

解説



最近の Lisp 言語†

松田 裕幸††

1. はじめに

Lisp* が初めてプログラミング言語の世界に登場して約 30 年が過ぎようとしている。Lisp と Fortran はほぼ同時期に生まれ、Fortran が数値処理を目的として開発されたのに対し、Lisp は記号処理を初めから前提とした当時としては非常にユニークな言語であった。それでも最初は Fortran を記号処理可能なように拡張することや、当時ヨーロッパで勢いを増しつつあった Algol60 の仕様を考慮することなども考えていたようである。しかし、まもなく McCarthy らは quote と eval の発見によって今日われわれが使用している Lisp の原形を提案する¹⁷⁾。

eval の実現によってプログラムをインタプリティブに実行することが可能になり、同時にプログラムをインタプリートするためにプログラムを S 式としてネストするアイデアが生まれる。当時はコンパイラを先に作ることが念頭にあったようで、いわゆる括弧式も処理系作成の手間を簡単にするためのアイデアにすぎなかった。しかし、S 式の発見により、今日でも Lisp の最大の特徴の一つである、プログラムをデータとみなし、逆にデータをプログラムとみなすプログラムとデータの相互交換性を獲得する。これは他の言語にはない特徴で、Lisp が現在まで AI 分野で広く使われている理由の一つになっている。このあたりの歴史的概観は文献 14) 2.2, 2.3 節に詳しい。また、22) には日本での状況も含めた Lisp の歴史的経緯が紹介されている。

7 年前『情報処理』が Lisp を取りあげたときは¹²⁾、COMMONLISP が現れ始めたころでまだ Lisp

マシンに対する期待も大きかった。COMMONLISP は予想どおり Lisp コミュニティで広く認知されるに至ったが、Lisp 専用マシンのほうはいくつかのメーカーが撤退し、汎用機、特に RISC チップをベースにしたワークステーション上の優れた Lisp コンパイラにその市場を奪われつつある。当時の記事を振り返ってみると、COMMONLISP²⁴⁾、Lisp マシンのアーキテクチャ¹⁰⁾、プログラミング環境¹⁹⁾などが主な話題であった。しかし、その後の変化をみると、オブジェクト指向の枠組み、ウィンドウシステムとのインタフェース、並列実行などの新しい概念が Lisp にも持ち込まれてきたことが分かる。

本稿では総花的な解説はできるだけ避け、オブジェクト指向枠組みの例として CLOS, COMMONLISP と X ウィンドウとのインタフェースとして CLX, そして並列 Lisp の例として MultiLisp を取り上げその特徴を概観する。さらに、Lisp の方言であり COMMONLISP で採用されている静的スコープ、クロージャの考え方を早くから取り入れながら、一般にはあまり馴染みのなかった Scheme の紹介も行う。Scheme のそのシンプルで美しいセマンテックスは巨大化しつつある COMMONLISP と対照をなしている。

『情報処理』の記事がソフトウェア、ハードウェアの内部機構中心の解説になっていたのに対し、本稿では Lisp プログラマあるいは Lisp を仕事の中で使う機会がある方々を念頭において、プログラムを実際に組むうえでの言語的特徴、さらに Lisp に関する情報アクセスの方法などを紹介する。したがって、Lisp 処理系の内部構造、Lisp 向きアーキテクチャなどの話は今回は省略する。

2. プログラミングからみた Lisp 言語の特徴

2.1 Lisp の基本的特徴

最近の Lisp 言語の特徴を解説するまえに、Lisp

† New Features of Lisp Languages by Hiroyuki MATSUDA (Computer Center, Tokyo Institute of Technology).

†† 東京工業大学総合情報処理センター

* LISP は LISP Processing の略。表記として LISP, Lisp, Lisp が用いられているがここでは Lisp を使用。他の言語、たとえば、Fortran などに対しても同じ扱いをする。

の基本的特徴を復習する。Lispプログラムはすべて式と関数定義からなる。式 $x+1$ は $(+ x 1)$, 関数定義 $f(x)=x+1$ は $(\text{defun } f(x)(+ x 1))$ と書く。また、代入 $x:=2$ も式として扱い $(\text{setq } x 2)$ と書く。Lispは他の言語にないデータタイプを提供する。たとえば、シンボル自身を値として扱うことができ、 $(\text{setq name 'akina})$ は変数 *name* にシンボル値 *akina* を代入する。ここで *'akina* は (quote akina) の簡略表現で、シンボル *akina* を評価しないでそのまま値として用いることを意味する。もし、 (setq name akina) のように *quote* をはざすと、変数 *akina* を一度評価し、その結果を変数 *name* に代入するというように意味が変わる。ちなみに *setq* は *set quote* の略で、シンボル *name* を (評価しないで) そのまま変数として扱いその変数が指す番地に第2引数の値を代入する。

あと、他の言語に比べ特徴的なデータタイプとしてリストがある。リスト要素はセルと呼ばれ動的に生成/回収される。たとえば、

```
(setq *friend* '(太郎 花子 次郎))
```

```
(setq *enemy* '(ゴジラ モスラ ガメラ))
```

では、変数 **friend** にシンボル太郎、花子、次郎を要素とするリスト (図-1) が代入される。変数 **enemy** についても同様である。もし、 $(\text{setq *friend* (太郎 花子 次郎)})$ とすると太郎を関数名、花子、次郎を引数とする評価が起きる。これは意図したことなく、最初に示したようにクォートが必要となる。

次にリスト要素を削除、追加する例を示す:

```
(setq *enemy* (remove 'ガメラ *enemy*))
```

```
(setq *friend* (cons 'ガメラ *friend*))
```

