Scheme 前編

過去 現在 未来

Guy L. Steele Jr. 訳 井田 昌之

はじめに——訳者より

Scheme の開発者の一人である Guy L. Steele 博士が，1995年7月10日に Lisp 協議会のシンポジウムで講演を行なった．場所は青学会館である．本稿は，その講演をもとにまとめたものである．その内容は，開発の当事者でしか語れないものが多く，たとえば，Scheme の発端のいきさつには，Carl Hewitt 先生も大きく関係している点など，訳者にとっても，なるほどと新鮮で興味深いものであった．当時の MIT の人工知能研究所のホットな様子を垣間見た気がした．

本文では敬称を略した．登場人物の肩書はどんどん変化したので，それを完全に追跡するのは困難だったこと，また，博士の講演そのものでも，まったく敬称が使われなかったからである．

Scheme 関連の研究・調査を行ないたい読者の便宜を計れるよう，本文にはできるだけ，脚注として参照文献を付した．

挨 拶

こんにちは，LISP 協議会にお招きいただいてありがとうございます．今日は，Scheme 言語のはじまりについてお話しします．また，そもそものいきさつから今日に至るまでの歴史を議論します．今後についてもいくつか述べます．おそらく，Common Lisp†のことをご存知な方もいらっしゃるでしょうが，Common Lisp はたいへん大きく複雑な言語です．一方，Scheme はたいへん小さいです．今日の話を聞いて，なぜ Scheme が小さいのか，なぜ Common Lisp よりエレガントだと私が考えているか，みなさんに理解していただけることを期待します．いくつかの明確な理由があるのです．

その発端——Planner から Plasma へ

まず，どのようにして Scheme がはじまったのかについてお話しします．Scheme はたいへん注意深く設計された言語であって，プログラミング言語のあり方についての十分な研究と理解に基づいたものであったと言えればよかったかもしれません．しかし，事実はぜんぜん違います．

この言語は，MIT の二人の聡明な人間がしていた，大きな論争にはじまります．それは，私が MIT に学生として来るちょっと前のことでした．この論争というのは，Carl Hewitt と，新しく学生として入った Gerry Sussman の間に起こったものです．Carl Hewitt は，MIT の人工知能研究所にいて，ロボットのための言語の設計に興味をもっていました．60年代の半ばでは，ロボットが自分でなすべきことについて定理証明ができるかどうかが重要のようでした．特

† 彼は，Common Lisp の開発者としても有名である．Common Lisp の仕様については，『Common Lisp 第二版』，共立出版，1991.がある．

に，ロボットが具体的な行動をする前にその行動のプランニングをできるかどうかが重要でした．そこでCarl Hewitt は Planner[†1]というプログラミング言語を設計しました．

それは，プログラマやロボットが定理証明をするのを助けるためのものでした．アイディアというのは，プログラマがルールを書き出し，Planner 言語が自動的にルールから定理を演繹するというものでした．したがって，Planner はおそらく最初のルールベースのプログラミング言語でした．

Gerry Sussman は MIT に来て，Planner を実装するプロジェクトのメンバーになりました．Carl Hewitt は，簡単な言語を設計するということは決してありませんでしたので，Planner はたいへん複雑な言語でした．そこで彼らはまず，Muddle[†2]と呼ぶシンプルなプログラミング言語の設計からはじめることにしました．そして，Muddle を使って，Planner を作成しようとしたのです．Carl Hewitt は Muddle の設計を受け持ち，Gerry Sussman は Muddle の実装を受け持ち，そして Planner の実装をしようとしたのです．これもなお，困難な仕事で，完全な Planner 言語はついに実現されませんでした．しかし，Muddle は完全に作られ，他の研究プロジェクトで用いられました．Common Lisp の機能のうち，引数リストの，&optional，&rest，&keyword，その他のキーワードなどは直接 Muddle からきています．

一方，ロボットのためのなんらかのプログラミング言語が必要となり，Planner の小型版，Microplanner[†3]をつくる別のプロジェクトがはじまりました．それは，Muddle で実現する代わりに，Common Lisp の前身の一つである MacLisp[†4]で作られました．この Microplanner はまずまずの成功をおさめ，MIT の人工知能プロジェクトのいくつかはこれで書かれました．おそらく，その中で最も有名なプロジェクトは，Terry Winograd による「積み木の世界」で積み木を動かす SHRDLU でしょう．

