●●●●機能拡張言語としての●●●●

GNU Emacs Lisp

リチャード・ストールマン　訳 井田昌之・土井　巧

1回 Emacs前史 ——TECOではなぜ駄目だったか

これから話そうと思っているのは，Emacs と Emacs Lisp の歴史，Emacs Lisp でのデシジョンの根拠，それから Lisp から見た GNU プロジェクトの今後の計画のことだ．

まず，Emacs 以前のいくつかの話から始めなければならない．それには TECO というものがからんでいる．これは確か，1961 年かそこらにボストン周辺のどこかで，たぶん PDP-1 コンピュータ用に書かれたテキストエディタだ．

TECO はリアルタイムのテキストエディタではなかった．たくさんのコマンドを 1 列にまとめたコマンド文字列全体をタイプして，入力の終りには Esc を 2 回タイプする．すると，そのシーケンス全部がいっぺんに実行される．このコマンド文字列の中に，カーソル移動コマンドや検索コマンドや挿入コマンドや削除コマンドなど，いろいろな種類のコマンドを入れるわけだ．一つのコマンド文字列には，好きなだけたくさんのコマンドを入れることができる．

だけどみんなは，コマンドを一つにかためて与えるより，もっと強力なことをしたいと思っていることがわかった．たとえば，エディタを使ってやりたいことの一つに，全置換がある．すべての FOO を BAR で置き換える，というやつだ．まあ，多くのエディタにはたいてい，全置換をするようなコマンドがある．

TECO をやってた人間は，別のアイディアをもっていた．ループの構文要素を追加しよう，というわけだ．そうすると，次のように言うことができる．「ループを開始せよ．FOO を検索せよ．もし見つからなければループを終了せよ．見つかった 3 文字を削除せよ．BAR を挿入せよ．ループを終了せよ」（“Begin a loop, search for foo, exit if it failed to find it, delete previous three characters, insert bar, end the loop.”）この簡単なプログラムは，バッファを検索して，逐一 FOO を見つけ出し，削除してから BAR を挿入する．このように，エディタ用言語にプログラミング構文が追加されるようになった．

時がたつにつれて，彼らはさらにプログラミング構文要素を追加していった．たとえば，変数，算術演算，条件文，それからテキスト文字列について関数呼出しができる機能を．そうこうするうち，TECO はチューリングマシンなみのパワーをもつようになった．同時に，チューリングマシンと同じくらいプログラマにとって醜いものになった．

1970 年代になると，TECO を使って書かれるプログラムはどんどん大きくなりだした．けれど，TECO の矛盾に満ちたシンタックスのために，こうしたプログラムで仕事をするのはいつも苦痛だった．

人々は表示機能を追加した．これは，カーソルの周

† 本稿は，1994 年 12 月 7 日に青山学院大学で行なわれた公開講演 “GNU Emacs Lisp as an Extension Language” を翻訳・編集したものである．

囲のバッファの内容を TECO が表示する，というものだ．コマンド文字列を全部実行したあとごとに，再表示するわけだ．これは便利な機能で，最初のディスプレイ指向の TECO となった．

1972 年だったと思うが，Carl Mikkelson がリアルタイムディスプレイモードを追加した．リアルタイムディスプレイモードは，1 文字打つたびに実行する．すべての文字がそれぞれ，何かをするコマンドであり，そのコマンドはすぐさま表示をするわけだ．この最初の実装は，たいして強力でも効率的でもなかった．私はこれをもっと効率が良くなるように書き直して，実用的に使えるようにした．けれども，それはまだパワーが限られていた．できたことは，局所的にテキストを編集することだけだった．だから，ファイルのセーブや別のファイルの表示といった別のことをしたければ，対話モードを抜けて，通常の TECO のコマンド文字列に戻らなければならなかった．

そのうち，誰かがこう言った．「2，3 文字の指示で好きな TECO プログラムを実行できるようにしたらどうだろう？ そうすれば，リアルタイムエディティングモードのプログラムが変更できる．」私はこれについて考え，すべての文字を再定義可能にすればたやすいと判断した．で，私はそれを作った．たちまち，たくさんのいろいろな連中がリアルタイムエディティングモードの再プログラムを始め，たくさんのコマンドを定義しだした．

2 Emacs誕生

そうこうするうちに，リアルタイムエディティングモードの中で何でもできるようになった．抜ける必要もなければ，実際の TECO コマンドをタイプする必要もなくなった．こうして TECO は，プログラミング拡張機能をもったエディタから，エディタ記述用のプログラミング言語へと変身していった．エディティング専用でもなく，プログラミング専用でもなくなっていった．けれども TECO は，その生まれゆえに，矛盾したシンタックスをもった汚らしいプログラミング言語だった．私たちは，TECO がテキスト編集言語として生まれた運命まで変えることはできなかった．

2 年くらいして，Guy Steele があるアイディアを出した．それは，「TECO で書かれた様々なエディタを見比べて，最良のアイディアを採り入れ，それらをうまくフィットさせて，新しいエディタを TECO で書こう」というものだった．そこで，彼はコマンド言語を設計した．そして，私たちは一緒に処理系を作り始めた．が，しばらくして彼は抜けたので，それからは私が処理系を作った．それが，Emacs だ．

私たちがそれを Emacs と呼んだ理由だが，まず，TECO で書かれたプログラムはみな，「TECO マクロ」と呼ばれていた．まだ私には，これはマクロ機能をもったエディティング言語だという考えがあった．TECO で書かれたエディタは通常，macro や macros からとって ○○MAC とか ○○MACS という名前が付けられていた．そこで私は，この規約に従うことにした．E をつけた理由は，まだ使われていない 1 文字の略称がほしかったのだが，まだ E は使われていないことがわかったからだ．で，私はこれを EMACS と呼べば，E と略すことができて良いと思った．