この例では敵 (**enemy**) からガメラを除き (*remove*)、味方 (**friend**) に追加 (*cons*) してい

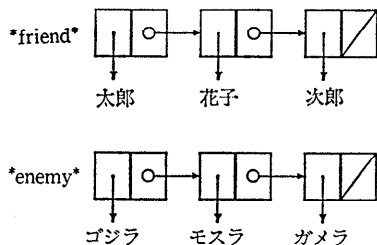


図-1 リスト処理

★COMMONLISP では大域変数は名前の左右に * をつける習慣がある。Cで大域変数に大文字を用いるのと似ている。可読性を高めるには有効である。

る。これによってリスト **enemy** のほうはセルを一つ失い (ゴジラ モスラ), リスト **friend** はセルを一つ追加する (ガメラ 太郎 花子 次郎)。これを図示すると、図-2 のようになる。ここで注意が必要なのは新しく追加、削除されるセルに対し、その領域確保、解放に対する *alloc*, *free* などの明示的な命令を用いていないことである。したがって、プログラマは、対象に対する記号処理のみ、今の例だと *cons*, *remove*, に専念でき、メモリ回りなどのインプリメントを意識する必要がなくなる。

個々の言語機能だけでなく、プログラミング全体からみた特徴も重要である。始めからアルゴリズムが完全に決定されているようなプログラミングとは異なり、AI分野で顕著な実験的、探索的プログラミングにおいては、すべての部品が揃わなくても部分だけで実行できる機構が必要不可欠である。その点、Lispプログラムをインタプリティブに実行できることの意味は大きい。インタプリタにおいては、部分だけの実行が可能だけではなく、その環境を抜けることなく、あらたにデータ、プログラムを対話的に追加でき、インクリメンタルなプログラム開発を可能にする。もちろん、こうして対話的に追加/修正されたプログラムを本来のプログラムに反映する機構もインタプリタは当然用意していなくてはならず、事実、ほとんどのシステムは構造エディタの形でこの機構を提供している。Mosesの次の言葉がLispプログラミングの性格を良く表している。

“LISP, on the other hand, is like a ball of mud. You can add any amount of mud to it and it still looks like a ball of mud.” J. Moses (MIT)

Lispは泥ボールのようなもので、どれほどつけ加えても泥ボールであることには変わらない、そん

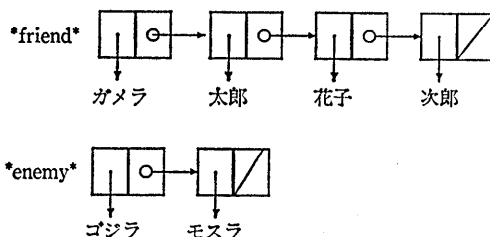


図-2 リスト処理 (追加/削除)

な意味である。

これ以外の高階関数, first-class-object などの話題については 6. Scheme との比較の中で触れる。また, データとプログラムの相互交換性については Lisp 自身による Lisp インタプリタの例で説明する:

〈インタプリタ〉

...

(loop

(setq exp (read))

exp のタイプによる条件分岐

場合 1: (eval exp)

場合 2: ...

)

...

〈実行〉

(setq x 1)

インタプリタはプログラム (setq x 1) をデータとして読み込み, 式のタイプを判別してしかるべき条件の元で評価する (eval). データかプログラムかはその形によって区別されるのではなく, それらがどの文脈で使われているかに依存する。プログラムを入力する側からは入力した文字列はプログラムのつもりであり, 他方, インタプリタの側からはその文字列はデータであり, しかるべき段階でプログラムと解釈される。Lisp による Lisp インタプリタの例としてはたとえば文献 23) 18 章 Lisp in Lisp が参考になる。

2.2 Lisp の新しい特徴

最近の Lisp, 特に COMMONLISP 以降導入された, あるいは, 広く認められるようになった新しい特徴—タイプ, 変数に関するレキシカルスコープ, クロージャ, 構造体, パッケージ—を紹介する。

2.2.1 タ イ プ

COMMONLISP 以前の Lisp においてもタイプは存在し, シンボル, 文字列, 整数, 浮動小数点, ベクタなどは異なるタイプとして区別された。そして, 今でも, 関数によっては特定のタイプの組合せしか認めないものもあり, たとえば, (+ 'apple 'banana), (car 1)* などは実行時にエラーとなる。ではタイプに関し, 何が COMMONLISP とそれ以前の Lisp とで異なるのか? 一つはタイプエ

ラーの検出時期に関するもので, COMMONLISP ではタイプ宣言を基にコンパイル時にエラーを検出することができるのに対し, それ以前の Lisp においてはこれはプログラマの責任で対処するか, すでにみたように実行時のエラーを待つしかなかった。もう一つは, タイプ宣言を最適コンパイルのための情報として積極的に利用するかどうかで, COMMONLISP とそれ以前の Lisp とで機能に差が生じる。

次に CMU COMMONLISP を例にとりてタイプエラー検出と最適化のためのタイプ宣言の例を示す。プログラム

```
(defun bar (x)
```

```
(let (a)
```

```
(declare (fixnum a))
```

```
(setq a (foo x))
```

```
a))
```

をコンパイルすると, let によって変数 a の初期値は nil になるのに対し, (declare (fixnum x)) で値として整数を要求しているため, 警告が発せられる:

In: DEFUN BAR

(LET (A) (DECLARE (FIXNUM A))

(SETQ A (FOO X)) A)

Warning: The binding of A is not a FIXNUM:

NIL

プログラム

```
(defun foo (a)
```

```
(declare (type (array fixnum (5 5)) a))
```

```
(setf (aref a 2 7) 1))
```

をコンパイルすると, 宣言によって変数 a には任意の要素タイプ, 任意サイズの配列ではなく, 整数を要素にもつ 5×5 の配列が割り当てられることが分かるので, 配列への参照 (aref a 2 7) はこの情報を元にした最適コードが生成される。ただし, この情報によって配列参照における範囲オーバーバを検出するかどうかは処理系に依存する。

