The Scheme of Things: The June 1992 Meeting[1]

Jonathan Rees
Cornell University
`jar@cs.cornell.edu`

An informally constituted group of people interested in the future of the Scheme programming language met at the Xerox Palo Alto Research Center on 25 June 1992. The main purpose of the meeting was to work on the technical content of the next revision of the Scheme report.

We made progress on several fronts:

- Some differences with the IEEE Scheme standard were resolved.
- Proposals for multiple return values and `dynamic-wind` were adopted.
- A proposal for an `eval` procedure was adopted.
- The high-level macro facility described in the Revised^4 Report's appendix will be moved into the report proper.

Two subcommittees were formed: one to work on exceptions, and one to charter the formation of a standard library. The subcommittees will report back to the group with proposals for inclusion in the report.

It had been hoped that there would be progress on some other fronts (user-defined types, dynamic binding, improvements to "rest" parameters), but after inconclusive discussion these topics were dropped. These topics will probably be taken up again in the future.

Norman Adams was appointed the Revised^5 Report's editor. It is hoped that it will be ready by early 1993, so as to precede the reconstitution of the IEEE standard group.

This article is my own interpretation of what transpired, and should not be construed as definitive.

Agreement with the IEEE Scheme standard

Until now, the Scheme reports have encouraged but not required the empty list `()` and the boolean false value `#f` to be distinct. It has been the intent

[1]This article appeared in *Lisp Pointers* V(4), October–December 1992.

ever since the Revised Revised Report, however, that this distinction would eventually be required. The IEEE Scheme standard bit the bullet in 1990, and now the Revised^5 report follows.

The standard also dropped the distinction between essential and not-essential language features; most features that were formerly not essential, such as n-ary `+` and `apply`, are now required. The Revised^5 Report will adopt this stance, at least as regards language features that are shared with the IEEE standard. Non-essential non-IEEE oddities such as `transcript-on` and `transcript-off` and the proposed `interaction-environment` (see below) were not discussed at the meeting, however, and consensus on their status will have to be reached via electronic mail.

A third aspect of the standard that was adopted was a certain obscure paragraph regarding assignments to top-level variables (section 6, paragraph 2). The effect of this is that if a program contains an assignment to any top-level variable, then the program must contain a `define` for that variable; it is not sufficient that the variable be bound. This has been the case for most variables, but the rule applies as well to variables such as `car` that have built-in bindings. In addition, it is clarified that if a program makes such a definition or assignment, then the behavior of built-in procedures will not be affected. For example, redefining `length` cannot affect the behavior of the built-in `list->vector` procedure. If in some particular implementation `list->vector` is written in Scheme and calls `length`, then it must be sure to call the built-in `length` procedure, not whatever happens to be the value of the variable `length`.

Multiple return values

The `call-with-values` and `values` procedures were described in an earlier Scheme of Things (*Lisp Pointers*, volume IV, number 1), but I'll review them here. The following is adapted from John Ramsdell's concise description:

> `(values` *object* ...`)` essential procedure
>
> `values` delivers all of its arguments to its continuation.
>
> `(call-with-values` *thunk receiver*`)` essential procedure
>
> `call-with-values` calls its *thunk* argument with a continuation that, when passed some values, calls the *receiver* procedure with those values as arguments. The continuation for the call to *receiver* is the continuation of the call to `call-with-values`.

Except for continuations created by the `call-with-values` procedure, all continuations take exactly one value, as now; the effect of passing no value or more than one value to continuations that were not created by `call-with-values` is unspecified (as indeed it is unspecified now).

`values` might be defined as follows:

```
(define (values . things)
  (call-with-current-continuation
    (lambda (cont) (apply cont things))))
```

That is, the procedures supplied by `call-with-current-continuation` must be passed the same number of arguments as values expected by the continuation.

Because the behavior of a number-of-values mismatch between a continuation and its invoker is unspecified, some implementations may assign some specific meaning to such situations; for example, extra values might be ignored, or defaults might be supplied for missing values. Thus this multiple return value proposal is compatible with Common Lisp's multiple values, but strictly more conservative than it. The behavior of programs in such situations was a point of contention among the authors, which is why only the least common denominator behavior was specified.

