第1章

CLOS概説

井田 昌之

1. CLOS への道筋

1.1 オブジェクト指向の流れ

オブジェクトの概念の形成は，1960年代後半のSIMULAにさかのぼることができる．そしてSmalltalkを経て現在のオブジェクト指向言語へと流れている．このことを疑う余地はないが，それらすべてが必ずしも同一のオブジェクト指向概念に基づいているものではないことには留意する必要がある．逆に言えば，オブジェクト指向には多くの知恵が集められ，現在もなお進歩しているということができよう．CLOS(Common Lisp Object System)[21),22)] はその中にあって，ANSI において Common Lisp[1)] 用の標準オブジェクト指向機能として取り上げられたものである．

書棚の中のほこりをかぶったSIMULAのマニュアルを久しぶりに取り出してみた．SIMULA には class 宣言があり，そこでは，ブロック構造の手続きを定義する．その手続きに属する局所データの宣言機構が付属する．SIMULA でのオブジェクト生成とは手続きの呼出しそのものである．すなわち，オブジェクトというのは関数定義であり，それを生成するというのはその定義を実行させることだというイメージのほうが近い．Common Lispでたとえれば，クロージャの定義機構＋単一継承がSIMULA のオブジェクト指向機能の中心概念といえよう．シミュレーション言語であることを反映する部分は後の Flavors などにある mixin の元祖のようであり，前もってシステムに用意されたクラスを指定し呼び出す仕組みとなっている．

Smalltalk はその後に現われた．Smalltalk からオブジェクト指向が始まったとし，そこでの概念を鏡としている場合が多い．一般には，Smalltalk をもってオブジェクト指向の元祖としている．Smalltalk ではすべてがオブジェクトであるという大きな特徴が評価されている．Smalltalk 以後，すなわち，1970年代後半になって，オブジェクト指向は新しいプログラミング・パラダイムとして認知されるようになった．

Smalltalk 以後には，たくさんのオブジェクト指向言語が現われてきた．そのほとんどは，Smalltalk のような独自の独立した言語というよりも，通常の言語の上にオブジェクト指向機能を加えたハイブリット言語である．たとえば，C言語に対しての Objective C あるいは C++，Lisp に対しての Flavors，LOOPS などである．本書で扱っている CLOS も独立した言語ではなく，Common Lisp に対するオブジェクト指向機能である．

CLOS は現在，X3J13 において Common Lisp の仕様の一部としてまとめられている．CLOS は万能薬ではないが，上述した Flavors，LOOPS の機能の延長線上にあり，Common Lisp を使って実際的なソフトウェアを作成するには欠かせないものである．CLOS は基本的に Common Lisp の基本仕様の一部となるので，その位置付けは Common Lisp のそれと等しくなる．その仕様は3章に分かれている．第1章と第2章は文献21)としてまとめられており，主に，CLOS を使う上での仕様を規定している．第3章は文献22)としてまとめられており，メタオブジェクトに関する仕様を規定している．本書は第1章と第2章について主に扱っている．

1.2 オブジェクト指向に対する伝統的な見方

従来，オブジェクトとは，

（1） それ自身のプライベートデータをもつ．

（2） そのデータの上で行なわれる操作の組がある．

という認識をもとに，これら(1)，および(2)を包括するものとして考えられてきた．プライベートデータは外

部から見える場合もあるし，見えない場合もある．また，操作の組が実際にどんな手順で何をするのかはわからなくとも，これらの提供された操作の組を用いて規定された機能を働かせることができるようになっている．そして，これらを一括する次のような機能をもったものとしてオブジェクト指向機能（システム）を捕えることができた．

（1） Encapsulation：プライベートデータの実際の構造，扱いを外部から隠す（あるいは，1つにまとめる）．オブジェクトに external view と internal view をもたせる．これにより，内部の変更を外部への影響なしに行なえる．また，外部のアクセスから内部を守ることができる．

（2） Integrity：同一の情報に対する重複した（冗長な），矛盾のある記述を避ける．

（3） Operationality：それらを，複数のソフトウェアの間で共有する機構，実行時に有効なサポートをする機構を置くことにより，実用的な道具として実現させる．

また，こうした基本的な機能から，特に，大規模なソフトウェア開発において多人数が共同開発する上での便利さ，拡張，変更への対応力などが着目され，

（4） Modularity：ソフトウェアをモジュラーに開発する手法，コンポネントプログラミングとも呼ぶべきソフトウェア開発法を可能とした．また，application domain に沿った開発を可能にした．

これらを字面で捕えるのであれば，言い換えれば，「データと操作の定義に対して抽象データ型と uniform external interface が提供され，それを用いて，ソフトウェア開発者はモジュラーにプログラムを作成できる」ということだけが特質であるのなら，たとえば，ADA のパッケージや CLU クラスターもオブジェクト指向といえることになる．それらはそうは呼ばれていない．

オブジェクト指向を特徴付けるもう1つの重要な概念は，「継承」である．それがどんなものであっても，継承（もしくはその発展概念）がなければ意味がない．上記の4つは「継承」の裏付けをもっているのである．

継承とは，これを，現実的な機能から意味付けるとするなら，「データおよびそれらに対する操作の組を再定義する手間を省き，それらを部分共有しながら新しいオブジェクトを定義する」概念である．

たとえば，「人間」を定義するのに，『「哺乳類」としての性質・機能に関連する部分は，「哺乳類」オブジェクトでの定義を部分利用する，言い換えれば，継承することで行なう』，あるいは，『画面上での線分のグラフィック表示は，「画面上に線を引く」ということと「二次元座標の処理」という，2つの側面があり，これらおのおのの操作は，他との共通性があるので，「画面上の線分のグラフィック表示」という特定の機能だけの中に記述するのではなく，「画面の処理」と「二次元座標の処理」という2つの部分に分けて考え，これらを継承することで，「画面上の線分のグラフィック表示」に関する機能を定義する』といったことを思い浮かべることができる．

複数のものを継承することを**多重継承**，1つだけを継承することを単一継承という．**図1**に多重継承をするような階層的なクラス構造を示す．

このようにして，たとえばウィンドウ・システムでは，ウィンドウ・オブジェクトを考え，そのローカルなデータとして，ウィンドウ・サイズ，画面上の位置，その表示内容の種類といったものを，操作としては，移動，サイズの変更，削除などを考え，それらを一まとめにして定義する．加えて，これを継承することにより，後でシェルのウィンドウとか，グラフィック表示用ウィンドウといったものを，それに本質的な部分だけ，あるいは継承させたくない部分だけを記述することで作成できる．これらの内部の変更は外部インタフェースに変更を与えない限り，変更の影響はそのオブジェクトの中だけにとどめ，最小限にすることができる．

