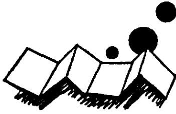
解説

Common Lisp†



湯浅太一††

1. はじめに

ここで取り上げるCommon Lispは，Lispの国際的標準化を目指して設計されたLisp仕様である．Lispシステムの画一化を図るものではなく，Lisp言語レベルの共通仕様を与えることによって，Lispプログラムの移植性を飛躍的に向上させることを目標としている．以下では，設計の経緯，仕様の概要，実現上の問題点，そして現状と将来の展望について解説する．

2. 設計の経緯

Common Lisp設計の動機は，MacLisp系Lispの言語仕様統一にあった．1970年代の始めまでは，MacLisp[1),2)]の開発機関はMIT，対象機種はDEC (Digital Equipment Corporation) 社のPDP-10，OSはITS (Incompatible Time-sharing System) と決まっていた．ITSはMITで開発されたOSで，その名のとおりPDP-10の標準的なOS (TENEXとTOPS-10) と互換性がなく，MITでしか動いていなかった．したがって，MacLispのユーザはMITとその周辺の人々に限られていたし，仕様変更の伝達もオンライン・ファイルに蓄積されたchange noteでことたりていた．

しかし1970年代半ば頃から，MacLispの流れを汲むLispがさまざまな機種上に実現され始めた．Lisp Machine[3),4)]を対象とするZetaLisp[5)]，VAX上のNIL[6)]，Lawrence Livermore研究所で開発中のスーパ・コンピュータS-1[7),8)]の上のS1-Lisp[9)]，パーソナル・マシーンPERQ上のSpice Lisp[10)~12)]などがそうである．これらのLispは，基本構造こそMacLispのものを継承していたが，それぞれ独立したプロジェクトで開発されたために，しだいに言語仕様が食い違い始め，相互の互換性さえ危うくなっていた．ZetaLispはMacLispを若干拡張した言語仕様でスタートしたが，PDP-10の18ビット・アドレス空間の制約や，汎用機による実行効率の悪さのためにMacLispでは実現できなかった機能が加わり，MacLispとはほど遠い巨大なものになってしまった．NILはZetaLispの影響を強くうけていたものの，MacLispで記述されていた数式処理システムMACSYMA実現上の要求などから独自の機能拡張がすすめられた．S-1 Lispのプロジェクトは，はじめはNILとの共同開発をすすめていたが，ハードウェアでLispをサポートするS-1の特徴を生かすために，いつしか独立したプロジェクトとなった．Spice Lispの開発をすすめていたカーネギー・メロン大学 (CMU) のSpiceプロジェクト[13),14)]は，PERQに限らず，マイクロ・コード可能なパーソナル・マシン一般を対象としていたので，Spice Lispも移植性を強く考慮して設計されていた．ZetaLispの機能のうち，複雑すぎるもの，移植性を損なうものを除去することによって言語仕様を固めていった．

第一回Lisp会議[15)]が1980年に開かれた．参加者の最大の関心事はLispの標準化であった．John McCarthyがLispを考案したのが1950年代の末だから，20年以上を経てやっとLisp会議が開かれたことになる．この間に，数え切れないくらいのLispが登場し，言語仕様も千差万別，インプリメンタごとに仕様が異なるといった状態になってしまっていた．MacLisp系だけをみても，上のような状態だったのである．次々と登場する人工知能や数式処理関係のソフトウェアの普及にとって，これは大きな障害である．同じく1980年には，米国の人工知能関係のプロジェクトの大部分をサポートしているDARPA (Defence Advanced Research Projects Agency) がLisp標準化のためのワーク・ショップを開いている．この時点でLisp標準化のお膳立てはそろっていたと言えよう．

Common Lispの設計作業は1981年11月に始まっ

---

† Common Lisp by Taiichi YUASA (Research Institute for Mathematical Sciences, Kyoto University, Japan).
†† 京都大学数理解析研究所

表-1 Common Lisp 設計グループ(文献[26]による)

Alan Bawden[2]	*Richard P. Gabriel[5,6]	William L. Scherlis[1]
Rodney A. Brooks[3]	Joseph Ginder[1]	Richard M. Stallman[3]
Richard L. Bryan[2]	Richard Greenblatt[7]	Barbara K. Steele[1]
Glenn S. Burke[3]	Martin L. Griss[8]	*Guy L. Steele Jr.[1]
Howard I. Cannon[2]	Charles L. Hedrick[9]	William van Melle[10]
George J. Carrette[3]	Earl A. Killian[6]	Walter van Roggen[1]
David Dill[1]	John L. Kulp[2]	Allan C. Wechsler[2]
*Scott E. Fahlman[1]	Larry M. Masinter[10]	*Daniel L. Weinreb[2]
Richard J. Fateman[4]	John McCarthy[5]	Jon L. White[10]
Neal Feinberg[1]	Don Morrison[8]	Richard Zippel[3]
John Foderaro[4]	*David A. Moon[2]	Leonard Zubkoff[1]

1. Computer Science Department, Carnegie-Mellon University, Schenley Park, Pittsburgh, Pennsylvania 15213.
2. Symbolics, Inc., Cambridge, Massachusetts 02139.
3. Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139.
4. Computer Science Division, Department of EECS, University of California, Berkeley, California 94720.
5. Computer Science Department, Stanford University, Stanford, California 94305.
6. University of California, Lawrence Livermore National Laboratory, Livermore, California 94550.
7. Lisp Machines Incorporated (LMI), Cambridge, Massachusetts 02139.
8. Department of Computer Science, University of Utah. Salt Lake City, Utah 84112.
9. Laboratory for Computer Science Research, Rutgers University, New Brunswick, New Jersey 08903.
10. Xerox Palo Alto Research Center, Palo Alto, California 94306.