こうして，Emacs が書かれ，また Emacs を書くために，私たちは TECO のプログラミング機能にたくさんの拡張を施した．たとえば，1，2 文字ではなくもっと長い名前の関数をもたせられる機能や，変数名を長くできる能力，それに UNWIND-PROTECT や非局所的脱出などの強力なプログラミング機能だ．

けれども，シンタックスの醜さを変えることはできなかった．そして私は，今までたどってきた道をさらに進むことは間違いであるという結論に達した．ユーザ向けのコマンド用に設計された言語をもってきて，これをプログラミング言語に変えようとするのは間違いだ．そうすると悪いプログラミング言語ができてしまうから．良いプログラミング言語がほしければ，最初からプログラミング言語たるべく設計するべきだ．すでにあるプログラミング言語を調べて，自分でどれが好きかを判断し，これを自らのプログラミング言語とすると仮定した上で最善を尽くすべきだ．コマンド言語をベースにすることで，“きみの”プログラミング言語を妥協してはいけない．

私が設計した Emacs には，エディティング言語と，Guy Steele が設計した 1，2 文字のコマンド群が含まれており，元々は TECO，そして後に新しく出てきた Emacs の処理系は，別のプログラミング言

語をもっていた．できあがった設計を見れば，プログラミング言語が TECO でなければならない理由は何もなかった．

けれど，そのプログラミング言語はインタプリタでなければならない．ユーザが作業中に簡単に機能拡張できるように，インタプリタがなければならない．さて，インタプリタの利点を最もよく活用する言語は，Lisp である．したがって Lisp は Emacs のプログラミング言語として使うのにとても良い言語である．

かくして，最初に Lisp を使った Emacs は Multics Emacs だった．これは 70 年代後半に Bernie Greenberg が開発した．彼は，エディタをまるごと Multics マクロで書いた．これは，彼が真のコンパイラをもっていたから可能だった．だから彼は，再表示といった，とても高速性が要求される部分を含め，エディタをまるごと Lisp で書いた．そしてこれは機械語にコンパイルされたので，高速に実行した．

3 GNU Emacs Lisp登場

GNU プロジェクトに Emacs がほしいと決心したときに，私はいくらか違った設計をした．私はとても強力な Lisp システムをもっていなかったし，自分で真の Lisp コンパイラを書きたいとは思わなかった．なぜなら，たくさんの種類のコンピュータがあるので，Lisp コンパイラの可搬性を高めることは大仕事だからだ．

そこで，私は違った設計をとった．単純な Lisp システムを作る．それは Common Lisp のような大きなものではなく，最小限の機能をもった Lisp であり，それ自体，可搬性を高くする．C で書き，どんなコンピュータでも走るようにする．そして，この Lisp の実行はあまり速くないので，エディタのうち速く走る必要のある部分は直接 C で書く．そして，高速実行の必要のない部分は Lisp で書く．

私が Common Lisp を却下したのはこのためである．Common Lisp 処理系は，数多くの機能をもっているため，とてもでかい．私は，第一に，Common Lisp を書くのは仕事が大きすぎ，第二に，プログラムがそんなに大きいとエディタとしては実用的でないことを自覚していた．だから，私はたくさんの気のき

いた機能をよく考えた上で却下した．ふつうなら，「こりゃいかした機能だね．OK，これも入れよう」というところを，「これなしでもできるさ．私たちは小さくしなきゃならないんだ．出ていけ！」と言ったのだ．

たとえば，Emacs Lisp にはループ構文は一つしかない．それは WHILE で，とても単純なループ文だ．けど，それで十分なんだ．ほかのは必要ない．

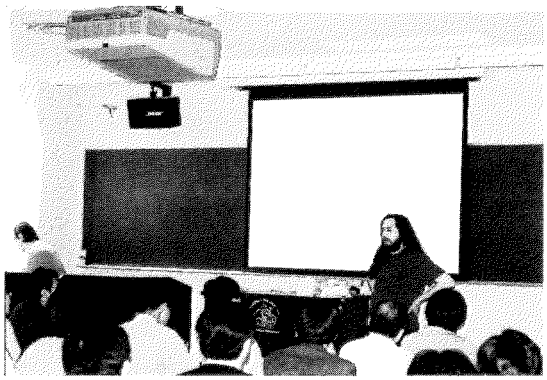
私はダイナミックスコーピングを使うことに決めた．ダイナミックスコープをもつ Lisp は，インプリメントが単純だからだ．それに，レキシカルスコーピングだけしかもたないというのは良くない．レキシカルスコーピングは実行スピードの点，そして名前の衝突を避けるという点では良いものだ．けれども，システムが小さければ，名前の衝突を避けるには，名前にプレフィクスをつければ十分だ．だから，どちらか1種類のスコーピングを選べと言われたら，それはダイナミックだ，というのが私たちがダイナミックスコープを選んだ理由なのだ．

うん，でも私は Lisp にいくつかの機能を付け加えもした．ほかの Lisp ではもっていないような機能を．これらは基本的な言語機能ではないけれど，エディティング用にいいので入れた．その一つにバッファローカル変数がある．普通の Lisp にはそんなものはない．でも，Emacs Lisp には，Emacs で編集中のテキストに使用するカレントバッファというのがある．カレントバッファに対して，変数束縛の集まりがある．名前と値の対の集合である．バッファを切り替えると，これらの変数の値が全部いっぺんに変更される．これは，編集方法をカスタマイズするという点で非常に大事なことである．なぜなら，バッファを切り替えるということは，編集用のパラメタをすべて変更することを意味するのだから．

私が Lisp に追加したもう一つの機能は，対話型呼出しの機能だ．

（講演の要旨を書き出している井田に向かって）あぁ，INTERACTIVE を使った例を書き出してよ．簡単なやつでいい．ペンをもってるんだからさ．（会場，笑）じゃ，関数を定義して．そう．二つ引数をとる．OK，そこで INTERACTIVE “r”…