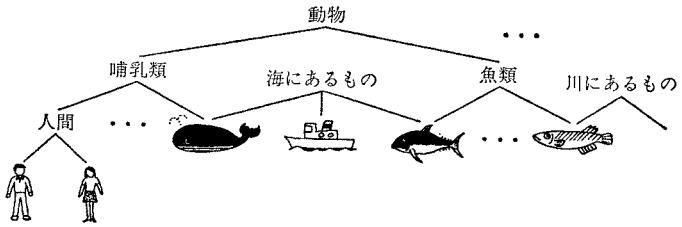

図1 多重継承とクラスの階層

2. クラス

2.1 クラスの基本概念

はっきりとした共通の定義がないオブジェクトという言葉にかえて，我々はここで，「**クラス**」という言葉を導入する．クラスの実現形をインスタンスという．「インスタンスを生成する」というようにして用いる．また，**メタクラス**という言葉を導入する．メタクラスは，クラスを定義するクラスである．クラスの概念は，構造をもつデータ表現に対するテンプレートが階層的に相互に影響し合っているものとして捕えることができよう．

クラスの定義には**4つの要素**がある．

- クラス名
- 継承するスーパークラス
- スロット定義
- クラスオプション

である．クラス名はそのクラスの名前である．スーパークラスとは継承関係の中でいわばそのクラスの「親」となるクラス，スロットとはそのクラスに固有な局所データである．クラスオプションはそのクラスに対する指示である．

なお，これに加えて，後述するメソッドもクラスの中に閉じ込めるように考える仕組みもあるが，CLOS ではこの encapsulation のイメージをあまり強く主張していない．CLOS ではメソッドの定義はクラスの定義から分離している．これが CLOS の第1の特徴である．

「クラス定義とメソッド定義は分離している．」

もちろん CLOS の上にメソッドとクラスを強く結び付けるような機構をかぶせることができる．しかし，CLOS はそうした強連結性を欲しない応用にも対応できるような柔軟性を与えている．

2.2 クラス定義の基本的な方法

（1） defstruct からの導入

defstruct は Common Lisp がもつ構造型の定義手段であり，

```
(defstruct 構造名  スロット定義 …)
```

という形式をとり，構造の定義を行なう．

たとえば，平面上の点を表わす，次のような定義が書ける．

```
(defstruct point
           (x 0)
           (y 0))
```

point 型のデータは自動的に生成される make-point という関数を使い，単に，

```
(make-point)
```

もしくは，スロットの初期値を同時に与えて，

```
(make-point :x 1.5 :y 3.0)
```

により生成する．

各スロットのアクセスのために，point-x および point-y という関数が自動的に作られる．

```
(setq foo (make-point :x 1.5 :y 3.0))
```

ののち，

```
(point-x foo)
```

を実行すると 1.5 が返される．

defstruct には多くのオプションがあるが，CLOS との類似性が強いものとしては，他の defstruct を引用して定義を行なう :include をあげることができる．これは，継承の概念の初歩とみなすこともできる．

この defstruct からの発展としてクラス定義との関連を導入してみよう．

（2） CLOS でのクラス定義

最も簡単な例を次に示す．

```
(defclass point ()
      ((x :initform 0)
       (y :initform 0)))
```

また，線は，

```
(defclass line (point)
      ((length :initform 1)
       (direction :initform 0)))
```

と書ける．

表 1　defclass 対 defstruct

	defstruct	defclass
対象	構造型	クラス
スロット	スロット	スロット
他の引用	:include による静的な引用	多重継承
生成	make-XXX	make-instance
生成時のスロット初期値付与機能	:X で自動的に可	クラス定義時に指定
スロットアクセサの生成	自動生成	定義時の指定による

このように defcalss は

```
(defclass クラス名
    (継承するスーパークラス名のリスト)
    ((スロット定義)...)
    クラスオプション)
```

という形をしている．クラス定義に関するその他の指示は，もしあればスロット定義の後に，クラスオプションリストとして追加する．継承されるクラスをスーパークラス，継承するクラスをサブクラスという．

上の例は，point クラスには x と y というスロットを置き，その初期値は 0 としている．初期値設定のためには :initform というキーワードが欠かせない（この他に :initarg というキーワードを使う方法もある）．point は何も継承していない．一方，line クラスは point を継承している．すなわち，line では length と direction というそこで定義されたスロットの他に x と y というスロットが含められる．これだけでは defstruct での :include と同じであるが，加えて，この継承関係は後述するメソッドの継承をも含み，この結果，非常に便利な機構を提供している．

表 1 は defstruct と defclass を比べたものである．defstruct に慣れた者が defclass を知る上でポイントになることをまとめてみた．いうまでもなく，defclass は CLOS のほんの一部の機構であり，「defstruct の延長に CLOS/defclass がある」と単純に考えることはたいへん危険である．

（3）　多重継承

継承には多重継承と単一継承とがある．CLOS は多重継承をする．言い換えれば，複数のスーパークラスをもつことができる．もし，

```
(defclass baz (foo1 foo2)
  ...)
```

とあると，baz は 2 つのクラス foo1 と foo2 の定義をこの順に継承する．これが多重継承である．同名のスロットが foo1 と foo2 で存在する場合，foo1 のものが受け継がれる．また，同名のメソッドが，両者にある場合も同様に foo1 のものが優先して受け継がれる．このサーチの手順は left-to-right-and-up-to-joins といわれることもある．

似たような処理をいちいち個別に書かねばならないとしたら不便である．共有できるものは共有したい．そうすれば節約になるし，それだけではなく構造をきれいにすることができる．こうした要求に沿うのが継承という概念である．

2.3　インスタンスの生成

Smalltalk 的な概念では，そのクラスのインスタンスを生成するメソッドは，そのクラスのメタクラスにおいて new というセレクタにより定義される．CLOS では，make-instance という総称関数がこれを行なう．

make-instance はクラス名を必須の第一引数として取り，たとえば，

```
(make-instance 'line)
```

のようにして実行させる．

この結果，line クラスのインスタンスが 1 つ作られる．インスタンス生成時にスロットの値を初期化することができる．それについては次節 2.4(3) に述べる．

次の例は多重継承をする plane-in-the-radar クラスの定義とそのインスタンスの生成例である．

```
(defclass 2d-object (point)
    ((name :initarg :name :type
      string)))
(defclass plane-in-the-radar
     (2d-object rectangle)
     ((destination :initarg
      :destination :type string)
      (departing-point :initarg
      :departing-point :type :string)
      (hours-to-fly :initarg
      :hours-to-fly :type number)))
(make-instance 'plane-in-the-radar
     :name "UA525"
     :destination "San Fransisco"
     :hours-to-fly 2.5
     :departing-point "Denvor")
```

