Scheme 後編

過去 現在 未来

Guy L. Steele Jr. 訳 井田 昌之

訳者より

前回は，Carl Hewitt と Gerry Sussman の間にあって Guy Steele がどのような役割を演じてきたか，それをどのように自分で感じていたかが述べられた．Carl Hewitt の設計した Plasma の仕組みを知るために簡単な処理系を作り，それを，Planner そして Conniver の次という意味で Schemer と呼んだこと，それが，当時のコンピュータの制限から 6 文字に切られ，Scheme という名前になったいきさつなどが語られた．今回の部分では，アルファ式として導入したもののクロージャとの関係，アクタとの関係などにまず触れられ，次にラムダ式を用いたコンパイラの最適化のはなし，そして，Scheme の言語仕様の変化に対する興味深い洞察が話されている．

前回と同様，登場する人物の氏名には一切敬称を略した．また，読者の便宜のためにできるだけ多くの参考文献を付した．

クロージャとアクタ

しかし，設計とネーミングの二つの偶然だけでなく，もう一つの偶然もありました．それはおもしろいものでした．私達は，関数とアクタをサポートする Lisp インタプリタを作りました．引数をつけて関数を呼び出したり，メッセージをアクタへ送ることができます．そしてそのインタプリタにアクタが実装された仕方を見て，私達は，たいへん驚きました．ラムダクロージャの実装とアルファアクタの実装は，同一のコードだったのです．

「すごいですねぇ．」

私達はアクタと関数クロージャは同一のものだと決断しました．これはたいへん有益な結論です．というのは，このことから，アクタはラムダカリキュラスで説明できるということを意味すると気がついたのです．これは，実際は偶然ではありませんでした．しかし，偶然のように感じました．Lisp の中のレキシカルスコーピングはそれと同一のものです．Sussman と私はレキシカルスコープの Lisp は，ラムダカリキュラスと同一であるということを発見しました．

それは，1975 年のことです．すぐにラムダカリキュラスの論文を読み出しました．特に，私達は Alonzo Church の 1941 年の論文† に戻りました．彼の論文の中では，アクタとしてのコンスセルのようなペアのシミュレーションが論じられていることに気がつきます．彼がそれをメッセージ送信になるべきだと理解していたとは思いませんが，1941 年の論文でコンスセルのような何かがでているのを見つけるのはたいへん興味深いことです．

それから 4 年後，1979 年に私は結婚しました．妻と私は二人とも MIT の学生でした．もう少しお話しすると，私達は Sussman の研究室にいました．結婚して 6 か月後に妻は，彼女の母から，親戚のことがたくさん書かれている手紙を受け取りました．彼女は祖

† Church, Alonzo: The Calculi of Lambda Conversion, Anunals of Mathematics Studies 6, Princeton University Press, 1941.

父の Herbert Taylor と叔父の Sam Church について話してくれました．また，「いとこの Alonzo からの手紙をもらった．彼はプリンストンにいました．」と書かれていました．それを彼女は私に見せてくれました．

私は言いました．「バーバラ，プリンストンにいて，引退した Alonzo Church という人は何人いると思う？」彼女は，「たった一人よ．」「なんてこった！」Alonzo Church は，彼女のいとこだったのです．まったく世界は狭いですね．

Sussman と私は関数とアクタは同一だと認めました．どんなことになるのでしょう？　関数は値を返すものと考えられています．アクタはそうではありません．かわりにアクタはメッセージを送ります．この質問に対する正しい答えは「それがどうした．」「なんでも，いいじゃないか．」です．

引数を受け取り，何かをするからということで，何かがアクタであるか関数であるかを，識別することはできないのです．

アクタ，あるいはクロージャがどんなことをできるか考えてみましょう．いくつかのことがあります．まず，他のアクタやクロージャを呼び出せます．再帰的に行なってもかまいません．たぶん条件テストもできるでしょう．これも再帰的な場合もあります．then 部あるいは else 部が何か別のことをしてもいいです．あるいは，言語に用意されたプリミティブを呼び出すこともできます．帰納の基礎となるケースです．しかし Sussman と私は偶然，もし言語中のプリミティブが値を返すなら，すべてのアクタあるいは labels クロージャも値を返すことを発見しました．もしすべてのプリミティブがメッセージを送るなら，すべてのアクタやクロージャもメッセージを送るのです．

