トピックス

Common Loops——Common Lispオブジェクト指向機能の標準化原案

井田 昌之

1 はじめに

CommonLoopsは，Common Lisp[19]へのオブジェクト指向機能原案である．その概要はIJCAI'85 Common Lisp MeetingでXerox Parcより発表され[1]，その後，ポータブル処理系(Portable CommonLoops)の公開による試用評価を経て，ACM OOPSLA'86への論文[2]としてまとめられている．CommonLoopsの名は"COMMON Lisp Object Oriented Programming System"に由来する．その最終仕様書はまだ出されていないので，若干の言語仕様細部の変更がありうる．著者は，その仕様の作成以前より研究交流を持っていたので，およその流れを知る位置にいる．日本国内においても，Portable CommonLoopsの移植があちこちで始められ，また，CommonLoopsの最終仕様はCommon Lispに組込まれるという合意が米国内では行われている．ここでは，そうした位置付けで，CommonLoopsとそのポータブル処理系の概要の紹介を行う．

Common Lispへのオブジェクト指向機能の標準案の模索は，1984年10月に始まった．モンタレーでの会議を経て，Object-Oriented-Subcommitteeが作られた．K. Kahn (Xerox Parc)がその議長である．当初は，標準案を作るか，それとも共通的な核の仕様を決めるのかといったレベルの議論がARPAnet上で行われた．1年弱の一般的な議論を経て，CommonLoopsの他にObject Lisp[5]，Snyder案[15][18]，newFlavors[14]が提案された．これらの提案の中でCommonLoopsが中心となり，電子メール討論のメールボックスもスタンフォード大学からXerox Parcへ移り，今日に至っている．国内における資料としては，[4][6][7][8][9][10][17]などがある．[4]にはこれらの経緯がまとめられている．

Masayuki Ida，青山学院大学理工学部．
1986年12月15日受付．

いくつかの提案の間でCommonLoopsが良い評価をを得た理由は，その言語設計の優秀さと斬新さによるが，加えて，Xerox Parcの努力による点も大きい．たとえば，オープンな議論に基づき，[5][14][15][18]の提案に盛り込まれた仕様のCommonLoops上での実現可能性を実証するなどしている．(たとえば，[16]はそのベースとなった資料である)．

ポータブル処理系(Portable CommonLoops．以下PCLと呼ぶ)は，Common Lispで書かれており，Xerox Common Lisp (1100SIP)，Zetalisp (Symbolics 3600)，Lucid-Common Lisp (Sun3)，TI-Common Lisp (TI-explorer)，Vaxlisp (DEC VAX/Ultrix)，KCL (VAX 4BSD Unix)，ExCL Common Lispなどでの動作が確認されている．これら各々のための低レベルフックを含めて，PCLのソースコードはARPAnet上でXerox Parcより公開され，パブリックドメインにある．

2 CommonLoopsの特徴

CommonLoopsの特徴はオブジェクト指向機能を二つの部分に分けることから始まる．すなわち，

defstructの拡張としてのクラス定義，

関数呼出し機構の一般化としてのメッセージ送信，

である．

そして，論文[1][2]のタイトルにもあるように，Lispとオブジェクト指向機能の融合に目標がある．

2.1 defstructの拡張によるクラス定義

1) defstruct

defstructはCommon Lispが持つ構造型の定義手段である．その仕様の詳細は[19]に記されているが，

```
(defstruct  構造名
            スロット定義
            …)
```

という形式をとり，構造の定義を行う．

たとえば，平面上の点を表す，次のような定義が書ける．

```
(defstruct position
        (x 0)
        (y 0))
```

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

```
(make-position)
```

もしくは，スロットの初期値を同時に与えて，

```
(make-position :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，position-xおよびposition-yという関数が自動的に作られる．

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

ののち，

```
(position-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，CommonLoopsに影響が強いものとしては，他のdefstructを引用して定義を行う:includeをあげることができる．これは，のちに述べる多重継承の実現に拡張される．