た. ZetaLisp, NIL, S-1 Lisp, Spice Lisp のメンバが集まり,第一回の会合を開いている.設計作業には,後に Standard Lisp[16], Franz Lisp[17], さらには MacLisp のライバルとも言われる InterLisp[18)~20)] のメンバまで参加している(表-1 参照).設計の中心となったのは,当時 CMU にいた Guy L. Steele Jr. で,彼は MIT 時代に並列ごみ集めに関する論文[21]で ACM 学生論文賞を受賞したり,SCHEME[22]とそのチップ[23]を設計したことで知られている.また末期の MacLisp のメンテナンスを一人で引き受けていた経験もある.Steele が試案を作成し,設計協力者がこれについて議論を交し,次の試案に反映させていった.意見の交換はもっぱら ARPA ネットの電子メイルを通じて行われた[24].設計に要した2年半の間に交換されたメッセージは3000通,550万文字に及ぶ.可能な限り設計グループの同意をとり,必要となれば ARPA ネットで多数決をとっている[25].

設計の基本方針は1982年までには固まっており,第二回 Lisp 会議で次のように報告されている[26].

- 特定の機種に依存しない設計であり,プログラムの移植性が高いこと
- コンパイラとインタプリタに同一のセマンティックスを採用する
- ツールについては規定しない
- 言語の表現力が豊かであること
- 既存の Lisp, 特に ZetaLisp, MacLisp, InterLisp との互換性が高いこと
- 効率の良いコンパイラが実現できること
- 仕様が安定していること

仕様の大筋もこの時点でほぼ固まっていたが,まだ不完全なもので,細かな点で最新の仕様と食い違う点が多い.事実,会議録中の例のほとんどは,いまとなってはそのままでは動かない.設計協力者の第二回の会合がこの Lisp 会議を利用して開かれ,残った詳細設計は,Steele を始めとする5人の「キー・デザイナ」に託された(表-1 に*印で示す).その後いくつかの試案[27),28)]を経て,1984年には仕様書が完成し,現在 DEC 社から "Common Lisp" というタイトルで出版されている[29].

3. 仕様の概要

Common Lisp は Lisp には違いない.近代的な

Lisp に共通する特徴はもちろん備えている．特に MacLisp 系の Lisp で書かれたプログラムの多くは変更なしに Common Lisp でも使える．たとえばフィボナッチ数列（1, 1, 2, 3, 5, …）の第 n 番めの要素をもとめる関数 fib は再帰呼び出しを使えば

```
(defun fib (n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (t (+ (fib (- n 1))
              (fib (- n 2))))))
```

と定義できる．（つまり，n が0あるいは1であれば fib の値は1，そうでなければ $n-1$ と $n-2$ の fib の値をそれぞれ求め，その和が fib の値となる．）

以下では Common Lisp の最も基本的な部分を，特徴的な点に注目して説明することにする．

3.1 フォーム

Lisp でフォーム (form) といえば，評価の対象となる式を意味する．Common Lisp のフォームは次の三つに大別される．

- 数値や文字列のような，自分自身を値とするもの
- 変数名を表わすシンボル
- リスト・フォーム

さらにリスト・フォームはそのリストの第一要素によって

- 関数呼び出し
- マクロ・フォーム
- スペシャル・フォーム

に分類される．関数呼び出しのフォームは

（関数名　引数…引数）

または

（ラムダ式　引数…引数）

に限られ，各「引数」が順次評価されて，それらの値が実引数 (argument) として関数にうけわたされる（ラムダ式については後述）．「関数名」は評価されず，したがって呼び出す関数の名前はコンパイル時に静的に決定できる．関数名を実行時に動的に決定したければ funcall などの関数を用いる．

(funcall 関数　引数…引数)

は，(funcall が関数だから)「関数」とすべての「引数」がまず評価され，「関数」の値を関数，あるいは関数名だと思ってそれを呼び出す．

スペシャル・フォームとは，**表-2** にあげた「スペシャル・フォーム名」で始まるリスト・フォームで，Common Lisp の基本的文法を与えるものである．たとえば，ALGOL 系言語の if～then～else～ に相当するものとして if がある．

```
(cond ((= n 0) 1)
      ((= n 1) 1)
      (t (+ (fib (- n 1))
            (fib (- n 2)))))
          ⇩
(if (= n 0) 1
    (if (= n 1) 1
        (+ (fib (- n 1))
           (fib (- n 2)))))
```

図-1 cond のマクロ展開

表-2 スペシャル・フォーム

block	if	progv
catch	labels	quote
compiler-let	let	return-from
declare	let*	setq
eval-when	macrolet	tagbody
flet	multiple-value-call	the
function	multiple-value-progl	throw
go	progn	unwind-protect

(if フォーム1　フォーム2　フォーム3)