Microplanner を経験したあと，Gerry Sussman ともう一人の学生 Drew McDermott はそれに満足しませんでした．プログラミングするのが難しかったからです．彼らはさらに別のプログラミング言語を発明することを決心しました．それは，Conniver[†5]と呼ばれました．

Microplanner と Conniver との違いは，Microplanner では定理証明も制御構造もオートマティックであったことです．Microplanner は定理証明をするのにルールを使っていましたので，自動バックトラックで次々とテストをしていきました．ときどき，この自動バックトラックは非効率的です．Conniver 言語では，Sussman と McDermott は，プログラマがこのバックトラックをもっと制御できるように，言い換えれば，バックトラック以外の何かもできるようにしようとしました（ここで，McDermott と Sussman によって書かれた論文があることを指摘しておきます．そのタイトルは，「なぜ Conniver は Planner より良いのか」[†6]です．）．

そこで，ある種の進歩があったのです．Carl Hewitt と Gerry Sussman の間の議論は，ロボットをプログラミングする最も良い方法，そして，人工知能をプログラミングする最も良い方法に関してであったのです．しかし，互いに争うのではなく，やったことは競って別の言語の設計をしたのです．「私の言語のほうが君のよりいいよ」と互いに言い合ったのです．

Conniver で終りではありませんでした．Carl Hewitt は，Plasma[†7]という名の言語を設計しました．Plasma は，Conniver より良かったのです．

Plasma は，アクタモデルという Carl Hewitt の新

---

†1 Hewitt, Carl E.: PLANNER: A Language for Proving Theorems in Robots, Proc. of the First Int'l Joint Conference on AI, pp 295-301, 1969.

†2 Galley, S.W. and Pfister, Greg.: The MDL Language, Programming Technology Division Document SYS. 11.01. Project MAC, MIT, Nov. 1975.

†3 Sussman, Gerald Jay, Terry Winograd, and Eugene Charniak: Microplanner Reference Manual, Report AIM-203 A, AI Lab, MIT, 1971.

†4 Moon, David: MacLisp Reference Manual version 0, Technical Report, MIT LCS, 1978.
Pitman, Kent: The revised MacLisp Manual (Saturday evening edition), Technical report 295, MIT LCS, 1983.

†5 Sussman, G.J., and McDermott, Drew V.: From PLANNER to CONNIVER——A Genetic Approach, Proc. the 1972 FJCC, pp 1171-1179, Aug. 1972.
McDermott, Drew V., and Sussman Gerald Jay: The CONNIVER Reference Manual, AI Memo 295 A, MIT AI Lab., Jan. 1974.

†6 Sussman, G.J., and McDermott, Drew V.: Why Conniving is Better than Planning, AI Memo 255 A, MIT AI Lab., April 1972.

しいアイディアに基づいていたので，以前の言語とは異なっているものでした．だから，Scheme を理解するには，まず Carl Hewitt のアクタモデルを理解する必要があります．

Scheme の理解は，まずアクタの理解から

アクタ（Actor）とは何か？　そのアイディアは，SIMULA 67 と Smalltalk にヒントを得ています．アクタは，オブジェクトのようなものです．これらの説明をしますが，私の言い方は Carl Hewitt が使った言葉とはまったく同じではありません．

アクタはオブジェクトです．アクタはメッセージを送受できます．アクタは他のアクタを知っています．もう少し言えば，他のアクタへのポインタをもっています．

Carl Hewitt は，すべての計算をアクタを使って説明しようとしました．今日，Plasma はまったくオブジェクト指向言語であったということができます．しかし，あの時点では，オブジェクト指向言語という言葉はあまり使われていませんでした．それで，この言語を理解するのにもがいていました．Carl Hewitt の大きな貢献は，彼は「オブジェクト」という言葉を，それ自身何もしないがそこにあるものとし，「アクタ」は何かをする何かとして見たことです．それは，エージェントです．事実，私はアクタではなくエージェントという言葉が使われなかったことにたいへん驚きました．ラテン語では，アクタとエージェントは同一の動詞から来ています．

1 に，メッセージ '+2 c' を送ると，c に 3 が送られる!?