2) CommonLoopsでのクラス定義

CommonLoopsのクラス定義は，基本的にdefstructで行う．このために，defstructオプションに:class指定が追加されている．:class指定があると，CommonLoopsのクラス定義で許されたいくつかのオプションが可能になる．

最も簡単な例を次に示す．

```
(defstruct (position (:class class))
        (x 0) (y 0))
```

(:class class)は，positionを「標準的なクラス概念」で定義することを表す．これは逆に，「標準的でないクラス概念」によるシステムをその上に組込めることを意味している．そのベースになるものがメタクラスである．自分でメタクラスを記述することにより，セマンティクスをある程度変更することができる．

3) 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．(:class class)という指定はCommonLoopsで標準としているメタクラス，すなわちclassという名のメタクラスによりすべてが司られることを指定している．型階層のセットアップを始めとして，クラスclassに関するすべてのメカニズムは起動時に初期ロードされる．すなわち，CommonLoops処理系とクラスを扱う機構(メタクラス)は分離されており，ここからCommonLoopsの柔軟性が得られている．

4) 組込みクラスとしての型階層

図1にCommon Lispの型階層を示す．そこに記され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図に示される階層的な順序関係をそのままもちこんでいる．

たとえば，number型>integer型>fixnum型の関係から，もし，fixnum型の実体のメッセージ送信の際，fixnum型に対して定義されたメソッドが無く，number型に対するメソッドがあるならば，number型に対して定義されたメソッドが起動される．

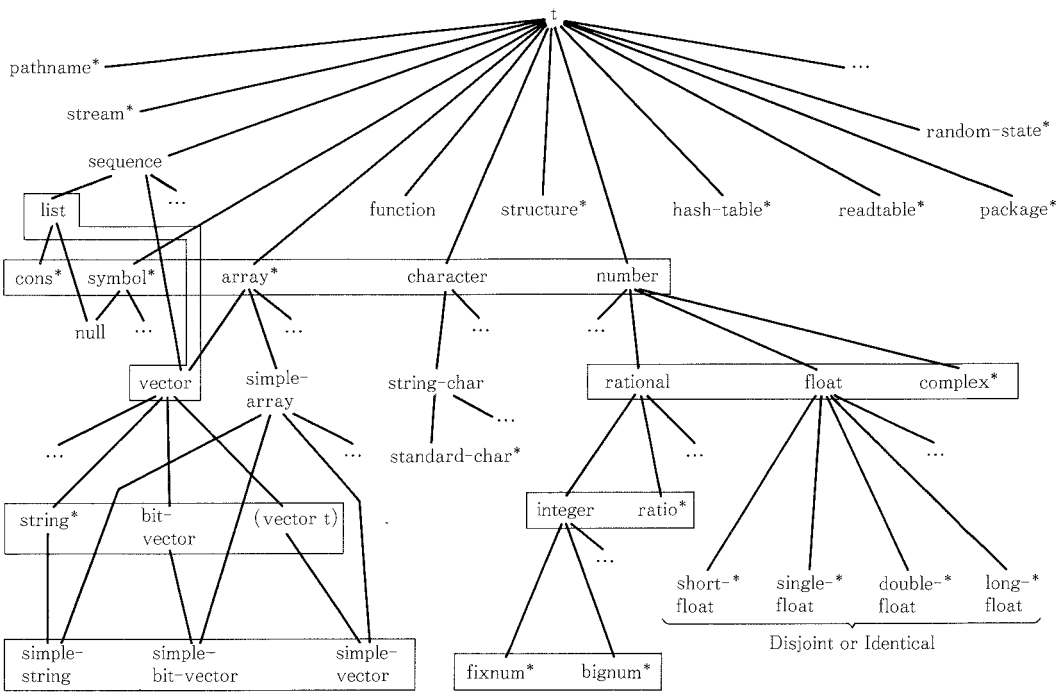
型階層に関連して注意したいことは2点ある．一つはsimple-arrayの導入により，束(lattice)にならなくなる点である．束は任意の二要素に対して，交わり(glb)と結び(lub)をもつ半順序集合のことであるが，vector型とsimple-array型の二つをとった時に交わりを決定できない．もう一つは，vector(そしてその下のクラス)とnullには各々二つのスーパークラスがあり，いずれを優先するかという点である．null型にはlist型とsymbol型があり，simple-…型にはvector型とsimple-array型の二つがある．[1]は，null型(nil)はlist優先，すなわちlist型はsymbol型に優先することを定めている．vector型とarray型の関係については定めていない．[6]は，これを補う形で，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．