だから，アクタやクロージャとして振る舞うものというのは，用いているプリミティブに依存するのです．したがって，言語が純粋に関数なのか純粋にアクタなのかはそのプリミティブに依存します．

Initial Report on Scheme とそれに続く論文

Sussman と私は Scheme に関する最初のレポート[†1]を書きました．1975 年 12 月のことです．lambda という関数コンストラクタをおきました．見てきたように，それはアクタコンストラクタでもあります．labels という不動点オペレータも加えました．これは Lisp 1.5 の label のようなものです．モダンなプログラミング言語では letrec と呼ばれています．条件オペレータとして if を加えました．if のかわりに cond を置けたかもしれませんが，そのほうがシンプルだと思ったのです．

setq に似た aset と呼ぶ副作用オペレータを置きました．それらに加えて，catch と呼ぶ暗黙の関数コンティニュエーションを得る方法を加えました．catch は特殊形式で，最近の Scheme では callcc と普通呼んでいるものです．関数の適用は，関数に引数を与えます．そして，変数を参照することももちろんできます．リスト，数，などの基本的なデータタイプがあります．それが最初の Scheme でした．いろいろなことにはできるだけ一つのことを入れるだけですませようとしました．開発用というのではなく，教育のために，この言語をたいへん小さいものに保とうとしてきたからです．

Sussman と私は，次にこの小さな言語を使って制御構造全体というものを，もう一度説明しようとしました．Carl Hewitt は，アクタに関する論文を書きました．“Viewing Control Structures as Patterns of Passing Messages”[†2]です．1974 年か 75 年のことです．Hewitt の論文は，繰返し，再帰，ループ，そしてあらゆる種類の複雑な制御構造をアクタとメッセージによって説明しました．Sussman と私は，Scheme を使って，Carl Hewitt のやっていることをシンプルな方法で説明しようとしました．

それで，“Lambda : the Ultimate Imperative”[†3]という論文を書きました．そこでは，制御構造をラムダカリキュラスを使って説明しようとしたのです．それは新しいアイディアではありませんでした．Peter Landin と John Reynolds は，Algol をラムダカリキ

†1 Sussman, Gerald Jay and Steele, Guy Lewis Jr. : SCHEME : An Interpreter for Extended Lambda Calculus, AI Memo 349 MIT AI Lab., December 1975.

†2 Hewitt, Carl E. : Viewing Control Structures as Patterns of Passing Messages, *Artificial Intelligence*, Vol. 8, No. 3, pp. 323-364, 1977.

†3 Steele, Guy Lewis Jr. and Sussman, Gerald Jay : LAMBDA : The Ultimate Imperative, AI Memo 353 MIT AI Lab., March 1976.

ュラスで説明しようとしました[†1]. denotational semantics[†2]という理論領域がそのころ現われてきました. 事実, Sussman と私はこれらのアイディアを自分達のフレームワークで繰り返しました.

私達の新しい貢献だというのは, ラムダカリキュラスでの制御構造のモデルは実行できるということでした. ラムダカリキュラスの理論モデルをすべて Lisp 構文則に翻訳し, それを走らせました. それで, MIT のほかの人たちはたいへん便利だと言ってくれました. 次の論文 "Lambda : The Ultimate Declarative"[†3]では, ラムダの宣言的サイドを見ていました.

この論文では, 2点を強調しました. まず, ラムダの主な目的は, メッセージに到着する引数に名を与えることです. もう一つは, 関数の呼出しは必ずしも値を返すわけではないことです. もし, 関数呼出しが値を返さなくてもよいとすると, 帰る必要もないのです. このことを, 関数はアクタと同じであり, 呼出しはメッセージと同じなので, 理解できます.