2.4　スロットのアクセス，初期化

（1）　スロットの可視性

各インスタンスのスロットに対して，次のような選択がある．

（a） そのスロットはいつ生成されるか？

（b） そのスロットはどこに置かれるか？ ローカルかグローバルか？

（c） そのスロットの値をインスタンス生成時に設定できるか？

（d） そのスロットの値を外部から参照できるか？

（e） そのスロットの値は外部から変更できるか？

（a）については，たとえば，クラス定義時，ロード時，コンパイル時などのタイミングが考えられる．CLOSでは基本的には実行時に最初のスロット参照すなわちインスタンス生成時に作られる．

（b）については，ローカルであればそのスロットは共有されず，そのインスタンスの中に取られる．グローバルであればそのスロットは共有され，そのクラス全体に1つだけ取られることを意味するようになる．2.4（2）に述べる方法で利用者が選択できる．

（c）は，具体的には：initarg である．2.4（3）に述べる方法で指定できる．

（d）および（e）は，2.4（4）に述べる総称関数により選択する．また，CLOSでは低レベルのインタフェースが用意されており，2.4（4）に述べる関数がないとしても，slot-value により内容を参照/変更できる．2.4(5)を参照してほしい．

（2） :allocation {:class, :instance} によるスロットのアロケーション

スロットが置かれる場所は :allocation で指定する．:class 指定はグローバルにクラス定義の上に置くことを，:instance 指定はインスタンスに割り付けることを，おのおの意味する．無指定の場合は :instance 指定が仮定される．たとえば，

```
(defclass foo ()
    ((x :allocation :class)
     (y :allocation :instance)
     (z :initform 0)))
```

は，xはクラス foo に対してグローバルに割り付けられる．したがって，foo の各インスタンスに対してただ1つだけしか存在せず，その値の変更は全インスタンスに影響する．yおよびzは，各インスタンスに割り付けられる．

（3） スロットの初期化の2つの方法——:initform および :initarg による生成時の初期化

インスタンスの生成時にスロットの値（初期値）を与えることができる．これには，2つの方法がある．1つは :initform によりクラス定義中に与える方法，もう1つは :initarg に名前を指定し，その名で，make-instance 時に値を与える方法である．後者は，異なるスロットに同一の初期値を与えるような場合にさらに便利である．:initform による初期化は2.2（2）に例がある．(make-instance 'point) とすると，スロットxおよびスロットyの値は指定に従って0になる．:initarg の例は2.3にある．make-instance 時に引数として初期値を与える．(なお，:initarg ではすべてキーワードシンボルを書く必要はない．4. の例では普通のシンボルを用いている．)

2.2の(2)にある point クラスは，スロットを make-instance 時に初期化することはできない．これをするには :initarg 機能を用いる．

```
>(defclass new-point ()
    ((x :initarg :x) (y :initarg :y)))
new-point
>(setq new-pos (make-instance
  'new-point :x 1 :y 2))
#<Standard-Instance...>
>(slot-value new-pos 'x)
1
```

のようになる．

（4） :accessor, :reader, :writer スロットオプション

:reader に続けて総称関数名を書くと，そのスロットの読出しはその関数が行なうことになる．その関数は利用者が与えることができるので，そこに読出し時に行ないたいことを書いておくとそれがすべて実行されることになる．利用者が与えない場合には単にそのスロットをアクセスするメソッドが自動的に用意されている．

:writer も :accessor も同様であるが，ライタ(:writer で指定された関数）は書込み時のため，アクセサ(:accessor で指定された関数）は読出し/書込みともに用いられる．

```
(defclass test ()
  ((x :accessor access-x)))
(defclass test2 ()
  ((x :accessor access-x)))
(defmethod access-x ((v test2))
  (incf *global-counter*)
  (slot-value v 'x))
```

とあったとすると，test クラスのインスタンスのスロッ

トxに対するアクセスは自動的に作られた access-x により，たとえば，(access-x test-class-instance) によりそのスロットのアクセスだけが行なわれるが，test2 クラスの場合，上に定義したメソッドが呼び出され，*global-counter* の値を1加える操作が同時に起こる．

これらの使い方はいろいろと考えられる．基本的にはアクセス時に生ずる諸々の手順を利用者から隠ぺいする点である．たとえば，point インスタンスが常に画面に表示されているとすれば，座標を変更すると同時に画面上の位置も変わるのが望ましい．すなわち，xスロット，yスロットの値を変更すると自動的に表示も変更するようにするとよい場合もある．そのような場合には，ライタを書き，その中で自分で slot-value を用いてスロット値を変更するのに加えて，表示の変更をする手続きを加えておけばよい．

（5） slot-value によるアクセス

インスタンスの各スロットの値をセットしたり，各スロットの値を参照するには，読み書きともに基本的には slot-value を用いる．slot-value にはインスタンスとスロット名を与える．

```
>(setq a (make-instance 'point))
#<Standard-Instace …>
>(slot-value a 'x)
0
>(setf (slot-value a 'x) 1)
1
>(slot-value a 'x)
1
```

slot-value と書くことを避けるために，with-slots という構文が用意されている．この構文の中では，各スロットをアクセスするのに (slot-value …) とせずにただ単に，変数のように書くことができる．これについては3.6を参照してほしい．

2.5 型とクラスとの統合

CLOS では，まったくクラス定義をせずに組込みのクラスに対して直接メソッドを定義できる．Common Lisp 組込みの型をクラスとして使う場合である．これも CLOS の大きな特徴の1つである．

型とクラスの関係をまとめると次のようになる．

第1に，型にはクラスになるものがある．array, bit-vector, character, complex, cons, float, integer, list, null, number, ratio, rational, sequence, string, symbol, t, vector がそれである．

第2に，クラスは型になる．

第3に，構造型はクラスになる．しかし，注意したいことは，構造体型のデータを make-instance で生成できるわけではない．

array から vector までの上に記した型はクラス名として利用できる．クラスとしての相互関係は，階層的な型の順序関係をそのまま持ち込む．たとえば，number 型は rational 型の，rational 型は integer 型のそれぞれスーパークラスである．

型階層では vector と null の2つに注意したい．null 型のスーパークラスは list 型と symbol 型がある．参考文献2) は null 型 (nil) は list 優先，すなわち，list 型は symbol 型に優先することを定めている．しかし，CLOS では symbol が優先するようになった．vector 型と array 型の関係については参考文献14)は，sequence 型優先の大原則を考え，array 型に対する vector 型の優先を提案している．CLOS ではこのようになっている．

2.6 メタクラス

メタクラスはクラスをインスタンスとするクラスである．そこには，それに属するクラスに対するすべての操作が含められる．通常は CLOS で標準としているメタクラスによりすべてがつかさどられる．独自の概念を実現させたい場合には，メタオブジェクト・プロトコルを用いてメタクラスを定義する．これらについては参考文献22)で規定されている．