アクタ計算の例をみてみましょう（図 1 参照）．Carl Hewitt にとって，1 とか 2 とかいった数というのはアクタです．

普通のプログラミング言語では，1 と 2 を加算に送ると考えるでしょう．そして，加算関数が 3 を返します．しかし Hewitt は違った考え方をしました．1 は一つのアクタだとします．もし，1 にメッセージを送ると，コンティニュエーション c を得ます．特に，1 にメッセージ（+ 2 c）を送ると，最終的にアクタ c はメッセージとして 3 を受け取ります．74 年ころの時点で，これは非常に奇妙な考え方でした．Gerry Sussman と，そのころ MIT に学生として入った私には，理解するには困難なものでした．

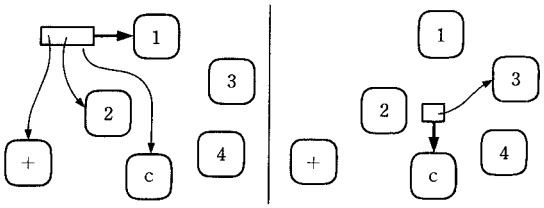


図 1

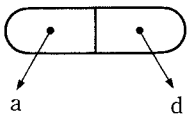


図 2

Conses as Actors

- A cons cell is an actor that knows about two other actors *a* and *d*.
- If you send it the message 'car *c*' then actor *c* receives the message 'a'.
- If you send it the message 'cdr *c*' then actor *c* receives the message 'd'.
- If you send it the message 'atom ? *c*' then actor *c* receives the message 'false'.

アクタの別の例をお見せしましょう．Lisp のコンスセルがアクタだとします．図で示しましょう（図 2）．コンスセルは，二つの別のアクタを知っているアクタです．

普通，この二つのアクタを *a* と *d* あるいは car と cdr と呼びます．もし，'car c' のメッセージをこのコンスセルに送ると，アクタ *c* はメッセージとして a を受け取ります．したがって，コンスアクタに，「あなたの car は何か」と尋ねることができるのです．そしてそれをメッセージとして返します．実際にはアクタ *a* を私たちに返すのではありません．それをアクタ *c* に渡すのです．今日，私たちはアクタ *c* を「コンティニュエーション」と呼びます．しかし，そのころはそうした用語はありませんでした．コンスセルに

(前ページ) †7 Smith, Brian C., and Hewitt, Carl : A PLASMA Primer, Working Paper 92, MIT AI Lab., October 1975.

送れるもう一つのメッセージは，'cdr c' です．そうすると，アクタ c は，メッセージ d を受け取ります．このコンスセルに送れるもう一つのメッセージに 'atom ? c' があります．この場合，c は，偽というメッセージ，「私はアトムではない」ということを受け取ります．

1974年の Lisp プログラマにとって，これは非常に奇妙な概念です．すべての人はコンスセルはただ36ビットのメモリだということを知っていました．18ビットが car で，18ビットが cdr です．誰でも，その car や cdr をのぞいてみることができ，rplaca や rplacd で置き換えることができます．

Carl Hewitt は言います．「そうじゃない．コンスセルはオブジェクトではない．それはアクタだ．『お渡しするオブジェクトであなたの car を置き換えてください．お願いします』と言わなければいけない．」

PLASMA が投げかけたもの ——Scheme のはじまり

Hewitt は，アクタ言語 Plasma を設計しました．この言語の中のすべてのものはアクタです．これは，たいへん複雑な言語で，たいへん複雑な文法をもっています．Sussman と私は，それがどう動くのか理解できませんでした．

MacLisp で書かれた Plasma の処理系がありました．それを試すことができました．また，Plasma でプログラムを書いてみました．それは動作しましたが，なぜ，動くのか理解できませんでした！

Sussman と私はそれがどう動くのか理解しようとしました．Sussman と私は，小さなインタプリタを書きました．言語を理解する最も簡単な方法はそのインタプリタを書くことです．インタプリタを書く最も良い言語は Lisp です．

そこで，私達は Plasma のたいへん小さなバージョンを書くことにしました．次の AI 言語を設計しようという気はありませんでした．そうではなく，不要な機能を省いた Plasma を理解しようとしたのです．それは小さな言語でした．そのインタプリタは Lisp で書いたので，そのおもちゃの言語は Lisp 構文をもっていました．Plasma の複雑な構文をもつのではなく，単純な Lisp 構文をもたせたのです．

