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

井田 昌之
青山学院大学 理工学部

1。序

現在、CommonLisp[1] へのオブジェクト指向機能原案がまとめられつつある。CommonLoops[2] , Object Lisp,[3] new Flavors[4] がベースになる素案である。(IJCAI'85 において、発表されたプロポーザルとしては、この他に、A.Snyderによるもの[5] がある。Snyderの興味はオブジェクト機構のhookの提案へ移行しているように思える。[6] .) 総括的な動向、各案の比較などについては[11]にまとめてあるのでここでは触れない。

この発表では、CommonLoops に絞って議論を進める。

2 . Common Loopsの核仕様

ここでは、[2] に述べられた目標・核仕様を中心にまとめる。この作業は既に[10]に発表したが、xerox parcからの[10]に対するコメントも得たので、それによる修正を加えて、要約する。

2.1.Goals of the Common Loops

Common Loopsのゴールは次の通り

1)互換性・一般性(Compatibility)

Common Loops は通常のLispの関数的なプログラミングスタイルにできるだけあわせたい。

このために、

・メッセージセンドは関数呼出しと同一の構文による。

・Common Lisp の単純な拡張として考えられるような仕様にする

2)スモールカーネル

3)Powefull Base ... 他の、より高位のオブジェクト言語を用いなくともアプリケーションが書ける。

4)Universal Kernel... Flavors,smalltalk,LOOPS を共通的に実現できる普偏性

このためにメタクラスのアイデアがある。

5)Common Lisp Style ... Common Lisp に準拠する。

6)効率... 特殊なハードウェアが無くとも効率良く走る。

7)Extensiblity... 将来の夢への発展性

2.2.核となる仕様

a)クラス定義とメソッド定義の分離

クラス定義はdefstruct の拡張で行なう。

メソッドの定義のためには、defmethod を新設する。

b)defstruct の拡張

:classオプションを設ける。

(:class class)によりLOOPS 流の機能を

(:class flavor) によりFlavors 流の機能を

c)defmethod の新設

defun と並行した(類似した) 記述法を持つ。selfは単に、第一引数のこととなる。defun との基本的な違いは、ラムダリスト（仮引数リスト）の中のシンボルに対して、それが属するべきクラスを指定できるという点にあると考えるとわかりやすい。

例 (defstruct move ((obj block) x y) ...)

クラスとしては、defstruct 定義によるもの、Common Lisp の基本的な型があると考えるととりあえず先に進むことができる。

同一のセレクタ（関数名）を持つメソッドが複数あってよい。

d)関数呼出し形によるメッセージ送信

block1を位置33から位置120 へ移動させるには、

(send block1 :move 33 120) ではなく、

(move block1 33 120) とする。

前者は、

(funcall (method-specified-by ':move (type-of block1)) block1 33 120)

後者は、

```
(funcall (method-specified-by 'move (type-of block1) 'fixnum 'fixnum)
         block1 33 120)
```

とモデル化できる。

e)multi-method,multiple-discrimination

上記の例のように、セレクタに対する引数は、「全て」disicrimination に関与する。このような規則のもとでのメッソドをmulti-methodといい、そこでのdiscriminationをmulti-discrimination という。

例　(defmethod insidep ((w window)(x integer)(y integer))...)

関連した用語として次のようなものがある。class-specifiersとは、各引数のシンボルの属するクラスのリストである。discriminator objectとはメッセージ送信に際して、適用するメソッドの特定を行なうためのコードオブジェクトである。

f)クラス、メタクラス、class-precedence-list

メタクラスが鍵であり、そこには、それに属するクラスに対するすべての操作の組が含められる。最初のメタクラスのローディングは「きわめつけ」のプログラムとなる。class-precedence-list はclass hierarchy を規定する。Common Lisp のtype hierarchyはessential class のclass precedenceを決定する。図1 を参照。

g)多重継承

多重継承は、:includeオプションの拡張として定義する。

h)メソッドコンビネーション

run-super という関数呼出し機構を用意し、これによりメソッドコンビネーションをさせる。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。

3．拡張仕様－－－－現在の時点での論点

ここに記述された仕様は、［2］では含めるとは完全に決定されてはいなかったものである。現在においても未実装、未定のものもある。現在の仕様についての簡単なチェックリストを図2 に示す。

a)individual-specialization

class-specifiersの中に、特定の値の指定を許す機能。

例　(defmethod find-file (name (host-name 'MIT-AI))　...)