CommonLoopsでは，Common Lisp組込みの型のクラスしか使わない場合には，全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．これもCommonLoopsの大きな特徴の一つである．

2.2 関数呼出しの一般化としてのメッセージ送信

1) メソッドの特定とclassical-method

Lispの基本的な手続き呼出し機構は関数呼出しである．CommonLoopsではこの関数呼出しの一般化として



情報処理学会記号処理研究会資料 30-2 井田昌之：ポータブルバックエンド型 LISP プロセッサ ALPS/III の構想より．

図 1 Common Lisp の Type Hierarchy

メッセージ送信を扱う．

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，この例だけで言えば，Common Lisp の symbol-function などを想定すればよい．CommonLoops を考えない Lisp だけの世界では，f という名を持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．symbol-function への setf などをしている特殊な例を除けば，defun で定義されたものか，静的に束縛されたもののいずれかである．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無いことである．たとえば，第一引数は数型，第二引数は文字列型を仮定した二引数の関数 f があるとする．Lisp の言語仕様としては，f の呼出し時に，引数の個数が合わなくとも，あるいは，想定された引数の型と合わなくともそれらに対処することは求められていない．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[20]([14]で提案されている 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 CommonLoops 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，メッセージ送信は旧 Flavors での send のような特定の関数の中に閉込めず，単に，関数呼出しの形式で記述する．次のようになる．

(f a b)

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，これだけを見てもわからない．(区別の方法については，2.5 で述べる)．

[12]に立脚して眺めるならば，

(send a :f b)⟶(f a b)

という変換が生じていると思ってもとりあえず理解することができる．なお，newFlavors[14]ではすでに，この関数呼出しとしてのメッセージ送信を採用し，send によるシンタックスを採っていない．この点が Common-Loops を中心とした標準化の印象を強くさせている．

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 と呼ばれる．また，メソッドの特定に，一つの引数が関与しているので，single-method とも呼ばれる．

2) multi-method

メソッドの特定に第一引数を参加させるだけに留める必然性は特に無い．与える総ての引数をメソッドの特定に関与させる仕組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)

という解釈が行われる機構を考えることができる．メソッドの特定に二つの引数を関与させる．複数の引数がメソッドの特定に関与するメソッドは multi-method と呼ばれる．

multi-method の場合，lambda-list keyword problem というものが存在する．これは，第一に，メソッドの定義で &optional, &rest, &key などを許すかということ，第二に，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．第一の問題に対しては「許す」，第二の問題に対しては「関与しない」という方針を[6]は与え，PCL に取入れている．

2.3 defmeth：メソッドの定義

メソッドは defmeth を用いて定義する．defmeth の構文は defun に似ている．異なる点は，引数の記述の中に，それが属するクラス名を合わせて書ける点である．また，同じ名前のメソッドは複数存在しうるから，その点も必然的に defun とは異なることになる．

```
(defmeth 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 classical-method である．

一方，

