記 号 処 理 16－3
（1981. 3. 27）

Hyperlisp とその implementation

萩谷昌己・佐藤雅彦
（東大理・情報科学）

1. prolog(ue)

Prolog の話から始めよう．Prolog で，次のような predicate eq を考える．

```
+eq(X, X, true).
+eq(X, Y, false).
```

これで，2つの term の equality を判断しようというわけだ．さて，次のような関数定義があるとする．

$$f(X,Y) = \underline{\text{if}}\ X \neq Y\ \underline{\text{then}}\ g(X,Y)\ \underline{\text{else}}\ h(X,Y)\ \underline{\text{fi}};$$

今，gは，g(X,Y,Z)，hは，h(X,Y,Z)という predicate で表現されているとしよう．Zは，いわゆる output variable だ．g, h から，f(X,Y,Z) を定義したい．すぐに考えつくのは，次の定義だろう．

```
+f(X, Y, Z) - eq(X, Y, false) - g(X, Y, Z).
+f(X, Y, Z) - eq(X, Y, true) - h(X, Y, Z).
```

これでよいか．明らかにまちがっている．原因は，eq の定義にある．eq の定義は，programming language としての Prolog に特有の，'評価の順番'というものを利用している．では，こうしないと eq は定義できないのだろうか．Prolog の data structure は，簡単にいえば，closed term の全体で，それは，function symbol を定めることによって決まる．ところが，Prolog は，function symbol の全体をきちんと定めてはいない．

今，仮に function symbol を，0(nullary)と s(unary)に限ってみる．すると，eq は次のように定義できる．

```
+eq(0, 0, s(0)).
+eq(0, s(X), 0).
+eq(s(X), 0, 0).
+eq(s(X), s(Y), Z) - eq(X, Y, Z).
```

この定義は，'評価の順番'に全く頼っていない．（true を s(0), false を 0 で表現している．）しかし，実際の Prolog では，function symbol が不特定無限個あるため，このような定義は不可能である．

2. 

我々が認識できる object とは，どんなものだろう．ここでいう'我々'の中には，高速計数型電子計算機という subject も含まれることはいうまでもない．明らかに，我々は有限個の element からなる object は認識できる．では，無限個ならどうか．これが極めてむつかしい問題なことは確かだが，次のことは最小限いえる．"我々は，有限個からなる object の element に，有限個の operation を，有限回作用させてできる(ものからなる)object は認識できる．" この原理を正確にいえば，いわゆる

generalized inductive definitionという形になる．たとえば，自然数というobjectは，

i) 0は自然数．
ii) nが自然数ならば，n'は自然数．
iii) i), ii)からできたもののみが自然数(extremal clause)．

こうして，自然数は，無限個あるけれども，認識できるobjectなことがわかる．

3. lisp

Lispのdataは，Sexpressionと呼ばれる．Sexpressionは，次のように定義される．

i) atomはSexpression．
ii) α, βがSexpressionならば，(α.β)はSexpression．
iii) extremal clause．

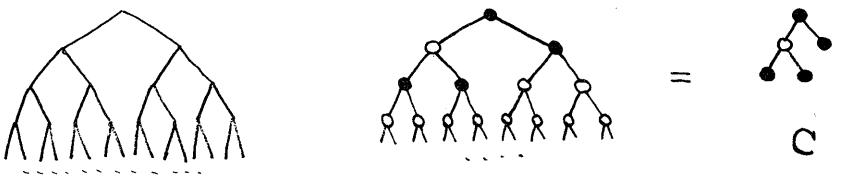
こうみると，Sexpressionは，えでいうobjectと認められそうだが，iiが問題だ．いったいatomとは何なのか．実際のLisp systemでは，atomは実に種々雑多である．integerであり，Stringであり，real numberであり…．もうめちゃめちゃだ．

