



## The First Report on Scheme Revisited

GERALD JAY SUSSMAN

*Massachusetts Institute of Technology, 545 Tech Square, Room 428, Cambridge, MA 02139, USA*

[gjs@mit.edu](mailto:gjs@mit.edu)

GUY L. STEELE JR.

*Sun Microsystems Labs, 2 Elizabeth Drive, MS UCHL03-207, Chelmsford, MA 01824, USA*

[guy.steele@east.sun.com](mailto:guy.steele@east.sun.com)

Designs and standards tend to go in cycles. After a specific new design or standard becomes established, it is incrementally “improved”—usually by the accretion of features, until it becomes unwieldy. Then it is time to step back and reassess the situation.

Sometimes a judicious designer can sort through the accumulated set of ideas, discard the less important ones, and produce a new design that is small and clean. Pascal and Modula-2 were produced in this way.

Scheme wasn’t like that at all. We were actually trying to build something complicated and discovered, serendipitously, that we had accidentally designed something that met all our goals but was much simpler than we had intended.

We thought that Scheme would turn out to be the next in a long series of programming languages that had been designed at MIT over the course of 17 years. The principal themes of this series were complex data structures with automatic pattern matching, and complex control structures with automatic backtracking. These languages were specifically envisioned as tools to support explorations into theorem proving, linguistics, and artificial intelligence, in much the same way that Fortran was intended to support numerical computation or COBOL to support business applications.

There was Lisp, of course, designed in 1958 by John McCarthy. Less well-known today is Victor Yngve’s COMIT (1962), a pattern-matching language intended for use in linguistic analysis. A COMIT program repeatedly matched a set of rules against the contents of a flat, linear workspace of symbolic tokens; it was a precursor of SNOBOL and an ancestor of such rule-based languages as OPS5. In 1964, Daniel Bobrow implemented a version of COMIT in Lisp and (punningly) called the result METEOR. This in turn inspired Adolfo Guzman in 1969 to design CONVERT, which merged the pattern-matching features of COMIT with the recursive data structures of Lisp, allowing the matching of recursively defined patterns to arbitrary Lisp data structures.

In 1969, Carl Hewitt designed an extremely ambitious Lisp-like language for theorem-proving called Planner, based on the use of pattern-directed procedure invocation and the use of automatic backtracking as an implementation mechanism for goal-directed search. It was never completely implemented as originally envisioned, but it spurred three other important developments: Muddle, Micro-Planner, and Conniver.

The language Muddle (later MDL) was an extended version of Lisp designed in 1970 by Sussman, Carl Hewitt, Chris Reeve, and David Cressey (later work was done by Bruce Daniels, Greg Pfister, and Stu Galley). One of its goals was to serve as a base for a full-blown implementation of Planner. (This in itself was an interesting development: Planner, which was intended to support AI applications too difficult to program “directly” in Lisp, was itself so complex that it was thought to require a special implementation language.)

Muddle introduced the lambda-list syntax markers `OPTIONAL`, `REST`, and `AUX` that were later adopted by `Conniver`, `Lisp Machine Lisp`, and `Common Lisp`.

In 1971, Sussman, Drew McDermott, and Eugene Charniak implemented a subset of `Planner` called `Micro-Planner`. The semantics of the language were not completely formalized and the implementation did not work correctly in certain complicated cases; for example, the matcher was designed to match two patterns, each of which might contain variables, but did not use a complete unification algorithm. (Much later, Sussman, on learning about `Prolog`, remarked to Steele that `Prolog` appeared to be the first correct implementation of `Micro-Planner`.) `Micro-Planner` was successfully used for a number of AI projects, notably Terry Winograd's `SHRDLU` system.

The language `Conniver` was designed in 1972 by Drew McDermott and Sussman in reaction to certain limitations of `Micro-Planner`. In their paper *Why Conniving Is Better Than Planning*, they argued that automatic nested backtracking was merely an overly complicated way to express a set of `FORALL` loops used to perform exhaustive search:

It is our contention that the backtrack control structure that is the backbone of `Planner` is more of a hindrance in the solution of problems than a help. In particular, automatic backtracking encourages inefficient algorithms, conceals what is happening from the user, and misleads him with primitives having powerful names whose power is only superficial.

The design of `Conniver` put the flow of control very explicitly in the hands of the programmer. The model was an extreme generalization of coroutines; there was only one active locus of control, but arbitrarily many logical threads and primitives for explicitly transferring the active locus from one to another. This design was strongly influenced by the “spaghetti stack” model of Daniel Bobrow and Ben Wegbreit, which provided separate notions of a data environment and a control environment and the possibility of creating closures over either. The implementation of spaghetti stacks addressed efficiency issues by allowing stack-like allocation and deallocation behavior wherever possible. This required a certain amount of low-level coding. In contrast, `Conniver` was implemented in `Lisp`; control and data environments were represented as heap-allocated `Lisp` list structures, relying on the `Lisp` garbage collector to handle reclamation of abandoned environments.