Unwind/wind protection

`dynamic-wind`, which was described previously in this column (when it was The Scheme Environment; *Lisp Pointers*, volume I, number 2), is already implemented in many Scheme dialects. `dynamic-wind` takes three arguments, all of which are thunks (procedures of no arguments). It behaves as if it were defined with

```
(define (dynamic-wind before during after)
  (before)
  (call-with-values during
    (lambda results
      (after)
      (apply values results))))
```

except that the execution of the `during` thunk is "protected" against non-local entries and exits: a throw out of the execution of `during` will cause the `after` thunk to be invoked, and a throw from outside back in will cause the `before` thunk to be invoked. (By "throw" I mean an invocation of an explicit continuation as obtained from `call-with-current-continuation`.)

For details, the earlier Scheme Environment column refers the reader to Friedman and Haynes's paper "Constraining Control" in POPL 1985, but to save you the trouble of looking that up, I have supplied a more direct implementation of `dynamic-wind` in an appendix to the present column.

`dynamic-wind` was adopted with the following clarifications: The semantics of `(dynamic-wind` *before during after*`)` should leave unspecified what happens if a throw occurs out of *before* or *after*; and it is best to defer interrupts during *before* and *after*.

Evaluating computed expressions

The original 1975 memo on Scheme described `evaluate`, which was analogous to Lisp's traditional `eval` function. `evaluate` took a single argument, an S-expression, and invoked an interpreter on it. For example:

```
(let ((name '+)) (evaluate (list name 2 3)))
    ⟶  5
```

Scheme being lexically scoped, however, there was some confusion over which environment the expression was to be evaluated in. Should

```
(let ((name '+))
  (let ((+ *))
    (evaluate (list name 2 3))))
```

evaluate to 5 or to 6?

To clarify matters, the Revised Report replaced `evaluate` with `enclose`, which took two arguments, a `lambda`-expression and a representation of an environment from which to supply bindings of the `lambda`-expression's free variables. For example:

```
(let ((name '+))
  (let ((+ *))
    ((enclose (list 'lambda '() (list name 2 3))
              (list (cons '+ +))))))
    ⟶  6
```

This forced the programmer to be explicit about the `lambda`-expression's enclosing environment.

For various technical and practical reasons, there was no `eval` analogue in subsequent Scheme reports. The major stumbling blocks were how to describe `eval` formally and how to define something that makes sense in all extant variants of the language. Some Scheme implementations contain a distinguished top-level environment, while others extend the language by providing ways to create multiple environments, any of which might serve equally well.

The `eval` proposal adopted at the June meeting, which I reproduce here, is one that comes from Bill Rozas.

> `(eval` *expression environment-specifier*`)` essential procedure
>
> `eval` evaluates *expression* in the environment indicated by *environment-specifier*. *environment-specifier* may be the return value of one of the three procedures described below, or implementation-specific extensions. No other operations on environment specifiers are defined by this proposal.
>
> Implementations may allow non-expression programs (i.e. definitions) as the first argument to `eval` *only* when the second argument is the return value of `interaction-environment` or some implementation extension. In other words, `eval` will never create new bindings in the return value of `null-environment` or `scheme-report-environment`.
>
> `(scheme-report-environment` *version*`)` essential procedure
>
> *Version* must be an exact non-negative integer corresponding to a version of one of the Revised^n Reports on Scheme. This procedure returns a specifier for an environment that contains exactly the set of bindings specified in the corresponding report that the implementation supports. Not all versions may be available in all implementations at all times. However, an implementation that conforms to version n of the Revised^n Reports on Scheme must accept version n. If `scheme-report-environment` is available, but the specified version is not, the procedure will signal an error.
>
> The effect of assigning (through the use of `eval`) a variable bound in a `scheme-report-environment` (e.g. `car`) is unspecified. Thus the environments specified by the return values of `scheme-report-environment` may be immutable.
>
> `(null-environment)` essential procedure
>
> This procedure returns a specifier for an environment that contains no variable bindings, but contains (syntactic) bindings for all the syntactic keywords defined in the report, and no others.

`(interaction-environment)` procedure

> This procedure returns a specifier for an environment that contains implementation-defined bindings, typically a superset of those listed in the report. The intent is that this procedure will return a specifier for the environment in which the implementation would evaluate expressions dynamically typed by the user.