（井田は，受講者のために RMS の講演の要旨を OHP シートに超高速で書き留め続けている．井田，プログラム例を書き始める．）



講演中のリチャード・ストールマン
（左端は訳者の井田）

```
(defun foo(bar baz)
(interactive"r")...)
```

さてと，この関数を Lisp 式から呼ぶと，ふつうの Lisp 関数と同じように二つの引数にプログラムで与えた値を受ける．けれども，INTERACTIVE は，この foo が同時にエディタコマンドでもあることを告げている．ユーザが編集作業中に対話的にこの関数を呼び出すこともできるんだ．そしてその場合，“r” が引数の作り方を指定する．小文字の “r” で，リージョンの境界を引数として使うことを指定している．つまり，この指定はテキストの一部を操作するような関数の中で使える．だから，これらの引数にはふつう，START と END という名前が使われる．プログラムの中では開始位置と終了位置のアドレスを渡すことになるのだけれど，ユーザが対話的に呼び出すときには操作すべきテキストを指定するポイントとマークが与えられる．INTERACTIVE の指定にはこのほかにも，数値プレフィクスを引数として使うもの，ミニバッファを使ってファイル名やバッファ名なんかを読むもの，などがある．

このように，私は他のエディタによく見られる余分なレベルの複雑さを回避した．Lisp Machine エディタを含め，多くの Emacs エディタでは，新しいコマンドを書くたびに関数を二つ作らなければならなかった．一つは，Lisp プログラムから呼ぶのに適しているような，Lisp スタイルの引数をとる関数．そして次に，エディタコマンドであるようなもう一つの関数．これは，対話的に引数を読み込んでから前者を呼

び出す．私は思った．「なぜ一つのオペレーションにいちいち二つの関数を定義する必要がある？」関数は一つだけでよい．Lisp からもユーザからも簡便に呼び出せるような一つの関数があれば，この仕組みは，引数渡しという点ではとてもうまくいく．

しかしながら，今のところ Emacs では，もう一方のことがまだできていない．結果の表示に関して，対話レベルとプログラムレベルを結合するということだ．というのは，もし結果を表示したければ，MESSAGE の呼出しを付け加えればよい．これは，スクリーンの下端にメッセージを表示する．私が思うに，プログラム中で呼ばれたら普通の Lisp 関数のように値を返し，対話的に呼ばれたら結果をおしゃれなメッセージで表示することを一つの同じ関数でできれば都合がいい．

そこで，私は改良案の一つとして，INTERACTIVE フォームに第2の要素または第3の要素を追加して，値を対話的に表示する方法を指定できるようにすることを考えている．数を返すような関数を定義すると，Lisp から呼んだときにはその数を返すが，対話的に呼ばれたときには“%d”といった文字列をINTERACTIVE の引数として書いておくことができて，もっと長いメッセージ表示の中にその値を表示できるようにすればいい．このようにすれば，引数渡しの側と同じように，値を返す側も面倒を見ることができるだろう．これは将来の話だけれど．

4 なぜLispなのか？

さて，なぜ Lisp はインタプリタのシステムにとってそんなに具合いが良いのだろうか？　何よりもまず，Lisp はとても単純なシンタックスをもっているところが良い．Lisp で仕事をしている間は，シンタックスに思い悩む必要がない．

もう一つ Lisp の良いところは，強力にして独立したデータ型の集まりをもっていることだ．なるほど，多くの言語では独自のデータ型を定義できる．けれども，それらはみな，同じ主題の変奏にすぎない．つまり，それらは基本的に，ありきたりのものを集めただけなのだ．しかし Lisp だと，違ったアルゴリズム的特性をもったいくつかの違ったデータ型が使える．たとえば，リストとベクタというデータ型があるが，それらはそれぞれ違った操作に適している．リストでは，簡単に要素の挿入と削除ができる．ベクタでは，要素の挿入と削除はできないが，一定時間で任意の要素を取り出せる．私たちが Lisp に新しいデータ型を導入するのは，それが実際に何か違うことをするからだ．そうして，強力なデータ型の集まりが得られる．Lisp シンボルもその一つだ．シンボルは，他の言語では類を見ない特徴をもっている．C には Lisp シンボルのようなものはない．これは，Lisp の強力な機能の一つなのだ．

そしてこれは，プログラムがきわめて自然なやり方でデータとして表現されるという事実とあいまって，Lisp を他のどんなインタプリタ言語よりも強力にしている．

さて，世に文字列処理言語はあまたある．たとえば TECO のように，プログラムを構築して実行できるようなものもある．TECO では，単に文字列を作り，これをレジスタに格納してから，そのレジスタを実行させる．そうすると，TECO インタプリタはその文字列を構文解析する．ほかにはたとえば，Tcl のような言語がある．古い時代には，それこそいくつもの文字列処理インタプリタがあった．けれども，文字列というものはその性質として構造をもっていない．リストとして表現された Lisp プログラムは，プログラムの構造が歴然とわかるような構造を本来もっている．だからリストは，文字列でプログラムを表わすのに比べてはるかに強力である．

さて，Lisp の長所の一つに，単純な Lisp プログラムがとても簡単に書ける，ということがある．Lisp を Emacs のようなエディタに突っ込めば，とても単純な関数で，テキストに対して役に立つような仕事をさせることができ，一挙両得になる．そして，プログラミングを勉強中の入門プログラマでも，とても単純な関数を書いて実行し，テキストの変化を見ることができる．よって，このプログラムは何をして，その変更の仕方はこうで，このようにデバッグして，といったことについて直接的なフィードバックを得ることができる．

