

# どう転んでも Lisp

東京大学 情報理工学系研究科 創造情報学専攻 竹内郁雄

## 1 はじめに

本稿は、これまでにいろいろなところで書いたり喋ったりしてきたものをベースにしている。特に電子情報通信に書いた招待論文 [1] をベースにしている。どこかで見た覚えがあるなあと思われたらごめんなさい。

1966年にP. J. Landinの書いた“Next 700 Programming Languages”という少し古い論文 [2] がある。この論文の要旨は「これからざっと700個のプログラミング言語が設計されるだろうが、どれも同じ穴のムジナの変種。言語の設計には2つの独立な選択、つまり、文法の選択とデータ型の選択がある。これさえ済ませば言語は一丁上がり。それ以外の言語の骨格はラムダ計算で尽きている。それ以上なにを望むことがあるか。」という哲学的達観である。もし、Landinの主張が正しければ（あるいは、正しくなくても世の中の人に受け入れられれば）、プログラミング言語に関する議論はとっくの昔に落着いていたはずである。

たしかに、Landinがこの論文を書いた時代は、言語研究者が百家争鳴状態であった。しかし、現在は、いわば群雄割拠、すなわち、比較的少数のプログラミング言語がそれぞれのテリトリーで自己の存在を主張している。プログラミング言語（あるいはそれを使う人々）に生態学的棲み分けが起こることを、Landinには予測できなかったのかもしれない。その後、R. Floydが1978年のTuring賞受賞講演 [3] で、プログラミングパラダイムという言葉を使い、そのような棲み分けの意味を明らかにした。

言語の棲み分けの固定化は、言語それぞれの得意・不得意もあるが、その言語で書かれたプログラムの巨大な蓄積 — これは文化以外のなにものでもない — や、プログラムを書く人間の（どの言語がプログラミングの母国語だったか、どのコミュニティの中でプログラムを書いているかといった）嗜好・環境など、純粋な言語設計論の枠の外の理由によって起こる。プログラミング言語と人間社会の関係を見るときに、これを見損ってはならない。この事情は自然言語と同じである。

群雄割拠の進行と並行して、1960～70年代をピークとしたプログラミング言語に関する研究（特に言語設計に関するもの）は明らかに減少した。これはLandinの哲学的達観に真実が含まれているということでもある。多数の研究者が研究のタネを求めるには、もはや豊かな収穫の見込めない分野になってしまったのであろう。

言語設計の量的な衰退は、現在のコンピュータハードウェアの基本構造に基づくかぎり、画期的に斬新な、あるいは強力な計算モデルがもうないこと、既存の言語の中で、固有の特徴をもった少数が、それを支えるコミュニティとともに、プログラミング言語の世界における群雄としての地位を固めてきたことによる。言語の実践研究は、このような群雄言語の小改良や実装技術の開拓に限られてきているとって過言ではなからう。

Lisp もこういった群雄の一つだが、やっぱりほかの群雄とは一味違うということをここでは述べたい。

## 2 記号処理

我々が今日「記号処理」というとき、具体的になにを指すか、それほど明確ではない。人によって定義が少しずつ異なる。私は大学院で「記号処理特論」という講義をしているが、学生に「記号に対立する処理対象はなに?」聞くと、ほとんどまともな答えが返ってこない。実はそれ以前に「まともな答え」がないのかもしれない。

大昔, “Non-numerical Computation” といったタイトルの本があった。記号処理と非数値計算を意図的に区別するために工夫されたタイトルだったと思う。非数値計算には文字や文字列の扱いも含まれた。しかし, C 言語は, 設計者によって書かれた見事な文字列処理ライブラリによって, 文字列処理を「記号処理」の花園から追い払ったように思われる。それ以前に Snobol[4] という文字列処理言語があったが, 当時はたしかに「記号処理言語」に分類されていた。

C 言語の構造体はもちろんであるが, オブジェクト指向言語の中に出てくるオブジェクトを「記号」と明確に断言する人は少ないようである。たしかに, 記号と呼ぶにはそれ自身が大きすぎるという印象がある。

記号とは端的にいうと, や のようにそれ 1 個でなにかを指示でき, しかしそれ自身は固有の意味をもたないものである。このような「記号」の言葉の使い方をプログラミング言語の歴史の中で最初にしたのは, A. Newell と H. Simon であろう。彼らは J. McCarthy らが Lisp を設計する数年前に IPL (Information Processing Language) という言語を設計開発した。彼らの目標は記号論理を扱う Logic Theorist というシステムをつくることだったので, 文字どおり記号を扱う言語が必要であった。彼らの発想の根源は彼らの Turing 賞受賞記念講演で述べられている [5]。

彼らは, 神経回路網モデルの先駆者 W. McCulloch の「人間が使う数とはなにか, 数を使う人間とはなにか?」という 1961 年の論文をもじって, 「知能が使う記号とはなにか, 記号を使う知能とはなにか」という問題設定を行ない, 最終的に「物理記号系」という仮説を導いた。物理記号系とは, 時間の経過にしたがって, 記号構造が進展するような集合をつくり出す機械的な系のことである。彼らによれば, 記号はある物理法則に従う物理的実体である — ここがユニーク。これらの記号から式, すなわち記号構造が合成される。記号と式には対象を指示する能力があり, 式が指示するプロセス (動作) は実行可能である, などとした上で, 記号と式に関して以下のような性質を仮定した。

- (1) 記号の指示には任意性があり, なんでも指せる。
- (2) ある機械が実行できるすべてのプロセスに対してそれを指示する式がある。
- (3) 式をつくり, 変更できるプロセスがある。

- (4) 式は実体として安定である (一度つくったら消さないかぎり消えない).
- (5) 系が保持できる式の量は非有界である.

