

Common Lisp について

今村 有祐
(日本電信電話株式会社 武蔵野電気通信研究所)

筆者は研究室の同僚とともに、Common Lisp とは骨格の異なる新しい Lisp オブジェクトを開発中である¹⁾。本報告ではこの立場から Common Lisp に対してコメントする。

1. Lexical scoping

Lexical scoping は局所変数の宣言とプログラマテキスト上の静的範囲に閉じさせようというもので、普通の算術言語ではとくく昔から常識だったものである。自由変数が無秩序に出てくる Lisp プログラムは、プログラミング作法の観点から好ましくない。Lisp が dynamic scoping をなるべく減らそうというのは正しい姿勢である。

Common Lisp の lexical scoping の目的と根拠はインタープリタとコンパイラの完全両立性であると発表されている。しかし、両立性を達成するだけならば、これは“過剰仕様”である。なにしろ、米国の Lisp 文化を支えた MacLisp がこの両立性に関してインチキであったのど少し反省と連が効きすぎたようだ。

筆者らは現在の TAO の原型であるミニコン上の Lisp (TAO/60) でこの両立性をもっと単純な方法で解決していた²⁾。それは、deep binding において変数の探索モードを静的範囲と越えるところを変え、外側では special 変数のみしか捜さないという方法である。このため、インタープリタでも special 宣言のない変数は静的範囲を越えてアクセスできない。これを筆者らは static scoping と呼んだ。TAO も当然この方法を採用している。通常の Lisp プログラミングで lexical scoping と static scoping はほとんど差がない。

しかし、Common Lisp の lexical scoping はこれだけではなく本質的にインタープリタと違える罫(あるいは鉛)といくつかは込んでいる。

・鉛1: lambda による完全な関数閉包 — コンパイラでは文脈による最適化が容易だが、インタープリタでは生真面目に閉包を作ってやる必要がある。このとき binding がスタック上にあると、それをヒープに移さなければならぬ。だから毎直にインタープリタと作る時昔ながらの a-list の登場となる。Lisp 実現技術の発展と逆行するというわけだ。しかも、lambda による完全な関数閉包が与える自身意味をもつようなプログラムは実際の Lisp プログラミングではほとんど現れない! 陽に関数閉包を作らせる従来の Lisp に比べて一体何が得になったのだろうか? また、special 変数が閉包に入らないのも実際の使用局面で不便を感じるものが多いうのではないだろうか。

・鉛2: go tag, exit point の lexical-scope dynamic-extent — 少なくとも筆者には高速の実現技法が思い浮かばない。

・鉛3: flet, flabels, fmacro — これもコンパイラでは別な方法があるが、インタープリタでは大変な重荷となる。たとえば、car と見て直ちに基本関数の car を思ってしまうからである。実際のところ、これも Common Lisp の仕様上の他の部分 (symbol-function, form の car は評価しない...) と整合性を保つたまま高速に解釈する実現技法はどんなものなのだろうか? これも筆者に

は思い浮かばない。(もちろん、多重プログラミングの拡張にも耐え得る技法でなければならぬ。)

ただし、こちらの鉛はコンパイラにとって、コンパイル時には多少の手間を伴うが、生成されたオブジェクトコードにとっては鉛にはならないことに注意しよう。また逆に lexical scoping によってより高速のオブジェクトコードが生成されるということもないことにも注意しよう。すなわち、lexical scoping は言語仕様上の多少の美しさをもたらしたが、ほとんど一斉的にインタープリタにだけ重い鉛を仕込んだ。上に挙げた鉛はいずれも「1人のために100人が迷惑する」という性質のものである。

2. Setf

Setf は Lisp にあける汎用代入機構として米国ではすでに高い評価を受けている。しかし、筆者らが開発中の TAO では setf よりさらに汎用的で強力な代入機構を導入したため、setf は極めて色退せて見える。たとえば setf はマクロという同様の計算であるのに対し、TAO の代入機構は LIFO に基づく通常の計算機構と逆行したもう一つの(要する extent を持つ)直接的計算機構である。このため setf では不可能な代入が容易に行なえる。詳細は文献③に譲るが、以下に少し例を示す。(TAO では代入を左ガッコの右に続けて打つ、タコロンで表す。)

```
(:(cond (flag (car x)) (+ (cadr x))) 1234)
```

```
(:(contents dir) nil)
```

```
(ただし、ここぞ (defun contents (x) (nthcdr 10 x))
```

このほかにも、直接的計算機構であるが故に、C 言語風に

```
(:@(p++) (-- q))
```

といった参照後(前)自動増加(減少)などと代入機構を容易に組み合わせることができる。さらに、TAO で自己代入と呼んでいる機構も自然に実現できる。自己代入は

```
(::fn ... (gn ... :x ...) ...)
```

と書き、x の評価に副作用がなければ

```
(:x (fn ... (gn ... x ...) ...)
```

と等価である。これを使うと

```
(::max :max-temp current-temp)
```

; 最高温度計

```
(::or :(status file) 'deleted)
```

; file の status が nil だったら

; deleted にする

```
(::+ :(table i j k) n)
```

; 配列要素のインクリメント

; (配列要素の参照計算は1回)

など極めて快適なプログラミングが可能になる。これは一度使うともう止められない計算機構の一つである。TAO ではこれをマイクロコードではなくオーバーヘッドなしに実現しているが、TAO(-like)の言語を汎用機上で実現中の橋村恭司君(筆者の同僚の1人)によると、multiple-value の実現技法の中にオーバーヘッドを吸収できるので、TAO の代入機構は鉛を仕込まずに実現可能とのことである。