==> Yes.しかし、未実装

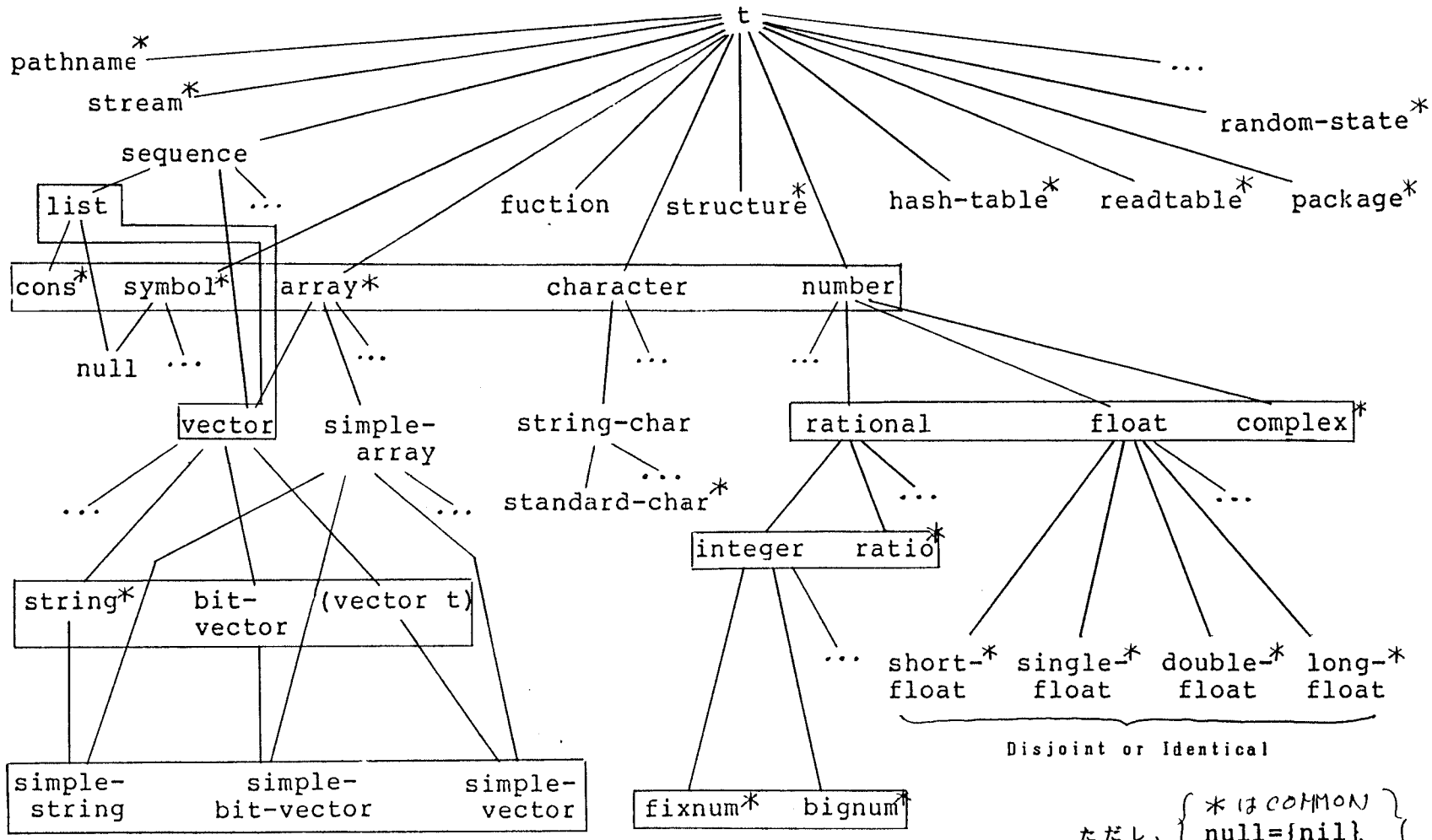

図1 Common Lisp の Type Hierarchy

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

check list of the Common Loops syntax

1. defstructs

```
(ndefstruct
     (name (:class class)
           [(:include local-supers)]
           [ other-defstruct-options ]
       )
  [slot-name only]...
  [(slot-name initial-value
        [:accessor accessor]
        [:allocation {instance|class|dynamic} ]
        [:get-function lambda-expression]
        [:put-function lambda-expression]
    ]...
  )
```

example of the get-function:

```
      (ndefstruct (foo (:class class))
                  ( a ()  :get-function (lambda (obj)
                                (print (get-slot obj 'a)))
```

do not write codes in get-function which makes a recursion!
Be carefull! the argument of the lambda is the instance!
using get-function/put-function, annotated value can be easily coded.