というスペシャル・フォームは，もし「フォーム 1」の値が nil（真偽値の偽に相当）であれば，「フォーム2」は評価せずに，「フォーム3」だけ評価する．逆に nil 以外の値であれば「フォーム2」だけを評価する．

マクロ・フォームは原則的には，一度マクロ展開されて，その結果が評価の対象となる．先の fib に現われた cond は（システムに組み込みの）マクロであり，(cond …) の部分は if を使った**図-1** のようなフォームにマクロ展開される．マクロはユーザが追加，変更できるのに対して，スペシャル・フォームは表-2 の24個が固定されており，追加，変更はできない．個個のインプリメンタが勝手に追加や変更をすることさえ許されない．スペシャル・フォームを固定し，その数を必要最小限にとどめることによって，Lisp 文法の単純化と安定を図ろうという意図である．これによって，Common Lisp で書かれたプログラムを解析するプログラムのポータビリティが保てるし，それらのプログラムの作成も容易になる．

3.2 ラムダ・リスト

MacLisp 系なので，関数定義には通常は defun を使う．

(defun plus1 (x) (1+ x))

は，厳密には plus1 という関数名にラムダ式

```
(lambda (x) (1+ x))
```

によって定義される関数，つまり引数 x に 1 加えたものを値とする関数を割り付けることにほかならない．Common Lisp ではこのように関数はすべてラムダ式で定義される．ラムダ式自体は，

```
(lambda ラムダ・リスト
        フォーム1…フォームn)
```

の形で，フォーム 1 からフォーム n までが本体であり，関数が呼び出されたときに順次評価される．ラムダ・リストは仮引数 (parameter) の宣言で，その一般形は

```
({変数}*
  [&optional {変数|
             (変数[初期値 [判定変数]])}*]
  [&rest 変数]
  [&key {変数|({変数|
         (キーワード 変数)}[初期値 [判定変
          数]])}*
         [&allow-other-keys]]
  [&aux {変数 | (変数 [初期値])}*]
)
```

である．ここで {A}* は空であるか，さもなければ A の複数個の並びであり，[A] はオプション，A|B は A または B を表わす．

&optional, &rest, &key, &allow-other-keys, &aux はラムダ・リスト・キーワードまたは単にラムダ・キーワードとよばれる．どのラムダ・キーワードよりも前に現われる変数は required で，関数が呼ばれたときに必ずそれらに対応する実引数がなければならない．特に，ラムダ・キーワードが一つもなければ，仮引数はすべて required である．&optional に続く仮引数は文字どおり optional で，関数呼び出しの際に，対応する実引数が与えられていなければ「初期値」にバインドされる．初期値の指定がないときは nil になる．「判定変数」は，対応する実引数が与えられたかどうかを判定するための変数で，与えられていれば t，そうでなければ nil を値とする．実引数のうち，required と optional に対応するもの以外は，リストになって &rest の次の仮引数にバインドされる．&aux は局所変数を定義するためのもので仮引数とは関係ない．例を挙げよう．(&key と &allow-other-keys は後述)

```
(defun foo
  (a b &optional c (d 10) (e 11 f)
       &rest g
       &aux h (i 12))
  (list a b c d e f g h i))
```

a と b は required. c, d, e はデフォルト値がそれぞれ nil, 10, 11 の optional. f は optional e の判定変数．h と i は局所変数で初期値は nil と 12. これらの仮引数と局所変数を一本のリストにして返すだけの関数である．foo への実引数の数によって foo の値がどのようになるかを以下に示す．(本文では Common Lisp の仕様書にあわせて，フォームとその値を ⇒ で示し，値が重要でない場合は省略する.)

```
(foo 1 2)
⇒ (1 2 nil 10 11 nil nil nil 12)
(foo 1 2 3)
⇒ (1 2 3 10 11 nil nil nil 12)
(foo 1 2 3 4)
⇒ (1 2 3 4 11 nil nil nil 12)
(foo 1 2 3 4 5)
⇒ (1 2 3 4 5 t nil nil 12)
(foo 1 2 3 4 5 6)
⇒ (1 2 3 4 5 t (6) nil 12)
(foo 1 2 3 4 5 6 7)
⇒ (1 2 3 4 5 t (6 7) nil 12)
```

&optional と &rest はほかの Lisp にもみられるが，&key は Common Lisp 特有のものである．たとえば次のような関数を定義できる．

```
(defun list2 (&key (first 1) (second 2))
  (list first second))
```

(first 1) は，list2 の実引数に :first なる「キーワード」があればその直後の実引数を first にバインドし，もしなければ first のデフォルト値として 1 をとることを意味する．したがって，

```
(list2 :first 10 :second 20) ⇒ (10 20)
(list2 :second 20 :first 10) ⇒ (10 20)
(list2 :first 10) ⇒ (10 2)
(list2) ⇒ (1 2)
```

となる．先に関数への引数は必ず評価されると書いた．実は上の例で :first や :second も評価されているのである．コロン“:”はこれらがキーワード・パッケージと呼ばれるパッケージに属するシンボルであることを表わす．そしてキーワード・パッケージのシンボルの値はそのシンボル自身と決められているのである．

ラムダ・リストに &allow-other-keys があれば，

実引数の中にラムダ・リストで宣言されていないキーワードがあってもかまわない.

```
(list2 :third 3 :second 20)
```

はエラーになるが,

```
((lambda (&key (first 1) (second 2)
              &allow-other-keys)
   (list first second))
 :third 3 :second 20)
⇒   (1 20)
```

となる.