```
(defmeth f ((x foo) (y bar))
           ...)
```

は，x が foo 型，y が bar 型であることを指定している．したがって，これは multi-method である．

これらの定義における f をセレクタと呼ぶ．

2.4 例

次の例は CommonLoops 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．

elt 関数は，[19]で定義された二引数の関数であり，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，その列の中の取出したい位置を第二引数とする．

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

この三つの定義により，第一引数が simple-string である場合のもの，simple-vector である場合のもの，list である場合のものの各々についての処理を記述できる．最初のものから順に説明をすると，

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

という意味になる．

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，一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，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無い．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[3, 第9章])．

defmethを用いると，引数の各々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．システムに任される．あるいは，もっとも良い場合には，全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．

2.5 メソッドサーチ

メソッドサーチのアルゴリズムは，最も特殊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，multi-methodの場合，左優先(left-to-right)のサーチが組合わされて形成される．

multi-methodの例を示すと，

(defmeth foo ((x number) (y list)) …)

と，

(defmeth foo ((x fixnum) (y sequence)) …)

という二つのメソッドが定義されていたとすると，第一引数がfixnumであれば，必ず，後者のメソッドが呼出される．もちろん，第二引数はその場合，sequenceでなければならない．CommonLoopsは，複数のmulti-method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．

このメソッドサーチ機構の効率はCommonLoops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一つ作っておいてそれに総てを任せるようでは実行効率は良くない．また，(f a b)という形式それ自身のみでは，それが普通の関数呼出しか，メソッド呼出しかを判定できないという問題もある．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defmeth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．少なくとも，コンパイラでは2.4で述べられたような最適化を極力行うことが必要である．インタプリタではサーチを行う機構の組込みには各々の工夫が必要となる．たとえば，PCLではセレクタの関数セルにクロージャとしてサーチ手続きとメソッドの融合体をコンパイルし，与えている．これにより，従来からevalを変更せずに，普通の関数でもメソッドでも随意に効率良く呼出せるようにしている．

3 Portable CommonLoops(PCL)でのクラス定義

3.1 ndefstruct

クラス定義のできるdefstructをPCLでは便宜上，区別してndefstructと呼んでいる．ndefstructのおおよそのsyntaxを次に示す．

```
(ndefstruct
        (name (:class class)
                [(:include local-supers)]
                [その他の defstruct-options]
        )
    [スロット名]
```

```
[(スロット名　初期値
  [:accessor　アクセサ]
  [:allocation {instance, class,
                dynamic}]
  [:get-function　ラムダ式]
  [:put-function　ラムダ式]
 … )]
…
)
```

クラスのインスタンスの生成は，defstruct の基本機能である make-xxx を用いても良いが，一引数の make という形式が用意されており，(make class-name) としても良い．なお，

(make class-name :slot the-value)

のように，スロットの初期値を同時にセットすることもできる．

また，(:class class) を指定することが多いので，ndefstruct の上に，

(defclass クラス名　スーパクラスの並び
　スロット定義の並び　オプション)

のような構文を用意し，Loops や newFlavors に近い記述を許すことなどもでき，実際に試みられている．

3.2　:include 指定の拡張による継承，多重継承

:include オプションの拡張として継承を定義する．継承されたクラス定義でさらに，:include があってもよい．:include では，(:include (クラス1 クラス2…)) のように，複数のクラスを指定できるという言語仕様の拡張も行われている．これにより，多重継承が実現される．スロットだけでなくメソッドも受けつがれるが，この場合のメソッドのサーチは，「:include で記述された順序に従い，かつ，自分自身のクラスにあるものを優先する」というアルゴリズムが，標準メタクラスである class クラスでは採用されている．このサーチ手順は，left-to-right and depth first up to joins と呼ばれる[2].

:include の簡単な例を示す．

```
(ndefstruct (foo (:class class))
            x
            y
            )
(ndefstruct (bar (:class class)
                 (:include foo))
            z)
```

とあると，bar は x, y, z のスロットを持つ．あるセレクタに対し，bar クラスを指定した defmeth が無ければ，foo を探しに行く．

3.3　ndefstruct のスロットオプション

1)　:allocation {class, dynamic, 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，dynamic 指定は最初のアクセスでクラス定義上に生成することを，instance 指定はインスタンスに割付けることを，各々意味する．無指定の場合は instance 指定が仮定される．たとえば，

