The Scheme of Things:
The June 1987 Meeting

William Clinger
Tektronix Labs

This month I am pleased to feature a poem by Tigger, of the Hundred Acre Wood, as submitted by John Ramsdell.

```
Posted-Date: Mon, 6 Jul 87 07:46:39 EDT
Date: Mon, 6 Jul 87 07:46:39 EDT
From: Tigger@Hundred-Acre-Wood.Milne.Disney
To: ramsdell@linus.uucp
Subject: Scheme

    The wonderful thing about Scheme is:
    Scheme is a wonderful thing.
    Complex procedural ideas
    Are expressed via simple strings.
    Its clear semantics, and lack of pedantics,
    Help make programs run, run, RUN!
    But the most wonderful thing about Scheme is:
    Programming in it is fun,
    Programming in it is FUN!
```

* * *

The second occasional meeting of people interested in the continuing evolution of the Scheme programming language took place at the end of June on the MIT campus. This meeting was less satisfying than the remarkably successful first meeting, at Brandeis in the autumn of 1984. At Brandeis we had come prepared to agree on the core language, which is described in the *Revised^3 Report on the Algorithmic Language Scheme*. At MIT we came prepared to agree on what problems remain to be solved, but we did not bring complete solutions.

Henry Kissinger is said to have remarked that the reason academic politics are so bitterly contested is that so little is at stake. Lacking major proposals that were ready for acceptance, we spent too much time haggling over minor issues. We did accomplish a few things. Indeed, if I count an informal session Tuesday night after most participants had left following the weekend meeting, then we accomplished more than I had expected. It wasn't fun, though, prompting John Ramsdell to contribute the poem as a reminder of what we should be about.

We are generally pleased with Scheme as it stands. So far as it goes, it is pretty much the right thing. The design has been very conservative, in the sense that we have tried to avoid creeping featuritis—or *feeping creaturitis*, as the disease is known in the highly technical language of detonational simplistics—by refusing to agree on new features until a need for them has been clearly demonstrated and we think we understand the semantics of the feature and its implications for the rest of the language. This conservatism has been frustrating at times, but has kept the language healthy.

We agree that Scheme needs a few more features, particularly for large-scale programming. Scheme needs a standard macro facility, a standard module facility, a standard way to define new opaque types, and more extensive libraries. At MIT we heard a progress report on research into better macro facilities, we discussed the kind of module facilities that we would like, we heard Julian Padget explain an object system being proposed for EuLisp, and we established a standard repository for user-contributed libraries.

Alan Bawden reported on recent research toward solving the problems of Scheme macros. His report grew out of an earlier workshop that brought together the leading researchers in this area—many of them graduate students like Alan, or young faculty like Eugene Kohlbecker and Kent Dybvig. It appears that Alan and his colleagues have designed an architecture capable of solving most of the problems that have heretofore plagued Lisp macro facilities. There is much more work to be done, but the macro issue seems to be in good hands.

One of the more controversial issues dividing the Scheme community has been the use of environments as a module facility, as in *Structure and Interpretation of Computer Programs.* Environments have been controversial primarily because they have been associated with MIT's peculiar semantics for "incremental definition" and with the use of `eval` to extract values from an environment. MIT-style incremental definitions make compilers more complicated and less effective; they break the important rule that says "the names of bound variables don't matter"; and the use of `eval` to extract values from environments may mean that a compiler, or at least an interpreter, must be present at run time.

At MIT, however, we heard the implementors of MIT Scheme say that they considered the semantics of incremental definitions to be a feature of their programming environment, not a language feature that they would expect anyone else to support. They also reported their experience that the mechanisms they have been using to create environments are more powerful than necessary, and they proposed a new mechanism called `build-environment` that seems to have all the power needed in practice. The syntax of `build-environment` would be:

```
(build-environment <base-environment>
  (define <var1> ...)
  (define <var2> ...)
  ...
)
```

This expression would evaluate to an environment that contains all the bindings in the given base environment plus bindings for the variables defined in the body of the `build-environment` expression. If a variable is defined in both the body and in the base environment, then the definition in the body would shadow the binding in the base environment. Though we did not formalize the semantics of `build-environment`, I assume that the environment returned by the `build-environment` expression would not contain bindings for variables in the lexical environment in which the `build-environment` expression is evaluated unless they happen to be in the base environment as well. I would also guess that the right hand sides of the definitions are evaluated in the lexical environment of the `build-environment` expression, augmented by the definitions in its body, rather than in the environment returned by the `build-environment` expression. Clearly we do not yet have enough experience with this new proposal to adopt it, but we should have plenty of time to try it out before our next meeting.

If, as in MIT Scheme, a special-purpose syntax like `access` is used instead of `eval` to fetch values from environments, then `build-environment` answers most of the objections that have been raised to environments in Scheme. That is not to say that environments are the best mechanism possible, but they certainly form one of the best module facilities that has ever been used in a Lisp-like language.

A library of user-contributed portable code has been established, with Bill Rozas as first librarian and referee. Contributions should be sent to scheme-librarian@mc.lcs.mit.edu When you submit code, please specify it well, and note the implementations you have used to test it. Rozas has indicated that code might not be accepted into the library unless it has been shown to run in at least two different implementations of Scheme.

On Tuesday night, we agreed on a syntax and a family of about five possible semantics for multiple return values. The possibilities are linearly ordered by upward compatibility, so each will be a compatible extension or subset of whichever semantics we eventually choose.

Participants workings in private sessions also developed a proposal for minor changes in the syntax of numbers, improved the description of exact and inexact numbers, and agreed on the concept of a proposal to rewrite the section on the equivalence predicates eq?, eqv?, and equal?.

* * *

In a previous article on continuations I left you to puzzle over Eugene Kohlbecker's classic mind-bender: What does the following program print?

```
(define (mondo-bizarro)
  (let ((k (call-with-current-continuation (lambda (c) c))))
    (write 1)
    (call-with-current-continuation (lambda (c) (k c)))
    (write 2)
    (call-with-current-continuation (lambda (c) (k c)))
    (write 3)))
```

Several of you have complained that I owe you an explanation of this contorted, nay twisted, example. Remember, you asked for it.

When mondo-bizarro is called, k is bound to a continuation κ_1 that will accept a value and bind k to it before continuing with the body of the let. The program then prints 1. Then κ_1 is called with a continuation κ_2 that will accept a value, throw it away, and continue the program at the expression (write 2). Since κ_1 is called, we get a new binding for k, to κ_2, and we start back at the body of the let. Hence the program prints another 1. Then κ_2 is called with a continuation κ_3 that will accept a value, throw it away, and continue the program at the expression (write 2). Obviously κ_3 is very much like κ_2, but there is one major difference: within the continuation κ_2, k is bound to κ_1, but within κ_3 k is bound to κ_2. This doesn't matter, though, because κ_2 just ignores its argument κ_3.

By calling κ_2 we continue at the expression (write 2) with k bound to κ_1. Hence we print a 2, so the output thus far is 112. Then we call κ_1 with a continuation κ_4 that will ignore its argument and continue at the expression (write 3) with k bound to κ_1. By calling κ_1 with argument κ_4, we bind k to κ_4 and start over with the body of the let.

This is beginning to look circular. But we dutifully print a 1, and then call κ_4 with a continuation κ_5 that will accept a value, throw it away, and continue the program at the expression (write 2) with k bound to κ_4. Since κ_4 ignores its argument, however, the fate of κ_5 is the same as that of κ_3. By calling κ_4, we execute (write 3) and then return from mondo-bizarro. The complete output is therefore 11213.

My computer agrees, so this explanation must be correct.