2. defmeth

```
(defmeth name ( (arg1 class1) ...)
       function-body)
```

3. making an instance of a class

```
(make 'class) or (make-class)
        or (make 'class :slot the-value ...)
```

4. direct accessing of a slot

```
  (get-slot instance 'slot-name)
  (put-slot instance 'slot-name the-value)
```

5. Undeclared-slots

```
  (get-slot-always instance 'slot-name)
  (put-slot-always instance 'slot the-value)
  (remove-dynamic-slot instance 'slot-name)
```

slots which are defined with :allocation dynamic are FIRST referred by get-slot-always,put-slot-always.
True undeclared-slots which are not declared in a ndefstruct may be reffered by get-slot-always/put-slot-always also.
The difference between :allocationdynamic and non-:allocation dynamic slot is whether it is included in class or not.
(the later is only added to the instance)

6. changing class

```
  ( change-class instance (class-named 'new-class) )
```

7. run-super
the same definition as the paper at IJCAI.

8. with with*
Described in manual.tx

9. lambda-list keywords: &optional,&rest, and &key
They do not play a role in discrimination,
but can be used in the method body. as described in Ida85
(An Interpretation of the Common Loops specification, WGSYM 35-3 IPSJ)

```
(defmeth foo ((x class) &optional (y 1)) ...)
(foo (make 'foo)) and (foo (make 'foo) 100) will fire the same method with different arguments
```

図2 Common Loops チェック リスト 'Feb 末日 '86.

b)with構文

Flavors のdefmethod とは異なり、defun-likeであるので、スロットアクセスの記述へのエイドが必要になる。

例　(defmethod move ((b block) new-x new-y)
　　(with (b)
　　(setq x new-x y new-x)))

xとyはblock のスロットである。

==> yes.実装済み.

c)スロットのアロケーション

:allocation {class,dynamic,instance,none}

==> yes.実装済み。ただし、noneを除く。

classs指定はグローバル、dynamic指定は最初の出現で生成、instanceはデフォルト、noneは実際の割付けをしない。

d)get-dynamic-slot,remove-dynamic-slotを用意する。

==> yes.実装済み。しかし、関数名は、get-slot-always,put-slot-always,remove-dynamic-slot となっている。

e)annotated-value

:get-function,:put-function スロットオプションを用意する。

==>yes.実装済み。

f)change-class

既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。

==> yes.実装済み

g)run-super 時の引数の変更

==>no.

h)Flavorsスタイルのメッソド結合(:before,:after)

==> big project ....newFlavors対応の処理へ

i)ref

j)mlet

==>未解決。おそらく無くなるらしい。

k)complex specialization

(or ...)あるいは範囲指定など

==> GregorはNo.

l)lambda-list keyword problem

&optional,&rest,&keyなどを許すか?

==> 許す。しかし、discriminationには参加させない。この点は、bboardでも議論のあった所である。また、[10]が役立った所でもあった。

m)method-slot class problem

==> 未解決。Gregorは疑問視

4。ＰＣＬでの約束の変更点

defmethod では無く、defmeth という名前になった。

defstruct は当分の間の区別のために、ndefstructという名が用いられる。

(:class class)はLOOPS 流と[2] ではされていたが、実際にはCommon Loops流である。LOOPS 流にするには、(:class loops)とする。しかし、未実装である。:class flavor は、未実装というより、[4] の出現と、仕様の歩み寄りにより、不要になるかもしれない。

5。クラスオブジェクトの構造　（図3参照）

次のようなスロットを持つ。これは、ユーザがdefstruct したクラス、組込み型のクラス、メタクラスなど、すべてに共通する。

1)name... クラス名

2)class-precedence-list... たとえば、list->sequence->common->object->T という関係(object とはclass メタクラスにはいるものすべてのクラス、T はしんらばんしょうをあらわす)