よく読むと、これは Lisp の性質の表明にほかならない。それはともかく、Simon と Newell は、物理記号系が知的行為の必要十分な実現手段であると主張した。これが人工知能においてコネクショニズムと対照される記号主義の原点である。

このような概念整理が 1950 年代前半に本当になされていたかどうかを細かく詮索してもしょうがないが、彼らのつくった IPL はポインタを陽に表現できるアセンブラのような言語であった。当時の人々は、ポインタの先にまたポインタがあって...、とはまるでタマネギの皮ではないか？ どこに計算すべき中身があるのかといぶかしがったという。

ところが、記号処理の本質は Simon と Newell によって見事に総括されている。や のような文字どおりの記号だけではなく、記号と記号のなす合成的関係、すなわち式という記号構造が記号処理の対象なのである。特に記号構造が動的に変化するときにこそ「記号処理」という言葉はふさわしい。文字列は文字と文字の関係が変わっても所詮一次元配列の中の位置関係の変化にすぎない。それは構造的変化と呼ぶには単純すぎる。ところが、動的に個々の要素の関係が変化するのであれば、オブジェクト全体のなす関係構造は、Simon と Newell の記号構造と似たものになり、逆にオブジェクト指向言語のオブジェクトを記号と呼ぶことに違和感がなくなる。

記号という、それ自身には意味のない軽いものと、記号構造を組み立てる合成規則の単純さのバランスが最もよくとれている言語の一つが Lisp である。Lisp のオリジナルでは、記号はその名もアトムであり、合成規則は cons と呼ばれる対構造の生成だけである。これによって任意の二進木が合成できる。文字列との決定的な差は、一次元配列から、二次元的な二進木への飛躍である。このことの重要な意味合いを、1988 年の McCarthy の京都賞受賞の記念セミナーで佐藤雅彦が強調している。

記号処理がほかと違うということを最も直観的に理解できる例は、やはり数式処理であろう。数値計算ではもちろんないし、文字列処理でもない。数式の中の記号が数を指示するという制限はあるが、あとは上の Simon と Newell の 5 つの性質を比較的良好に満たしている。

### 3 Lisp の今日的意義

Lisp がどのような言語であるか、どのような特徴をもった言語であるかについては、すでに多くのことが語られている。ここで、現代的な視点を加味をしながら、もう一度それを復習する。

#### 3.1 前言語としての S 式

1958 年 9 月からの Lisp 開拓史 — まさに、開拓史と呼ぶにふさわしい波瀾万丈があった — をひもとくと、今日の二進構造の源である cons セルは、最初のうち、car, cdr のほかに、cpr,

cir, csr, ctr など、もっとたくさんの (IBM704 のアーキテクチャに依存した) フィールドをもっていて、明確な二進構造として意識されたものではなかった。基本操作である cons も瑣末な差しかない3種類のものが用意されていた。しかし、これが整理・統合されるまでに、2ヵ月もかからなかった。当時の計算機科学の水準を考えると、これは恐るべき早さだった。

このように機械依存の世界から一挙に高級言語へと抽象化が進んだが、その時点ではプログラムは数学の関数表記に近い M 式で書くものとされ、S 式はデータあるいはプログラムの内部表現を書くための低レベルの (まるでバイナリダンプのような) 表記法とされていた。しかし、いまや M 式は排除され、プログラムもデータも S 式で書くことが Lisper の間で標準になった。ただし、カッコだらけの S 式の見にくさはいつまでたっても一般から問題視され続けた。

## [エピソード 1] Lisp (リスプ) とは bit 悪魔の辞典より引用

Parentoxin (カッコ毒) 中毒症患者の書く文章やプログラムの総称。(を多用 (たよう) し,) も多用するため (と) の対応 (開く (と閉じる) との) の対応が取り難 (にく) い。また開きカッコ ((の) こと) と閉じカッコ () のこと) が対称的なため) と (を間違えたり (この間は間の間違い (よくある間違い (これは一種の再帰呼び出しか?) なので要注意)) 誤値 (ごしょく ( のような打ち誤りなどが多く、(結果的に) 非常に読み難い。

Lisp の基本は () (別名 nil) である。すなわち、何もしないことや正しくないことが基本である。この前提により、Lisp のプログラムは必ず正しい。もちろん、これでは実用的なプログラムが書けないことは当然である。実際、世の中で唯一使われている Lisp の応用は、Lisp で書かれた Prolog インタプリタのみである。

英語としての lisp は本来、舌もつれ (舌足らず) の発音をすること ([s] の音を, [th] のように発音すること) である。すなわち、“Lisp lisps lithp (「リスプ」は「リすプ」と発音する)” なのであり (英語ではこれに類した法則がたくさんある。たとえば、“Left left right (左は右を残した)”, “Right writes right (右は正しく書く)” など。) ここですでに Lisp の自己再帰性は崩れている。すなわち、Lisp は ((と) の多用による混乱のみならず) 理論的にも不完全である。もっとも、日本人には舌がもつれなくても th の発音が難しいので、これから日本は Plorog でいくとあっており、米国人に笑われている。

Lisp 嫌いの多くは、S 式を Lisp のほかの特徴をすべて覆い隠す必要悪のように考えているらしい。McCarthy 自身も S 式は好きでなかったようで、S 式を使わない Lisp 2 の開発プロジェクトに関わった。S 式忌避のちょっと前の例では、Lisp と Smalltalk のいいとこどりをしたという動的オブジェクト指向言語 Dylan がある [6]。筆者が見た 1990 年代始めの Dylan の