2.2.2 レキシカルスコープ

COMMONLISP がそれまでの Lisp と大きく変わった点は, 自由変数の扱いである。従来の Lisp (MacLisp と呼ぶ) において, 自由変数はインタプリタ時とコンパイル時とでその扱いが異なるのが問題となっていた。例題をもとに説明する。

```
(defun foo (a)
```

* (car '(a b))=a のようにリストの先頭要素を取る関数

```
(defun goo (x) (list x a))
```

```
(hoo 'goo a))
```

```
(defun hoo (a b) (funcall a b))
```

MacLisp は動的スコープを採用しているため、関数 `goo` 内の自由変数 `a` の値は、この関数 `goo` が呼ばれる直前の値が用いられる。関数 `goo` が呼ばれるのは関数 `hoo` 内であるが、その際、関数 `hoo` の引数 `a` の値 `goo` が、関数 `goo` 内の `(list x a)` で今変数 `a` が参照しようとしている値なので、`(foo 1)` の結果は `(1 goo)` となる (図-3)。さらに、このプログラムを MacLisp でコンパイルすると、自由変数 `a` は通常のローカル変数と同じだと判断されるため、コンパイルはとおっても実行時に未束縛変数としてエラーとなる。

こうした問題を解消するために、COMMONLISP では動的スコープに対してレキシカルスコープを採用した。これは、自由変数であってもその変数が存在する位置がテキスト (プログラムの字面) 内であるときは、その変数が定義された場所をその変数の参照位置とする立場である。上記プログラムの例で言うと、関数 `goo` 内の自由変数 `a` はその参照を関数 `foo` の引数 `a` にもっている。したがって、今度は MacLisp と異なり `(foo 1)` の結果は `(1 1)` となる。これはこのプログラムを COMMONLISP でコンパイルしても同じ結果を得る。ただし、プログラムに次のような宣言を加えることで、MacLisp と同じ効果を得ることもできる：

```
(defun foo (a)
```

```
(declare (special a))
```

```
(defun goo (x) (list x a))
```

```
(hoo 'goo a))
```

```
(defun hoo (a b)
```

```
(declare (special a))
```

```
(funcall a b))
```

レキシカルスコープにすることで得られるもう

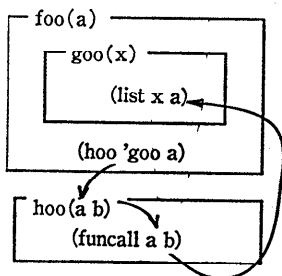


図-3 ダイナミックスコープ

一つのメリットは、コンパイルによる実行時速度の向上である。たとえば、先ほどの自由変数 `a` の参照に対し、レキシカルスコープの立場では、変数 `a` の参照は関数 `goo` の環境の中に収まっているのに対し、動的スコープでは実行時に参照場所を毎回探しにいく必要があり、当然後者の方式のほうが遅くなる。

2.2.3 クロージャ

レキシカルスコープによって自由変数を局所に閉じ込めることが可能になった。たとえば、呼ばれるたびにカウントアップするプログラムを大域変数を用いずに作ることができる：

```
(defun new-counter (initial)
```

```
#'(lambda ( ) (setq initial (1+ initial))))
```

```
(setq *my-counter* (new-counter 4))
```

```
(funcall *my-counter*) -> 5
```

```
(funcall *my-counter*) -> 6
```

...

ここで、`#'(lambda...)` は `(function (lambda...))` の略で、クロージャ closure を形成する。自由変数 `initial` はレキシカルスコープの性質から関数 `new-counter` の引数位置を参照とし、かつ、その参照時間はこのクロージャが使われるかぎり存続する。変数 `initial` は参照が存続するのでそれへの副作用を残すことができ、かつ、大域変数のように外部からの参照からは隠蔽されているという、良好な性質を有する。したがって、初期値 4 から始まるカウントを一つ作り (`*my-counter*`)、毎回これと呼び出す (`funcall *my-counter*`) と、そのたびに 1 増えた値を返してくる。

2.2.4 構造体

いくつかの基本要素からなるリストを用意した場合、個々の要素の参照は、リスト中の各要素位置に依存する。たとえば、名前、生年月日、血液型、身長からなるデータを考えると、(小泉今日子 66.2.4 O 155) のようなデータ `d` が作れるが、たとえば、血液型を知りたいとすると `(caddr d)*` のように、要素位置を意識した参照関数を用いなければならない。これ自身面倒だけでなく、仮に、仕様の変更で、名前と生年月日の順番が変わった、あるいは、新しい要素が間に追加されたりしたら、このデータを扱っているプログラムすべての箇所で順番の移動に対応した参照関数

☆ `(caddr '(1 2 3 4))=3`

の変更が必要になってくる。

このような不便を回避するために、COMMON-LISP では構造体という新しいデータタイプが用意されている。構造体によると、各要素はその位置ではなく、あらかじめ決めた名前で参照することが可能になる。上記にあげたデータを例にとると、次のような構造体 `idol` が定義できる：

```
(defstruct idol
  (name nil)
  (birthday nil)
  (blood-type nil)
  (height nil))
```

構造体を定義すると、自動的にそのインスタンスを作る関数“make-構造体名”，各要素を参照する関数“構造体名-要素名”が生成される。今問題にしている例だと、

```
(setq *K2*
  (make-idol
    :name '小泉今日子 :birthday '66.2.4
    :blood-type 'O :height '155))
(idol-name *K2*) -> 小泉今日子
(idol-height *K2*) -> 155
...
```

のように、仮に構造体 `idol` の構成要素が変わっても、参照関数 `idol-name` などを用いるプログラム部分は変更する必要がないことが分かる。

2.2.5 パッケージ

Lisp は、名前（印字名）から対応するシンボルを探すための名前空間を用意する。もし、名前空間が一つしかないとなると大規模なプログラム開発を行う場合、名前の衝突を回避するための手間が大変である。COMMONLISP はパッケージという形で複数の名前空間が共存できる環境を用意する。例で説明する。今、図-4にあるような二つのパッケージ `pac1`, `pac2` を用意する。パッケージ `pac1` は三つの変数をもっていてそのうち、変数 `ex2` のみパッケージ外からの参照を許している (`export`)。このパッケージプログラムは

```
(in-package 'pac1)
(export '(ex2))
(defvar ex1 1)
(defvar ex2 2)
(defvar x1 3)
```