```
(ndefstruct (foo (:class class))
            (x :allocation class)
            (y :allocation dynamic)
            (z 0)
            )
```

と記述すれば，x はクラス foo に対してグローバルに割付けられる．従って，foo の各インスタンスに対して，ただ一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 は dynamic 指定なので，インスタンス生成時に自動的に y スロットが割付けられることは無く，y のアクセスがあった時に初めて作られる．z は，デフォルトで各インスタンスに割付けられる．

2)　:get-function, :put-function スロットオプション

:get-function および :put-function の後に各々ラムダ式（もしくは関数名）を置く．これらの関数は，スロットアクセス時に呼出される関数である．これらは一引数の関数でなければならない．

簡単な例を示す．

```
(ndefstruct (foo (:class class))
            (a () :get-function
               (lambda (obj)
               ;obj は foo 型のインスタンス
                  (or (get-slot obj 'a)
               ;obj の a スロットをアクセス
```

(read)))))

(make 'foo) をした後，(foo-a) のアクセスがあると get-function で指定された関数が起動される．その時のスロット a の値があれば(nil 以外であれば)，それが返され，そうでなければ read が生じるという annotated-value のようなことが起る．

なお，この定義の中で用いられている get-slot は，直接スロットアクセスを行う CommonLoops の関数である．対応する put 関数として，put-slot があり，(put-slot instance slot-name value) の形式を持つ．なお，これらは伝家の宝刀ともいうべき関数で，これらを用いれば外部から自由にスロットの内容を変更できる．

3.4 Undeclared-slots

これは動的に付加されるスロットに対する概念の総称である．Undeclared-slots に属するものとしては，次の二種類がある．

1. :allocation dynamic により，動的生成が指定されたスロット．このスロットは最初のアクセス時に生成される．
2. 全く定義が無いスロット．このスロットは get-slot-always もしくは put-slot-always というアクセス関数の使用により，実行時に生成されるものである．これは，そのインスタンスに割付けられる．

Undeclared-slots に対する操作として，次の三つが用意されている．

(get-slot-always instance slot-name)
スロットの内容を取出す
(put-slot-always instance slot-name value)
スロットの内容をセットする
(remove-dynamic-slot instance slot-name)
スロットを削除する

(なお，get(put)-slot-always は[2]の get(put)-dynamic-slot に各々対応する)．

3.5 クラスとメタクラス

一般の利用者はメタクラスを理解しなくともよい．2. 1の2)と3)で述べたように標準のクラス概念(メタクラス)に従ってプログラムを作るからである．ここでは，より深い興味を持つ研究者のためにクラス機構の概要を説明する．