3)local-supers... 直接の親のクラスが並ぶ

4)direct-subclasses...直接の子供のクラス

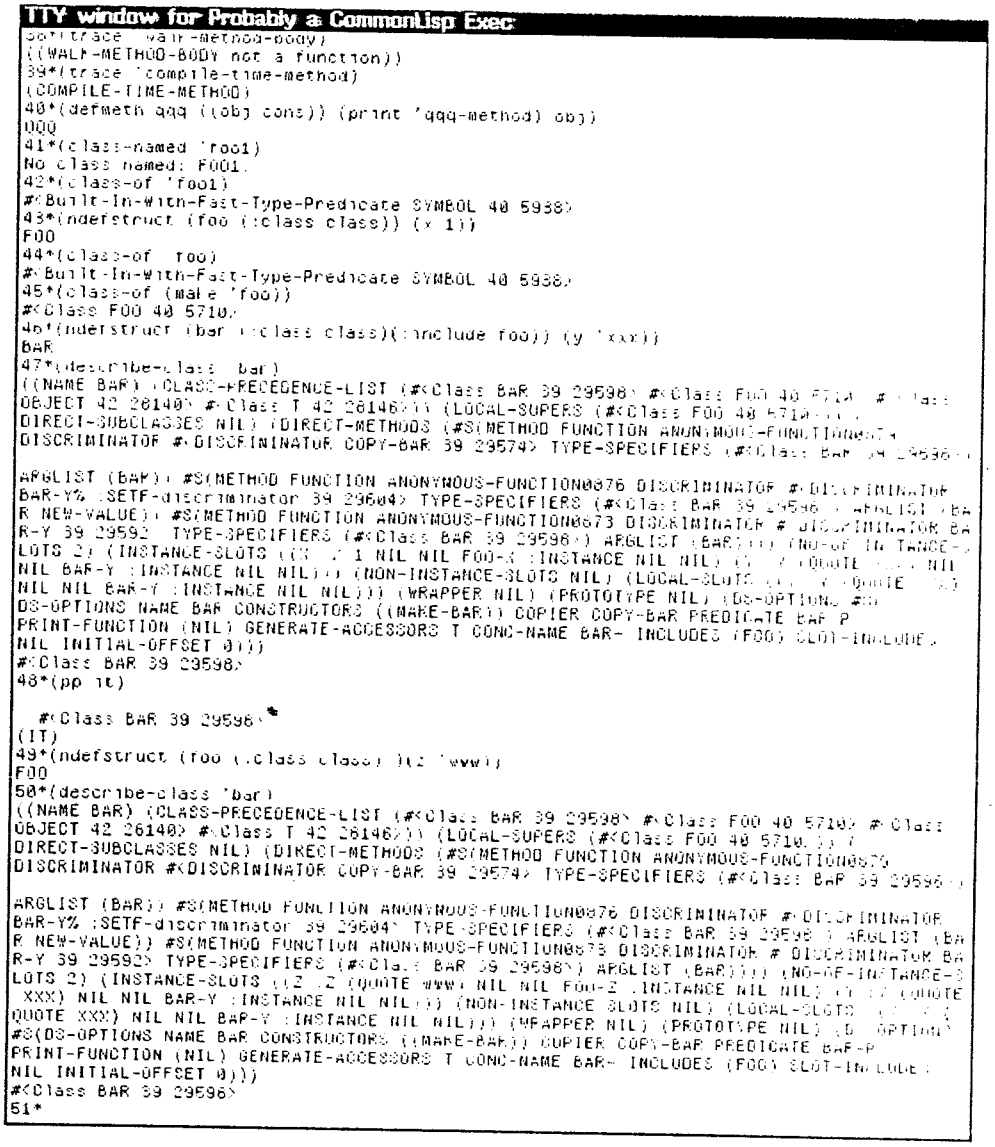
TTY window for Probably a CommonLisp Exec
38*(trace walk-method-body)
((WALK-METHOD-BODY not a function))
39*(trace 'compile-time-method)
(COMPILE-TIME-METHOD)
40*(defmeth qqq ((obj cons)) (print 'qqq-method) obj)
QQQ
41*(class-named 'foo1)
No class named: FOO1.
42*(class-of 'foo1)
#<Built-In-With-Fast-Type-Predicate SYMBOL 40 5938>
43*(ndefstruct (foo (:class class)) (x 1))
FOO
44*(class-of 'foo)
#<Built-In-With-Fast-Type-Predicate SYMBOL 40 5938>
45*(class-of (make 'foo))
#<Class FOO 40 5710>
46*(ndefstruct (bar (:class class)(:include foo)) (y 'xxx))
BAR
47*(describe-class 'bar)
((NAME BAR) (CLASS-PRECEDENCE-LIST (#<Class BAR 39 29598> #<Class FOO 40 5710> #<Class
OBJECT 42 26140> #<Class T 42 26146>)) (LOCAL-SUPERS (#<Class FOO 40 5710>)) (
DIRECT-SUBCLASSES NIL) (DIRECT-METHODS (#S(METHOD FUNCTION ANONYMOUS-FUNCTION0673
DISCRIMINATOR #<DISCRIMINATOR COPY-BAR 39 29574> TYPE-SPECIFIERS (#<Class BAR 39 29598>)
ARGLIST (BAR)) #S(METHOD FUNCTION ANONYMOUS-FUNCTION0676 DISCRIMINATOR #<DISCRIMINATOR
BAR-Y% :SETF-discriminator 39 29604> TYPE-SPECIFIERS (#<Class BAR 39 29598>) ARGLIST (BA
R NEW-VALUE)) #S(METHOD FUNCTION ANONYMOUS-FUNCTION0673 DISCRIMINATOR # DISCRIMINATOR BA
R-Y 39 29592> TYPE-SPECIFIERS (#<Class BAR 39 29598>) ARGLIST (BAR))) (NO-OF-INSTANCE-S
LOTS 2) (INSTANCE-SLOTS ((X . 1 NIL NIL FOO-X :INSTANCE NIL NIL) (Y . (QUOTE XXX) NIL
NIL BAR-Y :INSTANCE NIL NIL))) (NON-INSTANCE-SLOTS NIL) (LOCAL-SLOTS ((Y . (QUOTE XXX)