3.3 Lexical Scope

Common Lisp は lexical scope を採用している. 正確に言えば,

- 変数のバインディング
- 局所関数と局所マクロの定義
- ブロック (return の戻り先) とタグ (go の行き先)

の有効範囲は, それらを設定 (establish) したフォーム内に限られる. 変数バインディングについてのみ説明しよう. 次の2つの関数を考える.

```
(defun foo (x) (baa))
(defun baa ( ) (print x))
```

(foo 1) を実行すると, foo の仮引数 x は1にバインドされるが, このバインディングの有効範囲は foo の本体のみに限られる. したがって, foo から別の関数 baa が呼ばれるが, baa の実行中は先の x のバインディングは無効で, baa 内で参照される x はグローバルな変数である. このようなバインディングを lexical binding と言う. lexical binding は ALGOL 系言語のユーザにとっては当然なものに見えるかもしれないが, Lisp の場合はむしろまれで, 大部分の Lisp ではインタプリタの効率を高めるために dynamic binding をとっており, baa の中で1がプリントされる. dynamic binding ならばスタックが使えるが, lexical binding だとメモリを消費する連想リスト (A-list. 変数バインディングに限って言えば, 変数名とその値のペアからなるリスト.) を使うのが普通である. lexical binding には, コンパイルすれば dynamic binding より効率が良くなる (少なくともその可能性はある) というメリットがあるものの, それをはるかにしのぐほどインタプリタの効率低下は大きい.

しかし dynamic binding はしばしば Lisp プログラムのエラーの原因となる. 上の baa の定義をみてその中の x が foo のそれであると判断するのは困難である. また dynamic binding はインタプリタとコンパイラの整合性を崩す原因となる. 通常の Lisp コンパイラは上の関数 foo をコンパイルするときに, x を局所変数として取り扱い, foo 以外からは参照できない領域 (スタックまたはレジスタ上) に割り当てる. すなわち, コンパイラは lexical binding を採用しているという不都合が生じているわけである. インタプリタも lexical binding を採用することでこの問題が解決されるのは言うまでもない.

dynamic binding ができないわけではない. 上の foo の定義でスペシャル宣言というのをつけて

```
(defun foo (x) (declare (special x))
  (baa))
```

とすると, x のバインディングは foo の実行中有効となり, この x の値が baa の中でプリントされる. MacLisp などではもともとインタプリタが dynamic binding を採用しているので, スペシャル宣言はコンパイラへの情報であって, インタプリタはこれを無視する. しかし Common Lisp の場合はインタプリタが必ずスペシャル宣言の有無をチェックしなければならない. インタプリタにとってかなりのオーバヘッドである.

3.4 関数閉包

Lisp には funarg 問題と呼ばれる歴史的な大問題がある[30),31)]. ラムダ式で与えられた関数の実行中に, 自由変数 (そのラムダ式の中ではバインドされていない変数) が思いもかけないエラーを引き起こすことがある. 簡単な例で説明しよう.

```
(defun down-foo (x)
  (down-baa '(lambda ( ) (1+ x))))
(defun down-baa (f &aux (x 100))
  (funcall f))
```

down-foo からラムダ式 (lambda ( ) (1+ x)) で表わされる関数が down-baa への引数として渡される. ラムダ式自身は x のバインディングを行っていないので x は自由変数である. これを書いたプログラマの意図としてはこの x は down-foo の引数のつもりであろう. ところが down-baa 内で (funcall f) が評価されラムダ式の本体が実行されるときには, 大部分の Lisp は x を down-baa の局所変数だとみなす. 関数がラムダ式を値として返す場合にも同じような問題が生じる.

```
(defun up-foo (x) '(lambda ( ) (1+ x)))
```

```
(defun up-baa (x) (funcall (up-foo 100)))
```

前の down-foo と down-baa のような場合を downward funarg, この up-foo と up-baa のような場合を upward funarg の問題と呼んで区別することがある. Common Lisp はこの両方の funarg 問題を関数閉包によって解決している.

関数閉包 (function closure) の概念自体はほかの Lisp にもみられるが, Common Lisp では関数閉包の生成された時点のすべての lexical な環境がその中に生きつづける. これは ZetaLisp や Franz Lisp のように, 関数閉包に含まれる環境をユーザが陽に指定するものよりもはるかに強力なものとなっている.

```
(defun init-count (x)
  #'(lambda () (setq x (1+ x))))
(setq count (init-count 100))
(funcall count) ⇒ 101
(funcall count) ⇒ 102
(setq x 0) ⇒ 0
(funcall count) ⇒ 103
```

これは関数閉包を使った例である. #'(…) は (function (…)) の略記で, (lambda () (setq x (1+ x))) で定義される関数に lexical な環境, この例では x のバインディング, を取り込んだ関数閉包を生成する. (init-count 100) によって関数 init-count の変数 x が 100 にバインドされ, このバインディングが init-count の実行が終った後も関数閉包の中に生き続けるのである. はじめの (funcall count) によって関数閉包が呼びだされ, この変数 x の値が 1 増加して 101 になる. 関数閉包の中に含まれる変数 x のバインディングはプログラムのほかの部分とはまったく無関係であり, 変数 x の値をセットしなおしても ((setq x 0)) バインディングはなんら影響を受けない.