PCL の上でメタクラスを利用者が定義するためには，「クラスを規定する構造」の規定が不可欠である．その構造は essential-class と呼ぶクラスに定義されている．essential-class は(:class class)指定により自動的に include される．メタクラスを作る場合には，(:include essential-class)により，その構造を継承することができる．

essential-class では，クラス定義は次のようなスロットを持つように規定している．

1) name…そのクラスの名前．
2) class-precedence-list…そのクラスの持つクラス優先順位リスト．クラスの階層はここからたどる．
3) local-supers…直接の親のクラスのリスト．
4) direct-subclasses…直接の子供のクラスのリスト．
5) direct-methods…そのクラスを class-specifiers (引数並び)に持つ defmeth されたメソッドおよび，それに関連した setf-discriminator (setf 用のメソッド特定のための手続き)，あるいは ds-option によるメソッドの並び．

ここに入れられる各メソッドは，

関数名，discriminator のアドレス，
type-specifiers，引数リスト

が並んだ構造を持つ．

6) no-of-instance-slots…インスタンスに付くスロットの個数．
7) instance-slots…インスタンススロットのリスト．
8) non-instance-slots…dynamic, class などの割付けのスロットのリスト．
9) local-slots…ndefstruct されたスロットのリスト．

ここではこれ以上その詳細には触れないが，利用者がここに触れた構造を理解すれば，メタクラスを定義できる．その場合，さらに二つの方法がある．一つは，メタクラスを ndefstruct し，(その中では正しく各定義を埋め)そのクラスを指定した defmeth をする方法，もう一つは，CommonLoops の処理系に組込んでしまう方法である．後者の場合には，以後の処理で，(:class xxx)として，利用者の定義したメタクラス xxx を ndefstruct において指定できるようになる．

次に，クラスを make する(インスタンスを生成する)時に作られるものを示す．図2は，インスタンスの構造の概略である．インスタンスは，そのインスタンスが属

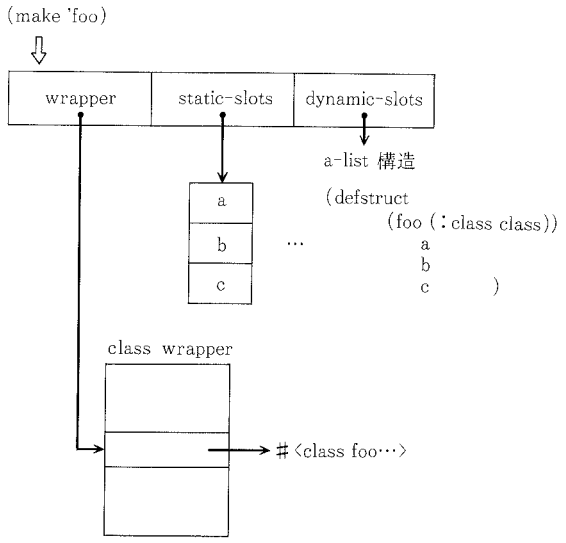


図 2　インスタンス(iwmc)の構造の概略

しているクラスへのポインタ表(class wrapper)へのエントリ，インスタンスに付いているスロットへのエントリ，動的なスロットへのエントリから成る．このオブジェクトはPCLでは，iwmc (instance-with-meta-class)と呼ばれている．これは，PCLを作る際にクラスのクラス，すなわち，クラスをインスタンスとするクラスの定義から始めたことに関係があるネーミングである．

なお，2.4で述べられた「型」が構成するBuilt-in-classメタクラスは独特のものである．Built-in-classのインスタンス，すなわち，「型」は特殊なクラスとして定義される．「型」にはスロットが無く，また，make (3.1節)によりインスタンスを生成することができない．

3.6　change-class

CommonLoopsの特徴的な機能をすべて紹介することはできないが，ここに示すのは，すでに生成したインスタンスの属するクラスをあとから動的に変える機能である．