第一版の仕様書はすべてが S 式で記述されていた。しかし、第二版からカッコがなくなり、一般的な言語の文法スタイルに改訂されてしまった。よほど S 式の評判が悪かったのであろう。

Lisper が S 式をなんの障害とも思わず、むしろそれを Lisp の本質と考え、Lisp を使わない人々が S 式を Lisp の最大の欠点だと思っている、この大きな食い違いの原因は論理的なものではないであろう。しかし、嫌いな人々が見向きもしない間に、S 式ですでに多くのことがなされていることは事実である。実際、Lisp 嫌いの人も認めざるを得ない Lisp の生産性の高さには S 式は障害になっているどころか、基礎となっている。

言語開発が盛んだった 1970 年代において、多くの研究者が Lisp を実験言語のベースにした。S 式のような、いわば文法らしいものがない言語は、実験言語をつくる時に「文法設計」に悩まず、言語の意味論の本質に単刀直入に迫ることを容易にした。規模は小さくなったが、この流れは現在でも続いている。

S 式が単なる記号構造を表すだけの最低の文法規則しかもたないことが、これらの例では重要な意味をもっている。S 式は前置記法によく適合しているという以外、意味的に無色透明である。逆に、S 式の上にほとんどありとあらゆる言語意味論を簡単に乗せることができる。S 式に慣れてしまえば、プログラミングの本質に迷うところなく単刀直入に入ることができるのである。今日のようにシステム開発サイクルが短くなっている状況では、言語設計のうちのある部分をさっと省略できることはありがたい。

文字で書く言語は所詮一次元の文字列である。それに最低限の内在的二次元化機能をもたせただけの S 式は言語構造化を最低保証しているだけである。筆者はそういった意味で「S 式は言語以前の言語、前言語である」と呼んでいる。これが Lisp の強みになっていることは、多くの人に理解されていないと思う。今日、XML がこれほど広く受け入れられているのは皮肉である。

## [エピソード 2] 老子から？

```
Tao produces nil.  
nil produces atom,  
atom produces S-expression, and  
S-expression produces ten thousand things. ...
```

## 3.2 解釈実行型言語

最近のコンパイラ技術の著しい発展により、解釈実行型言語の旗色はますます悪くなっているように見受けられる。しかし、解釈実行型言語の本質的な良さは「インタープリタは最良のデバッガ」であるという口承に最も適切に表現されている。コンパイラ言語では、デバッガは独立した大きなシステムとしてつくらないといけないが、解釈実行型言語では、インタプリタ自身がすでにデバッガである。ボトムアップにプログラムをつくることはもちろん、ボトムアップにデバッグすることも容易である。

しかし、単に解釈実行型言語であるということより — それだったら BASIC もそうだということになる —, Lisp の場合、前言語 S 式がプログラムを表現すると同時に、プログラムが扱う対象データも表現していること、すなわち次に述べる “Program as Data” が Lisp プログラミングの生産性の高さに本質的に効いている。

### 3.3 Program as Data

これは、プログラムとして書かれた S 式をそのまま処理対象のデータとして扱えるという意味と、データであった S 式を (突然) プログラムとして解釈することができるという意味の両方を表している。このことは、Lisp が発表された当初、自己書き換え可能なプログラミング言語、つまり成長するプログラムを書ける言語として大いに喧伝された。

“Program as Data” はとりもなおさずフォン・ノイマン・アーキテクチャの重要な特質である。J. Backus の Turing 賞受賞講演 [7] により、プログラムとデータが同じ線を流れることが、性能向上を抑止するフォン・ノイマンのボトルネックとして槍玉にあげられた。反面、自己言及や自己書き換えという重要な機能が忘れ去られた。Lisp は高級言語でありながら、ノイマンの蓄積プログラム方式を残した最初の言語である。いずれにせよ、自己書き換え能力をもつ言語にはある種の「創発」をもたらす可能性がある」と筆者は信じている。

このような大仰なことを言わずとも、“Program as Data” はもっと深く広く、Lisp の生産性を上げるのに寄与している。本格的なプログラミング環境の先駆けとして有名な InterLisp [8] は、Lisp であることがそのままプログラミング環境につながったといえるシステムである。プログラミング環境とはプログラムを処理対象として気持良く扱えるようにしたシステムにほかならない。Lisp には最初からその素質があり、InterLisp は、その素質を早い時期にしゃぶり尽くした。たとえば、S 式をそのまま編集できる Lisp 構造エディタ、シンボルのミススペルを自動修正する DWIM (Do What I Mean)、 $a+45*c$  のように算術式と解釈できるようなシンボルを  $(+ a (* 45 c))$  と善意で自動翻訳してしまうフィーチャ、エラー発生時にただちにソースの位置がエディタに出るなど、これでもかという親切ツール群が揃っていた。InterLisp はプログラミング言語のマニュアルが電話帳の厚さになった最初の例でもあった。

InterLisp は圧巻であったが、多かれ少なかれ、Lisp システムはすべからくプログラミング環境の特質を天性のものとして備えている。Lisp の生産性の高さ (これまでに聞いた人々の印象を総合すると、たとえば C 言語に比べて数倍は高い) は、結局、そこに行き着くともいえるが、その根底に “Program as Data” という基本原理があることは、強調してもしすぎることはない。

### 3.4 動的言語

解釈実行型、かつ “Program as Data” であることと、ほぼ同義といってもいいが、Lisp はプログラムの実行時まで呼び出すべき関数の実体が決まらない動的束縛 (late binding ともい

