e r k ss s1 s2 fl
              #'(lambda (arguments e r k ss s1 s2 fl)
                  (operate (first.argument.of.arguments.list e)
                           arguments r k ss s1 s2 fl ) ) )
    (wrong msg.requires.a.function f e r k ss s1 s2 fl) ) )
```

This ability already permits interesting effects. The user can obtain the oblist with

```
(setq oblist (lambda ()
               (reify (lambda (e r k oblist) oblist)) ))
```

We now add some functional behaviours to `operate` considering environments or continuations to be first class and applicable :

```
((type.argument.cont type.eval.function.cont type.apply.cont
  type.if.cont type.progn.cont type.setq.cont )
 (if (has.1.argument? e)
   (resume f (first.argument.of.arguments.list e) r ss s1 s2 fl)
   (wrong msg.requires.one.argument f e r k ss s1 s2 fl) ) )
((type.rib.env type.global.env)
 (if (has.1.argument? e)
   (if (is.a.symbol? (first.argument.of.arguments.list e))
     (lookup r (first.argument.of.arguments.list e) r k ss s1 s2 fl)
     (wrong msg.requires.a.symbol f e r k ss s1 s2 fl) )
   (if (has.2.arguments? e)
     (if (is.a.symbol? (first.argument.of.arguments.list e))
       (modify r (first.argument.of.arguments.list e)
                 (second.argument.of.arguments.list e)
                 r k ss s1 s2 fl )
       (wrong msg.requires.a.symbol f e r k ss s1 s2 fl) )
     (wrong msg.requires.one.or.two.arguments f e r k ss s1 s2 fl) ) ) )
```

It is thus possible for the users to write their own `call/cc` as

```
(setq call/cc (lambda (fn)
                (reify (lambda (e r k oblist)
                         (fn k) )) ))
```

[8] The three other registers `s1`, `s2` and `fl` represent the store and are out of our reflection mechanism.

Note that this one is "pushy", not "jumpy" (see [Danvy & Malmkjær 88]). First class environments eases debugging :

```
(reify (lambda (e r k ss)
           (r 'x (cons t (r 'x))) ))
```

is just `(setq x (cons t x))`

The inverse of `reify` is `reflect` which installs its arguments in the `e`, `r`, `k` and `ss` registers of the virtual machine. The invocation form is

`(reflect` *e r k ss*`)`

The `reflect` function is only a few lines in `operate`:

```
(type.subr.four
  (if (has.4.arguments? e)
    (eval.case (tag.of.subr.four f)
       (tag.reflect
         (progn
           (evaluate (first.argument.of.arguments.list e)
                     (second.argument.of.arguments.list e)
                     (third.argument.of.arguments.list e)
                     (fourth.argument.of.arguments.list e)
                     s1 s2 fl ) ) )
      (else (wrong msg.unknown.tetradic.primitive f e r k ss s1 s2 fl)) )
    (wrong msg.requires.four.arguments f e r k ss s1 s2 fl) ) )
```

The whole reflective process would be much more interesting provided the existence of some read and write accessors to these data. Had we implemented environments or continuations by $_u$cons-cells instead of $_s$cons-cells[9], users would have been able to exercise their surgery skill with the traditional $_u$`rplaca` or $_u$`rplacd`. Debuggers and escapes like $_u$`catch` and $_u$`throw` would have become possible. Had the oblist be coded by $_u$cons, one can obtain it, modify it and reinstall it as the "current package". Esoteric read-time effects such as `unintern` or other weird package manipulations can then be attempted.

5 Conclusions

The presented interpreter can be summarized with a few figures.

8	running primitives	: `car cdr rplaca rplacd eq read-char write-char end`
3	more features	: variable reference, functional applications, alternatives
3	initialization features	: `setq lambda cons`
5	offered special forms	: `if quote setq progn lambda`
15	offered primitives	: `end car cdr rplaca rplacd cons eq readch` `princh type.of eval/ce explode implode reify reflect`
–	other offered features	: first class fonctions, continuations and environments
3	offered libraries	: `read print error`

The code of the interpreter is approximatively (before expansion) 500 lines of code. The boot expression represents approximatively 750 $_u$cons-cells (i.e., 1500 $_s$cons-cells). These figures are small compared to the level of details embedded in this definition. More amazing is that the description is complete i.e., can be translated into a stand alone program. One more time, this confirms the local optimum reached by Lisp, the sole programming language to have a metacircular definition of this size. The inefficiency of FEMTO is related to the small number of concepts upon which it is built. This makes it a good candidate for bench-marking but also for pedagogical exercices since no weird microcode is involved in it.

FEMTO is built of only a few means but its structure is driven by modern concepts such as generic functions, automatic garbage collector synthesis and reflection. The power of the specified language compares

[9] This is a simple modification. Just change $_s$`cons`, $_s$`car`, $_s$`cdr` ... by $_s$`a.cons`, $_s$`car.of.cons`, $_s$`cdr.of.cons` in the definitions of the $_s$data types definition.

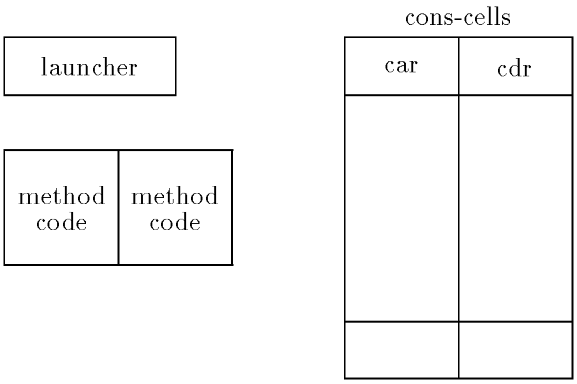


Figure 3: The Turing-McCarthy World

well with the specifying language which is so restricted that stand-alone translations or compilations are straightforward. We offer a final view of FEMTO consisting of a world of cons-cells coupled with a few pieces of code. To increase the reflectivity we can also add means to retrieve or alter methods from types. The small number of primitive entities lying in this world confers FEMTO the status of a kind of "Turing-McCarthy" machine devoted to crunching lists.

References

[Abelson & Sussman 85] Harold Abelson, Gerald Sussman, with Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge MA, 1985.

[Bawden 88] Alan Bawden *Reification without Evaluations*, 1988 ACM Conference on Lisp and Functional Programming, pp 342–351, Snowbird, Utah.

[Bobrow & Wegbreit 73] Daniel G. Bobrow, Ben Wegbreit, *A model and Stack Implementation of Multiple Environments*, CACM 16, 10 (Oct 1973), pp 591-603.

[Bobrow et al. 88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales and David A. Moon, *Common Lisp Object System Specification*, ISO/IEC JTC1/SC 22/WG 16 Lisp N 10, March 1988.

[Boyer & Moore 82] Robert S. Boyer and J. Strother Moore, *A Mechanical Proof of the unsolvability of the Halting Problem*, Technical Report ICSCA-CMP-28, July 1982.

[Brooks & Gabriel & Steele 82] Rodney A. Brooks, Richard P. Gabriel, Guy L. Steele Jr., *An Optimizing Compiler for LexicallyD Scoped Lisp*, Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, SIGPLAN Notices 17,6 (June 1982), pp. 261-273.

[Clinger 84] William Clinger, *The Scheme 311 compiler: An exercise in denotational semantics*, 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, pp 356 – 364.

[Comer 84] Douglas Comer, *Operating System Design, the XINU Approach*, Prentice-Hall International Editions, 1984.

[Danvy & Malmkjær 88] Olivier Danvy, Karoline Malmkjæer, *Intensions and Extensions in a Reflective Tower*, 1988 ACM Conference on Lisp and Functional Programming, pp 327–341, Snowbird, Utah.

[des Rivieres & Smith 84] Jim des Rivières, Brian Cantwell Smith, *The Implementation of Procedurally Reflective Languages*, 1984 ACM Conference on Lisp and Functional Programming, pp 331–347.

[Dybvig 87] R. Kent Dybvig, *The Scheme Programming Language*, Prentice-Hall, 1987.

[Fenichel & Yochelson 69] R.R. Fenichel, J.C. Yochelson, *A Lisp Garbage Collector for Virtual-Memory Computer Systems*, Communications of ACM, Vol. 12, No 11, November 1969.

[Friedman & Wand 84] Daniel P. Friedman, Mitchell Wand, *Reification: Reflection without Metaphysics*, 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, pp 348 – 355.

[Gordon 75] Michael J. C. Gordon, *Operational Reasoning and Denotational Semantics*, Actes du colloque IRIA Construction et Justification de Programmes, G. Huet & G. Kahn (eds), Arc et Senans (Juillet 1975).

[Hudak 86] Paul Hudak, *A semantic Model of Reference Counting and its Abstraction (Detailed Summary)*, 1986 ACM Symposium on Lisp and Functional Programming, Boston, Mass., pp 351–363.

[Kessler 88] Robert R. Kessler, *Lisp, Objects, and Symbolic Programming*, Scott, Foreman/Little, Brown College Division, Glenview, Illinois, 1988.

[Lakin 80] Fred H. Lakin, *Computing with Text-Graphic Forms*, Conference Record of the 1980 Lisp Conference, 1980, pp 100–105.

[Lieberman 87] Henry Lieberman, *Reversible Object-Oriented Interpreters*, ECOOP'87, Special Issue of BIGRE 54, June 87, pp 13–21.

[Lieberman & Fry 88] Henry Lieberman, Christopher Fry, *Common EVAL*, Lisp Pointers, Vol 2, Number 1, July-August-September 1988, pp 23–33.

[McCarthy et al. 62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, Michael I. Levin, *Lisp 1.5 Programmer's Manual*, MIT Press, Cambridge, Massachusetts, 1962.

[McCarthy 78] John McCarthy, *A Micro-Manual for Lisp - Not the whole Truth*, ACM SIGPLAN Notices, Volume 13, Number 8, August 1978, pp 215–216.

[Muchnick & Pleban 80] Steven S. Muchnick, Uwe F. Pleban, *A Semantic comparison of Lisp and Scheme*, Lisp Conference 1980, pp 56–64.

[Queinnec 84] Christian Queinnec, *Lisp*, Macmillan, 1984.

[Queinnec & Cointe 88] Christian Queinnec, Pierre Cointe, *An open-ended Data Representation Model for* EuLisp, 1988 ACM Conference on Lisp and Functional Programming, pp 298–308, Snowbird, Utah.

[Rees & Clinger 86] Jonathan A. Rees, William Clinger, *Revised^3 Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, 21, 12, Dec 86, pp 37 – 79.

[Reynolds 72] John C. Reynolds, *Definitional Interpreters for Higher-Order Programming Languages*, ACM National Conference 1972, Vol. 2, pp 717–740.

[Saint James 87] Emmanuel Saint-James, *De la Méta-Récursivité comme Outil d'Implémentation*, Thèse d'état, Université Paris VI, Décembre 1987.

[Schmidt 86] David A. Schmidt, *Denotational Semantics, A Methodology for Language Development*, Allyn and Bacon, Inc., Newton, Mass., 1986.

[Steele & Sussman 78] Guy L. Steele Jr., Gerald J. Sussman, *The Art of the Interpreter, or the modularity complex (parts zero, one, and two)*, MIT AI Memo 453, May 1978.

[Steele 84] Guy L. Steele Jr., *Common Lisp, the Language*, Digital Press, Burlington MA, 1984.

[Stoy 77] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass., 1977.

[Sussman & Steele 75] Gerald J. Sussman, Guy L. Steele, Jr., *Scheme: an Interpreter for extended lambda Calculus*, MIT AI Memo 349, December 1975.

[Wand & Friedman 86] Mitchell Wand, Daniel Friedman, *The Mystery of the Tower revealed: A non reflective Description of the Reflective Tower*, 1986 ACM Conference on Lisp and Functional Programming, pp 233–248.

[Winston 88] Patrick H. Winston, Berthold K. Horn, *Lisp*, 3rd Edition, Addison Wesley, 1988.

Annex