NIL NIL BAR-Y :INSTANCE NIL NIL))) (WRAPPER NIL) (PROTOTYPE NIL) (DS-OPTIONS #S(
DS-OPTIONS NAME BAR CONSTRUCTORS ((MAKE-BAR)) COPIER COPY-BAR PREDICATE BAR-P
PRINT-FUNCTION (NIL) GENERATE-ACCESSORS T CONC-NAME BAR- INCLUDES (FOO) SLOT-INCLUDE
NIL INITIAL-OFFSET 0)))
#<Class BAR 39 29598>
48*(pp it)
#<Class BAR 39 29598>
(IT)
49*(ndefstruct (foo (:class class))(z 'www))
FOO
50*(describe-class 'bar)
((NAME BAR) (CLASS-PRECEDENCE-LIST (#<Class BAR 39 29598> #<Class FOO 40 5710> #<Class
OBJECT 42 26140> #<Class T 42 26146>)) (LOCAL-SUPERS (#<Class FOO 40 5710>)) (
DIRECT-SUBCLASSES NIL) (DIRECT-METHODS (#S(METHOD FUNCTION ANONYMOUS-FUNCTION0673
DISCRIMINATOR #<DISCRIMINATOR COPY-BAR 39 29574> TYPE-SPECIFIERS (#<Class BAR 39 29598>)
ARGLIST (BAR)) #S(METHOD FUNCTION ANONYMOUS-FUNCTION0676 DISCRIMINATOR #<DISCRIMINATOR
BAR-Y% :SETF-discriminator 39 29604> TYPE-SPECIFIERS (#<Class BAR 39 29598>) ARGLIST (BA
R NEW-VALUE)) #S(METHOD FUNCTION ANONYMOUS-FUNCTION0673 DISCRIMINATOR # DISCRIMINATOR BA
R-Y 39 29592> TYPE-SPECIFIERS (#<Class BAR 39 29598>) ARGLIST (BAR))) (NO-OF-INSTANCE-S
LOTS 2) (INSTANCE-SLOTS ((Z .Z (QUOTE www) NIL NIL FOO-Z :INSTANCE NIL NIL) (Y . (QUOTE
XXX) NIL NIL BAR-Y :INSTANCE NIL NIL))) (NON-INSTANCE-SLOTS NIL) (LOCAL-SLOTS ((Y . (
QUOTE XXX) NIL NIL BAR-Y :INSTANCE NIL NIL))) (WRAPPER NIL) (PROTOTYPE NIL) (DS-OPTIONS
#S(DS-OPTIONS NAME BAR CONSTRUCTORS ((MAKE-BAR)) COPIER COPY-BAR PREDICATE BAR-P
PRINT-FUNCTION (NIL) GENERATE-ACCESSORS T CONC-NAME BAR- INCLUDES (FOO) SLOT-INCLUDE
NIL INITIAL-OFFSET 0)))
#<Class BAR 39 29598>
51*