話をPure Lispに限ろう．Pure Lispのatomは，a, aa, …, ab, …といった，literal atomである．atomが認識できるobjectである以上，それらは有限個のobjectから組み立てられていなくてはならない．事実，literal atomは，a, b, …, zという26文字からできている．ところが，Pure Lispのprimitiveは car, cdr, cons, atom, eqの5つで，これだけではatomの内部構造を知ることはできない．その代りに，eqという，atomのequalityを与えるpredicateをprimitiveにしているわけだ．この点がPure Lispの大きな問題点で，(Turing machineのような)計算のmodalとしてPure Lispを採用することをむつかしくしている．

4. sexp

そこで，Hyperlispが登場してくる．Hyperlispのdataはsexpという．Sexpressionの略である．まず，直観的な定義から始めよう．

下図Aのような，無限にのびる葉のないbinary treeを考える．



A　　B　　C

これを碁盤とみる．有限個の黒石を用意して，それらをこの碁盤上に適当におく．黒石がおかれなかった所には，白石をうめていく．できあがった図形をsexpと呼ぶ．(B)　実際の絵に書かれていない所には，すべて白石がうまっていると約束する．黒石の数は有限としたから，sexpは有限の図形として描ける．BとCは同じsexpである．sexpの全体をSで表わす．

黒石の1つもないsexpを0で表わす．

次に，S上のprimitiveを定義する．

atom : $S \to Bool$

atom(x) = if x の root が黒 then true else false fi

car, cdr : $S \to S$

$car\left(\overset{\circ}{x}\ \overset{\circ}{y}\right) = car\left(\overset{\bullet}{x}\ \overset{\circ}{y}\right) = x$

$cdr\left(\overset{\circ}{x}\ \overset{\circ}{y}\right) = cdr\left(\overset{\bullet}{x}\ \overset{\bullet}{y}\right) = y$

cons, snoc : $S \times S \to S$

$cons(x, y) = \overset{\circ}{x}\ \overset{\circ}{y}$

$snoc(x, y) = \overset{\bullet}{x}\ \overset{\bullet}{y}$

atom(x) = true となる x を atom と呼ぶ．この名前はよくない．なぜなら，atom の car はとれるのだから．

次の等式に注意して欲しい．

$car(0) = cdr(0) = 0$

$cons(0, 0) = 0$

5. notation

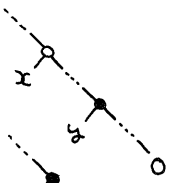
sexp に対する notation を導入する．

dot notation :

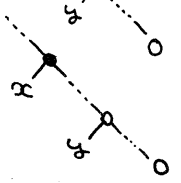
$(x . y) = \overset{\circ}{x}\ \overset{\circ}{y}$　　$[x . y] = \overset{\bullet}{x}\ \overset{\bullet}{y}$

list notation :

$(\cdots, x, \cdots, 'y, \cdots) =$



$[\cdots, x, \cdots, 'y, \cdots] =$



たとえば，$(x, 'y, z) = (x . [y . (z . 0)])$．, は ; でもよい．

6. inductive definition.

ここで，sexp を，(こでいう) inductive に，定義しておこう．

i) 0 は sexp.

ii) x, y が sexp ならば，$(x . y)$, $[x . y]$ は sexp.

iii) extremal clause.

iv) $(0 . 0) = 0$.

7. literal

Pure Lisp の literal atom に相当するものを S 上に実現する。a, b, …, z 上の空でない string を literal と呼ぶ。literal を S の中へうめこむ。つまり，sexp で表現する。そのうめこみ方は任意だが，現在の Hyperlisp では，次のようになっている。

abc = [[1,1,0,0,0,0,1], [1,1,0,0,0,1,0], [1,1,0,0,0,1,1]]

a の ascii code は，8進で141，2進で1100001なことに注意。

8. natural number

自然数も S へうめこんでしまう。次のようにする。

0 = 0　（自然数0は，sexp 0で表現する。）
n' = [n . n]

たとえば，1 = [0.0]，2 = [1.1]，3 = [2.2]，…。

9. eval

Hyperlisp の evaluator (eval) を定義しよう。eval は，S 上の partial recursive function である。

以下，Algolic に eval を定義する。**eq, cond, lambda, label** は，7の literal である。env という global variable に，関数定義が alist として入っているとする。