At about this time, Carl Hewitt was developing his theory of *actors* as a model of computation. This model was object-oriented and strongly influenced by `Smalltalk`. Every object was a computationally active entity capable of receiving and reacting to messages. The objects were called actors, and the messages themselves were also actors. Every computational entity was an actor and message-passing was the only means of interaction. An actor could have arbitrarily many *acquaintances*; that is, it could “know about” other actors and send them messages or send acquaintances as (parts of) messages. Hewitt then went on to model many traditional control structures as patterns of message-passing among actors. Functional interactions were modeled with the use of continuations; one might send the actor named “factorial” a message containing two other actors: the number 5 and another actor to which to send the eventually computed value (presumably 120). While `Conniver` made control points into first-class data, the actors model went to the logical extreme by making *everything* be first-class data.

Hewitt and his students developed and implemented in Lisp a new language to make concrete the actor model of computation. This language was first called Planner-73 but the name was later changed to PLASMA (PLanner-like System Modeled on Actors).

It was at this point that we (Sussman and Steele) put our heads together. (Steele had just become a graduate student at MIT, but he had been following this progression of language designs because he had had a part-time job at MIT Project MAC as one of the maintainers of MacLisp.) We wanted to better understand Hewitt's actors model but were having trouble relating the actors model and its unusual terminology to familiar programming notions. We decided to construct a toy implementation of an actor language so that we could play with it. Using MacLisp as a working environment, we wrote a tiny Lisp interpreter and then added mechanisms for creating actors and sending messages.

Sussman had just been studying Algol. He suggested starting with a lexically scoped dialect of Lisp, because that seemed necessary to model the way names could refer to acquaintances in PLASMA. Lexical scoping would allow actors and functions to be created by almost identical mechanisms. Evaluating a form beginning with the word `lambda` would capture the current variable-lookup environment and create a closure; evaluating a form beginning with the word `alpha` would also capture the current environment but create an actor. Message passing could be expressed syntactically in the same way as function invocation. The difference between an actor and a function would be detected in the part of the interpreter traditionally known as `apply`. A function would return a value, but an actor would never return; instead, it would typically invoke a *continuation*, another actor that it knew about. Our interpreter also provided the necessary primitives for implementing the internal behavior of primitive actors, such as an addition operator that could accept two numbers and a continuation actor.

We were very pleased with this toy actor implementation and named it “Schemer” because we thought it might become another AI language in the tradition of Planner and Conniver. However, the ITS operating system had a 6-character limitation on file names and so the name was truncated to simply SCHEME and that name stuck. (Yes, the names “Planner” and “Conniver” also have more than six characters. Under ITS, their names were abbreviated to PLNR and CNVR. We can no longer remember why we chose SCHEME rather than SCHMR—maybe it just looked nicer.)

Then came a crucial discovery. Once we got the interpreter working correctly and had played with it for a while, writing small actors programs, we were astonished to discover that the program fragments in `apply` that implemented function application and actor invocation were identical! Further inspection of other parts of the interpreter, such as the code for creating functions and actors, confirmed this insight: the fact that functions were intended to return values and actors were not made no difference anywhere in their implementation. The difference lay purely in the primitives used in their bodies. If the underlying primitives all returned values, then the user could (and must) write functions that return values; if all primitives expected continuations, then the user could (and must) write actors. Our interpreter provided both kinds of primitives, so it was possible to mix the two styles, which was our original objective. But the `lambda` and `alpha` mechanisms were themselves absolutely identical. We concluded that actors and closures were effectively the same concept. (Hewitt later agreed with this, but noted that two types of primitive actors in his theory, namely cells (which have modifiable state) and synchronizers (which enforce

exclusive access), cannot be expressed as closures in a lexically scoped pure Lisp without adding equivalent primitive extensions.)

That is how Scheme got started. This led us to three important ideas:

- First, we realized that all the patterns of control structure that Hewitt had described in terms of actors could equally well be described by the  $\lambda$ -calculus. This was no surprise to theoreticians, of course, especially those working in the area of denotational semantics, but Scheme became the vehicle by which those theoretical concepts became much more accessible to the more practical side of the programming language community.
- Second, we realized that the  $\lambda$ -calculus—a small, simple formalism—could serve as the core of a powerful and expressive programming language. (Lisp had adopted the  $\lambda$ -notation for functions but had failed to support the appropriate behavior for free variables. The original theoretical core of Lisp was recursion equations, not the  $\lambda$ -calculus.) An important consequence of the existence of an interpreter for the  $\lambda$ -calculus was that suddenly a denotational semantics could be regarded as an operational semantics. (Some styles of denotational specifications require a call-by-name interpreter rather than the call-by-value evaluation order used by Scheme. We went on to study this distinction carefully and settled on call-by-value as the “official” evaluation order for Scheme only after a fair amount of deliberation and debate.)
- Third, we realized that in our quest for the “ultimate AI language” we had come full circle. As the MIT school had struggled to find more and more powerful ways to express and manipulate control structure to support heuristic search, we had progressed from Lisp to CONVERT to Planner to Conniver to PLASMA to a simple variation of Lisp!