う)の言語である。データ型についても寛容であり、実行時にデータ型をCPU負荷を伴って調整することを厭わない。最近のプログラミング言語のコンパイラ技術の著しい進歩は、静的な宣言を言語にどんどん採り入れていく傾向を押し進めている。実際、そうすることにより速度性能が上がり、プログラムの形式的な誤りを生じにくくしている。ソフトウェア工学的には動的言語はすでに出る幕をなくしたようにも見える。

しかし、本当に動的言語を捨て去ってよいのだろうか。動的言語の最大の利点は柔軟性である。たとえば、実行時にプログラムの一部の関数実体を変更しても、プログラムは動作し続けることができる。一時的には、古い関数実体と新しい関数実体が並行して動いていることもあり得る。それが致命的の矛盾を起こすような変更でなければ、システムは稼働したまま、なだらかに旧バージョンから新バージョンへが移行していく。いわゆる on-the-fly のアップデートである。また、実行時に、ある関数にプログラムのプロンプトを入れて、動作を観察をするといったことも容易である。

オブジェクト指向言語でオブジェクトのクラス定義が動的に変更されていくような動的オブジェクトが話題になったこともあるが、このようなことを可能にできるのも動的言語の特徴である。

### [エピソード 3] またも老子から？

<http://www.edepot.com/taohumor.html> (sorensen@ecse.rpi.edu) より

```
The tao that can be tar(1)ed  
is not the entire Tao.  
The path that can be specified  
is not the Full Path.  
We declare the names  
of all variables and functions.  
Yet the Tao has no type specifier.  
Dynamically binding, you realize the magic.  
Statically binding, you see only the hierarchy.  
Yet magic and hierarchy  
arise from the same source,  
and this source has a null pointer.  
Reference the NULL within NULL,  
it is the gateway to all wizardry.
```

システムが環境条件の変化に伴い、走らせたまま修正や更新ができることは、無停止でサービスをしないといけない応用システム（たとえば、電話システム、24時間データサービスシステムなど）では重要な必要条件である。Lispは、この性質を生まれながらに持っている。応用事例に、この性質のおかげでLispが積極的に採用されたものが出始めている。

動的言語は、このほか、人工知能研究のように、問題対象の解析が十分でないときの実験プログラムやラピッドプロトタイピングにも有効である。これまで Lisp が研究室での実験用言語としてよく使われてきたことがそれを物語っている。

**[エピソード 4]** Gary Entsminger: The Tao of Objects, M&T Publishing, 1991 から

Ancient Chinese philosophers believed in a unifying reality called the Tao. These philosophers of the Tao, or Way, emphasized that the world is not a static entity, but a dynamic process. The objects that compose the world are reflections of that process — in flux and ever-changing. Thus, the ability to model or capture an object's essence is illusory; we really only capture a bit of it for a short while.

A computer program is a process, a tool, a way to model, capture, or simulate part of the world. In traditional programming, static concepts impose themselves on a dynamic world. It becomes difficult to model the world or modify the model. Object-oriented programming is a major step toward making programming and the programs themselves dynamic. Using object-oriented programming techniques, you can design programs that are better suited for modeling a dynamic world, create and destroy complex processes at both compile time and run time, and design programs as if they evolved (which they do).

これは動的プログラミングの必要性をオブジェクト指向という枠組で的確に表現している。

Lisp が動的言語であることは次の自動メモリ管理を備えていることにもよるが、Lisp 自体が「動的」であることにも注意しておこう。Lisp 言語自体は世の中の変化につれ、ダイナミックに変転・流転していく。これは Lisp という言語のたくましさにつながっている。Lisp は生きている! 昔、ある人が、「Lisp はいくらでも大きくできる泥の玉だ。どこまで行っても泥の玉は泥の玉で見掛けは同じ。しかし、一番中心部にはキラリと光る真珠の玉がある。」というような意味のことを言った。プログラミングコミュニティの手にかかると、泥の玉、いや雪の玉は、このように自然に転がりながら大きな雪ダルマになるのである。

**[エピソード 5]** またもまたも老子から?

第 1 章の「道可道 非常道」の英訳 “What is said to be the Tao is not the true Tao.” の Tao を Lisp に置き換えると…,

What is said to be the Lisp is not the true Lisp.

### 3.5 自動メモリ管理

不要になったメモリブロックを回収し、再利用することである。通常、ゴミ集めと呼ばれる。最初の Lisp においてすでに実装された技術であるが、以来今日に至るまで 40 年以上にわたっ

て汲めども尽きぬ研究課題として君臨している。Lisp 以外の実用（商用）言語に実装された例は驚くほど少なかったが、ようやく認知度が高まったように見える。ゴミ集めの有用性が正しく一般に認識されてこなかったのは、名前が悪かったせいかもしれない。

ゴミ集めは、時間軸方向に仮想的にメモリ資源が無限に供給されるというメカニズムである。これと、空間軸方向に仮想的にメモリ資源が実装量以上あるという仮想記憶メカニズムとは、メモリ資源の拡大に関して時間軸と空間軸が異なるだけで、どちらも等しく重要な概念である。しかし、空間軸方向の仮想記憶だけが商用コンピュータに大々的に採り入れられ、時間軸方向のゴミ集めは研究室から大きく飛び出すことはなかった。大規模配列計算の需要が歴史的に先行したためとも考えられるが、今日、Java の登場をまって初めて、ゴミ集めの重要性が徐々に一般に理解されるようになったのは、やはり遅すぎたといわざるを得ない。

ゴミ集め、あるいは自動メモリ管理によって、動的なメモリ消費を行なうプログラムの作成がどれほど容易になるかの定量的評価がなされたという話は筆者は寡聞にして知らない。しかし、筆者および仲間の Lisper 連中の直観では、平均して 30 パーセントほどプログラミング能率が向上する。解くべき問題の解析が進んでいない段階でのプログラミングではこれより大きな値になるだろう。コンピュータの適用範囲の拡大とともに、雑多で動的な実世界的な問題を扱う局面がさらに増えている今日、ゴミ集めの重要性はますます高まっている。