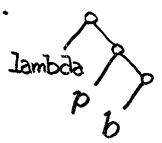
```
eval(x)
= if atom(x) then apply(car(x), cdr(x))
               else apply(car(x), evlis(cdr(x))) fi

evlis(x)
= if x = 0 then 0
  elif atom(x) then cons(car(x), evlis(cdr(x)))
               else cons(eval(car(x)), evlis(cdr(x))) fi

apply(f, x)
= if f = 0 then 0
  elif atom(f) then
       if f = 1 then car(x)
       elif f = eq then
            if car(x) = car(cdr(x)) then 1 else 0 fi
       elif f = cond then evcon(x)
       elif assoc(f, env) ≠ 0 then apply(cdr(assoc(f, env)), x)
       else apply(eval(f), x) fi
  else
       if car(f) = lambda then
            eval(subst(x, car(cdr(f)), car(cdr(cdr(f)))))
       elif car(f) = label then
```

```
                    apply(subst(f, car(cdr(f)), car(cdr(cdr(f)))), x)
          else      apply(eval(f), x)   fi fi

evcon(x)
= if x=0 then 0
  elif atom(eval(car(car(x)))) then eval(car(cdr(car(x))))
  else  evcon(cdr(x)) fi

subst(x, p, b)
= if p=0 then b
  elif atom(p) then point(x, car(p))
  elif atom(b) then
        snoc(subst(x, car(p), car(b)), subst(x, cdr(p), cdr(b)))
  else  cons(subst(x, car(p), car(b)), subst(x, cdr(p), cdr(b))) fi


point(x, q)
= if q=0 then 0
  elif atom(q) then x
  elif cdr(q)=0 then point(car(x), car(q))
                else  point(cdr(x), cdr(q)) fi

assoc(x, a)
= if a=0 then 0
  elif x=car(car(a)) then  car(a)
  else  assoc(x, cdr(a)) fi
```

まず，evalとevlisをみれば，実引数の評価のされ方がわかると思う．つまり，評価されるsexpがatomのときは，引数は全く評価されず，そうでないときは，evlisへ渡される．引数listが，(…, x, …, 'y, …)のとき，evlisは，xは評価し，yはquoteするという動作をする．以上によって，Hyperlispでは，FEXPRとEXPRを分ける必要がなく，またQUOTEもいらない．

applyでは，Lispと同様に，primitiveやlambda expressionを認識して，applicationの処理をする．primitiveは，0，1，eq，condの4つである．その意味は，applyとevconから明らかだろう．carやconsに相当するprimitiveがないのは，それらがlambdaで定義可能だからだ．

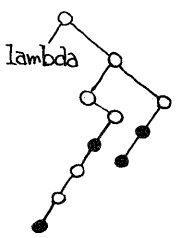
Hyperlispのlambda expressionとparameter bindingは，Lispと大きく異っている．強いていえば，Lispのmacroに近い．lambda expressionは，左のような形をしている．



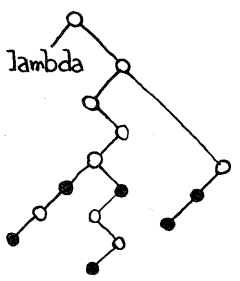
lambdaは，literalで，全体がlambda expressionであることを示す．bが関数bodyで，pは，実引数がbodyのどこへ，代入されるかを表すsexpである．詳しくは，substとpointの定義をみて欲しい．applyは，関数がlambda expressionであることを知ると，実引数をbodyへ代入する．そして，そ

の結果を，もう一度evaluateする．この点が，Lispのmacroと似ているところである．lambda expressionの例として，carとconsを表してみよう．

car：



cons：



両者とも，1というprimitiveが本質的である．

10. reference language

Hyperlispのnotationのsystemは，5で導入した簡単なnotationよりも凄まじく拡張されている．(これは，evaluator(特にlambda)を考えれば明らかだ．)これを，Hyperlispのreference languageと呼んでいる．以下，そのsyntaxをBNFで定義する．