で表される。一方、パッケージ `pac2` は4つの変

数をもっており、変数 `ex1`, `ex3` に対し外部参照を許している。このプログラムは次のとおり：

```
(in-package 'pac2)
(export '(ex1 ex3))
(defvar ex1 5)
(defvar ex2 6)
(defvar ex3 7)
(defvar x1 8)
```

実際にこれらのパッケージを使うにはそれらを定義したファイルを Lisp 環境にロードする（図-4太い矢印線に対応）：

```
(load "pac 1")
(load "pac 2")
```

この段階では、`pac1:ex2`, `pac2:ex3` のような形で、各パッケージの `export` 変数への参照が可能になる。一方、`export` されない変数は各パッケージ内で `intern` されており、外部からの参照はできない。したがって、たとえば、`pac1:x1` などへの参照はエラーとなる。しかし、参照する手段はあり、その場合には `pac1::x1` のように書く。

`export` されている変数を実際に自分の環境に取り込み `intern` することも可能である：`(import 'pac1:ex2)`。あるいは、特定のパッケージはよく使うので、毎回パッケージ名による修飾が面倒だというときには、`export` されている変数を継承することもできる：`(use-package 'pac2)`。この場合、変数参照に際し、先ほどのように `pac2:ex3` のように書く必要はなく、単に `ex3` でよい。ただし、現在の環境の中にすでに `ex3` という変数が

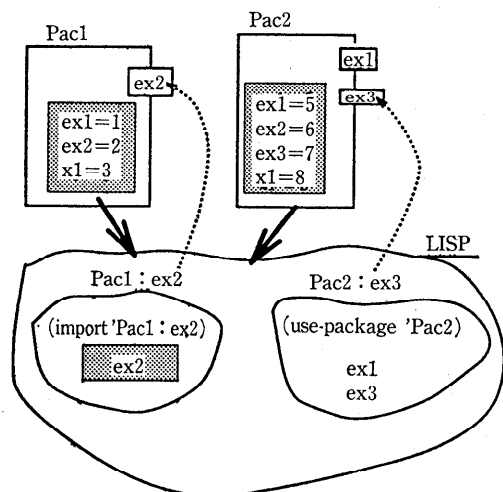


図-4 パッケージ

存在する場合には、名前の衝突が起き、エラーとなる。

パッケージに関しては、CLtL 1²⁰⁾、CLtL 2²¹⁾との間で大部仕様が揺れ動いており、これが ANSI X3J13 で最終的にどういうふうになるかは未定である。

3. オブジェクト指向 Lisp 言語: CLOS

オブジェクト指向 Lisp は Flavor, LOOPS などさまざま知られているが、ここでは CommonLisp Object System として、COMMONLISP に正式に組み込まれた CLOS を紹介する。やはりここでも、CLOS の一般的な解説というよりは具体例一並行プロセスにおける相互排除管理のためのロック lock 設計問題¹³⁾—によって直観的なイメージを理解することに主眼を置く。ただし、ここでは相互排除そのものの機構についてはまったく触れないので、したがって、問題そのものを理解する必要はない。なお、オブジェクト指向に関する基礎知識は仮定している。

3.1 クラスの設計

この問題では最初に 2 種類のロック null-lock, simple-lock を設計する。すべてのロックは固有の名前をもち、両ロック共通の (スーパ) クラスとしてロック lock を定義する:

```
(defclass lock ()
  ((name :initarg :name :reader lock-name))
  (:documentation
    "The foundation of all locks."))
```

クラス lock は唯一のスロット name をもち、その初期化にはキーワードオプション :name を用い、スロット name 値の参照はメソッド lock-name を用いる。クラス名 lock の次の () にはスーパクラスを書くが今の場合にはないので空のままになっている。ドキュメント部分はなくてもよいが、プログラム管理上は書くのが好ましい。実際にロックを一つ (インスタンス) 作ってみる:

```
(setq *my-lock*
  (make-instance 'lock :name "secret"))
(print (lock-name *my-lock*))
```

一行目でスロット name への初期値を与えてインスタンスを一つ作る。2 行目はスロット name の値を印字している。

共通クラス lock が準備できたので、次に、二つ

のクラス null-lock と simple-lock を設計する。二つのロックの違いは、simple-lock がスロット owner の値によってロックの busy, free (nil) を管理するのに対し、null-lock は常に free であるため特別なスロットをもたない点にある。したがって、null-lock はスーパクラス lock から継承したスロット name とメソッド lock-name 以外もたない単純なクラスになる:

```
(defclass null-lock (lock)
  ()
  (:documentation
    "A lock that is always free."))
```

一方、simple-lock はロック状態を管理するスロット owner をもつ:

```
(defclass simple-lock (lock)
  ((owner :initform nil
    :accessor lock-owner))
  (:documentation
    "A lock that is either free or busy."))
simple-lock のインスタンスを一つ作ってみる:
(setq *simple-lock*
  (make-instance 'simple-lock
    :name "Simple lock"))
(setf (lock-owner *simple-lock*) 3401)
(setf (lock-name *simple-lock*)
  "Another lock")
```

2 行目はスロット owner の値を変更する。COMMONLISP には参照場所の値を変更する機能があり、setf は (lock-owner *simple-lock*) で参照された場所の値を 3401 で置き換える。setf は強力で*, 実は、setf を用いればプログラムにおいて setq は一切に使わずに済ますことができる。3 行目では、スロット owner と同じようにスロット name を書き換えようとしているが、メソッド lock-name は読みだし専用として作られているため (:reader lock-name), エラーとなる。