Rozas explains: "The proposal does not imply the existence or support of first-class environments, although it is compatible with them. The proposal only requires a way of associating tags with a finite set of distinguished environments which the implementations can maintain implicitly (without reification).

"'Pascal-like' implementations can support both `null-environment` and `scheme-report-environment` since the environments specified by the return values of these procedures need not share any bindings with the current program. A version of `eval` that supports these but not `interaction-environment` can be written portably, but can be better written by the implementor, since it can share code with the default evaluator or compiler."

Here "Pascal-like" refers to implementations that are restricted to static compilation and linking. Because an `eval` that doesn't support `interaction-environment` can be written entirely in the Scheme language described by the rest of the report, it raises no troublesome questions about its formal semantics.

Macros

The consensus of the meeting was that `define-syntax`, `syntax-rules`, `let-syntax`, and `letrec-syntax` should be moved out of the report's appendix into the main body of the report. Although everyone agrees that a low-level macro facility is important, the subject is too contentious at present, with three or more competing proposals at present. The disposition of the rest of the appendix and of the other low-level proposals will be left up to the report's editor.

Committee work

There is a strong sense that some kind of exception system is needed. However, no specific proposal was ready at the time of the meeting. A committee has been formed to work on one. What seems to be in the air might be described as a highly distilled version of the condition system that Kent

Pitman developed for Common Lisp. I hope that I'll be able to report on this in a future column.

On the subject of libraries, Will Clinger's minutes report that "the authors perceive a need to give some library official status. In fact, we need to give official sanction to multiple libraries. There is reason to distinguish between accepted (or standard) libraries, experimental libraries, and proposals. The accepted libraries can reduce the intellectual size of the language by removing things like `string->list` from the report. The experimental libraries would contain solid implementations of experimental features, including things that might never deserve to be in the report. The proposal libraries could contain anything implemented in portable Scheme."

Among the content of the accepted libraries, some features (such as those that may be moved out of the body of the report) may be required to be built in to implementations, while others will be expected to be available on demand (perhaps using something similar to, but not the same as, `require` as found in Common Lisp and GNU Emacs).

A librarian was appointed (Rees), and a library committee is developing proposals for the charter, structure, and content of the libraries.

* * *

I would like to acknowledge Will Clinger, who prepared the minutes of the meeting, and the various people who contributed proposals, including Bill Rozas and John Ramsdell. Any errors here are my responsibility, however. Thanks also to Norman Adams and Richard Kelsey for corrections to a draft of this article.

I would also like to belatedly acknowledge Norman Adams, Pavel Curtis, Bruce Donald, and Richard Kelsey for their comments on drafts of my previous column.

For future columns, I am entertaining various topic possibilities, including `eval`, threads, `amb`, and monads. If you have other ideas, and particularly if you think the written record on the language is particularly poor in certain areas, please write and let me know.

Appendix: An implementation of `dynamic-wind`

This program is based on my vague recollection of an ancient manuscript by Chris Hanson and John Lamping. I apologize for the lack of data abstraction, but the code is more concise this way.

A state space is a tree with the current state at the root. Each node other than the root is a triple $\langle$*before*, *after*, *parent*$\rangle$, represented in this implementation as two pairs `((`*before* `.` *after*`)` `.` *parent*`)`. Navigating between states requires re-rooting the tree by reversing parent-child links.

Since `dynamic-wind` interacts with `call-with-current-continuation`, this implementation must replace the usual definition of the latter.

```
(define *here* (list #f))

(define original-cwcc call-with-current-continuation)

(define (call-with-current-continuation proc)
  (let ((here *here*))
    (original-cwcc (lambda (cont)
                     (proc (lambda results
                             (reroot! here)
                             (apply cont results)))))))

(define (dynamic-wind before during after)
  (let ((here *here*))
    (reroot! (cons (cons before after) here))
    (call-with-values during
      (lambda results
        (reroot! here)
        (apply values results)))))

(define (reroot! there)
  (if (not (eq? *here* there))
      (begin (reroot! (cdr there))
             (let ((before (caar there))
                   (after (cdar there)))
               (set-car! *here* (cons after before))
               (set-cdr! *here* there)
               (set-car! there #f)
               (set-cdr! there '())
               (set! *here* there)
               (before)))))
```