図3 クラスの作り

5)direct-methods... そのクラスをclass-specifiers中に持つdefmeth されたメソッド及び、それに関連したsetf-discriminator，あるいはds-optionによるメソッドの並び
各メソッドのstructure は、
　関数名、discriminator のアドレス、type-specifiers,引数リスト
が並ぶ
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
9)local-slots ... defstruct されたスロットのリスト
10) wrapper
11) prototype
12) ds-options

6．discriminator オブジェクトの構造

discriminator には4 つのスロットがある。
　セレクタ名、同一セレクタのメソッドのリスト、discriminator のラムダ式、
　キャッシュ表
である。

7。Common Loopsの位置付け、将来、まとめ

Common Loopsは冒頭で述べた 3つの中でもっとも大きな位置を占めている。基本的には、Common Loopsがベースになると判断している。この裏付けの状況として、次のような事がある。
i)オブジェクト指向専門委員会の委員長はxerox parcのKen. Kahn であること、
ii) Common Loops，正確に言えば、Portable Common Loops (PCL) 及びその開発過程はARPAnet 上で公開されている。1100SIP,S3600,Lucid,TI-Explorer で動くlow-level hookが「xerox parcから」公開されている。また、筆者がparcに一週間滞在した際の約束として、日本からのものとして、正式にKCL を認知したかたちがとられるように、KCL インタフェースが含められるようになった。ただし、現在の時点では、ま

だ完全にKCL上では動かない。また、VAX Lispの上への移植を進めているグループが認知されてはいた。

iii) new Flavors の概念は従来のFlavors からの展開を意識してはいるが、直接の互換性は無く、CommonLoops とFlavors(従来) との橋渡しとしての概念で設計されている。

iv) 「核」としての役割を果すために、ii) に述べた普及の努力だけでなく、当初の夢にむけて、prolog的な論理型の機構のmerge への模索も既に始まっている(CommonLog[7][8]). また、他の提案との交流として、例えば、Common Loops上でのObject Lisp の実現方法などの議論も行なわれている。[9]

Common Loopsの処理系は、まだ完全に完成しているわけではない。言語仕様としても、細部に「つめ」が残されている。また、それぞれの処理系、機械の上での適合設計という問題の前に処理アルゴリズムの改良といったテ～マがある。こうした点への貢献を求めてARPAnet 上での公開を行なっていると解釈すべきであろう。

なお、PCL のソースはUS export control lawsの対象となっていることが明記されているので注意されたい。xerox parcの正式な許可の無いかたちでのソースの米国外への持出しは、たとえ善意であっても、問題となりうる。とくに、技術的な点だけに絞ってみても、現在のPCL は製作の真最中，ということがあり、今の時点のPCL で誤解されたくないというparc内の製作者たちの意向がある。更に、ややっこしいのは、同時に、これはむしろ技術者として当然な態度ではあるが、parcの技術者は非常にオープンであり、技術的な交流を歓迎していることがある。12月上旬に送ったアイデア[10]の一部（メソッドの内部形式をlambda式にならわせ、かつdiscriminationのルールを変更する）、が採用されていて嬉しくなったが、今でも、これこれの機械に移植した、というだけでは無く、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。これに関連して、少なくとも現在の時点では、筆者が直接に電子的な手段でupdateできるところへはロードして良いという許可をもらっている。青学大のmicroVAXII，東大の8600の上に当面、のせる予定である。

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、1985)

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

[3] G.L.Drescher, ObjectLISP for experienced LISP programmers, LMI aug.1985

[4] S.E.Keene and D.A.Moon, Flavors: Object-oriented Programming on Symbolics Computers, Symbolics, dec. 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] K.Kahn, Ideas about CommonLoops& Logic programming, draft, Nov.27.1985

[8] K.Kahn, Notes from discussion about "Common Log", draft, dec.3. 1985

[9] G.Kiszales, CommonLoops meets Object Lisp, CL-Object-Oriented-Programming@SU-AI.ARPA, jan.3, 1986

[10] M.Ida, An Interpretation of the Common Loops specification , WGSYM35-3, IPSJ, dec.1985

[11] 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