これは, 関数呼出しを, 引数を渡せる goto のようなものとしてコンパイルできることを教えてくれます. 特に, 関数呼出しは, 戻り番地をスタックに積む必要がなくなるのです. 戻りたいときにだけ積めばよい. あるいはこのことを別の考え方でみると, 戻り番地はまさにコンティニュエーションです. もし, 新しいコンティニュエーションを作りたくなければ, 戻り番地をプッシュする必要はないのです.

この原則に基づいてコンパイラを設計するなら, 末尾再帰は自動的に処理できるようになります. そしてそのコンパイラは末尾再帰の正しい処理を保証します. そして, Lisp でのオブジェクト指向プログラミングができるようになるのです.

著者 (左) と訳者 (右)

これらのアイディアに従って, 私は自分の修士論文[†4]の一部として, Scheme のコンパイラを実際に作りました. その中で, 私は今まで話してきたようなコードコンパイルのアイディアについてまとめました. また, ラムダカリキュラスを使って最適化コンパイルをする方法をまとめました. そのころ, 70年代では, ソースレベルのプログラム変換はなおも新しいアイディアでした. 私は denotational semantics として説明されるプログラム変換は, コンパイラ中で直接使えることを認識しました.

そのころ, 私は「その言語でコンパイラが書けなければ, いい言語ではない」という言葉に強く影響を受けていました. だから, 私は Scheme でコンパイラを書きました. ブートストラッピングでの別の問題を発見しました. 自分自身コンパイルするようにするには, インタプリタを走らせなければなりませんでした. さらに悪いことに, 改良をどんどんすると自分自身をコンパイルするコンパイル時間はどんどん長くなります. そこで, 家に端末を持って帰って毎晩コンパイルしました. 朝起きるといつもできていました. 3か月で100回くらいコンパイルしたことになります. コンパイラをうまく作れると, それを使ってそのコンパイラをコンパイルして, もっと速く実行できるようにします.

けれども, 何回かはコンパイラにバグを入れてしま

---

†1 Landin, Peter : A correspondence between Algol 60 and Church's lambda notation : Part I, *CACM*, Vol. 8, No. 2, pp. 89-101, 1965.
Reynolds, John : Definitional Interpreters for Higher Order Programming Languages, Proc. ACM National conference, pp. 717-740, ACM, 1972.

†2 Stoy, Joseph E. : *Denotational Semantics : The Scott-Strachey Approach to Programming Theory*, MIT Press, 1977.

†3 Steele, Guy Lewis Jr. : LAMBDA : The Ultimate Declarative, AI Memo 379 MIT AI Lab., November 1976.

†4 Steele, Guy Lewis Jr. : Compiler Optimization Based on Viewing LAMBDA as Rename plus Goto, S. M. thesis, MIT, May 1977.
Published as "RABBIT : A Compiler for SCHEME (A Study in Compiler Optimization)", AI TR 474 MIT AI Lab., May 1978.

ったこともありました．その場合，インタプリタを使ってブートストラップし直しました．これはとてもあきあきするような作業でした．

ラムダカリキュラスに基づくコンパイラ最適化の二，三の例を示しましょう．

ラムダカリキュラスによるコンパイラ最適化

Example : OR

```
(or x y)

    (if x x y)

            ((lambda (p)
                (if p p y))
              x)
                        ((lambda (p q)
                           (if p p (q)))
                         x
                         (lambda () y))
```

最初の例は or です．(or x y)．この意味はLisp の場合と同じで，もし x が真なら真を返し，y の評価はしない．しかも，もし x が偽ならば y を評価し，その値を返す．そこで，次のような変形もありえます；(if x x y)．しかし，これは，マクロとしてはやりたくありません．特に x が複雑な式の場合に．またそれは副作用があるかもしれません．2回実行はしたくありません．

したがって，やりたいことは x を1回だけ評価し，その値を見て，もしそれが真ならそれを返し，再び，実行しないようにさせます．多くのコンパイラはこのアイディアをすでにもっています．C コンパイラもすでにもっています．しかしそのころまでは，どんなコンパイラもソースからソースへの変換で扱ってはいませんでした．Rabbit では，この拡張を試みました．x を評価し，その値を変数 p に束縛し，それをテストします．ここのソースコードを見てください．x の評価は1回だけ行ないます．もう一つ別の問題があります．変数 p が y の中で使われていたとしたらどうしますか？ y の中では誤った p を参照してしまいます．