3. メソッドとメッセージ送信

3.1 メソッドの基礎概念

（1） Smalltalk 的な理解

データと操作の組が基本的なオブジェクトの概念であると 1. で述べた．Smalltalk 的なオブジェクト指向では，「オブジェクトにメッセージを送る」ということで，操作の実行を引き起こすようになっている．1つのメッセージにはセレクタをおく．セレクタは操作の種類を表わしている．引数を伴うこともありうる．メッセージの受け手（レシーバ）には，メソッドがあり，これが実際に行なうべきことを記述している．メソッドは手続きに似ているが，特定のオブジェクトと不可分である点が異なる．ウィンドウ・システムの例を考えてみると，move, display, delete, width, height といったセレクタと対象となるウィンドウ・オブジェクトとの組がメッセージである．たとえば，“shell-window delete:” とい

ったものをメッセージと考える．

（2） CLOSへの発展——関数呼出しの一般化としてのメッセージ送信

これに対して，CLOSでは関数の一般化としてメソッドの概念を包括する．Lispの基本的な手続き呼出し機構は関数呼出しである．CLOSではこの関数呼出しの一般化としてメッセージ送信を扱う．

二引数の関数fの呼出しは，

```
(f a b)
```

の形式をとる．aとbはその引数である．この形式は，ふつう次のように理解され，実行される．

```
(funcall (function-specified-by 'f)
         a b)
```

ただし，function-specified-byはここで説明のために導入したものであり，この例だけでいえば，Common Lispのsymbol-functionなどを想像すればよい．CLOSを考えないLispだけの世界では，fという名をもつ関数の実体はスコーピングルールによって決定される．ここで注意したいことは，関数の実体の決定は，引数の個数，型などには一切関係がないことである．Lispの言語仕様としては，引数は，実行すべき関数本体の特定には関与しない．

Smalltalk, LOOPS, Flavorsなどのメッセージ送信は，基本的に同等である．すなわち，旧Flavors[12]（参考文献7）で定義されているFlavorsと区別するため旧Flavorsと記す）で表現すると，次のようになる．

```
(send a :f b)
```

これは，:fというメッセージを，bという引数を伴ってaに与えるものである．その解釈と実行を上の例に倣って書くとすれば，

```
(funcall (method-specified-by :f a)
         a b)
```

となる．:fとaの属するクラスによってメソッドの特定が行なわれる．

関数呼出しの観点からすると，関数名だけでなく，第一引数が，関数の特定に参加していることになる．この点がCLOSを理解する上で重要な鍵となる．引数を実行させるべき関数の特定に参加させるという見方でメッセージ送信と関数呼出しとを融合させている．

すなわち，メッセージ送信はFlavorsでのsendのような特定の関数の中に閉じ込めず，単に関数呼出しの形式で記述する．(send a :f b)ではなく(f a b)のようになる．fが単純にdefunされた関数なのか，あるいはメソッドであるかは，これを見るだけではわからない．

```
(send a :f b) ⇒ (f a b)
```

という変化が生じていると思っても理解できる．

従来のオブジェクト指向機能と共通性のあるこのレベルの機能だけを利用したメソッドは，classical-methodということがある．また，メソッドの特定に，1つの引数が関与しているので，single-methodということもある．

（3） 多重メソッド（multi-method）

メソッドの特定に第一引数を参加させるだけにとどめる必然性は特にない．与えるすべての引数をメソッドの特定に関与させる仕組みが，ここまでの議論の自然な延長線上に存在する．

たとえば，ここまで使ってきた二引数の例でいえば，

```
(funcall (method-specified-by :f) a b)
```

あるいは，

```
(funcall (method-specified-by :f a)
         a b)
```

に加えて，

```
(funcall (method-specified-by :f a b)
         a b)
```

という解釈が行なわれる機構を考えることができる．この最後の解釈は，メソッドの特定に，2つの引数を関与させている．複数の引数がメソッドの特定に関与するメソッドは**多重メソッド**と呼ばれる．

多重メソッドの場合，lambda-list keyword problemというものが存在する．これは第1に，メソッドの定義で&optional, &rest, &keyなどを許すかということ，第2に許すとした場合に，それはメソッドの特定に関与するかという問題である．第1の問題に対しては「許す」，第2の問題に対しては「参与しない」という方針がとられることとなった．詳細は「第II部CLOSの仕様」を参照してほしい．

3.2 総称関数

Common Lispにはもともと総称関数（generic function）の概念があった．3.1のように捕えてきたメッセージ送信は，この総称関数となる．総称関数は，数型に対する関数や列型に対する関数の多くがあげられる．たとえば，eltである．これは従来のLispにあった同様の機能をもつが名前の異なるものを総称化している．総称化の基本は，引数の型に応じて，実行する手続きを切り換える実行手順を示唆している．この点が，オブジェクト指向における基本的な仕組みと合致できるものである．

3.3 メソッドの定義

CLOS では，メソッドは defmethod（もしくは defgeneric,…）により定義できる．defmethod は次の形式をしている．

```
(defmethod
    メソッド名　メソッド修飾子
    specialized-lambda-list
    フォーム …)
```

メソッド修飾子は付加的であり，なくともよい．これは，メソッド結合 (3.5) と関連して用いられる．defmethod のイメージを捕えるには，基本的に defun を想定すればよい．defun での単純なパラメータリストの代わりに，CLOS では特定化された (specialized) ラムダリストを置く．特定化されたラムダリストは，defun でのような単純なパラメータか，特定化されたパラメータからなる．特定化されたパラメータは次の形式をもつ．

```
(パラメータ名　引数特定子)
```

ただし，引数特定子は，クラス名か次の形式のリストである．

```
(eql インスタンス)
```

この後者の形式は，ある特定の（単一の）インスタンスの場合だけに用いられるメソッドを可能にしている．たとえば，比較的身近な例でいえば，非連続関数における不連続点の処理や，無限大の組込みなどである．あるいはカレンダーを扱う場合に，国民の休日を個別に扱う場合などを思い浮かべることができる．筆者はこうしたことのためのメソッドを特異メソッドと呼んでいる．

単純なパラメータを書いた場合，それは，

```
(パラメータ名 t)
```

の省略形として解釈される（t はクラス t を表わしている）．

簡単な例を示す．

```
(defmethod f ((x number) y) …)
```

は第一引数が number 型であるメソッドを定義している．第二引数は任意である．この定義は第一引数だけをメソッド特定に関与させていることになる．したがって，これは伝統的なメソッド (classical-method) である．一方，

```
(defmethod f ((x foo) (y bar))
    …)
```

は，x が foo クラス，y が bar クラスであることを指定している．これは多重メソッドである．

また，n! := if n= 0 then 1 else n・(n−1)! は，

```
(defmethod fact ((n (eql 0))) 1)
(defmethod fact ((n number))
  (* n (fact (1− n))))
```

と書くことができる．

総称関数とは，このようにして作られたメソッドの同名のものの集まりである．同名であればどんなものでもよいかというとそうではなく，後述するような細かな規則がある（第2章および「第II部 CLOS の仕様」参照）．ここでは，次の規則だけを挙げて置く．