5 Emacsでプログラミングも学べる

つまりこれは，Emacs はプログラミングを勉強しようとする人にとって非常に優れた環境だ，ということを意味する．そんなわけで，多くの人たちが Emacs Lisp のプログラミングを勉強する．Emacs Lisp は世界中でもっとも広く使われている Lisp システムの一つである．もっとも，ほかにも拡張可能なプログラムとして，AutoCAD がある．これも Lisp を使ったプログラムだと思う．ひょっとしたら AutoCAD 用の Lisp プログラムを書いている人のほうが，GNU Emacs Lisp のプログラマより多いかもしれない．実際どうなのだか私にはわからないけれど．

いずれにしろ，部分的な仕事をする簡単なプログラムが書けるということには変わりはない．たとえ，仕事がプログラミングそのものでないとしてもだ．たとえ主な興味が本を書いたり手紙を書いたりすることであったとしても，プログラミングを学ぶことは有益だと気がつくだろう．このように，Lisp でプログラミングができる Emacs は，人にプログラミングを教えるための強力なツールになる．たとえ彼らが当初はプログラミングの習得を望んでいなくてもね…．

実際，プログラミングを学んでいることにすら気づかなかった，という例がある．Bernie Greenberg の報告によれば，彼のオフィスの秘書たちは Multics Emacs 用の Lisp プログラムを書くことでプログラミングを学んでいたが，彼女たちは自分たちではそうは思っていなかったという．エディタの環境の変更方法を勉強しているとしか思っていなかったのだ．だが，実際に彼女たちは，Lisp プログラムの書き方を学んでいた．もし彼女たちがプログラミングしようとしていることを自覚していたなら，こう言うだろう．「私には覚えられないわ！　プログラミングなんてできっこない！」でも，誰もそれがプログラミングであることを教えなかったので，彼女たちはそれを学んだのだ．

これはむろん米国で起きたことで，日本でもそうかもしれないが，多くの女性は，自分ではプログラミングのような数学的なことはできないと教えられてきている．自分でできるとは信じていないのだ．これは本当に悲しむべきことだ．明らかに彼女たちはプログラミングができたのである．自分でできるとは信じていないにもかかわらず…．

6 GNU Emacs式機能設計法

Emacs の下に Emacs Lisp のプログラミング環境をもっていることが，Emacs をさらに強力にしている．けれども，このことがまた，設計を難しくすることもある．たとえば，文書の中に絵を入れることを考えてみよう．他のエディタだったら，「そこに絵があったらこれらのコマンドはどう動くべきか」といった設計上の決定事項を考えなければならない．「Ctrl-N をタイプして，絵から下へおりていくにはどうすればいい？」と，こういうのが下さなければならない設計上の決定事項である．さらに，データ構造を設計する必要があるかもしれないが，何でも好きなデータ構造を変更してよい．そのプログラムでしか見ていないデータ構造なのだから．

けれど，GNU Emacs ではこうは考えない．GNU Emacs では，ユーザのプログラムからまさに同じそのデータ構造が見えるのだ．データ構造がユーザプログラムから見えることで，二つの帰結が生じる．

一つは，私たちはデータ構造を拡張するときには，上位互換を保ってきれいに拡張する必要がある，ということだ．Emacs Lisp のデータの世界の中に自然にフィットさせなければならないのである．

二つ目は，たぶんすでに，ユーザが書いた Emacs Lisp のコードは 100 万行にも上っており，私たちはそれを壊したくない．つまり，今までにあるコードは新しい機能を入れてもそのままで動いてほしいということだ．だから私たちは，新しいデータ構造に対して，Lisp プリミティブが何をすべきかがわかっていなければならないのである．

たとえば，Emacs 19 にはテキスト属性がある．バッファや文字列は，文字の連なりである．Emacs 19 では，それぞれの文字は一つの属性リストを関連づけられている．概念的にはすべての文字が固有の属性リストをもっていると考えてよい．さて，既存のデータ構造と既存の Lisp プリミティブをきれいに拡張する

方法でこのことをしなければならない．私たちがやったことは，バッファからバッファへ，または文字列からバッファへテキストをコピーするときにはそれぞれの文字の属性リストも必ずコピーしなければならない，ということだった．これが自然な拡張というものだ．結果として，ユーザが書いたプログラムが，バッファから文字列へテキストをコピーしたり，文字列をバッファに挿入し直したりするときには，自動的にテキスト属性もコピーされるようになった．テキスト属性は保存されるわけである．このようにして，少なくとも大半のユーザの Lisp コードは，テキスト属性をもったテキストに対しても動き続けるようにできたのである．

けれども，私たちは未だに設計をし忘れた部分を見つけている．たとえば，ほんの先週のことだが，私たちが比較関数でテキスト属性をどう扱うかについて考えていなかったことを指摘された．違った属性をもつ二つの文字列が EQUAL になることがわかったのだ．これをどうするか決めなければならなくなった．そこで私は，関数 STRING-EQUAL では属性を無視し，文字だけを比較することに決めた．関数 EQUAL は，「これらのオブジェクトは同じ内容をもっているか？」を判断する伝統的な Lisp 関数だが，属性を比較させることにした．これを作り込んだのは2，3日前のことである．そういうことなのだ．

テキスト属性は，書式づけられたテキストの編集に関連して使っている．フォントの選択やマージンを狭めるべき位置やスペーシング幅といった，書式情報をすべてテキスト属性に記録する，というものだ．他の言語をサポートするような処理系がほしいと言われたら，たぶん Mule ——多国語版 Emacs ——のことは知ってると思うけど，テキストの表現が既存の Lisp プリミティブでもはまるようにきっちり考えなければならない．新しいデータ構造を作ったよ，などと言うのは無意味なことだ．だってすべてのユーザの Lisp コードが動かなくなってしまうかもしれないのだから．マルチバイトの文字列を導入することで，東洋の言語の文字やアルファベットでも，同一の関数を使ってテキストをコピーできるようにしたんだ．