しかし、それでもゴミ集めをしないで済ませたいという要求は依然として強い。なぜなら、問題の解析をきちんとすませないでプログラムを書くということに対する根本的な忌避が背景にあり、そもそもゴミ集めが本来の処理に対する意味のないオーバーヘッドでしかないからである。特に実時間性が要求される現代的なアプリケーションでは、ゴミ集めによって予測不可能な処理の中断が起こることは禁止的である。

ゴミ集めに関する研究の多くは、(1) ゴミ集めによる処理の中断をいかに回避するかということと、(2) ゴミ集めを続けることによって、仮想記憶が仮定しているようなメモリ使用状況から乖離し、メモリアクセスのスループットが下がってしまうのをいかに避けるかという、2 点に関するものである。研究室段階ではすでに多くの成果が出ている。

実際、筆者らは 1999 年に、専用マシン上ではあるが、割り込み反応遅延が最悪でも 150  $\mu$  秒以下、実時間応用を意識した状況では 30  $\mu$  秒以下の並行ゴミ集めを実装することに成功した [9]。これは Lisp のような記号処理システムを使っても、マルチメディア処理、ネットワーク制御、ロボット制御などが十分に可能であることを意味している。Java のゴミ集めに関しても、似たような成果が徐々につつある。もちろん、ゴミ集めのオーバーヘッドが処理時間の中に分散するだけであって、オーバーヘッドであることに変わりはない。しかし、ゴミ集めによって得られるプログラミング負担の軽減と、最近のプロセッサ性能の著しい向上を鑑みると、ゴミ集めは十分にペイする技術である。

大所高所に立って考えてみよう。今日のコンピュータの常識となっている仮想記憶メカニズムも、主記憶と二次記憶とのやりとりは本来の処理に対する単なるオーバーヘッドである。たまたま、I/O と CPU の並列動作の歴史が古いゆえに常識化しており、マルチプログラミング

において CPU を無駄にしなくて済んだので、オーバーヘッドがあまり意識されなかったにすぎない。当然、OS がそれを全面的にサポートした。

ゴミ集めが時間軸的仮想記憶とすれば、それに OS が — 特に実時間処理の文脈では — 関わらないで済むはずがない。上述した筆者らの実時間ゴミ集めでは OS (マイクロプログラムで書かれたカーネル) がゴミ集めの核心部に密接に連係した動作をしなければならなかった。そうしなければ、真の実時間性が達成できなかったのである。すなわち、実時間ゴミ集めは AP 層の問題ではない。筆者は実時間ゴミ集めが OS 技術の根幹に関わることもあまり認知されていないのではないかと恐れている。ゴミ集めが OS の基本的なトピックスの一つとなることによって、記号処理が将来のコンピュータの最も重要な領分になると筆者は断言したい。

#### 4 Lisp の特質ゆえに Lisp は使えない?

上に述べたような Lisp の長所は、それが望まれるところであっても、実行速度の低下とメモリ消費の増大という二つの代償を支払うことになる。

解釈実行型の動的言語では、実行時の束縛検索やデータ型チェックのオーバーヘッドを避けることができない。ゴミ集めのためのオーバーヘッドも必要になる。また、単純なゴミ集めのアルゴリズムを使うと、データが仮想記憶空間に散在することになり、使用メモリの局所性を仮定する仮想記憶メカニズムの速度性能が低下する。ゴミ集めの進歩の一つは、使用中のデータを局所的に配置するように移動 (またはコピー) する技術の開発であった。こうすると、移動中と移動中でないデータがポインタでリンクされる過渡的な状態が生ずるために、リンクたどりに小さなオーバーヘッドが生ずる。そもそも、通常の言語ではリンクたどり自身をあまり使わないから、リンクたどりがメモリアクセスの大きな割合を占める Lisp は、最初から速度性能では不利なのである。もっとも、これについても、典型的なリスト構造ではリンクを使わないですむ、cdr コーディングと呼ばれる方式が考案された。

記号の動的関係性、つまり動的な記号構造の実現にはリンク、すなわちポインタが不可欠である。ポインタは基本的にはアドレス空間を表現するのと同等のビット数を使うので、最も単純な線形リスト構造の実装を考えるとわかるように、ポインタなしのデータ構造に比べて最悪で 2 倍のメモリを消費する。今日の Lisp では、セル以外にオブジェクトや配列といったアドレス算術でアクセスするようなデータも多用するので、この最悪値は極端であり、複雑なデータ構造を扱うプログラムを書けば、C++ や Java に比べて明らかにメモリ消費量が大きいということはない。

実行速度とメモリ消費量はコンピュータプログラムの性能の二大指標である。これではかの言語に負ければ、商用アプリケーションをつくる時に最初から考慮の対象外となる。実際、過去 Lisp が広く使われなかった最大の原因はここにあった。

しかし、Lisp をめぐる状況は変化している。実行速度とメモリ消費量以外の、プログラムの性能指標の重要性が明確に抬頭してきた。たとえば、プログラムの生産性、保守性、可搬性、

実時間性 (これは実行速度とは独立の概念であることに注意), 耐故障性, 発展性, などなど, どれも昔から言われてきたものである. ハードウェア技術の指数的とも思われる進歩は, 皮肉なことに, プログラムの実行速度とメモリ消費量という性能指標の相対的な重要度を下げた.