「同一の総称関数に属するメソッドの引数の個数は，等しくなければならない．」

この結果，次のような例は引数の個数が異なるのでエラーになる．

```
(defmethod foo ((x class1) (y class2)
  (z class3)) …)
(defmethod foo ((x class4)(y class2))
  …)
```

引数の一致についてはさらに詳細な規則が決められている．「第II部 CLOS の仕様」の該当の箇所を参照してほしい．

具体例として次に，5つのメソッドをもつ総称関数F を考えてみよう．

```
F(x)=1/x (x <> 0)
      0 (x=+omega, −omega)
      undefined (x=0)
(defmethod f ((x number))
  (/ 1 x))
(defmethod f ((x (eql '+omega)))
  0)
(defmethod f ((x (eql '−omega)))
  0)
(defmethod f ((x (eql 0)))
  'undefined)
(defmethod f (x)
  (error "~%illegal argument for
   1/x, x=~s" x))
```

なお，ここでは無限大を表わすものとして，+omega および −omega を用いている．総称関数の呼出しは，通常の関数呼出しの形を取り，特殊な形式を用いない．したがって，上述の例に対しては，

```
(f 1)
```

のようにする．そうするとこの場合，総称関数 f の中から number クラスに対するものが呼び出され，(/ 1 x) が実行され，答1を得る．(f '+omega) の場合，(eql '+omega) をパラメータ指定子にもつメソッドが選択さ

れ，0が返される．

この例で示すように科学計算のライブラリ関数をCLOSの総称関数化すれば，無限大の処理や特異点の扱いを組み込むことができる．

次の例は多重メソッドからなる総称関数 rotate の一部である．

```
(defmethod rotate
        ((pos point) (theta number))
    (let ((pos-x (slot-value pos 'x))
          (pos-y (slot-value pos 'y))
          (new-2d (make-instance
           'point)))
      (setf (slot-value new-2d 'x)
            (- (* pos-x (cos theta))
               (* pos-y (sin theta)))
            (slot-value new-2d 'y)
            (+ (* pos-x (sin theta))
               (* pos-y (cos theta))))
      new-2d ;returns created instance
      ))
(defmethod rotate
        ((x line) (theta number))
    ...
    (incf (slot-value new-2d
     'direction) theta)) new-2d)
```

3.4 実行時にどのメソッドが選ばれるか？

（1） 基本メソッドの選択

関数呼出しの形式による総称関数コールは，複数あるメソッドの中から最もふさわしいものをただ1つだけ選び出し，それを実行する．これがいちばん基本的なルールである．すなわち，

「最もふさわしいメソッドが選ばれ実行される．」

このようにして選ばれるメソッドを基本メソッド（primary method）と呼ぶ．

メソッドサーチのアルゴリズムは，最も特定化されたものを探し実行するという基本ルールに，クラス-スーパークラス関係による上位クラスでのメソッド参照，その多重的な参照，加えて，多重メソッドの場合，左優先（left-to-right）のサーチが組み合わされて形成される．

3.3 の rotate の例では，第一引数が line クラスであれば，必ず後者のメソッドが呼び出される．もちろん，第二引数はその場合，number クラスでなければならない．CLOSは，複数の多重メソッドがあった場合，第一引数のクラスの関係から順に調べられていくこと（もしくはそれに等しいアルゴリズム）を要求している．

ここではその基本メソッドが選ばれるまでの手順，そしてそれ以外のメソッドの選ばれ方についてふれる．

実行時にはまず，実際に渡される引数の指定（特異メソッドを除けば引数のクラス）によりそれにマッチするメソッドが捜される．それらは基本的に複数あると思わなければならない．前節の例では（f 1）という呼出しに対しては（/ 1 x）を計算するメソッドだけでなく，最も一般的な場合のためのメソッド（この場合エラー表示）も候補となり得るはずである．（f 0）という呼出しでは，加えて，（eql 0）を指定したメソッドも対象となる．そしてそれが選ばれることになる．

こうした候補となり得るメソッドを「適用可能メソッド」（applicable method）と呼ぶ．実行時にまずされることは，この適用可能メソッドの取出しである．この取出しにおいて，メソッドは，その特定化の度合に応じて順に並べられる．CLOSでは，第一引数だけでなく，複数の引数について特定化される．そして適用可能メソッドは特定化の優先度によって並べられる．その並べ方は，引数特定子の中で左にあるものを最も先にした辞書式順序である．

たとえば，

```
(defclass foo () ...)
(defclass bar (foo) ...)
(defclass baz (bar) ...)
```

とあると，foo, bar, baz の順に優先度があがる．これに対して，

```
(defmethod test ((x bar) (y baz)) ...)
(defmethod test ((x foo) (y baz)) ...)
(defmethod test ((x foo) (y bar)) ...)
(defmethod test ((x bar) (y foo)) ...)
```

と4つのメソッドがあったとする．これらをそれぞれの引数特定子に指定されたクラスを取って，test[bar, baz]，test[foo, baz]，test[foo, bar]，test[bar, foo] と呼ぶとしよう．このとき，

```
(test x y)
```

という呼出しにおいて（x, y は baz のインスタンスとすると）test の適用可能メソッドは上の4つであり，それらは，

```
test[bar, baz], test[bar, foo],
test[foo, baz], test[foo, bar]
```

の順に並べられる．

その結果，test[bar, baz] が一番特定的であるので，

そのメソッドが実行されることになる．これが基本メソッドとなる．x, y がともに bar 型だったとすると適用可能メソッドは順に，test[bar, foo], test[foo, bar] の2つであり，test[bar, foo] が実行される．インスタンス指定がある場合，それはクラス指定に優先する．

このメソッドサーチ機構の効率は，CLOS を用いて作られたプログラムの性能に大きな影響を与える．汎用のサーチ手続きを1つ作っておいてそれにすべてを任せるようでは，実行効率は良くない．このため，良い手法の開発が将来への課題として期待される．

なお，将来の問題として，関数はすべてメソッドとして記述する（defun と defmethod の同一視）という考え方も存在し，この観点の可否の判断は良い手法の開発に委ねられる．

（2）メソッド定義の性能

次の例は CLOS を用いて，elt 関数の一部の機能を記述してみたものである．（なお，simple-string および simple-vector は CLOS では，正しいクラスとしては見なされていないので，このまま実行させることはできない．ここでは，性能を考えるための例として導入している．）elt 関数は，参考文献 1) で定義された2引数の関数であり，Common Lisp の特徴的な機能である列（sequence）のデータを第一引数とし，その列の中の取り出したい位置を第二引数とする．

```
(defmethod elt
        ((seq simple-string) index)
    (schar seq index))
(defmethod elt
        ((seq simple-vector) index)
    (svref seq index))
(defmethod elt ((seq list) index)
    (nth index seq))
```