```
;;; This rather lengthy expression describes all the necessary
;;; resources needed to synthetize the primitives.
(def.resources
    ;; the alphabet
    (characters
            #\a #\b #\c #\d #\e #\f #\g #\h #\i #\j #\k #\l #\m
            #\n #\o #\p #\q #\r #\s #\t #\u #\v #\w #\x #\y #\z
            #\A #\B #\C #\D #\E #\F #\G #\H #\I #\J #\K #\L #\M
            #\N #\O #\P #\Q #\R #\S #\T #\U #\V #\W #\X #\Y #\Z
            #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9 #\0 (star #\*)
            (dash #\-) (quote.mark #\') (back.slash #\\) (sharp #\#)
            (space #\ ) (bra #\() (ket #\)) (dot #\.) (comment #\;)
            (equal #\=) (question.mark #\?) (slash #\/) (newline #\
                                                                 ))
    ;; types used in the implementation.
    (types (if.cont (form env . cont))
           (no.cont ())
           (eval.function.cont (form env . cont))
           (apply.cont (func . cont))
           (build.arguments.list.cont (argument . cont))
           (build.arguments.frame.cont (argument . cont))
           (parameters.cont (parameters variables env . cont))
           (setq.cont (form env . cont))
           (progn.cont (forms env . cont))
           (alist.env (name value . env))
           (rib.env ((names . values) . env))
           (global.env ())
           (arguments.list (current.argument . other.arguments))
           (empty.argument.list ())
           (subr.zero name)
           (subr.one name)
           (subr.two name)
           (subr.four name)
           (boolean name)
           (null ())
           (closure (variables body . environment))
           (cons (car . cdr))
           (type name)
           (oblist (symbol . rest))
           (empty.oblist ())
           (constant.symbol (pname . global.value))
           (assignable.symbol (pname . global.value))
```