そのころの普通のトリックは gensym で変数名を

作ることです．しかし，私達はこの仕掛けでは満足できませんでした．そうしたマジックは言語の外にあるべきです．私達はラムダ変数を使って，完全にその言語の範囲内でソース変換をしたかったのです．

私達は，もっとラムダ式を使うことでこれが可能なことを発見しました．このラムダ式では，変数 x を変数 p に束縛し，変数 q を別のラムダ式に束縛しそしてそれが y を呼び出し，実行します．

Algol の分野では，このラムダ式は thunk と呼ばれます．さて，thunk は英語では変な（silly）名前です[†1]．

この意味は，このラムダ式（外側のもの）は単に二つのものを名付けます．変数 x とその thunk を名付けます．ここで名前 p と q を用います．もし p が真なら p，そうでないなら q を呼びます．この関数呼出しを，正しいスコープにある thunk への goto に実際に変更することを考えることができます．

Example : IF

```
(if (if a b c) d e)

      (if a (if b d e)
            (if c d e))

            ((lambda (x y)
               (if a (if b (x) (y))
                     (if c (x) (y))))
             (lambda () d)
             (lambda () e))
```

二つ目の例として，この if を考えてみましょう．入れ子になった if があるとします．ここで示すようなものへのソース変換がよいことが最近発見されています．

私はこの変換をカリフォルニア大学アーバイン校のカタログ[†2]の中で発見しました．アーバインのプログラム変換カタログは，まずソースレベルでの変換として変換をシステマティックに並べようとした最初のものでした．

この変換は必ずしも十分なものではありません．というのは，d と e のためのコードが重複しているからです．この考え方によると d と e に対して，ただ一つのコピーだけをもてばいいのです．そしてこの意思決定木を下がっていって正しいコードの断片に到達します．Scheme を使って Rabbit コンパイラは次のように表現します．x と y が d と e に対する二つの thunk を名付けます．それで d のコードは 1 回だけ現われます．e のコードも 1 回だけ現われます．その後，実際の実行できるコードは，a，b，c 上の意思決定木となります．この意思決定した後，d の thunk，あるいは e の thunk のどちらかへの直接の goto がおきます．Rabbit コンパイラはまさしくこのようにコンパイルします．これらの関数呼出しは，分岐命令になるのです．それで，興味深いことは，これらのラムダ変数（x と y）は，データ変数を現わすのではないことです．

x と y は，文のラベルのようなものです．それで，Scheme は変数の名と文ラベルの名を一つのメカニズムで説明できるのです．

Revised Report on Scheme

Sussman と私は Scheme に関してさらにいくつかの論文を書きました．次の論文は “Revised Report on Scheme”[†3]です．これは Scheme の定義を多少改訂したものです．主な新しい機能としては，ダイナミックバインディングを追加したことがあります．それで各変数はレキシカルかダイナミックのどちらであってもよく，プログラマが選択できるようになりました．

このときまでに，他のユーザは基本的にコードを書くために Scheme を使っていました．それで，

---

†1 なぜ thunk という名前が silly だと感じたかについての補足：
二つの理由があった．まず，第一に，音のひびき．thunk というのは何か大変堅いものにある程度堅いものがぶつかったときの音．たとえば，The bat hit the baseball, thunk! など．
第二に，子供たちはしばしば，think の過去形は thunk だと間違える．sink--sunk，stink--stunk などがあるから．Here is a new game I thunk up などといったりする．本当は thought up なのに．だから，その言葉は子供の言葉のように思える．それで，大人の英語の話し手には，thunk を名詞として使うのは silly だと言える．

†2 Standish, T.A. *et al. : The Irvine Program Transformation Catalogue*, University of California, Jan. 1976.

†3 Steele, Guy Lewis Jr. and Sussman, Gerald Jay : The Revised Report on SCHEME : A Dialect of LISP, AI Memo 452 MIT AI Lab., January 1978.