3.2 クラスとタイプ

CLOS で作られたクラスは COMMONLISP のタイプ階層に統合される。したがって、COMMONLISP のタイプ関数 type-of, typep, subtypep など

★ Lisp プログラムにおいては、条件検査のためにある場所の値を参照し、続いて変更のためにその場所へもう一度走査し、変更を行うということが多い。setf を用いると、参照箇所直接値の変更が可能になり、再度その場所への走査という余分なパスを省くことができる。

はそのまま、CLOS のオブジェクト（インスタンス）に適用できる：

```
(typep *simple-lock* 'lock) -> t
```

```
(typep *simple-lock* 'simple-lock) -> t
```

オブジェクト `*simple-lock*` のクラスがクラス `lock` か、`lock` のサブクラス `simple-lock` のとき、`typep` の結果は真となる。

3.3 汎関数

通常の Lisp 関数は図-5 のようにインタフェースとその実現が一体となっているため、たとえば、インタフェース+に対し実現として引数が数の場合とシンボルの場合とで `add`, `cons` の区別をしようとしても

```
(+ 1 2), (+ 'abc 'xyz)
```

通常はエラーとなるだけである。これを実現する関数は汎関数 (generic function) と呼ばれ、図-6 のようにインタフェース部と実現部が分離されている。CLOS ではインタフェース部の定義を `defgeneric`、実現部の定義を `defmethod` で行うことによって汎関数を提供する。汎関数は Lisp 関数と同じように使うが、引数にあたるオブジェクトが汎関数を定義したクラスのインスタンスであるとき、それに応じたメソッドが起動される。ロック `null-lock`, `simple-lock` は二つの汎関数 `seize`, `release` を必要とするが、ここでは `seize` に限ってそのインタフェースとメソッドを定義する：

```
(defgeneric seize (lock)
  (:documentation
    "Seizes the lock.
    Return the lock when the operation
    succeeds.
    ..."))
```

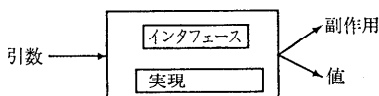


図-5 Lisp 関数

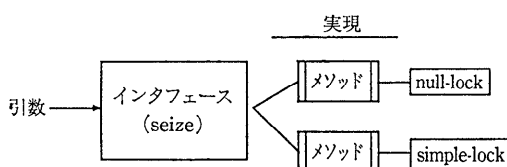


図-6 汎関数

```
(defmethod seize ((l null-lock))
```

```
  l); 待ちなし. 単にロック l を返す.
```

```
(defmethod seize ((l simple-lock))
```

```
  ; すでにロックされているかどうかチェック
```

```
(check-for-mylock l *current-process*)
```

```
(do ( )
```

```
  ; ロックが free のとき、現プロセスで
```

```
  ; busy にする.
```

```
((setf-if (lock-owner l) nil
```

```
          *current-process*))
```

```
  ; ロックが free になるまで待つ.
```

```
(process-wait "Seizing lock"
```

```
  #'(lambda ( ) (null (lock-wait l))))))
```

```
  1)
```

3.4 拡張

今までみてきた例は資源が一つの場合であったが、資源が複数（たとえば、バッテリーに対する皿とナイフ）になると獲得の順番によってはデッドロックが発生する。これを回避する一つの解として、資源獲得の順番をコントロールする方法がある。そのために、二つのロックに対し、このコントロールを施した別のクラス `ordered-lock`, `ordered-null-lock` を、コントロールクラス `ordered-lock-mixin` から作ることができる（図-7）。詳細は、本解説の内容から逸脱するので省略する（参考：文献 13)）。

また、メソッドにおいて、本来の仕事のほか、副作用として情報を加工したり、画面に表示したりしたいことがある。その場合、本来のメソッドの定義を変更することなく、メソッド本体 (primary) の前後に行いたい仕事を記述することができる。たとえば、

```
(defmethod describe :after
```

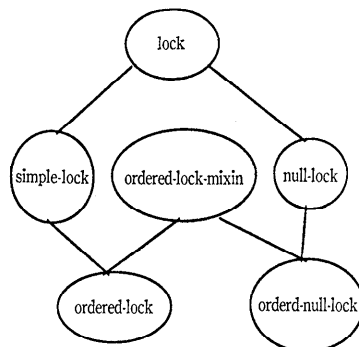


図-7 Mixin クラス

```
((1 ordered-lock-mixin))
```

```
...)
```

はメソッド `describe` の実行後、定義した処理を実行する。

なお、CLOS に関してはここで例題に使用した本以外に文献 9) が参考になる。9) では、共通問題に対し他のオブジェクト指向言語との比較も行われている。

4. ウィンドウインタフェース: CLX

Lisp とウィンドウシステムとのインタフェースの必要性は長い間、常に強く主張され続けてきた。かつては Lisp マシン上でしか、そのような機能を楽しむできなかったのに対し、高速で廉価なハードウェアの出現で現在では多くのシステム上で利用できるようになった。その中で、Common-Lisp と Xwindow との組合せにおいて、すでに *defacto standard* となっている COMMONLISP X interface (CLX)⁴⁾ をここでは紹介する。

CLX は X サーバに対しクライアントとして働き、X Window System Protocol Specification を理解する関数群からなっている。CLX によって Lisp からウィンドウを利用するアプリケーションが自由に書けるようになった。例として、図-8 のようなメニューを表示し、マウスによってメニュー項目を選択するプログラムを考えてみる(文献 4) 1 章)。マウスカーソルをメニュー項目に近付けると選択されたことを表す四角い枠が現れる。Lisp が選ばれると終了するが、その他の言語に対してはメニューを再表示して再びイベント待ちになる。ここでも、ウィンドウシステムに関する基本的知識は仮定する。実際のプログラムすべてを掲載することは物理的にも不可能なので、メニューウィンドウの作成、メニューの選択に CLX の関

