Scheme: The Next Generation

John D. Ramsdell*
The MITRE Corporation

Scheme has served the community for twenty years. It has demonstrated that a useful language can be constructed which has a very small number of rules for forming expressions. Scheme's set of rules were carefully chosen so as to produce a language flexible enough to support most major programming paradigms in use today, yet allow efficient implementations.

Scheme has evolved modestly over the years, but the computing world has not. Since the introduction of Scheme, many new programming languages have emerged with innovative ideas. Parallel computers have become common place. Finally, the extensive use of Scheme has taught both users and implementors much about the current definition of Scheme and how to improve it.

I believe now is the time to design a new dialect of Scheme for the next generation. This paper describes a set of changes and additions that were carefully chosen so as to retain the flavor of Scheme. I hope this paper marks the beginning of a community effort to design a next generation Scheme dialect.

Imitation is the most sincere form of flattery, and I believe the Scheme community should flatter the ML community. One practice we should copy from ML [5] is the pervasive use of immutable data structures. Most data structures created by Scheme programs are not modified. Programmers should be allowed to write code which creates and shares data with other modules, while being assured that no other module modifies that data. Implementations should be able to take advantage of the knowledge that some data structures are immutable.

In practice, this means that the procedure `cons` should return an immutable pair and `set-car!` and `set-cdr!` should be eliminated. Vectors should remain mutable, but an immutable vector creating procedure should be added. As in ML, the application of a `lambda` expression should bind variables to values, and not locations that contain values. The effect of mutable `lambda` bound variables should be provided by adding the three procedures `make-cell`, `cell-ref`, and `cell-set!`, which creates an initialized mutable location, retrieves the value in cell, and modifies the value in the cell.

Another practice we should copy from the ML community is to make the use of formal methods an integral part of the design process for the next generation Scheme dialect. While Scheme has a formal semantics, it has the markings of an afterthought. For example, though printed with the document, it is not a part of IEEE Std 1178-1990, IEEE Standard for the Scheme Programming Language [3]. Furthermore, optimizing Scheme compilers often perform program transformations that are difficult, if not impossible, to justify with the semantics.

The Vlisp Project [2] carefully studied and used the Scheme semantics. That work strongly motivates the following changes to `letrec` and equality testing.

`Letrec` should be restricted so as to bind variables only to `lambda` expressions. The semantics should include `letrec` as an expression constructor in the abstract syntax, and its meaning should be defined recursively, i.e. with the use of a fixed point operator.

Equality testing should provide implementors with more freedom. When `eqv?` is given two procedures as arguments, the result should be a boolean value, but implementations should be allowed to return either value at its discretion. With this change, compiler writers and the formal semantics would no longer need to label procedures to satisfy the needs of equal-

*This paper reflects the views of the author and makes no statement about the views of The MITRE Corporation. Author's address: The MITRE Corporation MS/K320, 202 Burlington Road, Bedford MA, 01730-0208. E-mail: ramsdell@mitre.org.

ity testing.

In combination with the change to immutable **lambda** bound variables, the above two changes will bring the semantics in line with the intuition displayed by compiler writers. **Lambda** expressions and the portion of an environment extended using `letrec` no longer depend on the store, which is the essence of the intuition.

Let me briefly list several other issues that should be addressed before I conclude with a proposal for parallel programming.

- Programmers should be able to rely on the order in which the elements of a call are evaluated. I believe the arguments should be evaluated left-to-right, followed by the operator. This change also makes the formal semantics more useful because it reduces the number of answers associated with each program.
- Support should be provided for the "little modules" approach to programming, in which the programming language provides facilities for partitioning code into many small independent parts. The module system included in Scheme 48 [4] provides a good starting point.
- The language should facilitate optional type checking, which might motivate the addition of a pattern matching conditional [6].

In order to support parallel processing on both distributed and shared memory machines, I believe the Scheme community should flatter the Erlang community. Parallel activity in Erlang [1] is synchronized by the use of message passing. All data structures are immutable.

Elimination of all mutable data structures would not be in the spirit of Scheme, but limiting mutations to one thread of control would. In a next generation Scheme dialect, parallel activity should be initiated by the use of a **spawn** procedure which creates a thread of control for the procedure given as an argument in a copy of the parent's store. Copying the parent's store ensures only one thread of control can access or modify each mutable data structure.

As in Erlang, parallel activity should be coordinated using asynchronous buffered message passing. Each message should be copied into the store of the receiver so as to prevent access to mutable data from more than one thread of control. Parallel algorithms which require a shared mutable database, would associate one thread of control with the database and use message passing to mediate database interactions from other threads.

Implementations on shared memory machines need only copy data structures which contain mutable data. Part of the motivation for making the use of immutable data structures pervasive is to reduce the amount of copying required by an implementation.

In conclusion, the ideas assembled retain the flavor of Scheme while incorporating innovative ideas from other programming languages. I hope these ideas will inspire an ambitious effort to design a new dialect of Scheme for the next century. It's time to embark on a new voyage in language design. Engage!

References

[1] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.

[2] Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.

[3] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[4] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[5] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.

[6] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. In *1994 ACM Conference on Lisp and Functional Programming*, volume 7 of *LISP Pointers*, pages 250–262, 1994.