```
(change-class instance
    (class-now new-class))
```

により，instanceが現在属しているクラスから，new-classクラスへ所属が変更される．

これは，後から作られた，より特殊化されたクラスに所属を変更する場合などに用いる．たとえば，多角形のインスタンスを，あとから，四角形というクラスができたとすると，もともとあったインスタンスの内，四角形のものを移してしまう，ということを可能にする．

4　PCLでのメソッドの処理機構

4.1　メソッドサーチと discriminator

メソッドサーチの基本的な機構については2.5に述べたが，ここでは，PCLでの補助機構を述べる．

defmethされたメソッドオブジェクトは，メソッド名(セレクタ)で識別される discriminator object(あるいは単に discriminatorと呼ぶ)に総て繋がれる．メソッドの呼出し(メッセージ送信)時に，関数名がメソッドであれば，その名の discriminator objectが調べられる．原理的に言えば，discriminatorの中にある discriminating functionにより，もっとも特殊なメソッドが一つだけ選ばれ，それが実行される．実際には，キャッシュが作られ，キャッシュサーチがおきる．このキャッシュの実現には各種の方法がありうるが，PCLでは，Smalltalkのようなグローバルな表ではなく，各 discriminatorごとの表が用意される．これらのメソッドサーチに関連するすべてのものは，セレクタの関数セルにコンパイルされたクロージャとして与えられる．[20]はこの機構から離れ，連想メモリチップを用いた手法を述べている．

discriminator には少なくとも次の4つのスロットが必要である(図3)．

1)　セレクタ名

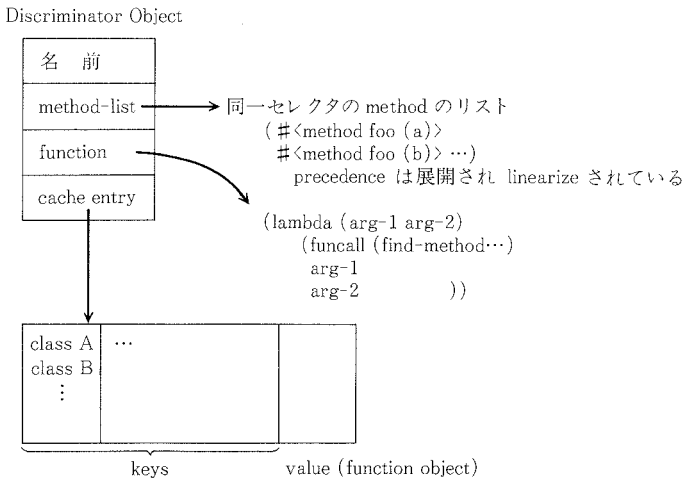


図 3 Discriminator Object の構造の概略

2) 同一セレクタのメソッドのリスト

3) discriminating function のラムダ式

4) キャッシュ表

実際の PCL では，この他に，default-method スロット，classical-method-table などを置いているが，基本的な理解には不要なのでここでは省略する．

4.2 メソッドコンビネーション

PCL では，run-super という関数呼出し機構を用意し，これにより CommonLoops で言うメソッドコンビネーションを実現している．run-super と書くと，“一つ上の”メソッドが同一引数で呼出される．“一つ上の”メソッドとは，そのメソッドを支配しているクラスの直接のスーパクラスで定義されたメソッドである．この時に，引数の変更を許すか？という問題提起が[1]にはあったが，現在では，許さないということになっている．

Flavors スタイルのメソッド結合は，new Flavors[14] ではすっきりとしたものになり，CommonLoops への融合においては:before および，:after メソッドの機構を組込むことは定められたが，全体としては，現在のところ(1986 年 10 月)まだ結論がでていない．

4.3 特異メソッド(individual method), individual specialization

class-specifiers の中に，特定の値の指定を許す機能である．

[例]