7回 GNU Common Lisp

さて，Emacs についてはもう十分に話した．

GNU プロジェクトにはほかにも，Lisp でやる計画やプロジェクトがある．たとえば，GNU Common Lisp がある．これは以前，Kyoto Common Lisp として知られていたものだけれど，GNU プロジェクトに寄贈された．これは Common Lisp 仕様書第1版の処理系だ．私たちは，Common Lisp 第2版の機能の追加を手伝ってくれるボランティアを必要としている．私自身はそのエキスパートじゃない．Common Lisp には，第1版の仕様の頃までは注目していたけれど，第2版が作られた頃には，その仕事をやめていた．でも，Lisp が好きで，Lisp の活気と繁栄を続かせたい人にとっては，GNU Common Lisp はうってつけのプロジェクトだ．井田先生なら，このプロジェクトの課題を教えてくれる保守担当者を紹介してくれる．

8回 共通機能拡張用言語の開発を目指して

加えて，ほんの最近のことだが，私たちは Modified Scheme†を共通の GNU 機能拡張言語として使う具体的な計画を立てた．

UNIX システムを見渡してみると，awk，bc，sh，それに csh といったプログラム可能なユーティリティがたくさんある．そのうちのどの二つをとっても，同じ言語をもってはいない．awk と bc と C-shell は C ライクに見えるけれど，それぞれ実際には C と違うし，みな互いに違う．

こりゃ考えうる最悪のシステム設計だ．ユーザは，混同するくらい似ているくせにそれぞれ違った四つのものを覚えなければならない，ということだ．かといって，混同してしまうとエラーになる．

システム設計における一つの原理として，同じシス

† 訳注：訳者らは，この語の訳として，改訂版 Scheme，修正版 Scheme，変更版 Scheme あるいは進化版 Scheme などを考えた．ストールマン氏はあきらかに Scheme そのものを GNU の共通機能拡張用言語に採用するのではなく，Scheme をベースにして，彼一流の味付けをしようとしている．しかし，まだ実体が現われていないこと，日本語になると意味が特定されてくるので今後の進行によっては彼自身の意図が変わることもありうること，などを考慮して英語のままとした．

テム中の二つのものは，同一であるかまたはまったく違うかのいずれかであるべきだ，というのがある．あるいは，同一であるか，または一方を覚えても他方を使う際に誤ることがない程度に自然に似通っているかのどちらかだ．

だから，私は同じ言語で書かれたユーティリティで awk や bc を置き換えたいと思った．そして，その同じ言語がシェルでも使えるようにしたいと思った．さらに，デバッガにプログラミング機能がほしい人のために，GNU のデバッガである GDB にその同じ言語を組み入れたいと思った．

そうすれば，ひとたびその言語を覚えれば，その言語を使うパッケージならどれでも使えるし，パッケージごとの違いで混同するようなことはなくなる．さて，そうなると，どの言語にすべきかを考えなければならなくなった．

これで，私はあるジレンマにとらわれた．まず，bc や awk は C ライクな言語だが，私は本当の C 言語に近づけた C ライクな言語を提供して，これらを置き換えたいと思っていた．awk や bc はゆがんだ構文則をもっており，たとえば行末にセミコロンを打つ必要がない．私はこれは馬鹿げていると思う．セミコロンをタイプするのは簡単なことじゃないか．一番重要なことは，これらのいろいろな言語を同じにすることだ．だから私は，式と文のシンタックスは C の構文そのもので，C を知っていれば使い方がわかる（その逆も言える）ような言語がほしいと思った．

もちろん，この言語に C と同じ意味でのポインタはもたせられないだろう．というのは，これはインタプリタ言語にするからだ．また，C が行なうのと同じような変数の強い型付けはしないだろう．その代わり，Lisp のようにそれぞれの値に型をもたせ，変数にはどんな型でも格納できるようにする．つまり，この言語は C ではない．ある局面では C とまったく同じであり，別の局面では C とはまったく違う．だが，これは可能性の一つであり，やる意味は十分にある．

けれども，同時に私は，Lisp を汎用 GNU 機能拡張言語にしたいと思っている．なぜなら Lisp は C や C もどき（Pseudo-C）よりもはるかに強力できれいだからである．で，私は悩んだ．

やがて，私はこう思った．「ひょっとしたら一つの機能拡張パッケージで Lisp と C の両方の言語をサポートできるかもしれないぞ」．そこで考えついて，おそらく Emacs から Emacs Lisp を切り出してきて，他のパッケージに突っ込むのがいいのではないかと思った．C もどきはその上に実装することにする，と．そうすれば，他のすべてのプログラムは Emacs Lisp と C もどきをサポートする．この夏までは，私は大まかなアイディアしかもっていなかったので，誰かこの仕事をする人を探し回っていた．

9月になり，私は Sun がポストしたメッセージを読んだ．それは，Tcl を汎用機能拡張言語にするという Sun の計画についてであった．Tcl の連中は，それを強力な標準に仕立てあげ，誰もが Tcl を使わなければならないようにもくろんでいるように思えた．これは恐ろしいことだ，と私は思った．早いとこ何とかしなくっちゃ．だって，私は Tcl なんか使いたくないからね．

Tcl というのは，結局のところ文字列言語で，非常に単純だ．確かにチューリングマシンなみに強力で，TECO よりもずっとましには決まっている．けれども，それでもなおセマンティクスはきれいではない．それに，文字列以外のデータ型がない．文字列処理言語の唯一のデータ型は文字列だというのは，ありがちなことだ．私は，それではパワーに欠けると思った．私のシステムにほしいのは，Tcl じゃなくて Lisp なんだ．

そこで，私はメッセージをポストして，こう言った．「Tcl の宣伝隊を盛り上げて，誰もがみんなその流行りに乗らなければならないような気にさせようとする計画があるようだ．私はここに，GNU プロジェクトはそれに乗らないことを発表する．だからすべての人がそうするわけではない．あなたが気に入らないなら，そうする必要はないのだ」．それから，ここで今まで話してきたような機能拡張言語についての大まかなアイディアを述べた．