Setf は TAO では (: ...) に受替すべからざる例外を除いてそのまま動く。例外は putprop と行なう (setf (get ...) ...) ぐらいである。これだけは setf と同じマ

7. 尾帰が必要だろう。いずれにせよ setf はベストではない。

3. keyword Parameter

これを統一原理と押し通したという意味で高く評価出来るが、慣用語語とにサッと切り落とした人工言語の匂いがする。実際のシステムでは慣用語語、すなわち nickname がまたさる視ゆえるのだろうが…。対話言語の性格が薄まったことは否めない。しかし、keyword parameter は Common Lisp の本質的な短所でも長所でもないだろう。ただ、インタープリタにとって重いことは確かである。

4. Control Structure

MacLisp 以来の伝統を色濃く受け継いでいる。ループなどについては、Lisp 以外の言語でいろいろ議論があったのだから、もう少し open-minded に新しい整構造を採用してもよかったと思う。

5. Package

Symbol の package への帰属関係を present と owned という 2 つの概念に分離しているが、これが必要以上に話を複雑にしている。両者を分離して考えないといけない状況がなまじしもあるというのは理解できるが、実面的により有益とは思えない。TAO では owned だけに絞っている。use-package 時の conflict エラックは現在のところ行なっていない。(行なったほうがよいことが判明した時とで再考)

6. Sequence

これは一種の overloading であるが、オブジェクト指向計算機構があるのだから type polymorphism へ吸収したほうがスマートであると思う。しかし、概念はよく整理されている。

7. 多重プログラミング

現在の Common Lisp ではなんの規定もされていない。多重プログラミングへの拡張を考えると、Common Lisp の実現はほろいような新釈を受けよう。Shallow binding には問題が多過ぎる。Package 内の protection 機構の導入も必要になる。Process 内の共有変数と Common Lisp の lexical scoping の採内どう実現するかも問題だろう。

これから Lisp プログラミングではエキスパートシステムにしろ CAD にしろ、大規模なデータベースを 1 つの Lisp 環境に抱えるものが多くなる。こういう場合、複数のユーザが 1 つの Lisp 環境の中で仕事を出来ることが望ましい。すなわち、アプリケーションが大規模になるほど、資源の有効利用からいって、1 つの Lisp 環境での多重プログラミング、多重ユーザをサポートする必要性が増す。現在の Common Lisp の仕様で、複数のユーザが同時に使えるようなパッケージソフトが作れるのかどうかは 1 つよくわからない。

8. オブジェクト指向プログラミング

Common Lisp は現在オブジェクト指向プログラミングを導入する方向で改訂中

とあると聞くが、オブジェクトのインスタンス変数をどう考えるかで多少混乱するのではなからうか？ 少し考えればわかるが、インスタンス変数を lexical 変数とすると、インスタンスは一種の関数閉包になる。しかし、メソッドがインスタンス変数に自由にアクセスできるようにするためには、メソッド中のインスタンス変数参照がインスタンス（クラスではない！）の lexical scope 内になければならない。これは、インスタンス生成のたびに、インスタンスに lexical に含まれたメソッドを複製しないといけないことを意味する！ つまり各インスタンスが全部固有のメソッドをばら下げるということになる（少なくとも概念的には）。ではインスタンス変数を indefinite scope, indefinite extent とするとどうか？ 残念ながらこれでは、メッセージ伝達の入れ子と通じて、他のインスタンスのインスタンス変数が兼通して見えてしまう。そのためインスタンス変数の参照を Loops 風にしないかぎり、scoping rule に何らかの変更を余儀なくされると思う。考えてみれば、Flavor 風のオブジェクト指向プログラミングは本質的に lexical scoping に合っていないのである。

9. インタープリタとコンパイラ

ほとんどの人が Common Lisp はコンパイルして使う言語であるということに意見が一致している。そして、コンパイラは非常時デバッグのためのつり足しであるという米国人が多い。そのため、多くの Common Lisp の実装でインタープリタが極端に速い。Lexical scoping でインタープリタとコンパイラの両立性をとるといつていたのは、これでは実質的に非両立であると言わざるを得ない。これは大いなる自己矛盾である。

筆者らの TAO は "実質的に" lexical scoping と同等であり、インタープリタとコンパイラの両立性を最大限保証しながら、現在のどの Common Lisp コンパイラよりも速いインタープリタを持つ。もうインタープリタの時代ではないというのは Lisp 実現者の怠慢である。

以上、Common Lisp に対して批判的なコメントを述べたが、筆者らが Common Lisp を否定しているわけではないことを強調してみよう。Lisp プログラムの移植というも最大の苦勞とする I/O については、TAO を完全に Common Lisp に含めて改定中である。このほか数値表現、関数名などは極力 Common Lisp に合わせるようにしてある。どうしても相容れないところは、Common Lisp Package を作って対処することになるだろう。いずれにせよ、TAO は Common Lisp で書かれたプログラムは吸収する。しかし、TAO の全機能（オブジェクト指向プログラミング、論理型プログラミング）を使った方が明らかに生産性と性能の上がるものについては Common Lisp の枠にこたわらうもりはまったくない。

文献

- [1] Takenuchi, Okuno, Orato: A New List Processing Language TAO and its Implementation [投稿中]
- [2] 大里, 杉内: 会話型 Lisp システム TAO/60 通研研究実用化報告 vol.31 No.11 (1982)
- [3] 大里, 杉内, 奥乃: TAO における代入計算機構・記号処理研究会資料 31-2