この3つの定義により，

もし，seq が simple-string 型であれば，(schar seq index) を，

もし，seq が simple-vector 型であれば，(svref seq index) を，

もし，seq が list 型であれば (nth index seq) を実行せよ

という処理を記述している．これらの定義は独立しており，後から別の型に対する同名のメソッドを追加することもできる．

これらを Common Lisp で，1つの関数として定義するとすれば，次のようになる．

```
(defun elt (seq index)
    (typecase seq
        (simple-string (schar seq
         index))
        (simple-vector (svref seq
         index))
        (list (nth index seq)) ))
```

この defun による elt の中の typecase による場合分けは，当然のことながら，コンパイルされたとしても必ず実行時に生じる．Common Lisp では declare あるいは the といった宣言の手段も用意されているが，たとえば，(elt "abcde" 2) のような型の確定した呼出しにおいても，それらの情報を生かして，(schar "abcde" 2) のみのコードを生成することは困難である．（もちろん，Common Lisp 組込みの elt ではこうした問題はない．ここでは例として elt を自分で定義することを題材としている．また，マクロを使い，生成コードの最適化をすることもできるが，マクロにはマクロとして考慮すべき問題が別に生じることになる．参考文献13)の第9章参照．）

defmethod を用いると，引数のおのおのの型に対する定義は独立しているので，実行時に，defun でのような利用者によって指定された場合分けは起きない．システムに任され，そのための効率の良いアルゴリズムを用意できる．そしてそのほうが一般に typecase をする場合より良い性能を達成できる．あるいは，最も良い場合には，まったく場合分けは起きずに直接対応するコードが起動される．

こうした性能に関する興味深いコメントが G. Kiczales 氏により報告されている．**囲み記事**を参照してほしい．

3.5　メソッド結合

3.4 の（1）で述べたように，「最もふさわしいメソッドが選ばれ実行される」というのが基本ルールであった．しかし，たった1つだけを選ぶというのではなく，他の考え方もある．たとえば，可能なものはすべて実行するなどである．メソッド結合は複数のメソッドを実行する枠組みである．

CLOS では三種類のメソッド結合ができる．

（1）　1つは，call-next-method による上位のメソッドの呼出しである．

CLOS では，call-next-method という関数呼出し機構を用意し，これにより陽なメソッド結合を実現させて

● CLOS の性能は良いのか　Gregor Kiczales

Date: Fri, 24 Jun 88 18:08 PDT
From: Gregor.pa@xerox.com.
Subject: Re: CLOS Speed

<Brad. Myers の以下の趣旨のメイルに対する返事として,『CLOS を使い続けていきたいが, 問題はスロットアクセスとメソッド呼出しの速度が大変遅く思われ, struct と通常の手続きより, 3～5倍かかることである』>

CLOS を用いるのと普通の関数を用いるのとの間での実行性能の違いを考えるのにはたくさんの方法がある. この2者の間の3つの比較を見てみたい.

(1) 総称関数か普通の関数か?

総称関数と普通の関数ではすることが異なる. 呼出しの性能の違いを比較するのは面白いが, そこで得られる数値は興味の対象とはならない. 本当に比較すべきなのは総称関数と, それと等しい効果をもつ普通の関数と typecase の組である.

例を次に示す.

```
(defclass c1 () ())
(defclass c2 () ())

(defun test-generic-function (x)
  (let ((c1 (make-instance 'c1)))
    (time
      (dotimes (i x)
        (test-generic-function-internal c1)))))

(defmethod test-generic-function-internal ((c1 c1)) 'c1)
(defmethod test-generic-function-internal ((c2 c2)) 'c2)


(defstruct (s1 (:constructor make-s1)) dummy)
(defstruct (s2 (:constructor make-s2)) dummy)

(defun test-typecase (x)
  (let ((s1 (make-s1)))
    (time
      (dotimes (i x)
        (test-typecase-internal s1)))))

(defun test-typecase-internal (x)
  (typecase x
    (s1 's1)
    (s2 's2)
    (otherwise (error "Don't know what to do with ~S." x))))
```

私がテストした2つの処理系では, test-generic-function は, test-typecase より速い. 幾つかの数値を示すと,

10万回	総称関数	typecase
処理系1	396	2048
処理系2	184	180
処理系3	560	795

これらの処理系のおのおので, 今後総称関数のルックアップは改良できると思う. その場合, 数値は次のようになるだろう.

処理系1	300	2048
処理系2	80	180
処理系3	400	795

(ここでは単位はわざとはずしてある. 重要なのは比であるから)

総称関数と, それに対応する機能と比較するといい値ができることがわかる.

しかし, 覚えておかねばならないのは, typecase が (将来とも) 不要で, 普通の関数だけでよいのであればそっちのほうがいい. それを必要としないなら, 総称関数を使うべきでない. 総称関数が使えるなら総称関数を使うことがしたいことをするためのもっとも速い方法だろう.

(2) defstruct アクセサ対 slot-value

slot-value を使うと, よく最適化された defstruct アクセサと比べて2倍かかると思われる. よく最適化された defstruct アクセサは, 基本的にメモリ読出し (aref) 1つである. スロットアクセスは基本的にメモリ読出し2つである. 2つ目の読出しとは, 多重継承をサポートするのに必要な間接アクセスの分である.

(3) defstruct アクセサ対アクセサメソッド

ほとんどの処理系で defstruct アクセサはコンパイラによりインラインに展開される. したがってメモリ読出し1回になる. このことは, 高い実効性能を与えるが, 一方で, そのアクセサに対する実際の呼出しはコンパイルされみえなくなるので, そのアクセサの変更は, そのアクセサを用いているすべてのプログラムの再コンパイルなしには不可能である.

defstruct アクセサは本質的にマクロである.

CLOS では, defclass によって生成されたメソッドであるアクセサメソッドをもつ総称関数は他の総称関数と同じである. つまり, コンパイラはそれらに対する呼出しをコンパイルして見えなくすることはない. 実行時にメソッドルックアップが生じ, 実行させるのに適切なメソッドを得る. これは, たくさんの柔軟性を与えている. これらのアクセサを特殊化 (specialize) するメソッドを定義することもできる.

また, CLOS アクセサをより高速にする最適化も存在する. CLOS アクセサの想定できるパフォーマンスは関数呼出しの1.3倍である.

以上で CLOS と defun+typecase+defstruct との間の真の比較の幾つかをクリアにしたと期待する. もちろん実際の問題はもっと複雑であり, ベンチマークで扱うようなものかもしれない. ここで強調したかったことは, CLOS が提供している機能を必要とするとき, その機能を達成するには CLOS が一番性能の良い機構である点である.

(注) この記事は G. Kiczales 氏により電子メイル討論の過程で, 電子メイルとして作成されたものを, 本人の指示と許可に基づいて掲載するものである. その意味で重要な主張を含んでいるが, 完全な論文となっていないことを, おことわりしておく.