Scheme をもっと便利にするためにいくつかの機能を加えました．let，cond，do をマクロとして加えました．また，ユーザがマクロを定義できるようにしました．この論文を私達は，“Revised Report on Scheme” と呼びました．というのは，私達は Algol 60 に敬意を払いたいと思っていたからです．“Revised Report on Algol 60”[†1]はたいへん良く書かれたレポートだと思っていました．私達はそれと同じくらいにきれいに書きたいと努力しました．

次に私達は，別の論文を書きました．“The Art of Interpreter”[†2]というものです．プログラミング言語の設計をどうやって実験できるのか説明しようとしたものです．Sussman と私が Scheme を設計している間に，“Lambda : The Ultimate Declarative” など，いくつかの論文を書き[†3]，また，たくさんの小さなインタプリタを書きました．あるときなどは，1週間に10あまりの異なるインタプリタを書きました．古い

†1 Naur, P., *et al.* : Revised Report on the Algorithmic Language ALGOL 60, *CACM*, Vol. 6, No. 1, pp. 1-17, Jan. 1963.

†2 Steele, Guy Lewis Jr. and Sussman, Gerald Jay : The Art of the Interpreter ; or, The Modularity Complex (Parts Zero, One, and Two), AI Memo 453 MIT AI Lab., May 1978.

†3 Steele, Guy Lewis Jr. : Macaroni is Better than Spaghetti, Proc. of the Symposium on Artifical Intelligence and Programming Languages, August 1977.
SIGPLAN Notices 12, 8, SIGART Newsletter 64 (August 1977), pp. 60-66.
Steele, Guy Lewis Jr. and Sussman, Gerald Jay : Constraints, AI Memo 502 MIT AI Lab., November 1978.
Invited paper, Proc. APL 79 conference.
ACM SIGPLAN STAPL APL Quote Quad 9, 4 (June 1979), pp. 208-225.
Steele, Guy Lewis Jr. and Sussman, Gerald Jay : Design of LISP-Based Processors ; or, SCHEME : A Dielectric LISP ; or, Finite Memories Considered Harmful ; or, LAMBDA : The Ultimate Opcode, AI memo 514 MIT AI Lab., March 1979.
Sussman, Gerald Jay and Steele, Guy Lewis Jr. : Constraints —— A Language for Expressing Almost-Hierarchical Descriptions, *Artificial Intelligence Journal*, Vol. 14, pp. 1-39, 1980.
Sussman, Gerald Jay, Holloway, Jack, Steele, Guy Lewis Jr. and Bell, Alan. : Scheme-79 —— LISP on a Chip., *IEEE Computer*, Vol. 14, No. 7, pp. 10-21, July 1981.

インタプリタを修正し，新しいものを作るのは，そして，言語が小さいのであればたいへん簡単な作業でした．インタプリタを書いて，インタプリタの中のさまざまな小さな違いが言語に大きな違いを与えることを示しました．

私達は，また，副作用とステートについても説明しました．副作用のもっともよい定義は Dan Weinreb によるものです．私達は「二つのオブジェクトが同一である」ことの意味を理解しようとしたことがあります．Sussman と私は，偉大な学者の本を調べてそれを理解しようとしました．しかし，Dan Weinreb は，言いました．「二つのコインがあって，一方がレールの上にあって，もう一つがつぶれたらそれは同一のものだ．」つまり，二つのオブジェクトがあって，一方での変化が常に他方に影響するのであればそれらは同一だ，という意味です．

Scheme の広がり

このころ，Scheme はもっとポピュラーになってきました．Sussman と私は 1970 年代にインディアナ大の研究者，Daniel Friedman と David Wise たちと，論文†を交換しました．インディアナ大学，エール大学などが Scheme の研究をしていました．

エールは二つのたいへん良い Scheme コンパイラを作りました．80 年代初期には，商用の Scheme が出現してきました．Texas Instruments は 100 ドル以下というたいへん安い価格で TI Scheme を出しました．Will Clinger のいたオレゴン大学の人たちは MacScheme を作りました．そのころ Common Lisp の開発がはじまったのです．