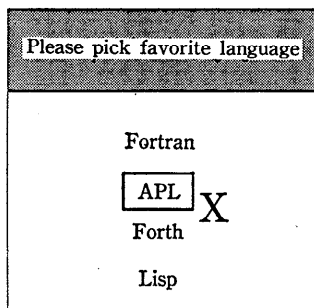


図-8 CLX プログラムの例: メニュー

数がどう関わっているかを説明することでだいたいの感触を得る。各関数の具体的な説明は 4) に記載されている。

1. ディスプレイ `display` を決定
(`OPEN-DISPLAY` サーバホスト名)
2. スクリーン `screen` を決定
(`car (DISPLAY-ROOTS display)`)
3. スクリーンのファア/バックグラウンドの色
`fg-color/bg-color`, 使用するフォント `nice-font` を指定。

```
(SCREEN-BLACK-PIXEL screen)
```

```
(SCREEN-WHITE-PIXEL screen)
```

```
(OPEN-FONT display font-name)
```

4. メニューウィンドウ `a-menu` (構造体) を作成。

```
(create-menu (SCREEN-ROOT screen)
```

```
fg-color bg-color nice-font)
```

```
(a) CREATE-GCONTEXT
```

グラフィックスに対する色, フォント, 親ウィンドウなどの情報を設定

```
(b) CREATE-WINDOW
```

親ウィンドウ, 図形的情報のほかに, このウィンドウが受け付けるイベントの種類を設定。

```
(MAKE-EVENT-MASK
```

```
: leave-window
```

```
: exposure)
```

カーソルがウィンドウから離れるときメニューウィンドウ内のイベントを掴む。

5. メニュータイトルを決定

```
"Please pick favorite language"
```

6. メニュー項目リストを与える。

```
(menu-set-item-list a-menu
```

```
"Fortran" "APL" "Forth" "Lisp")
```

(a) 各メニュー項目ごとにサブウィンドウを一つ割り当てる。

```
(CREATE-WINDOW
```

```
...
```

```
: event-mask (MAKE-EVENT-MASK
```

```
: enter-window
```

```
: leave-window
```

```
: button-press
```

```
: button-release))
```

(b) メニュー項目内では, マウスカーソルの出入り, ボタンの押し/離しをイベントとして検

知. 図-8 の例では, 項目 APL に入ったとき, 選ばれたことを検知して項目を四角い枠で囲む.

7. これでメニューを表示できる準備が整ったので, メニューを実際に表示し, 選ばれた項目が Lisp になるまで, ループするようにする.

```
(let ((choice
      (menu-choose a-menu 100 100)))
  (when (string-equal "Lisp" choice)
    (return)))
```

(a) menu-choose の中では, メニューの表示, マウスイベントのキャッチ, イベントに付随する値の処理などを行う. 一度, メニュー項目が選択されるとそのたびにメニュー全体を書き直しする.

(b) イベントキャッチ部分

(EVENT-CASE <イベントの発生検出>)

他のウィンドウとの重なりに対し再表示

(:exposure (count)

(when (zerop count)

(menu-refresh menu)))

マウスボタンが離された

(:button-release (event-window)

...)

5. 並列 Lisp 言語: MultiLisp

並列 Lisp を Lisp などによるエミュレーションではなく, 処理系の段階から並列機能を言語仕様に採用しているシステムはそれほど多くない. その中で, システムとしては少し古い, 実際に並列プロセッサ上に実装している MultiLisp⁸⁾ を代表例として紹介する. MultiLisp は最初 *Concert* と呼ばれる 16 台の MC 68000 と共有メモリからなる並列計算機上に future, pcall, delay を加えた Scheme の方言として実現された. すなわち, MultiLisp は明示的な並列命令をもち, 副作用を許し共有メモリを仮定した言語である.

式 (future X) を評価すると, X が未評価なフレーム future を返す. (cons (future A) (future B)) のようにセルへのポインタのみが必要な計算では, 実際に中身の値までは必要ないので, 計算は先に進む. そして, 実際に式 X の値が必要になった時点で評価が始まり, 値を計算する. これに対し, delay は無限リストを作成するときよく使

われるように, 実際に値が必要になるまでは, 何のフレームも作成しない. したがって, delay によって計算が停止するプログラムで delay を外した場合には計算が止まらない可能性があるのに対し, future によって正しく動いているプログラムでは future を外しても並列度は 0 になるが, 計算は正しく終了する. pcall は (pcall F A B) のように, 各引数 F, A, B を並列に評価し, その結果 f, a, b に対し (f a b) を実行する.

future によって飛躍的に並列度が上がる例をクイックソート (図-9) を例にとり説明する⁸⁾. 今, 分かりやすくするためにソートデータを (3 5 4 2) として話しを進める. 関数 partition はデータの先頭をキーとして, それ以下, 以上の要素のリストを計算する. たとえば, 今の例では ((2) (5 4)) が (partition 3 '(5 4 2)) の結果である. 関数 qs の中での future の使われ方は単純で分割の回数だけ, 関数 qs の並列度が上がる. 実際にデータを基に計算したトレース結果を図-10 に示す. ここで, 点. は cons を表す. たとえば,

(qs nil 2.(qs nil 3.(qs (5 4) nil)))

```
(defun qsort (l) (qs l nil))
(defun qs (l rest)
  (if (null l)
      rest
      (let ((parts (partition (car l) (cdr l))))
        (qs (left-part parts)
            (future (cons (car l) (qs (right-part parts) rest)))))))
(defun partition (elt lst)
  (if (null lst)
      (bundle-parts nil nil)
      (let ((cdrparts (future (partition elt (cdr lst)))))
        (if () elt (car lst))
        (bundle-parts (cons (car lst)
                             (future (left-part cdrparts)))
                       (future (right-part cdrparts)))
        (bundle-parts (future (left-part cdrparts))
                       (cons (car lst)
                             (future (right-part cdrparts)))))))
(defun bundle-parts (x y) (cons x y))
(defun left-part (p) (car p))
(defun right-part (p) (cdr p))
```