はたして，これに対して，いろいろなアイディアを示唆したり，すでになされている仕事があることを私に教えてくれる人が出てきた．そして，私は具体的なアイディアをまとめあげた．それは，根本の機能拡張言語として，いくらか変更と拡張を施した Scheme を使う，というものだ．そして，他のたくさんの言語は，Scheme に翻訳することでサポートする．C もどきは Scheme にコンパイルしてその Scheme プログ

ラムを評価することでサポートできる．Emacs Lisp は Scheme に変換してその Scheme プログラムを評価することでサポートできる．

クリーンアップ版の Tcl でさえサポートできる．すでに，Rush というプログラムがあって，これはだいたいの Tcl を Scheme にコンパイルする．これは，Tcl そのものではない．Tcl にはいくつか汚いセマンティクスがあって，それを実装する唯一の方法は実際に文字列をスキャンすることであるが，これは遅い．彼（Rush の開発者か）は，変数参照や何やらについて，きれいなセマンティクスを導入すれば，文字列のスキャンに比べたら桁違いに速くなることに気づいた．すると，妙ちくりんな Tcl の構文要素は動かなくなる．でも，同じことをするもっときれいなやり方があるのだから，そんな構文要素を使う必要はない．まあ，Tcl そのものもサポートできるかもしれないが，そうするともちろん実行は遅くなる．こっち（Rush）のほうが実行速度ははるかに速い．そして，他のいろいろな言語も同じように Scheme に翻訳できる．

これは実に，とても大きな問題を解決する．わかるかな…．いろいろな人がいて，いろいろな機能拡張言語があるということは，大きなゴールを達成する妨げになっているのだ．

9回 ユーザが機能拡張言語を選べるように

過去には，機能拡張パッケージはそれぞれ一つの言語をサポートしていた．そして，あなたが望むなら，ユーザはあなたが作った言語を使うチャンスがある．あなたはアプリケーション開発者を説得したりおだてたり脅したりしながらあなたの機能拡張パッケージを使うようにし向けなければならなかった．もしアプリケーション開発者がその機能拡張言語を選べば，ユーザもその言語を使うことになるからね．けれどもアプリケーション開発者が他の機能拡張言語を選べば，ユーザもその言語を使わなければならなくなる．ユーザに選択の余地はなかった．ユーザというものは，アプリケーション開発者が使った，機能拡張パッケージについてくる言語を使うものと相場が決まっていた．だから，もしあなたが Lisp が好きで，ユーザも Lisp で

プログラムが書けるようにしたければ，アプリケーション開発者のところに行くか何かして，Lisp インタプリタを使うように仕向けなければならなかった．でも，そうこうしているうちにほかの誰かがやって来て，Tcl インタプリタを使わせようとする．「なら戦わなくっちゃ！」ってことになる．だけど，今私が考えている解決策は，このような葛藤を排除すると思っている．

なぜなら，ユーザに言語の選択を委ねるような機能拡張パッケージにすればいいんだから．

ユーザは，Scheme を使うこともできれば，C もどきを使うこともできれば，Tcl/Rush，それに Python を使うこともできる．基本的にユーザが選択権をもつ．それぞれのアプリケーションは，一つのライブラリを使い，そのライブラリが間接的にいろいろな言語をサポートする．さらに，どんな言語であれ，特定言語のサポートとアプリケーションを独立させる．トランスレータなんだから．トランスレータは，ユーザのプログラムを Scheme に翻訳する．だから，トランスレータは特定のアプリケーションの中で動いている必要はない．それはアプリケーションに組み込む必要がない，ということだ．ユーザが自分でトランスレータを書いて，そのトランスレータをすべてのアプリケーションに使うことだってできるのだ．

このようにして，いろいろな，いわゆる機能拡張派閥の争いを解消して，機能拡張言語の有り様を根本的に改良することができると，私は信じている．ただ，個々のユーザに選択を委ねればいいのだ．

ここで，違うやり方でこれを達成してきた他のシステムについて触れる必要があるだろう．それらは，完全にこのことを達成したわけではない．そうしたシステムでは，アプリケーションとインタプリタの間に特定の API があって，アプリケーションでダイナミックリンクすることで，ユーザが好みの言語のインタプリタを動的にリンクできる．API を通じて話す限りは，どの言語も一緒に動く．ユーザはどの言語を使うこともできるけれど，言語の選択はその API で記述されたデータ型に制限される．そして，たいていそれは文字列だけである．

だからこの API アプローチはとても強力だけれど，使えるデータ型は制限される．私はこれでは不十分だと思った．それにもう一つ，GNU は一義的に，仕様ではなくて，システムなのだ．だから，使えるスペックを渡すだけ，ということはしたくなかった．私たちは人々に拡張性を与えたい．それにはこの拡張可能なパッケージを開発することに帰結する．最終的には適切な API が見つかるかもしれないし，ほかの人が同じ API に対して究極のインタプリタを開発するかもしれない．それはそれで構わない．重要なことは，私たちがほしいのは，GNU システムで使える一つの優れた拡張可能パッケージであるということなのだ．

10 Scheme との関係

とにかく，この拡張可能パッケージのほとんどの部分には，Scheme を使うことを計画している．Scheme は Lisp の方言の一つで，1970 年代に Guy Steele と Gerry Sussman が設計した．とても単純かつ強力になるように設計されている．

しかしながら，私たちは，いくつか機能を追加して，機能拡張言語として，より便利な言語にしたいと思っている．たとえば，UNIX のすべてのシステムコールを実行する手段を提供したい．それは，これらの機能拡張パッケージを「使う上で」必要なのだから．サブジョブを実行したり，入出力をリダイレクトしたり，パイプを生成したり，といったことの簡便な方法を提供したい．私たちは，高速で簡便な文字列処理関数を提供したい．なぜかというと，文字列だけでは十分だとは思っていないし，文字列だけに限定したいとも思っていないが，実際，こうした拡張可能ユーティリティではみんな，文字列を頻繁に使うからだ．だから私たちは，文字列の処理を簡便にするための良いユーティリティを提供する必要がある．