```
           )
    (special.form.keywords quote if setq progn lambda)
    (subr.zero   end readch )
    (subr.one    car cdr consp princh implode explode symbolp eval/ce
                 reify type.of )
    (subr.two    cons rplaca rplacd eq)
    (subr.four   reflect)
    (messages (msg.ill.formed.quotation      "Ill formed quotation")
              (msg.ill.formed.continuation   "Ill formed continuation")
              (msg.ill.formed.alternative    "Ill formed alternative")
              (msg.ill.formed.function       "Ill formed abstraction")
              (msg.ill.formed.assignment     "Ill formed assignment")
              (msg.constant.assignment       "Assignment on a constant")
              (msg.memory.exhausted          "Memory exhausted")
              (msg.undefined.environment     "Unknown environment frame")
              (msg.undefined.variable        "Undefined variable")
              (msg.not.applyable             "Not applyable")
              (msg.unknown.niladic.primitive "Unknown niladic primitive")
              (msg.unknown.monadic.primitive "Unknown monadic primitive")
              (msg.unknown.dyadic.primitive  "Unknown dyadic primitive")
              (msg.unknown.tetradic.primitive  "Unknown tetradic primitive")
              (msg.incorrect.number.of.arguments "Incorrect number of arguments")
              (msg.requires.no.argument      "Requires no arguments")
              (msg.requires.one.argument     "Requires one argument")
              (msg.requires.two.argument     "Requires two arguments")
              (msg.requires.a.cons           "Must be a cons-cell")
              (msg.requires.a.symbol         "Must be a symbol")
              (msg.requires.a.function       "Must be a function")
              (msg.requires.a.character      "Must be a character")
              (msg.requires.a.character.list "Must be a character list") )
    (boot (progn
            (setq prompt.out (quote \
\=\=\ ))
            (setq prompt.in (quote \?\?\ ))
            ((lambda (quote.char dot.char space.char newline.char
                      bra.char ket.char back.slash.char peekch
                      readch eq cons consp )
               (setq quote.char (quote \'))
               (setq dot.char (quote \.))
               (setq space.char (quote \ ))
               (setq newline.char (quote \
))
               (setq bra.char (quote \())
               (setq ket.char (quote \)))
               (setq back.slash.char (quote \\))
               ;;Low-level input functions
               ((lambda (ch old.readch)
                  (setq readch (lambda ()
                                 (if ch ((lambda (cn) (setq ch F) cn)
                                         ch )
                                     (old.readch) ) ))
                  (setq peekch (lambda ()
                                 (if ch ch
```