訳: 井田昌之

いる．call-next-method と書くと，“次の” メソッドが同一引数で呼び出される．“次の” メソッドとは，適用可能なメソッドの並びの中で次のものである．3.4 の（1）の test の例では，test[bar, baz] の次は test[bar, foo] である．これを同一引数で呼び出すことになる．

（2）　もう1つは，メソッド修飾子（method qualifier）を指定したメソッド実行の組合せである．

このためにメソッド修飾子として，:before，:after，:around の3つが用意されている．これらの修飾子をもつものは「補助メソッド」（auxiliary method）と呼ばれる．

適用可能な :before メソッドはすべて集められ，基本メソッドの前にすべて実行される．また，適用可能な :after メソッドはすべて集められ，基本メソッドの後にすべて実行される．

:around メソッドは，基本メソッドの「まわり」（around）で実行される．:around メソッドは，call-next-method によって呼び出され，またその中で別の call-next-method を用いて次に特定的なメソッドを呼び出すことができる．

（3）　第3の方法として，define-method-combination によるものがある．

これは，メソッド結合の仕方を利用者が定義するもので，単純に，結合法を and，or，その他の既定の方法に指定する形式と，メタオブジェクトの問題と関連するが，独自の結合方式を定義する形式とのの2つが可能である．

メソッド結合については後の章に詳しく説明されるので，ここではこれ以上ふれない．

3.6　with-slots および with-accessors 構文

CLOS を使ってプログラムを書くとき，スロットを扱うためには（slot-value …）もしくはアクセサ関数を多用することになる．書く上の手間も，書かれたプログラムを読む上でも煩雑となる．with-slots および with-accessors 構文は，スロットアクセスの記述を簡略化するシンタクスシュガーである．

例を示す．

```
(defmethod move
        ((r rectangle) new-x new-y)
    (with-slots (x y) r
          ;slot x and y of instance r
      (setq x new-x y new-y)))
```

ただし，x と y は rectangle のスロットである．setq の中の x は（slot-value r 'x）のように扱われる．with-accessors を使うと次のようになる．

```
(defmethod move
  ((r rectangle) new-x new-y)
   (with-accessors
     ((x rectangle-x) ;accessor for x
      (y rectangle-y));accessor for y
       r               ; instance r
      (setq x new-x y new-y)))
```

これにより，setq の中の x は（rectangle-x r）として扱われる．x のアクセスにおいて総称関数としての扱いが必要であれば，with-accessors を使うと便利である．ただし，アクセサがそのクラスの defclass において定義されていなければならない．

4.　CLOS によるプログラム例

CLOS プログラム例として，CLX[23] インタフェースを用いて，簡単なグラフィックシステムを書いてみた．この例は本章を通して参照されてきたものである．プログラムは S3620 で作動し，表示は Sun で行なっている．なお，これは CLOS/CLX のサンプルとして書いたものであり，動作はするがシステムとして完璧ではないし，また余分なものも入っていることをお断りしておく（図2）．

CLX は，X11 の配布の一部として R. Scheifler 自身が中心となって作成した Common Lisp によるツールパッケージである．CLX 自身はオブジェクト指向機能によっていない．そういった点からXと Common Lisp と CLOS がどういう関係になるかは今後の問題である．もちろんX以外のウィンドウ・システムもある．図2のプログラムは，こうした点に関連して筆者の研究室で行なっている実験でのプログラム例である．

このプログラムを動かす場合には，次の注意が必要である．図2の in-package の cluei の use は，このリストを取ったときには PCL でなく，TI の CLUE により CLOS の解釈をさせたためである．なお，CLUE の場合 :writer は動かないので，その部分はエラーになる．（Symbolics ではそれを無視させて実行を進めた．）

実行の手順としては，S3620 で（try）と入れると，ホスト kepa(SUN-4) にウィンドウがとられる．この後，たとえば，

```
(setq a (make-instance 'line 'x 100
```

```
;;; -*- Mode: lisp; Syntax: Common-lisp; Package: XLIB; Base: 10; Lowercase: Yes -*-
;;;
;;;
;;;  A Sample system with POINT LINE RECTANGLE
;;;    using CLOS and CLX
;;;

;;;  written by Masayuki Ida (Aoyama Gakuin University) 1988.08.15
;;;

(in-package 'xlib :use '( cluei lisp))

(defvar *display* nil)
(defvar *screen* nil)
(defvar *root-window* nil)
(defvar *window* nil)
(defvar *gcontext* nil)
(defvar *white-pixel* nil)
(defvar *black-pixel* nil)


(defun try (&optional (host "kepa"))
  (setq *display* (xlib:open-display host))
  (setf (xlib:display-after-function *display*) #'xlib:display-finish-output)
  (setq *screen* (car (xlib:display-roots *display*)))
  (setq *root-window* (xlib:screen-root *screen*))
  (setq *black-pixel* (xlib:screen-black-pixel *screen*))
  (setq *white-pixel* (xlib:screen-white-pixel *screen*))
  (setq *window*
        (xlib:create-window :parent *root-window*
                            ;; These are ignored.
                            :x 400
                            :y 3
                            :width 300
                            :height 300
                            ;; This is ignored if bs not supported.
                            :backing-store :always
                            :border-width 2
                            :border *white-pixel*
                            :background *black-pixel*
                            :event-mask (xlib:make-event-mask :button-press)))
  ;; Set up main window's properties.
  (setf (xlib:wm-name *window*) "Demo -- Draw")
  (setf (xlib:wm-icon-name *window*) "Demo -- Draw")
  (setf (xlib:wm-normal-hints *window*)
        (xlib:make-wm-size-hints :x 400 :y 3 :width 300 :height 300))
  (xlib:map-window *window*)
  (setq *gcontext* (xlib:create-gcontext :foreground *white-pixel*
                                         :background *black-pixel*
                                         :drawable *window*))
  (xlib:display-finish-output *display*)
  )
```

図 2 CLOS/CLX による線画システム例

```
;;; class POINT

(defclass point ()
  ((x :initform 0 :initarg x :writer move-x)
   (y :initform 0 :initarg y :writer move-y)))

;;; class LINE

(defclass line (point)
  ((length :initform 1 :initarg length :writer change-length)
   (direction :initform 0 :initarg direction :writer change-direction)))

;;; Class RECTANGLE

(defclass rectangle (line)
  ((height :initform 1 :initarg height :writer change-height)))


;;;
;;; Generic Function DRAW
;;;

(defmethod draw ((p point))
  (with-slots (x y) p
     (xlib::draw-point *window* *gcontext* x y)))

(defmethod draw ((l line))
  (with-slots (x y length direction) l
     (multiple-value-bind (new-x new-y)
         (get-another-end x y length direction)
       (draw-line-internal x y new-x new-y))))

(defun draw-line-internal (x y xx yy)
  (xlib::draw-line *window* *gcontext* x y xx yy))

(defmethod draw ((r rectangle))
  (with-slots (x y length direction height) r
     (multiple-value-bind (x1 y1)
         (get-another-end x y length direction)
       (multiple-value-bind (x2 y2)
           (get-another-end x1 y1 height (+ direction (/ pi 2)))
                          ; pi is the global constant defined in CLtL
         (multiple-value-bind (x3 y3)
             (get-another-end x y height (+ direction (/ pi 2)))
           (xlib::draw-lines *window* *gcontext*
                             (list x y x1 y1 x2 y2 x3 y3 x y)))))))

(defun get-another-end (x y length direction)
  (values (round (+ x (* length (cos direction))))
          (round (+ y (* length (sin direction))))))
```