一方で, ハードウェア技術の進歩は, ウェアラブルコンピュータのようなさらなる小型化をドライブしてきており, それに伴って資源制約型のプログラミング技術の必要性をリバイバルさせている. これは, 大型コンピュータのあとにパーソナルコンピュータが登場したときとよく似た状況である. 速度やメモリ資源の制約から解放されて, OS やプログラミング言語の健全な発展が進んでいたときに, (当時の) パーソナルコンピュータのような極端に資源制約の強い状況で, OS や言語が技術的に退行してしまう現象が見られた. E.W. Dijkstra は, パーソナルコンピュータが登場してまもなく「コンピュータ技術者は, また 20 年前の技術レベルの泥沼に戻っていくのか」と嘆いた.

1 円というコスト差が問題になる組込みコンピュータでは, 今日でも資源制約に合わせる事が至上目的のプログラミングが行なわれている. これとて, これまでに築き上げた技術蓄積があるので, 30 年前と同じレベルで開発が行なわれているわけではないが, 少なくとも Lisp がそこで使われる可能性は小さそうだ. しかし, 携帯電話に Java が入る時代である. 資源制約至上の方法論の生きる場はますます狭くなることは間違いない. 実際, ウェアラブルコンピュータにこそ Lisp を入れたいという研究者も出てきた. オリジナルなアイデアを含むので詳細はまだ書けないが, 筆者はパチパチと拍手を送りたい.

すでにコンピュータハードウェアは, 速度においてもメモリ容量においても十分に高性能である. むしろ, すでにかなり多くの基本プログラムや応用プログラムが要求している性能を超えた性能をもつようになった.

筆者がここで強調したいのは, この点である. すなわち, ハードウェア性能の指数的進歩は, コンピュータで解くべき, あるいはコンピュータで処理すべき問題の複雑さをいろいろな分野で超え始めた. 文房具として使うノートパソコンに必要な性能は, 筆者のように PC UNIX を使っているユーザにとっては, CPU クロックが 400MHz あればほぼ十分である (本当に文房具であればもっと遅くても十分だろう). これが 2GHz になっても性能を持て余すだけである. ノートパソコンの CPU はもはやクロック競争ではなく, 低消費電力競争へ変容してほしい.

## 5 Lisp が主役の一人に

筆者は, コンピュータでプログラムされる問題には, 人間や人間社会に本性的, あるいは固有の複雑さがあるという仮説を立てている. この仮説に従えば, 対象問題はコンピュータハードウェアの技術水準とは独立の固有の計算量的複雑さをもつはずである. ハードウェアが進歩すれば, コンピュータの性能がいろいろな問題の計算量的複雑さを次々に追い越していく.

プログラミング言語の間の性能差は、万能チューリングマシンの理論を持ち出すまでもなく定数倍である。その定数も研究者の精力が注ぎ込まれば、小さくなっていく。たとえば、当初 Java は C++ にくらべて一桁遅いとされていたが、最近では2倍程度、あるいはもっと小さい差になった。Lisp は遅いとされているが、それでも高々2~3倍程度である。この程度の差は、指数的なハードウェア技術の進歩の中で簡単に無視できるようになる。

たとえば、2年前に C++ で書いて実用的に使えた応用プログラムがあったとしよう。すると、それをいまのハードウェア上で Lisp で書いたら、要求性能を十分に満たして動く。では、これからそれと同規模の応用プログラムを書きたいとき、どのようにしてプログラミング言語を選択するだろうか。生産性など、プログラミングに関するほかの重要な性能指標が選択基準として浮上してくるはずである。選択基準の多くは、人間要因である。コンピュータはどんどん速くなり、記憶力（記憶容量）も増えるが、人間はそうはいかない。

経験者の多くが主張するように、Lisp の生産性がほかの言語に比べて5倍ほど高いとしよう。つくろうとしているプログラムの複雑さに対して、コンピュータの速度・メモリ性能面で不足が発生しないと見積もることができれば、Lisp を使わない手はない。Lisp はその生産性の高さからラピッドプロトタイピングに適した言語と言われていたが、今日の状況では、プロトタイピングとしてつくったプログラムは、機能仕様さえ満たせば、そのまま製品として使える。

これは Lisp に限ったことではない、Java でも、Prolog でも、ML でも、実験室レベルで快適なプロトタイピング言語として使われてきた言語が、より多くの応用分野で主流になる時代がやってくる。けだし、アセンブラ言語が高級言語に取って代わられたのは、これと似た理由であった。

## 6 事例

これまで、やや概念的に Lisp が主役の一人になる時代が到来したと述べてきた。これを裏付ける具体的な事例がすでに積み上げられている。

Lisp が人工知能用言語として商業的に喧伝されたのは1980年代である。筆者らが当時開発した TAO/ELIS という Lisp をベースにしたマルチパラダイム言語 (Prolog と Smalltalk+Flavors 風のオブジェクト指向を融合させていた) も、その一員としてビジネス路線に乗せられた。有名だったのは Symbolics, Xerox, Lambda などの Lisp マシンであったが、どれも最終的にビジネス的に失敗した。代表的なアプリケーションとしてのエキスパートシステムが大きな拡がりを得られなかったこともあろうが、結局コスト性能比で通常のコンピュータに太刀打ちできなかったことが失敗の最大の原因だった。

しかし、Lisp マシンが末期的な状況にあるとき、意外なニュースがあった。人工知能ブームの衰退とともに凋落していたはずの Symbolics マシンが、アニメ映画のコンピュータグラフィックス専用マシンとして売れ、かつ使われていたのである。10年ほど前になるが、ウォル