```
                      (setq ch (old.readch)) ) )) )
 F readch )
;;the reader
((lambda (read.object read.symbol read.list
          read.end.list read.then.peekch )
   (setq read (lambda ()
                (read.object (peekch)) ))
   (setq read.object
      (lambda (ch)
        (if (eq ch space.char)
          (read.object (read.then.peekch))
          (if (eq ch newline.char)
            (read.object (read.then.peekch))
            (if (eq ch quote.char)
              (cons (quote quote)
                    (cons (read.object (read.then.peekch))
                          nil ) )
              (if (eq ch bra.char)
                (read.list (read.then.peekch))
                (if (eq ch ket.char)
                  (implode (cons (readch) nil))
                  (implode (read.symbol ch)) ) ) ) ) ) ))
   ;;This code is sensible to left to right evaluation.
   (setq read.symbol
      (lambda (ch) ;ch is only poked
        (if (eq ch space.char)
          nil
          (if (eq ch bra.char)
            nil
            (if (eq ch ket.char)
              nil
              (if (eq ch newline.char)
                nil
                (if (eq ch back.slash.char)
                  (progn (readch)
                         (cons (readch)
                               (read.symbol (peekch)) ) )
                  (cons (readch)
                        (read.symbol (peekch)) ) ) ) ) ) ) ))
   (setq read.list
      (lambda (ch)
        (if (eq ch space.char)
          (read.list (read.then.peekch))
          (if (eq ch newline.char)
            (read.list (read.then.peekch))
            (if (eq ch ket.char)
              (progn (readch) nil)
              (if (eq ch dot.char)
                (read.end.list (read.list (read.then.peekch)))
                (cons (read.object ch)
                      (read.list (peekch))) ) ) ) ) ))
   (setq read.end.list
      (lambda (exps)
```

```
          (if (consp exps)
            (if (consp (cdr exps))
              (error (quote Incorrect\ dotted\ list) exps)
              (car exps) )
            (error (quote Incorrect\ dotted\ list) exps) ) ) )
     (setq read.then.peekch
        (lambda ()
          (progn (readch) (peekch)) ))
     t )
   nil nil nil nil nil )
  ;;the printer
  ((lambda (prin.list prin.symbol prin.chars)
     (setq print (lambda (e)
                   (progn (prin e)
                          (prin newline.char)
                          e ) ))
     (setq prin (lambda (e)
                  (if (consp e)
                    (prin.list e bra.char)
                    (if (symbolp e)
                      (prin.symbol e)
                      (prin.symbol (quote \#)) ) ) ))
     (setq prin.list
           (lambda (e ch)
                   (progn (prin ch)
                          (prin (car e))
                          (if (consp (cdr e))
                            (prin.list (cdr e) space.char)
                            (if (eq (cdr e) nil)
                              (prin ket.char)
                              (progn (prin (quote \ \.\ ))
                                     (prin (cdr e))
                                     (prin ket.char) ) ) ) ) ))
     (setq prin.symbol
           (lambda (s)
             (if (eq s nil)
               (prin (quote \(\)))
               (prin.chars (explode s)) ) ))
     (setq prin.chars
           (lambda (chars)
             (if (consp chars)
               (progn (princh (car chars))
                      (prin.chars (cdr chars)) )
               nil ) )) )
   nil nil nil ) )
 nil nil nil nil nil nil nil nil readch eq cons consp )
;;reflective utilities
(setq call/cc (lambda (fn)
                (reify (lambda (e r k oblist)
                         (fn k) )) ))
(setq oblist (lambda ()
                   (reify (lambda (e r k oblist) oblist)) ))
(setq error (lambda (msg culprit)
```

```
                (prin msg)
                (print culprit)
                (return.to.toplevel (quote Return\ from\ error)) ))
        ;;the Toplevel
        (setq toplevel
              ((lambda (f)
                 (setq f (lambda (it)
                           (progn (prin prompt.out)
                                  (print it)
                                  (prin prompt.in)
                                  (f (eval/ce (read))) ) ))
                 f )
               nil ) )
        (toplevel (call/cc (lambda (k)
                             (setq return.to.toplevel k)
                             (quote Hello\ Femto) )))
        ))
)
```