ほとんど全部，Scheme に対して上位互換な拡張だ．けれども，Scheme 言語を変更しなければならないようなところが何か所かある．そしてそれは，残念ながらデータ型に関することである．

Lisp は伝統的に，NIL というオブジェクトをもっている．これは三つの役割をもっている．一つは“NIL”という名前のシンボルとして，である．これは他のどんなシンボルとも同じように，一つのシンボルである．もちろん関数定義や属性リストをもたせることができる．同時にこのシンボルは，空リストとし

ての役割をもっている．リストの終端を示すものである．そして，NIL の三つめの意味は，偽という値である．真と偽を表わす値を使いたければ，NIL があれば，それが偽を意味し，それ以外の値はすべて真を意味する．

Steele と Sussman は，Scheme を設計したときに，これらの意味を分割した．彼らは，空リストであるようなオブジェクトをもたせ，これは同時に偽を意味するものとした．けれども，NIL はそれと無関係にした．シンボル NIL は特別な意味をもたないただのシンボルとしたのである．

さて，Emacs Lisp を Scheme に翻訳するときに，この違いをどうやって調停しようか．二つの言語が違う規約をもっているという事実を扱わなければならない．私はこのように決めた．Emacs Lisp の機能にはシンボル NIL は実際には空リストだと思わせるようにする，と．つまり，Lisp にシンボル NIL を要求すると，空リストが返ってくることになる．Lisp プログラムでは，「これは空リストかシンボルか？」と聞くと yes と答える．これはもちろん，Scheme と Lisp では「これはシンボルか？」について異なる述語をもつことを意味する．普通の Scheme では，「これはシンボルか？」の述語に空リストを渡すと no と答える．空リストはシンボルではない．だが，Lisp の述語は「そうです，こいつはシンボルです」と答える．さらに，その名前を尋ねると，Lisp は文字列“NIL”を返す．このようにして，このオブジェクトは実際にはシンボル NIL だと思わせることができる．

だが，事は悪いほうへ進んでいる．数年前，数学の理論家が Scheme の標準化委員会の大勢を占めるようになった．彼らは，Scheme を必要とし，処理系を作り，そして使う本当のハッカーたちの手から，言語設計の主導権を取り上げてしまった．そして，空リストと偽の値は分けたほうが数学的にきれいだと判断したのだ．というわけで，今では三つの別のオブジェクトがあるのである．二つに分かれたオブジェクトを扱う方法はわかったが，三つを扱う方法はわからなかった．で，公式標準の Scheme を実装するのではなく，Scheme を変更しなければならないかもしれない．だから，私たちはそれを Modified Scheme と呼んでいる．理論屋さんが公式標準を強制しようとしても，ことによるとこれは崩れていくだろう．

さらに，事態を解決するもう少しうまい方法の提案がきていて，それについては今，考慮している．それは，Scheme が三つの独立したオブジェクトをもっていてもなお，Emacs Lisp と Scheme の両方をサポートする方法についてだ．けれども，それがほんとに動くかどうかはわからない．調べてみなくては…．

さて，アプリケーションが Scheme インタプリタとインタフェースをとる方法は二つある．

アプリケーションを書いて，このような拡張性パッケージを使おうという段には，アプリケーションは，プログラミング言語に追加定義されたプリミティブ関数で構成されている．たとえば Tcl ではそのように動き，またこれは GNU の拡張パッケージでも同様だ．

問題は，追加定義した関数の呼出し規約がどうであるか，というところである．私たちは二つの異なる呼出し規約を考えている．一つは単純で，もう一つは強力である．

単純なほうは，文字列だけを使う．単純なほうの呼出し規約を使って作成したプリミティブ関数を

Scheme プログラムから呼び出すと，引数はすべて文字列に変換されて，その文字列の値が関数に渡される．また，戻り値も文字列になる．これは，とても簡単なプログラムでできるアプリケーションの作成に役立つ．実際，この呼出し規約は Tcl と互換にしたいと思っている．そうすれば，Tcl で動くように書かれたアプリケーションを GNU 拡張パッケージにもってこれるようになるのだ．

もう一方のインタフェースは強力で，Scheme のオブジェクトに直接アクセスする．つまり，実際の Scheme オブジェクトの値をアプリケーションの関数に渡す．実際には Emacs Lisp の呼出し規約にとてもよく似たものになってくるだろうと思う．それは，固定個の引数をもった関数であれば，C でぴったりその数の引数が得られ，それぞれの引数は単一の Scheme オブジェクトである，というものだ．そして，Scheme オブジェクトのアクセスは特別なマクロを使って行ない，その構成要素を取り出して Scheme の数と C の数との間の変換を行なう．こうして，引数はバッチリ扱える．

C のコードから別の Scheme のプリミティブを呼び出すのも簡単だ．普通の C 関数のコードの引数として，実際の Scheme オブジェクトを渡してやるだけでいい．ある点で，これは今の Emacs Lisp でやっていることよりも単純だ．今の Emacs Lisp では，ガーベジコレクタと通信する特殊なマクロを使わなければならない．ガーベジコレクタがマークしたりリロケートする必要のある変数を，すべて識別してやる必要がある．