ト・ディズニーの映画「トロン」は、実写と（いまとなつては稚拙な）コンピュータグラフィックスが重ねられて話題になった。これに Lisp マシンが大々的に使われた。

注意すべきは、その理由が、Lisp がコンピュータグラフィックスに特に向いていたからではないことである。コンピュータグラフィックスを実写と重ねて映画をつくるという当時前例のないことを試みるときに、試行錯誤的で対話的なプログラミングが容易であったことが、まず着目された。もっと重要だったのは、コンピュータグラフィックスに必要な部品を、ボトムアップにいくらかでもかつ簡単に積み上げていけるという、Lisp 言語（+オブジェクト指向概念）の内在的成長力である。堅い設計図面なしにボトムアップに積み上げ（泥の玉を大きくすること!）が可能ということは、複数の人々がある程度独立に積み上げに協力できるということである。まさに今日の Linux のような集団的システムビルドアップが可能だった。こうして一時期、Symbolics の Lisp マシンは CG マシンになったとまで言われた。

少し古いこのエピソードは Lisp のポテンシャルを的確に物語っている。現在、商業的に成功している Lisp は、Franz の Allegro Common Lisp (ACL) ぐらいしかない。しかし、その利用事例などを見るかぎり（半分に割り引いても）、Lisp を使用することによって、利用者が先行利益を得ているように見える。つまり、乗り遅れないうちに乗るべきという教訓が示唆されている。以下は、少し古い情報だが、企業として Lisp を採用したユーザが、Lisp の生産性に関する数値を（感覚ではなく、実経験に基づいて）挙げたものである。なお、ここでの Lisp は、CLOS (Common Lisp Object System) を含む。

- 60,000 リンクを含む HTML の処理に 3 週間を要したが、同じことを C++ で書いたらその 3~5 倍の時間がかかる — Web オーサリング環境の Schema 社。
- データマイニングのアルゴリズム部分を Lisp で行なうのに全予算の 25~30 パーセントを要したが、C++ だったら 60~70 パーセントかかっただろう。トータルでは 4 倍のコストがかかり、利益が出なかった — Ascent Technology Inc.
- LSI のネットリスト変換の設定に以前は 6 ヶ月もかかり、よく顧客を失っていたが、Lisp を採用してからはこの設定に 2 時間しかかからなくなった — American Microsystems Inc.
- カナダ宇宙局の RADARSAT システムの制御ソフトの開発が当初の見積りの 2,000 万ドルではなく、700 万ドルで済んだ — Blackboard Technology.
- あるユーザは、Lisp ベースの Web ページオーサリングツールを使うことにより、わずか 2 人で 10 人規模の会社と同じ売上を達成することができた。

## 7 展望とむすび

上で述べたユーザレポートの中で定量的な表現はなされていないが、注目すべき特徴が四つある。これは、記号処理システム Lisp の将来展望に示唆を与えている。

一つめは、自動ゴミ集めがあるため、その分プログラミングの負担が少ないことを明言す

るユーザが多いこと。上に述べた筆者の直観に符合している。ゴミ集めの技術はそろそろ研究室の論文のタネだけの位置から飛び出すべきだし、実際に飛び出すであろう。今後は、並列ゴミ集め技術を CPU 設計の中に織り込んでいくという方向性が出てくることを期待したい。これは元の取れる投資である。

二つめは、Lisp+オブジェクト指向の動的言語性が、Lisp 採用の核心的理由だとする企業ユーザが多いこと。Gabriel が指摘したとおりの動的言語の復興が起こっている。現在のように速く動く世の中では、昨日の標準が今日はもう古くなってしまふ。こういう状況で、つねにアップデートされたサービスを行なうためには、変化に即応できる適応力が、ソフトにも、それを保守する人にも必要である。しかも、サービスを中断することなく適応作業を終えないといけない。たとえば、電話会社の中のデータベースの機能の更新は、通信サービスを中断させずに行ないたい。無停止メンテナンスが可能で安全な Web の動的サーバを簡単に実現したい [10]。走行中のプログラムの上で、関数やクラスを簡単に変更できる Lisp+オブジェクト指向はこれらに必要な技術を本性的に備えている。

ゴミ集めも動的言語も、性能を保証するためにはハードウェアサポートが必要である。ただし、専用マシンの開発といった本格的なものは必要ないだろう。性能の本質的改善は、ごくわずかのハードウェアサポートだけで成し得る。言い替えれば、今日の商用プロセッサはちょっとの儉約で、実は大きな損失を被っているという気がしてならない。

三つめは、応用システムを単一の言語でつくるのではなく、多種類の言語を使い分けてつくっているユーザが多いこと。中には Java は GUI, Lisp は頭脳と割り切っているユーザもいる。オブジェクト指向が当たり前になり、分散オブジェクト指向のための交信標準ができ始めたおかげで、適材適所で言語を選ぶことが容易になった。遠隔分散ではなく、ヘテロ結合した並列プログラミングというジャンルが当たり前になりつつある。複雑な環境のもとで、生物の種類が多様化したのと同様、マイナーだが能力の高い Lisp のような特徴的な言語の存在理由が再認識され、大いなるニッチを獲得するチャンスが訪れている。

最後の四つめは、少し前には想像できなかった分野で Lisp が実用的に使われ始めていることである。5 章で、Lisp の出番がやってくると述べたが、概念論ではなく現実にそれが進行している。1 分単位を相手にする、いわば準実時間の空港管理システム（飛行機の運行管理やゲートの割り当てなど — ただし、まだ飛行制御は含まない）、NASA の火星探査機 Mars Pathfinder の準実時間的な行動計画立案（状況に応じて電池を節約をしながら、当初の計画の 30 日の 3 倍近い探査活動を可能にした）、任天堂のゲーム開発、2000 年問題に対応していない Cobol コードの自動発見・修正システムなどなど。いずれも開発時間が逼迫している状況で Lisp の生産性の高さが発揮された例である。