関数閉包によって funarg 問題が解決されているのは明らかであろう. 始めに挙げた二つの例の場合には, どちらも '(lambda () (1+ x)) を #'(lambda () (1+ x)) に置き替えるだけでプログラマの意図通りに動くのである.

3.5 仕様に含まれない機能

Common Lisp にまったく含まれない機能には,

- グラフィックス
- マルチ・プログラミング
- オブジェクト指向プログラミング

がある. グラフィックスはそれ単独でもいまだに標準化が大問題であり, マルチ・プログラミングもまだまだ先の話である. オブジェクト指向については, ZetaLisp 流の flavor の採用をめぐってずいぶん議論されたようだが, 結局時期尚早ということで採用されなかった.

標準化を目指す限りは, 特定の OS や機種に依存しない設計であるのは当然で, OS のファイル・システムとのインタフェースなどは, かなり柔軟性をもたせた仕様になっている. どうしても依存しないわけにはいかない点は, 大筋を仕様で規定し, 詳細は個々のインプリメンタにまかせている. プログラミング・ツールに関してもあまり規定していない. ツールは使用する端末機や OS に依存するし, 現在のソフトウェア水準の下でツールを規定しても将来新しいツールの出現によって無意味になるかも知れない. 最小限必要なツールとして, エディタ, トレーサ, ステッパなどが仕様書に含まれているが, これらの具体的な仕様はほとんどなく, トレーサとはなにかぐらいの記述にとどめている.

4. 実現上の問題点

Common Lisp はコンパイラを重視した言語仕様である. インタプリタの実行効率を多少犠牲にしてでもコンパイルされたコードを速く, 小さくするという設計方針が貫かれている. 前節であげた lexical scope の採用や, special 宣言の取り扱いなどをみてもこれは明らかだし, またスペシャル・フォームとマクロの明確な区別もその現われである. インタプリタによる実行の場合, マクロ・フォームはマクロ展開されてから評価されるので, スペシャル・フォームに比べると効率が悪い. にもかかわらず, Lisp プログラムで頻繁に使われる cond さえも Common Lisp ではスペシャル・フォームではなくマクロなのである. 従来の Lisp と同様, Common Lisp でもマクロはコンパイル時に展開されるので, プログラムをコンパイルしてしまえばマクロの実行時オーバヘッドはなくなるからである. このような言語仕様の枠組ではインタプリタの効率にはおのずと限度がある. インタプリタはデバッグ・ツールのひとつと考えて, コンパイラの性能向上を考えるほうが賢明であろう.

Common Lisp の詳細設計を担当した 5 人のキー・デザイナはすべて, ハードウェアで Lisp をサポートしたり, あるいは Lisp 用のマイクロ・コードを書いたりといった, いわゆる Lisp 専用機に携わっていた. 少なくとも設計期間中はそうであった. このため

表-3 Common Lisp ベンチマーク

Benchmark	専用機			汎用機		
	Symbolics 3600	S-1 CL Mark IIA	Spice PERQ	VAX CL 780	DG CL MV 10000	Kyoto CL MV 10000
Boyer	11.99	10.03	134.79	46.79	29.3	11.36
Browse	30.8	10.2	359.63	118.51	59.91	29.08
Destruct	3.03	0.91	17.78	6.38	6.95	3.13
Traverse	49.95	30.1	490.6	161.68	45.86	44.24
Tak	0.6	0.29	4.7	1.83	0.89	0.41
Stak	2.58	4.31	13.5	4.11	3.09	2.22
Ctak	7.65	0.82	8.4	8.09	1.79	4.89
Takl	6.44	2.92	24.0	7.34	5.52	5.25
Takr	0.6	0.58	7.7	3.42	1.21	0.56
Deriv	5.12	4.99	71.8	13.76	5.6	5.36
FFT	4.75	1.44	59.0	32.69	62.78	1.57
Puzzle	13.89	1.82	75.14	47.48	138.2	7.25
Triang	151.7	62.06	1488.85	360.85	151.2	139.28
Fprint	2.6	——	20.0	3.94	2.35	2.07
Fread	4.6	——	26.0	7.24	4.65	2.55
Tprint	4.9	——	22.6	2.85	2.83	1.85

(単位は秒．Spice と S-1 Common Lisp は実時間，ほかは CPU 時間．Spice と S-1 のデータは昨年のもので文献 33) から引用．Symbolics, VAX Common Lisp, DG Common Lisp のデータは本年4月のもので Richard Gabriel から直接入手した．Kyoto Common Lisp のデータは本年5月のもので筆者が実測した．テストの詳細については文献 33) を参照されたい．)

に，Common Lisp の仕様は汎用機よりも専用機に適した仕様であると言われる．このことはキー・デザイナの一人である Richard Gabriel も指摘している[32]し，仕様の随所にそれを裏付ける個所が見られる．しかし Gabriel のベンチマークの結果をみるかぎり，汎用機のハンディキャップは実際にはあまり大きくないようである（**表-3** 参照）．専用機がまだその能力を十分発揮していないとも考えられるが，汎用機上においても高性能のコンパイラの実現を可能にする言語仕様を Common Lisp が与えていることの現われであろう．