どうやって，そのおもちゃのアクタ言語を書いたのかお話ししましょう．まず，2ページくらいの長さのたいへん小さい Lisp インタプリタを書くことから始めました．

一つだけ Lisp 1.5 マニュアル†から特殊な変更をしました．それは，そのころ普通に使われていたダイナミックスコーピングの代わりに，レキシカルスコーピングを用いることでした．二つの理由でレキシカルスコーピングをしました．まず，Plasma 言語を調べた結果，アクタはレキシカルスコーピングがいるように思ったこと．もう一つは，Sussman はそのころ，Algol 60 を教えていて，レキシカルスコーピングに興味をもっていたことからです．彼は Algol のようなレキシカルスコーピングをもつ言語をつくることはおもしろそうだと感じたのです．これが，Scheme に Algol の設計が及ぼした直接の影響です．

私達はさらに二つのことを加えて，レキシカルスコーピングの Lisp の開発を始めました．一つは，アクタの生成手段，もう一つはメッセージ送信です．アクタの生成のためには，alpha 式を用意しました．これはラムダ式に似ています．アルファ式は，最初に alpha をおき，次にそのアクタに送られるメッセージの要素を表わす変数名を並べます．そのあとに，メッセージを受け取られたときに実行されるコードであるボディを書きます．

Carl Hewitt が言っていることから私達が理解できたのは，アクタと関数の違いは，アクタは値を返さないということでした．そのかわりにボディは何らかの方法で，値を他のアクタに送ります．普通，何かを送っている宛先のアクタは，元のメッセージの中に，コンティニュエーションとして存在します．

私達は，alpha という言葉を選びました．というのは，それがアクタというギリシャ語の最初の文字だからです．このような式の評価は，Lisp の値としてアクタを生成します．このアルファ式はアクタを生成するのです．つまり，その式の Lisp としての値はアクタです．それが私達がアクタを生成する仕方でありました．メッセージを送る仕方について Sussman に尋ねました．私は覚えていません．そして彼も覚えていません．しかし，私達は send という特殊形式をもっていたように思います．しかし，関数適用の構文は，

† McCarthy, J., *et al.* : *Lisp 1.5 Programmer's Manual*, The MIT Press, 1963.

また，メッセージ送信の構文となりうると，その後わかりました．というのは，もし，最初に関数があるのなら関数呼出しになるし，最初にアクタがあるのならメッセージ送信になるだろうということです．

アクタにメッセージを送るには，send，そのアクタ，そしてその引数という順に書きます．

それはアクタを生成し，メッセージを送るのには十分な，たいへん小さいシンプルな言語でした．それでもあらゆる関数，アクタ，そしてそれらの組合せを試してみることができました．Scheme のまさに初期の処理系でのサンプルコードを示しましょう．

関数と actor

Functions and Actors

・Factorial function:

```
(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))))
```

・Factorial actor:

```
(define actorial
  (alpha (n c)
    (if (= n 0)
        (send c 1)
        (send actorial (- n 1)
              (alpha (z) (send c (
* n z)))))))
```

これは，Lisp の教科書にもよく出てくる階乗(factorial) のプログラムです．factorial は，関数としてラムダ式で定義されています．この関数は，1引数，n をとり，もしそれがゼロなら1を値として返し，そうでなければ，値 $n-1$ で再帰呼出しをし，その値と n をかけ，結果を返します．この定義とアクタ版を比べてください．アクタ版は，英語のジョークで actorial という名にしています．

actorial はアクタを生成するアルファ式で定義されています．値を返すようには考えられていません．そのかわりに，値 c は，値をそこに送るコンティニュエーションです．

actorial を見てみましょう．actorial は，2引数，n と c を受け取ります．もし，n がゼロなら値1を c に送ります．そうでなければ，二つの値，$n-1$ と新しいコンティニュエーションを actorial に送ります．

この新しいコンティニュエーションはそれ自身アクタです．それは，z を受け取り，n と z をかけ，c に結果を送ります．したがって，最終的に c は正しい値にセットされます．

実際には，actorial のこの定義は純粋なアクタではありません．これは入り交じった記法です．というのは，actorial はアクタだけれど，等号とマイナスと乗算は関数だからです．等号とマイナスと乗算をアクタとすると，全体が純粋なアクタとなります．次のスライドでそれをお見せしましょう．