図-9 future を用いたクイックソートプログラム

```
(qs (3 5 4 2) nil)
(qs (2) 3.(qs (5 4) nil))
(qs nil 2.(qs nil 3.(qs (5 4) nil)))
2.3.(qs (5 4) nil)
2.3.(qs (4) 5.(qs nil nil))
2.3.(qs nil 4.(qs nil 5.nil))
2.3.4.(5)
(2 3 4 5)
```

図-10 関数 qs のトレース例

では関数 `qs` の呼び出しが3重になっているが、元のプログラムではそれぞれに `future` が掛かっているために、それぞれが独立して並列に実行が進む。

関数 `partition` での `future` の使われ方は多分に技巧的である。`future` がないときには、第2引数 `lst` が空になるまで自分自身を再帰的に呼びだし、その結果 (`nil`) に、キーと `lst` の終りから順に取り出した値とを比較し、小さいほう、大きいほうのリストを再帰を戻りながら作る。これに対し、`partition` の再帰呼び出しに対し `future` を掛けることで、`cdrparts` を作る過程で再帰による待ちが発生せず、したがって、分割をしながら同時に、キーとの大小比較による要素の括り出しを行うことができる。これは、また、`qs` によるソートとも同時に進行するため、高い並列度が得られる。なお、`partition` において、`cdrparts` が `future` で作られているため、`cdrparts` へのアクセス関数 `right-part`、`left-part` に関しても `future` をかけておかないと、その段階で `cdrparts` に対する計算が始まり、`partition` の呼び出しをブロックし、せっかく得られた並列度を失う結果になる。

なお、最近の並列 Lisp、並列 Lisp マシンに関しては文献 11) が参考になる。

6. Scheme

Lisp が動的スコーピングを堅持していたころ、Sussman らは、レキシカルスコーピングを基本とする Scheme を提案した。Scheme は当時としては、画期的な言語で、遅延評価、接続 `continuation` まで含んでいた。そのセマンティックスは簡潔で、*Revised⁴³⁾* には形式的シンタックス/セマンティックスの記述がわずか7ページで与えられている。そもそも言語仕様書として *Revised⁴⁾* が全体で52ページしかないというのも、巨大化した COMMONLISP と対照をなしている。

Lisp と Scheme の最大の違いの一つが一級オブジェクト `first-class object` としてとりうるデータの種類の差に現れている¹⁶⁾。Scheme では関数も含めてすべてのデータが一級オブジェクトになる。一方、Lisp では、関数は特殊な扱いを受ける。例に沿って違いを見ていく。Scheme では、`(SomeFunction arg1 arg2...)` は

```
(SomeFunction arg1 arg2...)
```

```
  ↑      ↑  ↑
  値 0    値 1 値 2
```

値 0 を値 1、値 2、...に適用。

のようにすべてを評価し、その結果の第一要素(値 0)を関数として残りの引数に適用するというシンプルな解釈になっている。したがって、次のプログラム

```
(define (double a) (* a 2))
(define (foo f n) (f n))
(foo double 4)
```

において、関数 `foo` は関数引数 `f` を受けてそれをそのまま関数位置に使うことができる：`(f n)`。一方、関数 `foo` を呼ぶ側でも、関数定義が束縛されている変数 `double` をそのまま引数として渡している。変数 `double` を評価すると関数の定義 `(lambda (a) (* a 2))` が得られる。これでも分かるように、関数 `double` の定義は

```
(define double (lambda (a) (* a 2)))
```

と書くこともできる。

一方、`(SomeFunction arg1 arg2...)` の Lisp での解釈は複雑で `SomeFunction` の扱いは単純ではない。すなわち、これが組み込み関数であるか、その他のマクロであるか、あるいは、ユーザ定義関数であるか等々によって判別が必要になってくる。先ほどのプログラムの Lisp 版を例にとるとその違いが分かる。

```
(defun double (a) (* a 2))
(defun foo (f n) (funcall f n))
(foo 'double 4)
```

関数 `foo` は引数として関数を受けとっているがそれをそのまま関数位置に書くことができないので、`funcall` によって第一引数 `f` が関数であることを処理系に知らせている。また、関数 `foo` を呼び出す側でも、変数 `double` を評価して関数定義そのものを得ることはできず、したがって、関数に対応したシンボル `double` を引数として渡ししか方法はない。これは、すべてのデータを一級オブジェクトとして扱うことで高階関数を実現している Scheme と比べて Lisp における問題点となっている。

この違いは関数の表記にも現れている。Scheme では `lambda` だけで

```
定義 (define foo (lambda (a)...))
```

```

呼び出し ((lambda (a) (1+ a)) 1)
値      (let ((n 1))
          (lambda (a) (+ a n)))
のすべてを表現できるのに対し、同じものを Lisp
で表現しようとするときそれぞれに対し
(defun foo (a)...)
(funcall #'(lambda (a) (1+ a)) 1)
(let ((n 1))
  #'(lambda (a) (+ a n)))
のような記述が必要になってくる。
次に遅延評価の例を紹介する。プログラム
(define (make-interval m n)
  (if (<= m n)
      (cons-stream m
                    (make-interval (+ m 1) n))
      ()))

```

は m から n までの数列を求める。ただし、最初の一つ以外は、必要があるまで計算しない。

```

(cons-stream <a> <b>)==(cons <a> (delay <b>))
(define (head stream) (car stream))
(define (tail stream) (force (cdr stream)))

```

この機能を利用すると、

```
(head (tail (make-interval 1 10000000)))
```

のような計算において、10000000 までのすべての数列を求めるまでもなく、2 番目の要素を瞬時に得ることができる。正確には `make-interval` を 2 回実行するだけの計算時間しかかかっていない。