これらを振り返ってみると、長い間、人工知能のための実験言語のような位置付けでしか見られなかった Lisp が、単なる人工知能ブームに乗った形でない根の張り方をしていることがわかる。しかし、日本ではまだ認知度が低い。企業で Lisp を使うことは担当者によほどの勇気がないと難しい [11]。米国から根が張り出してくるまで待たざるを得ないのだろうか。

さて、筆者はマルチパラダイム論を長い間主張してきた。マルチパラダイムにはパラダイムの接合という考え方と、パラダイムの融合という考え方がある [12]。筆者自身はパラダイム融合の研究を行なったが、現在、あるいは近未来の主流技術はパラダイム接合と言ってよい。つまるところ、プログラミング言語のメカニズム (関数型、論理型、手続き型、...) は所詮ソフトウェア全体から見ると、土壌ではあり得てもその上に豊かに実るものではない。メカニズム論争はプログラミング哲学の狭い分野に落ち着き始めた。どんな言語で書いても、大きくて複雑なプログラムは大きくて複雑なのである。

パラダイム接合の時代に向けて、異種言語間通信プロトコルの整備はさらに進むだろう。シリアルな記号表現としての S 式はそのための良いベースとなり得ると筆者は考えている。そういえば、RoboCup サッカーシミュレーションの選手たちが喋る言葉は S 式だ。また、パラダイム接合プログラミングの中で、Lisp が中枢的な仕事をすることは十分あり得る。

一時期、OS も言語も寡占化に収束するように思われたが、技術の進化は多様化への筋道を捨てなかった。どのような言語にも、それを支えるプログラミングコミュニティがある以上、ほかにないなんらかの魅力と特徴をもっている — 英語とフランス語の優劣を論ずるのは無意味である。パラダイム接合の時代には、それぞれの言語システムが固有のニッチを獲得する。Lisp もその一員である。Lisp がほかをすべて凌駕することはないだろうが、上に見た実用例からも伺えるように、Lisp がこれまでのようにシステムソフトウェアの範疇から外され続けることもあり得ないと筆者は考えている。

実際、Lisp に真の実時間性が備われば、アクティブネットワークや知的ロボットの制御など、元来システムソフトウェアの範疇に入る仕事が Lisp に可能になるだけでなく、その生産性の高さや動的言語の特質により、これらの分野で従来言語を凌駕してしまふことがあり得る。少なくともそれを筆者は努力目標として肝に銘じている。つまり、なんだかんだと言っても、つまりどう転んでも筆者は Lisp に転びたいのだ (押売はしないけれど...).

## 参考文献

- [1] 竹内郁雄: 記号処理システム Lisp の展望, 電子情報通信学会論文誌, Vol. J84-D-I, No. 6, pp.513-522, 2001 [招待論文]. (英訳: Ikuo Takeuchi: The Future of Lisp, Systems and Computers in Japan, Vol. 33, No. 6, pp.10-18, 2002 (Invited Paper).
- [2] P.J. Landin. The Next 700 Programming Languages, CACM Vol 9 No 3 March 1966.
- [3] R.W. Floyd. The Paradigms of Programming, CACM, Vol. 22, No. 8, 1979. [邦訳 有川節夫. プログラミングのパラダイム. 『ACM チューリング賞講演集』赤囁也, 共立出版, 1989.]
- [4] R.E. Griswold, J.F. Paoge, and I.P. Polonsky. *The SNOBOL4 qProgramming Language*, second edition, Prentice Hall, 1971.
- [5] A. Newell and H. Simon. Computer Science as Empirical Inquiry: Symbols and Search. CACM, Vol.19, No.3, 1976 [邦訳 横井俊夫, 横山晶一. 経験に基づく探求としての計算

- 機科学: 記号と探索. 『ACMチューリング賞講演集』赤囁也, 共立出版, 1989.]
- [6] <http://www.gwydiondylan.org/about-dylan.phtml>
  - [7] J. Backus. Can Computer Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. CACM, Vol.21, No.8, 1978 [邦訳 米澤明憲. プログラミングはフォン・ノイマン・スタイルから解放されるるか? 関数型プログラミング・スタイルとそのプログラム代数. 『ACMチューリング賞講演集』赤囁也, 共立出版, 1989.]
  - [8] S.H. Kaisler. INTERLISP, The Language and Its Usage. John Wiley & Sons, 1986.
  - [9] 竹内郁雄, 天海良治, 山崎憲一, 吉田雅治: 実時間記号処理システム TAO/SILENT の並行 GC, SPA'99, 日本ソフトウェア科学会, 1999.(正式の論文は, I.Takeuchi, Y. Amagai, M. Yoshida, and K. Yamazaki: A concurrent real-time garbage collector, 情報処理学会論文誌 プログラミング, Vol.44, No.SIG 16 (PRO20), pp.41-55, 2003.)
  - [10] 苫米地英人, 美馬秀樹. JLUGM — 日本 Lisp ユーザ会議 (3) 次世代 Web 技術としての動的サーバ技術 — マルチスレッド Lisp による可能性. bit Vol.32. No.11, 2000.
  - [11] 黒田寿男. JLUGM — 日本 Lisp ユーザ会議 (2) Lisp 応用事例: 自動車衝突実験データベースシステムの構築. bit Vol.32. No.10, 2000.
  - [12] 竹内郁雄. マルチパラダイム支援環境, 情報処理学会誌, Vol. 30, No. 4, 1989.