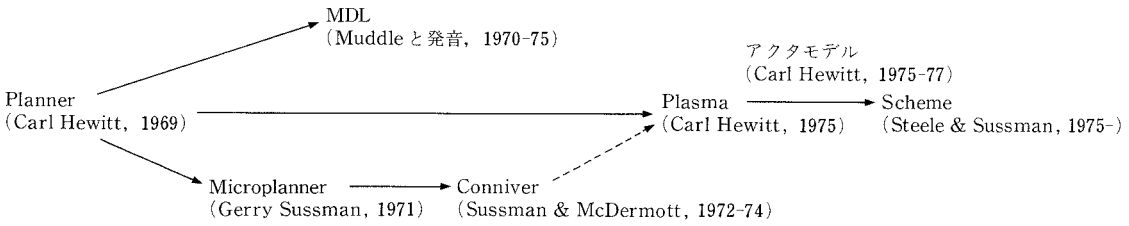
Detailed Factorial Actor

```
(define actorial
  (alpha (n c)
    (send = n 0
      (alpha (p)
        (if p
            (send c 1)
            (send - n 1
              (alpha (m)
                (send actorial m
                  (alpha (z)
                    (send * n z c)))
 )))))))
```

このスライドが私の話の中で最も複雑なものです．あとは簡単になります．

OK，さぁはじめましょう．これはちょっと複雑です．actorial は，メッセージとして二つのもの n と c を受け取ります．次に三つのものを等号アクタに送ります．値 n，値ゼロ，それとコンティニュエーションです．そして，そのコンティニュエーションは，値 p を受け取ります．もし，その値が真であれば，値1を c に送ります．もし，p が偽ならば，三つの値をマイナスに送ります．値は，n と1ともう一つのコンティニュエーションです．

このマイナスは，値をコンティニュエーションに送ります．この新しい値は，$n-1$ と等しいものです．これは m と呼ばれます．この新しい値 m は，actor-



ial に第3のコンティニュエーションとともに送られます．actorial が計算する値は z と呼ばれます．このコンティニュエーションは，その後，乗算アクタに三つの値，n と z と元々のコンティニュエーション c を送ります．次に乗算はその積を c に送ります．

それが，私達がほしいものです．もし，この計算過程を注意深く見てくれれば，これは同一の計算なのがわかるでしょう．なぜ，私達が理解が困難だと感じたかを理解してもらえると思います．

Plasma プログラムは，これに似ていますが，もっと複雑な構文をもっていました．Plasma プログラムは，Lisp に比べて2倍は長く，10倍は理解するのが困難でした．Gerry Sussman と私はシンプルな構文のシンプルな言語の，小さなインタプリタが大好きでした．楽しいので，この小さな言語でたくさんのプログラムを書きました．Carl Hewitt の論文からたくさんの例を引いてきて，それを私達のインタプリタで走らせました．

Scheme という名前の誕生

これは人工知能に良い言語かもしれないとわかってきて，私達はだんだん興奮してきました．

Carl Hewitt が複雑な言語を設計し，Gerry が私達が理解し使えるものを実現してくれるというのはこれが初めてではありませんでした．Carl Hewitt は，使うのには複雑すぎる言語である Planner を設計し，一方，Gerry Sussman は MicroPlanner を設計し，人々はそれを使いました．これはそれと同一のストーリーだと思いました．

Carl Hewitt は，複雑な Plasma 言語を設計し，Gerry Sussman（と私）は，Scheme を設計しました．これは使えるシンプルな Lisp だと思いました．私達は，新しい AI 言語を作ろうとしているのではありませんでした．私達は自分達が理解できるようなシンプルな言語を偶然作ったのです．

もし，これが Conniver や Plasma のあとの新しい AI 言語となり得るなら，良い AI 言語の名前をもつべきです．英語で Planner はプランする何かです．Conniver はスニーキー（こそこそする，卑劣）なプランナーです．だから，もっとスニーキーなプランナーを何と呼ぼうか．そりゃ，スキーマー（陰謀家，策略家）だ．それで，Schemer とつけたのです．

残念ながら，我々は60年代に設計された OS を使っていたので，すべてのファイル名は6文字以下でなければなりませんでした．それで，ファイル名 SCHEMER は最初の6文字だけに切り捨てられました．その結果，SCHEME というファイル名がポピュラーな名前となりました．言語全体は偶然に設計され，その名も偶然につけられました．多くの人は私達がすばらしい仕事をしたと考えています．私にはどうしてだかわかりません．

（いだ まさゆき　青山学院大学 情報科学研究センター）

［つづく］