以下では具体的に関数閉包に的をしぼってその実現方法を考えてみよう．まず，インタプリタが関数を実行する場合は，いつでも関数閉包を生成できるように準備しておく必要がある．これは lexical な環境を連想リストで表現すれば簡単である．関数閉包を，ラムダ式とそれが作成された時点の連想リストの組で表現する．関数閉包を実行するときには，この連想リストを環境だと思ってラムダ式を実行する．この方式だと，shallow binding[34),35)]という効率の良いテクニックは使えない．たとえ使えるにしても，非常に複雑な処理をともない，どの程度効率が向上するか疑問である[36)]．連想リストによる方式は実行効率は決してよくないものの，これまでの Common Lisp システムはすべてこの方式を採用している．

次に，前節に挙げた init-count という関数を例にとってコンパイラによる関数閉包の処理を考えよう．

```
(defun init-count (x)
  #'(lambda () (setq x (1+ x))))
```

init-count のコードは次のような動きをするのであろう．

（i） 仮引数 x のための特別な領域を確保する．

（ii） 受け取った引数をその領域に格納する．

（iii） ラムダ式に対応するコンパイル・コードとこの領域の組 (compiled closure) を init-count の値として返す．

そして，ラムダ式のコンパイル・コードはこの x のための領域の内容を1だけ増やすことになる．では x の領域はどこに，どのような形（データ）で確保するのだろう．ベクタ，つまり一次元の配列を使う方法が多いようである．init-count の例では関数閉包に取り込む必要があるのは変数 x（のバインディング）だけだが，一般には複数の変数を取り込む必要がある．各変数に対して，ベクタの決まった位置を割り当て，コンパイルされた関数閉包の中では，ベクタのインデックシングによって変数を参照する．つまり，上の(i)，

(ⅱ), (ⅲ)の各ステップは

（ⅰ） 長さ1のベクタ V を新しく生成する（関数閉包に取り込まれるのがxだけだから).

（ⅱ） V[0] に引数を格納する. (Common Lisp ではベクタの添字は0から始まる.)

（ⅲ） ラムダ式のコードと V の組を返す.

となる. ラムダ式のコードは

V[0] ← V[0]+1

を実行すればよい.

この方式の問題点は, ベクタを多用することにある. 関数閉包の処理に限らず, Common Lisp ではベクタおよび一般の配列を多用する傾向がある. 従来は, ベクタは一種の関数として扱われることが多かったが, Common Lisp では数値やリストと同じくデータである. ベクタを変数に代入することもできるし, 不用になればごみ集めの対象ともなる. ところが, ベクタの長さはベクタによって異なるので, 単に回収しただけでは, メモリ内に小さな, かつてベクタとして利用された領域が点在することになり, 大きなベクタを新しく割り当てることができなくなる. 使用中のベクタを一連のメモリ領域に移動させ, 大きな空き領域を確保するためのコンパクション (compaction) が望ましいが, コンパクションをするごみ集めには時間がかかる. 移動されるデータを指しているポインタをすべて書きかえる必要があるからである. この結果, ごみ集めが終るまで長い時間プログラムの実行が中断されることになり特にリアルタイムの応用には適さない. Baker の発案した方式[37]を使えば, プログラムの実行中に少しずつごみ集めをしてくれるので, この待ち時間はなくなる. しかし Baker の方式は Lisp のポインタをたどるたびにそのポインタを書きかえるべきかどうかを判定する必要があり, 汎用計算機で採用するとシステム全体の実行速度は著しく低下する[38]. マイクロ・コードや専用のハードウェアを使い, ポインタの参照とこの判定を並行処理することによって始めて実用に供すると考えられている[39].

関数閉包のコンパイルにはもう一つ問題がある. ある変数が関数閉包に取り込まれるかどうかの判定は, 関数閉包の生成の有無をコンパイラがチェックし, もしあればその関数閉包を構成するラムダ式を解析した後でないと行えない. 筆者のグループが作成した Kyoto Common Lisp のコンパイラはこのために2パスになっている. パス1で必要な情報を集め, この情報をもとにパス2でコードを生成する. 各変数をスタックやレジスタのような普通の領域に割り当てるのか, それとも関数閉包のための特別な領域に割り当てるのかを決定するためだけに, パスを一つ増やす必要があったのである. しかし多くの Common Lisp システムでは “1.5” パスのコンパイラになっている. 普通の領域と関数閉包のための特別な領域にアクセスするための二つの命令を用意しておく. コンパイラはとりあえず各変数が普通の領域に割り当てられると仮定してコードを生成していく. ある変数が関数閉包に取り込まれるとわかったら, それまでに生成したコードの中の, その変数を参照する命令を関数閉包のための特別な領域をアクセスする命令に置き替えるのである.

5. 現状と将来の展望

Common Lisp の設計作業と並行して, いくつかの Common Lisp システム開発プロジェクトが進行しており, そのほとんどがすでに稼働を始めている. **表-4** に現在入手可能な Common Lisp システムの一覧表を挙げておく.