Scheme は Common Lisp の設計に影響を与えました．NIL というちょっと奇妙な名前の Lisp があります．このプロジェクトの開始には多少かかわっていました．これは Digital VAX とスタンフォード S1 に対する Lisp を作ろうとして設計されたものです．NIL は Mac Lisp に似ていますが，レキシカルスコーピングをします．NIL はその後，Common Lisp の重要な前身となりました．それが Common Lisp がレキシカルスコーピングをする理由です．

たくさんの人々が Scheme に関係してきたので，標準化委員会が作られました．委員会がはじまり，Scheme を改良しようとしていくつものレポートが書かれました．委員会では，Revised Revised Report，Revised Revised Revised Report，Revised Revised Revised Revised Report，そして今は，Revised Revised Revised Revised Revised Report が作られています．

Scheme のレポートは 4 回改訂され，それが IEEE Scheme 標準となりました．IEEE Scheme 標準は 5 年前に出ています．それを改訂するか否かが問題になったことがあります．私達はそのままで良しとしました．Scheme コミュニティは，同一の言語仕様を 5 年間提供してきたのです．

Scheme コミュニティは，とてもフレンドリです．ほとんど彼らはコンセンサスで合意してきました．Revised report の作業をしていたときの委員会の大原則は，「もし変更が提案された場合，誰かが no といったらそれは入れられない」ということでした．別の言い方をすると，誰もがそれに同意しないと直さない．これは言語を小さいものに保ちます．

これと Common Lisp コミュニティと比べてもいいかもしれません．Common Lisp はフレンドリではないとも言えるでしょう．コンセンサスで動くのではなかった．5 年間私は委員長をしていました．論点の調停をしていました．それは私の一生でもっとも困難な仕事でした！ Common Lisp での原則は，「変更が提案されるとき，誰かが yes と言うとその機能は言語に入れられていきます！」だから，Common Lisp は大きくなりました．

この比較はフェアでないかもしれません．Common Lisp の目的はたいへん異なっているものだと思います．Common Lisp は産業用のプログラミング言語となるように意図されていました．教育のために Common Lisp を小さいままにすることはゴールではありませんでした．だから，Common Lisp はとってもとっても大きなサブルーチンパッケージとなったの

† Wand, Mitchell and Friedman, Daniel P. : Compiling Lambda Expressions Using Continuations and Factorization, Technical Report 55, Indiana University, July 1977.
Wand, Mitchell : Continuation-based program transformation strategies, *J. ACM*, Vol. 27, No. 1, pp. 164-180, 1978.

です．それを使うのは簡単ですが，教えるのは困難です．

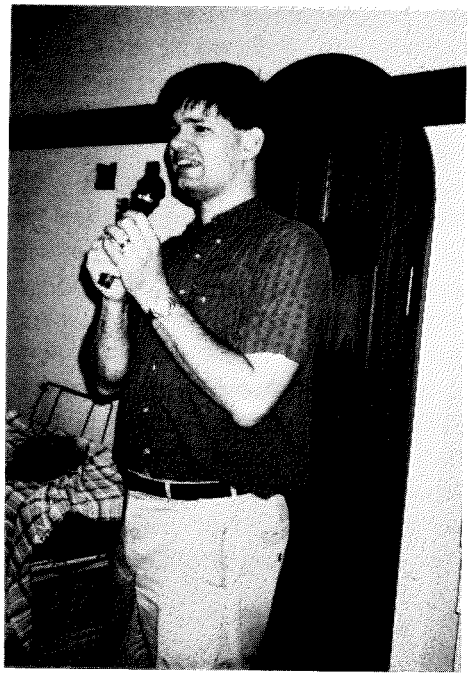
Scheme の貢献にはどんなものがあったのか，ひかえめにまとめてみたいと思います．まず，Scheme はたいへん小さな，しかし，便利な言語です．実際にそこには新しいアイディアは何もありません．そのかわりに，偶然，いくつかの古いアイディアがたくさんのことを説明できることを示しています．ことに，ラムダカリキュラスに基づく言語が有用なことを示しました．Scheme は小さいので，実現は容易です．そして，理論と実際の間の架け橋を与えています．理論的なラムダカリキュラスと denotational semantics のやり方は実際的なコンパイラを書くのにも便利なものです．