図 2 （続き）

```
                    'y 100 'length 10))
  (draw a)
```

とする．line クラスのインスタンスが作られ，draw により長さ10の線が（100, 100）より表示される．また，四角を表示するメソッドも用意してみた．テスト使用の終わりにウィンドウを閉じるには，(close-display *display*) を実行させる．

5. CLOS の位置付け，将来

CLOS 第1章及び第2章の最終仕様は，ようやく ANSI X3J13 によって確定した．一方，CLOS の処理系は，まだ完全にできているわけではない．それぞれの Common Lisp 処理系/機械の上での適合設計が必要なテーマである．Symbolics 上では関数呼出しと比べてメッセージ送信は2倍のステップしかかからない，という報告も筆者の手元には届いている．加えて，基本的な処理アルゴリズムの改良という大きなテーマがある．

こうした点などを含めて，主に CLOS の基準処理系である PCL のメイルを中心にネットワーク上での公開討論などが行なわれている．そのメイルボックスは，CommonLoops.pa@xerox.com である．また，ウィンドウ・システムに関する電子メイル討論にも参加している．これらについては ida@aoyama まで連絡をいただければ，国内での再配布をしている．

CLOS に対する現在の主要な研究課題を思い付くまま並べると，メソッドサーチの高速化，クラス・インスタンスの内部表現の検討，より良い機能の付加，グラフィック・インタフェース，マルチウィンドウ・インタフェースなどが考えられる．これらに加えて，メタクラス概念の柔軟性を生かしたコード生成の最適化や Prolog との融合などという視点も存在する[9)~11)]．また，プログラミング環境への影響も大きなテーマであろう．ソフトウェア工学的な視点も存在する．

なお，日本国内での PCL は，筆者が直接に電子的な手段で連絡できるところへはソースコードを含めて配布してよいという許可をもらっている．本稿執筆時点で28の大学/企業へ配布されている．

謝　辞

CLOS の開発は，D. G. Bobrow 氏，G. Kiczales 氏（Xerox PARC）らを中心とするものである．特に，早期から，資料の郵送，法的な問題の解決，筆者の PARC 訪問を含めた研究交流に対して配慮を受けたことを特記したい．

電子協 Lisp 技術専門委員会の諸氏，特に，PFU の大久保氏，NTT の石田氏（'86年度まで）を始めとするオブジェクト指向 WG との討論は有益であった．なおこのグループでは '86 年10月より PCL ソースの検討を始めている．また，関連した報告をまとめ，参考文献 17-1, 17-2, 17-3）に記している．

CSnet/junet を介した通信は，石田晴久教授（東大），和田英一教授（東大）を始めとする多くの研究者の助力がなければ継続できなかった．また，東京大学大型計算機センターにおける PCL の移植は，同センター石田晴久教授との共同研究による．ここに，謹んで感謝いたします．

参考文献

1) Guy L. Steele Jr. et al.: *Common Lisp: the Language*, Digital Press, 1984 (邦訳は，共立出版より，井田昌之訳，1985).
2) D. G. Bobrow, K. Kahn, G. Kiczales, et al.: COMMONLOOPS: Merging Common Lisp and Object-Oriented Programming, Xerox Parc ISL-85-8, Aug. 1985.
3) D. G. Bobrow, K. Kahn, G. Kiczales, et al: COMMONLOOPS: Merging Common Lisp and Object-Oriented Programming, *Proc. OOPSLA* '86 ACM, Sept. 1986.
4) G. L. Drescher: ObjectLISP for experienced LISP programmers, LMI, Aug. 1985.
5) A. Snyder: Object-Oriented Programming for Common Lisp, HPlabs ATC-85-1, July 1985.
6) J. Kempf, A. Snyder: Hooks for Common Objects, common-lisp@su-ai. ARPA, Jan. 8, 1986.
7) S. E. Keene and D. A. Moon: Flavors: Object-oriented Programming on Symbolics Computers, Symbolics, Dec. 1985.
8) G. Kiczales: CommonLoops meets Object Lisp, CL-Object-Oriented-Programming@SU-AI. ARPA, Jan. 3, 1986.
9) K. Kahn: Ideas about CommonLoops& Logic programming, draft, Nov. 27, 1985.
10) K. Kahn: Notes from discussion about "Common Log", draft, Dec. 3, 1985.
11) K. Kahn, et al.: Objects in Concurrent Logic Programming Languages, *Proc. OOPSLA* '86, ACM, Oct. 1, 1986.
12) D. Weinreb & D. Moon: Message Passing in the Lisp Machine, MIT AI-Lab, AI memo no. 602, Nov. 1980.
13) R. Brooks: *Programming in Common Lisp*, Wiley & Sons, 1985 (邦訳は，井田昌之訳『Common Lisp プログラミング』，丸善，1986).
14) M. Ida: An Interpretation of the Common Loops specification, WGSYM35-3, IPSJ, Dec. 1985.

15) 石田，井田，他：Common Lisp へのオブジェクト指向機能導入の動向，WGSYM36-7，IPSJ Mar. 1986.
16) 井田：Common Lisp へのオブジェクト指向機能原案の言語仕様上の特徴と問題点，WOOC '86 予稿，Mar. 1986.
17-1, -2, -3) 電子協 Lisp 技術専門委員会報告書，Mar. 1986, Mar. 1987, Mar. 1988.
18) 松木，泉ほか：CommonLoops の UtiLisp 上の実現，IPSJ 33 全国大会 2E-3，pp. 479，1986.
19) 井田：Discriminating EVAL，第三回ソフトウェア科学会全国大会，A-4-2，Nov. 1986.
20) 井田：CommonLoops のためのあるメッセージ送信機構，WGSYM 39-6，IPSJ，Nov. 1986.
21) D. Bobrow, G. Kiczales, et al.: Common Lisp Object System, Chapter 1. Programmer Interface Concepts, Chapter 2. Programmer Interface Facilities; ANSI X3J13 88-002R, June 1988.
22) D. Bobrow, G. Kiczales: Common Lisp Object System, Chapter 3. Metaobject Protocol; ANSI X3J13 88-003, Mar. 1988.
23) R. Scheifler: CLX Specification, 1987.