Common Lisp は言語仕様としては ZetaLisp に匹敵するほど巨大である. フル・システムを一から作成するのは容易でない. このために, 開発プロジェク

表-4 Common Lisp システム一覧表

名　称	開　発　機　関	対　象　機　種
Symbolics	Symbolics	Symbolics 3600
Spice Lisp	CMU	PERQ
VAX Common Lisp	DEC	VAX 11 (VMS)
Kyoto Common Lisp	京大数解研と NDG	ECLIPSE MV VAX 11 (Unix) SUN U-Station
DG Common Lisp	Data General	ECLIPSE MV
TOPS-20 Common Lisp	Rutgers 大と DEC	DECSYSTEM/20
S-1 Lisp	Lawrence Livermore 国立研	S-1 processor

トのなかにはシステムの移植性を考慮しているものがある．たとえば，CMU の Spice プロジェクトでは，CMU パッケージなるものを用意している．Lisp (Common Lisp) で書かれた Spice Lisp のソース・プログラムがはいっていて，マイクロ・コードの部分を書くか，あるいはマイクロ・コードをシミュレートするプログラムさえ書けばいいようになっている．また Kyoto Common Lisp はC言語と Common Lisp で記述されていて，簡単な修正だけで移植ができるようになっている．米国の Lucid 社では，Guy Steele や Richard Gabriel の作成したポータブル・コンパイラ[40]をベースにした Common Lisp システムを開発中で，近い将来，さまざまな機種上に同社の製品が発表される予定である．Common Lisp の普及はもはや時間の問題と言えそうである．

Common Lisp は Lisp プログラムの移植性を強く考慮して設計されているが，決して完全ではない．ある Common Lisp システムで作成されたプログラムがただちに別の Common Lisp システムで動くとは限らない．その理由の一つは，仕様に含まれない機能を個々のインプリメンタが自由に追加できることである．実際，Symbolics 社の Common Lisp 上では優れた Lisp プログラムが動いているにもかかわらず，Symbolics 固有の機能を多用しているために，ほかの Common Lisp システムへは簡単に移植できないという問題がすでに生じている．もう一つの理由は，インプリメンテーションの方法や機種に依存する部分が仕様では明確に規定されていないことである．型宣言がそのいい例である．MacLisp の伝統を引き継いで，Common Lisp でも変数や関数に型宣言を付加することによってコンパイル・コードの実行効率を高めることができる．Kyoto Common Lisp では fib 関数に次のような宣言をつけると最適なコード（C言語のような Lisp より効率がいいと一般に考えられている言語で直接記述するのに匹敵するほどのコード）が生成される．

```
(proclaim'(function fib (fixnum) fixnum))
(defun fib (n)
  (declare (fixnum n))
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (t (+ (fib (- n 1))
              (fib (- n 2))))))
```

(proclaim ...) で fib が fixnum(有限精度整数)を引数にとり fixnum を返す関数であることが宣言され，(declare ...) で仮引数 n の値が常に fixnum であることが宣言される．どちらの宣言も Common Lisp で定義されているものであるが，これらの型宣言をコンパイラが実際にどのように取り扱うかといった詳細は仕様には記されていない．したがって，この宣言つきのプログラムがほかの Common Lisp システムでも同じように効率のよいコードにコンパイルされる保証はまったくない．また Kyoto Common Lisp の場合は fixnum は 32 ビット長だが，仕様では 16 ビット長以上であればいいことになっている．このプログラムがほかの Common Lisp システムで正しく動く保証さえないのである．

このような移植性の問題はプログラミング言語一般の宿命みたいなもので，世界中のすべての計算機が同じ仕様と同じパフォーマンスを持ち，そして同じ仕様と同じパフォーマンスを持つ OS を持たない限り解決できそうにない．とにかく，上に挙げたのは些細な問題に過ぎず，Common Lisp が普及すれば Lisp プログラムの移植性がこれまでとは比較にならないほど向上することは確実である．

6. おわりに

Common Lisp の生い立ち，概要，問題点を眺めてきた．どうも批判がましい話が多くなったが，Common Lisp は全体的に非常によく設計されているというのが筆者のいつわりのない意見である．筆者は以前は Standard Lisp を主に使っていた．Common Lisp (Kyoto Common Lisp) に移行した直後は，その仕様のあまりの大きさに戸惑ったこともあったが，ひとたび使い始めると，Lisp プログラムを書く上で必要なありとあらゆる機能がそろっており，しかも無駄がない．もはや Standard Lisp には戻れそうにない．

筆者の勤める研究所では，ZetaLisp や MacLisp で書かれたソフトウェアを，Common Lisp に移植して使っている．移植にはもちろん時間と労力を要し，コンパイルしなおすだけで終りというわけにはいかないが，比較的容易に移植できるようである．現在のところ，はじめから Common Lisp で書いたソフトウェアはあまり発表されていないので，仕方なく Common Lisp 用のツールなどは自前のものを使っている．しかし将来は Common Lisp を介して大いに Lisp ソフトウェアの交換が行われるものと期待している．

なお，本文中の筆者の意見の多くは，ともに Kyoto

Common Lisp を開発した萩谷昌己氏との議論を通じて得られたものである．また本稿執筆にあたっては，多くの方の意見や疑問，そして批判を参考にした．これらの方々に心から感謝する．