```
(defmeth find-file (name (host-name
    'MIT-AI)) …)
```

これは，引数の定義域の中での特異な点，言いかえれば，特定の値に対してのみ動作するメソッドを定義する手段である．

たとえば，n! := if n=0 then 1 else n・(n-1)!は，

```
(defmeth fact((n 0)) 1)
(defmeth fact ((n number))
    (* n (fact (1-n))))
```

と書くことができる．

4.4 with 構文

with 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．

```
(defmeth move ((b block) new-x new-y)
        (with (b)
            (setq x new-x y new-y)))
```

ただし，x と y は block のスロットである．もし，with 構文が無いと，スロットアクセスをするには単に x と書くかわりに，(block-x b)と書かなければならない．with の導入により記述の負担が大幅に減る．これは Flavors の defmethod の記述力を部分的に導入したものである．

4.5 make-specializable

これは組込まれた関数，たとえば print に対して，

独自の処理を追加する機能を与える．

```
(defmeth print ((mat matrix)
    &optional stream) …)
```

というメソッド定義を追加し，

```
(make-specializable 'print)
```

とすると，print の関数セルにはメソッドサーチ手続きが組込まれ，利用者が定義した matrix クラスに属するオブジェクトに対する処理はこれが司り，他のものは組込みの print が対処するようになる．

5 CommonLoopsの位置付け，将来，まとめ

CommonLoops の最終仕様(CLOS)は前述したように未だ確定していない．細部の点に加え，newFlavors からの機能をどこまで含めるかがポイントとなっている．なお，本稿の3章および4章に記した内容は，基本的に CommonLoops の仕様に組入れられるものである．それらを各々2章から切離したのは文法が一部変わる可能性があるからである．

また，CommonLoops 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は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．こうした点への貢献を求めて ARPAnet 上での公開などが行われていると解釈すべきであろう．筆者の提案が現在の CommonLoops に多少影響を残せたのは嬉しいことだが，今でも(おそらく，1987年上期頃まで)，内部のアルゴリズムや言語仕様に対するコメントを送れる状況は続いているので，興味のある人は挑戦してほしい．そのメールボックスは CommonLoops. pa@xerox. com である．

CommonLcops に対する現在の主要な研究課題を思いつくままに並べると，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イス，マルチウィンドインタフェイ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 Prolog との融合などという視点も存在する[11][12][13]．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．さらに注目したいのは，それぞれのモジュールに与えられる引数の型/クラスの妥当性のチェックは CommonLoops が自動的に(実行時に)してくれるので，ソフトウェアの信頼性を高めることができ，ソフトウェア工学的なツールとしても有用であると考えられる点である．これらはいずれも将来のテーマである．

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(1986年10月)で17の大学/企業へ配布されている．

謝　辞

Portable CommonLoops の開発は D.G.Bobrow 氏，K. Kahn 氏，Gregor Kiczales 氏(Xerox Parc)らによるものである．特に，早期から，資料の郵送，法的な問題の解決，筆者の Parc 訪問を含め研究交流に配慮を受けたことを記したい．

電子協 Lisp 技術専門委員会の諸氏，特に，NTT 石田享氏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1986年10月より PCL ソースの検討を始めている．

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無ければ継続できなかった．東京大学大型計算機センタにおける移植は同センタ石田晴久教授との共同研究による．本稿をまとめるにあたって査読者より有益な助言を受けた．

ここに，謹んで感謝いたします．

参考文献

[1] Bobrow, D.G., Kahn, K., Kiczales, G., et. al.: *COMMONLOOPS: Merging Common Lisp and Object-Oriented Programming*, Xerox Parc ISL-85-8, 1985.

[2] Bobrow, D.G., Kahn, K., Kiczales, G., et. al.: COMMONLOOPS: Merging Common Lisp and Object-Oriented Programming, *Proc. OOPSLA '86*, ACM, 1986.

[3] Brooks, R.: *Programming in Common Lisp*, Wiley & Sons, 1985.

[4] 電子協コモンLisp動向専門委員会報告書, Mar. 1986.

[5] Drescher, G.L.: ObjectLISP for Experienced LISP Programmers, LMI, 1985.

[6] Ida, M.: An Interpretation of the Common Loops Specification, WGSYM 35-3, IPSJ, 1985.

[7] 井田昌之: 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, 1986.

[8] 井田昌之: Discriminating EVAL, 第三回ソフトウェア科学会全国大会 A-4-2, 1986.

[9] 井田昌之: CommonLoops のためのあるメッセージ送

信機構, WGSYM 39-6, IPSJ, 1986.
[10] 石田亨, 井田昌之ほか: Common Lisp へのオブジェクト指向機能導入の動向, WGSYM36-7, IPSJ, 1986.
[11] Kahn, K.: Ideas about CommonLoops & Logic programming, draft, 1985.
[12] Kahn, K.: Notes from Discussion about "Common Log", draft, 1985.
[13] Kahn, K., et. al.: Objects in Concurrent Logic Programming Languages, *Proc. OOPSLA '86*, ACM, 1986.
[14] Keene, S. E. and Moon, D. A.: *Flavors: Object-Oriented Programming on Symbolics Computers*, Symbolics, 1985.
[15] Kempf, J., Synder, A.: Hooks for Common Objects, *Common-lisp @ SU-AI. ARPA*, Jan. 8, 1986.
[16] Kiczales, G.: CommonLoops meets Object Lisp, *CL-Object-Oriented-Programming @ SU-AI. ARPA*, Jan. 3, 1986.
[17] 松木美保子, 泉寛幸ほか: CommonLoops の UtiLisp 上の実現, IPSJ33 全国大会 2E-3, 1986, pp. 479.
[18] Snyder, A.: Object-Oriented Programming for Common Lisp, HPlabs ATC-85-1, 1985.
[19] Steele, Guy L. Jr., et. al.: *Common Lisp: the Language*, Digital Press, 1984.
[20] Weinreb, D. and Moon, D.: Message Passing in the Lisp Machine, MIT AI-Lab, AI memo No. 602, 1980.