Scheme の将来

Scheme の将来について私の推理を述べましょう．もう言ったように IEEE 標準はしっかりとしたもので5年間変化がありませんでした．ユーザが変更してほしいと示唆したものは，マクロと，ある種のモジュールシステムでした．不運なことに，数年を経てもこれらの機能に良い合意はありませんでした．

マクロがたいへん良いものだという点には，かなり共通した合意があります．マクロにはいくつかの設計プロポーザルがありました．モジュールについても同じことが言えます．

私の推理では，Scheme はおそらく今と同じようなままでいるだろうと思っています．それで OK かもしれません．高校や大学や教育機関で広く使われているのです．また，CAD の機能拡張言語用，ちょうど Emacs には Emacs Lisp があるように，などのインダストリアルな応用も見られます．CAD のいくつかは Scheme を使っています．Scheme は全体としてたいへんよくできていて，変更の必要はないようにも見えます．



訳者の家でカラオケを楽しむ著者

最近の数年間，私は Scheme からほかの関数型言語への技術移転を認識しています．プログラミング言語 ML[†1]や Haskell[†2]は，Algol に似た構文をもっていますが，多くの点で，Scheme に似ています．かつて Scheme コミュニティにあったたぐいのプログラミング言語のイノベーションが，ML や Haskell でなされているのを知っています．

Scheme は凍結されているのか，Scheme にイノベーションが戻ってくるのかはよくわかりません．たとえば，Scheme は，徐々に ML のデザインに基づくモジュラーシステムになるのでしょう．

いつものように将来は不確定です．でも Scheme は今のところ，もっとも便利な Lisp だと思っています．

これで，私の話は終わりです．

どうもありがとうございました．

今後の勉強のために——訳者からの紹介

Scheme そのものについて，勉強するには次のような文献がある．

まず，Scheme 自身の文法については，*Revised*[5]

†1 Milner, Robin : A proposal for Standard ML, ACM sympo. on Lisp and Functional Programming, pp. 184-197, 1984.

†2 Hudak, Paul, *et al.* : Report on the Programming Language Haskell, Technical Report Yale University and Glasgow University, New Haven and Glasgow, Aug. 1991.

Report が現在まとめられつつあるとのことであるが，本文中にあるように，正式な最新のレポートは IEEE による IEEE CS : "IEEE Standard for the Scheme Programming Language", IEEE STD 1178-1990, 1991 である．

Revised Revised Revised Report と呼ばれているのは，Rees, Jonathan and William Clinger (Eds.) : "*Revised*^{3} Report on the Algorithmic Language Scheme", Vol. 21, No. 12, December 1986 であり，Revised Revised レポートというのは，Clinger, William : "The Revised Revised Report on Scheme : or, An Uncommon Lisp", AI Memo 848 MIT AI Lab., Aug. 1985, Revised Report というのは，Steele, Guy Lewis Jr. and Sussman, Gerald Jay : "The Revised Report on SCHEME : A Dialect of LISP", AI Memo 452 MIT AI Lab., January 1978 のことである．

これらの大もとの論文は，本文中にも引用されているが，Sussman, Gerald Jay and Steele, Guy Lewis Jr. : "SCHEME : AnInterpreter for Extended Lambda Calculus", AI Memo 349 MIT AI Lab., December 1975 がそれである．

また，Scheme だけではなく，Lisp 全般の歴史は，次の文献に詳しい．

Steele, G. L. and Gabriel, Richard : "The Evolution of Lisp", History of Programming Languages Conference Preprints, *ACM Sigplan Notices*, Vol. 28, No. 3, pp. 231-270, March 1993.

このコンファレンスには，他の多くのプログラミング言語の歴史についても論文が発表されているので，プログラミング言語の研究をする者は一読をされることをお勧めする．

（いだ まさゆき　青山学院大学 情報科学研究センター）