参考文献

1) Moon, D.: MacLisp Reference Manual. Revision 0, MIT Project MAC (1974).
2) Pitman, K.: The Revised MacLisp Manual, MIT/LCR/TR 295, MIT Lab. for Computer Science (1983).
3) Greenblatt, R.: The LISP Machine, Working Paper 79, MIT AI Lab. (1974).
4) Knight, T.: The CONS Microprocessor, Working Paper 80, MIT AI Lab. (1974).
5) Moon, D., Stallman, R. and Weinreb, D.: LISP Machine Manual, Fourth Edition, MIT AI Lab. (1981).
6) Burke. G. S., Carrette, G. J. and Eliot, C. R.: NIL Reference Manual, MIT/LCS/TR 311 (1984).
7) Correl, S.: S-1 Uniprocessor Architecture (SMA-4), S-1 Project 1979 Annual Reporr, Lawrence Livermore Laboratory (1979).
8) Hailpern, B. T. and Hitson, B. L.: S-1 Architecture Manual, Technical Report 161, Stanford University (1979).
9) Brooks, R. A. and Gabriel, R. P.: S-1 Common Lisp Implementation, Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, Pittsburgh (1982).
10) Fahlman, S. E., Large, J. and Cellio, J.: Spice Lisp User's Guide, Carnegie-Mellon University (1983).
11) Fahlman, S. E. et al.: Internal Design of Spice Lisp, Carnegie-Mellon University (1983).
12) Wholey, S. and Fahlman, S. E.: The Design of Instruction Set for Common Lisp, Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin (1984).
13) Ball, E., Barbacci, M. R., Fahlman, S. E. et al.: The Spice Project, CMU Computer Science Research Review (1982).
14) 斎藤信男：ソフトウェア開発環境とプログラミング環境のプロジェクト—Gandalf と Spice プロジェクト—, bit, Vol. 15, No. 1.
15) Conference Record of the 1980 LISP Conference, Stanford University (1980).
16) Marti, J. B., Hearn, A. C., Griss, M. L., and Griss, C.: Standard LISP Report, University of Utah (1978).
17) Foderaro, J. K. and Sklower, K. L.: The Franz Lisp Manual, University of California Berkeley (1982).
18) Teitelman, W. et al.: INTERLISP Reference Manual, Xerox Palo Alto Research Center (1978).
19) Teitelman, W., Bobrow, D. G., Hartley, A. K. and Murphy, D. L.: BBN-LISP: TENEX Reference Manual, Bolt, Beranek, and Newman, Inc. (1971).
20) Burton, R. R., Masinter, L. M., Bobrow, D. G., Haugeland, W. S., Kaplan, R. M., Sheil, B. A. and Bell, A.: Overview and Status of Inter LISP-D, Conference Record of the 1980 LISP Conference, Stanford University (1980).
20) Bates, R. L., Dyer, D. and Koomen, J. A. G. M.: Implementation of Interlisp on the VAX, Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, Pittsburgh (1982).
21) Steele, G. L.: Multiprocessing Compactifying Garbage Collection, CACM, Vol. 18, No. 9 (1975).
22) Steele, G. L. and Sussman, G. J.: The Revised Report on SCHEME A Dialect of LISP, AI Memo No. 452, MIT AI Lab. (1978).
23) Steele, G. L. and Sussman, G. J.: Design of LISP-Based Processors or, SCHEME: A Dialectric LISP or, Finite Memories Considered Harmful or, LAMBDA: The Ultimate Opcode, AI Memo No. 514, MIT AI Lab. (1979).
24) COMMON-LISP. MAIL, in ARPA net @SU-AI, ARPA.
25) Votes on the 1st Draft Common Lisp Manual, in ARPA net FEEDBACK. MSS @CMU-ZOU. (Nov. 1981).
26) Steele, G. L. et al.: An Overview of Common LISP, Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, Pittsburgh (1982).
27) Steele, G. L.: Common Lisp Reference Manual, Laser Edition, Carnegie-Mellon University (1982).
28) Steele, G. L.: Common Lisp Reference Manual, Mary Poppins Edition, Carnegie-Mellon University (1983).
29) Steele, G. L. et al.: Common Lisp, Digital Press (1984).
30) Moses, J.: The Function of FUNCTION in Lisp, MIT AI Memo, 199 (1970).
31) McCarthy, J.: History of Lisp, History of Programming Languages, ACM Monograph Series, Academic Press (1981).

32) Brooks, R. A. and Gabriel, R. P.: A Critique of Common Lisp, Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin (1984).
33) Gabriel, R. P.: Performance and Evaluation of Lisp Systems, Stanford University (1984).
34) Baker, H. G.: Shallow Binding in LISP 1.5., AI Working Paper 138, MIT AI Lab. (1977).
35) Teitelman, W. and Masinter, L.: Shallow bindings in Interlisp-10, Note to Interlisp Users, Xerox Memo (1976).
36) White, J. L.: Constant Time Interpretation for Shallow-bound Variables in the Presence of Mixed SPECIAL/LOCAL Declarations, Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, Pittsburgh (1982).
37) Baker, H. G.: List Processing in Real Time on a Serial Computer. CACM, Vol. 21, No. 4 (1978).
38) Dawson, J. L.: Improved Effectiveness from a Real Time Lisp Garbage Collector. Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, Pittsburgh (1982).
39) Moon, D. M.: Garbage Collection in a Large Lisp System, Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin (1984).
40) Brooks, R.A., Gobriel, R.P. and Steel, G.L.: An Optimizing Compiler for Lexically Scoped LISP. SIGPLAN Notices 17, 6 (1980).

(昭和60年3月27日受付)