Scheme に関してはすぐれた教科書が多く、文献 1), 7) が参考になる。

7. おわりに

今後 Lisp がどのように推移していくかははっきりしないが、COMMONLISP の影響が拡大することは間違いないと思われる。一方、Scheme はコンパクトな言語仕様によって軽い処理系を提供してきたが、COMMONLISP 同様、標準化の動きがあり、その行方は不透明である。Lisp 専用アーキテクチャについても商用レベルの開発はほとんど止まっている状況である。将来再び専用マシンの時代がくるかどうかは今の時点では予測がつかないが、処理系の再設計に必要な時間・金銭的成本などの経済的要因と処理速度の問題が克服されない限り難しいかも知れない。

本解説では省略したが、COMMONLISP コンパイ

ラの性能向上に関してカーネギメロン大学の CMU COMMONLISP の最適化コンパイラ戦略が参考になる¹⁵⁾。また、プログラミング環境に関しても同じく CMU COMMONLISP 上に開発された Hemlock²⁾ がエディタ、トレーサ、デバッガ、プロファイラ機能などに強力な環境を提供している。

Lisp に関する情報は USENET の comp. lang. lisp, comp. lang. scheme, fj. lang. lisp から得ることができる。特に、comp. lang. lisp に定期的にポストされる FAQ (Frequent Ask Questions)⁵⁾ は、Lisp システムの紹介、入手方法、Lisp に関する参考書、Lisp プログラミングに関するノウハウなどを集約してあり有用である。主に AI を対象としているが、COMMONLISP による膨大なプログラム例を解説した教科書もある¹⁸⁾。ここに掲載されたプログラムは ftp することも可能である。CACM 1991 年九月号⁶⁾は Lisp の特集号になっており一読の価値がある。最後に Lisp に関する論文が掲載される proceedings と雑誌を紹介しておく：

Proceedings of the biannual ACM Lisp and Functional Programming Conference, ACM (隔年).

LISP Pointers, ACM SIGPLAN (隔月)

Lisp はすでにプログラミングの現場に深く入り込んできているが、まだ、古い時代の Lisp に対する誤解を捨て切れない方も多いようである。本稿がそうした方々をも含め、多少なりとも Lisp に対する関心を増すきっかけになれば幸いである。

参 考 文 献

- 1) Abelson, H. and Sussman, G. L.: *Structure and Interpretation of Computer Programs*, MIT Press (1985). 邦訳プログラムの構造と実行(上下), マグロウヒル出版 (1989).
- 2) Chiles, B. and MacLachlan, R. A.: *Hemlock User's Manual*, CMU-CS-89-133-R 1 (Feb. 1992).
- 3) Clinger, W. and Rees, J. (Eds): *Revised⁴ Report on the Algorithmic Language Scheme* (Nov. 1991).
- 4) CLX: *Common Lisp X Interface*, Texas Instruments Inc. (1989).
- 5) Kantrowitz, M. (Moderator): *Lisp Frequently Asked Questions*, USENET, comp. lang. lisp.
- 6) Foderaro, J. (Ed.): *Special Section—LISP*,

- CACM, Vol. 34, No. 9 (1991).
- 7) Friedman, D. P., Wand, M. and Haynes, C. T.: *Essentials of Programming Languages*, McGraw-Hill (1992).
 - 8) Halstead, R. H.: MultiLisp: A Language for Concurrent Symbolic Computation, *ACM TOPLAS*, Vol. 7, No. 4 (1985).
 - 9) 井田昌之, 元吉文男, 大久保清貴: *CommonLisp* オブジェクトシステム—CLOS とその周辺—, bit, 1989年1月号別冊, 共立出版 (1989).
 - 10) 井田哲雄: LISP マシンのアーキテクチャ, 情報処理, in 12) (1985).
 - 11) Ito, T. and Halstead, R. H. (Eds.): *Parallel Lisp: Language and Systems*, Lecture Notes in Computer Science No. 441, Springer-Verlag (1990).
 - 12) 角田博保他: 小特集「Lisp の最近の動向」, 情報処理, Vol. 26, No. 7, pp. 710-749 (July 1985).
 - 13) Keene, S. E.: *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley (1989).
 - 14) Kreutzer, W. and McKenzie, B.: *Programming for Artificial Intelligence: Methods, Tools and Applications*, Addison-Wesley (1991).
 - 15) MacLachlan, R. A. (Ed.): *CMU Common Lisp User's Manual*, Carnegie Mellon University (May 1992).
 - 16) 松田裕幸: Scheme と Lisp の比較, 163 回 PTT 資料 (1990).
 - 17) McCarthy, J. et al.: *LISP 1.5 Programmer's Manual*, MIT Press (1963).
 - 18) Norvig, P.: *Paradigms of AI Programming: Case Studies in Common Lisp*, Morgan Kaufmann (1992).
 - 19) 奥乃 博, 丸山 宏: Lisp のプログラミング環境, 情報処理, in 12) (1985).
 - 20) Steele, G. L.: *Common Lisp: The Language*, Digital Press (1984). 邦訳 *Common Lisp* 第1版, bit.
 - 21) Steele, G. L.: *Common Lisp: The Language (2nd)*, Digital Press (1990). 邦訳 *Common Lisp* 第2版, bit, 1991年6月号別冊, 共立出版 (1991).
 - 22) 寺島元章: LISP—その発展の方向, 情報処理, in 12) (1985).
 - 23) Winston, P. H. and Horn, B. K. P.: *Lisp (3rd)*, Addison-Wesley (1989).
 - 24) 湯浅太一: *Common Lisp*, 情報処理, in 12) (1985).

(平成4年9月11日受付)



松田 裕幸 (正会員)

1952年生。1983年静岡大学工学部情報工学科卒業。1985年東京工業大学大学院理工学研究科情報科学専攻修了。日本電気技術情報システム開発(株)を経て、現在、東京工業大学総合情報処理センター助手。プログラミング言語設計論、プログラミング方法論に関する研究に従事。日本ソフトウェア科学会、ACM、IEEE 各会員。