しかし，Scheme では，保守的な（conservative）ガーベジコレクタを使用する．保守的なガーベジコレクタは，C プログラム中のすべての変数を探してマークする．だから，どの変数が実際の Scheme オブジェクトであるかを教えてやる必要はない．いずれにしろすべての変数を調べるのであるから．それで，もし値が妥当な，実際に何かを指している Scheme オブジェクトなら，ガーベジコレクタはそのオブジェクトにマークし，そうでなければ，無視する．そうすると，プログラム中に整数があると——まぁどんな値の整数でもいいのだが——その値がオブジェクトを指しているように見えなければ，何も悪いことは起こらない．ガーベジコレクタはその値を見て，「ああ，これは絶対妥当なオブジェクトじゃない」といって無視する．しかし，その値が偶然，たまたま Scheme オブジェクトのように見えたら，ガーベジコレクタはそのオブジェクトをマークしてしまう．これは間違いかもしれず，ひょっとしたらそのオブジェクトは実際にはゴミかもしれない．それでも，起こりうる最悪の事態は，一時的にいくらかの余分な領域を無駄遣いするだけのことだ．このようにして，私たちはプログラミングインタフェースを大幅に簡略化する．

浅草寺にて
（ストールマンと井田）

保守的なガーベジコレクタは，マークしたすべてをリロケートできるが，インクリメンタルにはできない．リアルタイムのアプリケーション向けの選択肢として，そのうちにインクリメンタルガーベジコレクタがほしくなるだろう．そのためには，マークの必要がある変数をマクロを使って決定するように API を変更する必要がある．これがインクリメンタルガーベジコレクタの処理に必要な情報を与えることになる．

訳者あとがき

ここに掲載したのは，いまさら人物を紹介するまでもないかもしれないが，フリーソフトウェアファンデーションの代表で，GNU プロジェクトのリーダであるリチャード・ストールマン氏の講演録である．1994 年 12 月 7 日青山学院大学 1173 番教室で行なわれた．講演時の録音テープから起こして作成した．井田がアウトラインを作り，土井が素訳を書き，それを井田が訂正するというプロセスをとった．佐藤衛氏（キヤノン）には校正に関して助力をいただいた．写真は六条範俊氏（富士通）による．

内容の正確さを保ちながら，できるだけストールマン氏の語り口を伝えられるように努力した．作成過程では，何

度か本人とやりとりをし，内容を確認していった．意訳はしないようにしたが，文脈がわかりにくい部分については，昨年度の AI 研での交流や，日本滞在中の井田の自宅での雑談なども含めて訳者には疑う余地のないものについては文脈の補強をした．Modified Shceme については，本文中の脚注にも記したが，現在その仕様を練っているところでもあるので，言い換えれば，彼自身プランをまとめている最中なので，彼自身があいまいな部分もあることを付け加えておく．

講演の背景を二，三述べる．アメリカに Association of Lisp Users (ALU) という団体がある．この前身は Symbolics Users Group および UNIX 上の Common Lisp 処理系のユーザたちのグループである．訳者の井田も何度か投票でそのボードメンバに選ばれている．

この ALU は，毎年，夏に LUV というコンファレンスを開催していて，LUV '94 は，あのマッカーシー教授とストールマン氏に二大基調講演をやってもらおうということになった．依頼をし，無事成功裏に終了したのだが，そのとき，ストールマン氏には，「自分の講演のアウトラインを書いてくれればその筋道でやる，そうでないと，どこへ話がいくか責任をもてないよ」，と言われ一同頭をかかえ，その結果作ったアウトラインがまあ彼本人にも好評で，それを日本で再現したのがこの講演である．だから，一度夏に予行演習をやり，さらにその後の半年で磨きをかけてできた講演だということができる．両方聞いた範囲で，賭け値なしに今回の講演のほうがまとまっている．

彼独特のするどい哲学が随所にあらわれているので，どうか，もう一度読み直していただきたい．

(いだ まさゆき　青山学院大学 情報科学研究センター・
どい たくみ　(株)CSK)

bit 悪魔の辞典

マルチ症候群 (multi-syndrome)

すべてのものに「マルチ」という枕言葉がつかないと落ち着かない，「マルチ」さえつけば満足する，「マルチ」さえつければ売れると思う，「マルチ」さえつければ研究として成立すると思うなどといった心の病の総称．これまでに以下に挙げるものを含め数十個の臨床例が報告されている．しかし「マルチ症候群」という命名をした医学者もその病にかかっているという全米医学会の報告もあり，いまや手をつけられない蔓延状態になっている．

◆**マルチーズ** (multiz)——複数の Z，すなわち ZZZ のこと．寝いびきの音を表わす．これは「枕言葉」から来た呼び方であるが，本来はズズズと呼ぶべきであろう．

◆**マルチックス** (Multix)——本来の意味は，X (伏せ字) が多いという意味だが，計算機の世界では，わけのわからない複雑なシステムを指すようになった (病にかかった作成者はそれを誇りに思っていたのである)．Multix をもじった MULTICS という OS がその代表例だが，実はマルチを超えようとして生まれたのが，偏在 X，すなわち UNIX (UNIversal X——伏せ字があまねく存在するソフトウェアといった意味) である．UNIX によって OS はまったく理解不能の境地に達した．

◆**マルチャンのタヌキウドン**——どーってことないインスタントウドンであるが，マルチがつかないと安心できない人が買う．

◆**マルチメディア**——中庸，凡庸を意味する medium の複数形にさらに「マルチ」を冠した典型的な症状．自分の凡庸さを認めたくない研究者をはじめ，多数の患者が感染している．

◆**マルチエージェント**——自らは責任を負えない，能力の低い代理人を何人か勝手に働かせれば，なにか意味のある仕事ができるという幻想．あの『心の病』を書いたミンスキーといった有名人も罹病している．

◆**マルチプロセッサ**——1 台の計算機の上ですら，ちゃんとしたプログラムの書けない研究者が，複数台の計算機の上でのプログラミングへと，問題を複雑化することによって，自分の無能を隠そうとする意図的な仮病．マルチ症候群の風下にもおけない．

◆**ハナマルチ味噌**——花丸つきともいえる究極の大量マルチ添加物で合成された味噌の登録商標．防腐剤が 7 種類も入っており，どんな顔や歌声でも腐ることがない．マルチ症候群の人々は，カラオケ屋で，この味噌を使った一品をつい注文してしまう．