In 1976, we wrote two more papers that explored programming language semantics using Scheme as a framework. *Lambda: The Ultimate Imperative* drew on earlier work by Peter Landin, John Reynolds, and others to demonstrate how a wide variety of control structures could be modeled in Scheme. *Lambda: The Ultimate Declarative* discussed  $\lambda$  as a renaming construct and related object-oriented programming generally and actors specifically to closures. These ideas in turn suggested a set of techniques for constructing a practical compiler for Scheme. Steele then wrote the first Scheme compiler as part of the work for his master’s degree.

Much of the content of these papers was not new; their main contribution was to bridge the gap between theory and practice. Scheme provided a vehicle for making certain theoretical contributions in such areas as denotational semantics much more accessible to Lisp hackers; it also provided a usable operational platform for experimentation by theoreticians.

In this process we learned some great lessons. The  $\lambda$ -calculus can be seen as a universal glue by which compound constructs can be built inductively from simpler ones and a set of primitives. In particular, if our primitives are functions, then our constructs are functions; if our primitives are relations, then our constructs are relations; if our primitives are actors, then our constructs are actors. The essence of the matter is that the  $\lambda$ -calculus provides a way of abstracting entities of any particular type and then combining such abstractions with other entities to make new entities of that same type, which are instances of the abstractions. This became apparent to us in, for example, the development of constraint languages. A  $\lambda$ -expression (or its “value”) may be a function (if its application yields a value), but it is

more general and more fruitful to regard it as an abstraction. Similarly, a combination is not always a “function call”; it is more general simply to speak of instantiating an abstraction. Given the right primitives, the  $\lambda$ -calculus can serve as a foundation for the abstraction and instantiation of virtually any kind of complex conceptual structure.

We also were struck by the great importance of *names* as a means of reference. Other notational systems can serve as universal glue, such as combinatory logic or the variant of the  $\lambda$ -calculus that uses de Bruijn numbers instead of names. These other systems can be used as the basis for programming languages, and indeed are better suited in some ways for machine execution; but, despite the fact that they are computationally equivalent to the  $\lambda$ -calculus in a fairly direct way, as notations they seem to be harder for people to read (and to write). Naming seems to correspond to some important cognitive mechanism that is, if not innate, then at least extremely well-developed in our culture. The  $\lambda$ -calculus embodies a provably coherent theory of naming.

In retrospect, we can also see that some aspects of the initial design of Scheme were flat-out wrong, most notably the approach to multiprocessing. Synchronization and mutual exclusion are matters of time, not of space, and are not properly addressed by lexical scoping, which governs textual structures rather than the dynamics of execution. We also now believe that Carl Hewitt was right: we would have been better off to have introduced cells as a separate, primitive kind of object, rather than allowing assignment to any and every  $\lambda$ -bound variable. (Worse yet, the assignment primitive was `ASET`, not `ASETQ`, which exposed the mapping from Lisp symbols to variables in our implementation—a decision we were later to regret.)

In 1978, we wrote a revised report on Scheme. Since then, to our amazement, Scheme caught on and spread. Scheme’s small size, roots in the  $\lambda$ -calculus, and generally compact semantic underpinnings began to make it popular as a vehicle for research and teaching. Because it was so small and simple, Scheme could be put up on top of some other Lisp system in a very short time. Local implementations and dialects sprang up at many other sites. In particular, strong groups of Scheme implementors and users developed early on at MIT, Indiana University, and Rice University.

At MIT, under the guidance of Sussman and Hal Abelson, Scheme was adopted for teaching undergraduate computing. The book *Structure and Interpretation of Computer Programs*, which was developed for this undergraduate course, has been in print since 1984, and is still used at many universities (now in a revised edition).

Once Scheme came into wide use, it began to accrete “improvements”; we are happy to say that most of these are in fact improvements. Scheme has served as a sturdy basis for experimentation in language design. Examples of this are a numerical system that distinguishes between exact and inexact numbers while also supporting arbitrary-precision integer and rational arithmetic, and the concept of hygienic macros, which can perform textual substitutions within programs while keeping possible name conflicts under control.

In 1991, Scheme became an IEEE and ANSI standard programming language. But we knew long before that that Scheme had found its place in the world.

We knew Scheme had made it when other researchers began to use it as an expository vehicle in their papers, published in such conferences as ACM POPL and ACM PLDI. We knew Scheme had *really* made it when researchers no longer cited our papers, but simply took Scheme for granted as part of the communication infrastructure for programming

language research. The most gratifying thing to us about Scheme is that it no longer belongs to us. We are happy to have made a discovery that many other people have found useful.