Hypernules：

<...-seq> ::= <empty> | <...-seq'>
<...-seq'> ::= <...> | <...>, <...-seq'>

Rules：

<form> ::= <term> | <term> : <term>
<term> ::= <unit> | <term><list>
<unit> ::= <literal> | <metaliteral> | <natural number> | <list>
"<unit> | <lambdaexp> | <labelexp>
<list> ::= <conslist> | <snoclist>
<conslist> ::= (<listelem-seq>) | (<listelem-seq> . <form>)
<snoclist> ::= [<listelem-seq>] | [<listelem-seq> . <form>]
<listelem> ::= <form> | '<form>
<lambdaexp> ::= Lambda(<metaterm> , <form>)
<labelexp> ::= Label(<metaterm> , <form>)
<metaterm> ::= <metaunit> | <metaunit> = <metaterm>
<metaunit> ::= <metaliteral> | <metalist> | 0
<metalist> ::= [<metaterm-seq>] | [<metaterm-seq> . <metaterm>]
<metaliteral> ::= 大文字で始まる英数字列 e.g. X, X1, Y, ...
<definition> ::= # <unit><metalist> = <form>

<top-level> ::= <form> | <definition>

，は；でもよい．

formがsexpをdenoteするわけで，このdenotationのruleを定めるということが，reference languageのsemanticsを与えるということになる．ここでは詳しく述べられないが若干のhintをあげておく．

```
"x = [1, x]
f[...] = [f, ...]
f(...) = (f, ...)
x:y = (x, y)
(x, y . z) = (x.(y.z))     (.x) = x

Lambda([X,Y]; "(X.Y)) = {9のconsを表わすsexp}
Lambda([[X]]; "X) = {carを表わすsexp}
```

11. function definition

たとえば,

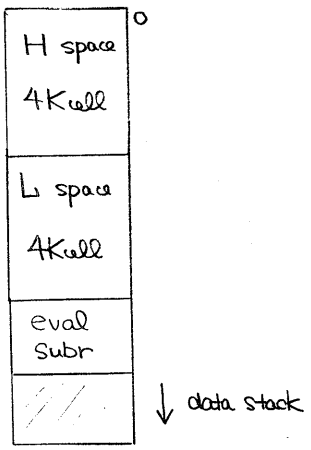
```
#foo[X] = fee[X, X]
```

という関数定義は, literal fooに, Lambda([X]; fee[X, X]) という定義を与えることを意味する. 任意のatomに, 関数定義を与えることができる.

関数定義をするごとに, 9のenvがupdateされると考えればよい.

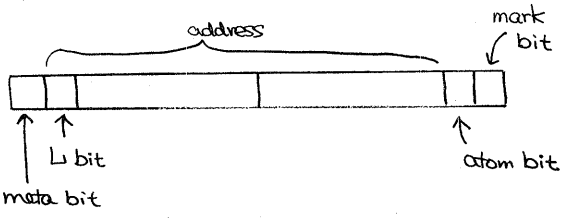
12 implementation

最初のinterpreterは, UNIX下のPDP11上に, Cとasでimplementした. pointerは16bitで, 1cellは4byteである. Sはmonocopyで実現した.(関数定義とliteralの出力が主な理由.) core mapを左下に示す. sexpはHspace上に作られる.



関数定義は, semi-compileされる. つまり, 11のspecificationをそのままimplementしているわけではない. このときに, L space (monocopyでない) が使われる. なるべく実引数の関数bodyへの代入を行わないような工夫をしている.

pointerの構造を下に示す.



pointerの中にatom bitを入れた.

sexp 0 は pointer 0 で表現される.

関数とその定義はhash tableで結びつけた.

8 QUEENが PDP11/55で 32秒で解けた.

13. examples of Hyperlisp programs

```
#cons[X,Y] = "(X.Y);
#null[X] = eq[X]

#append[X = [X1.X2], Y]
= cond[ null[X] : "Y;
        "1 : cons('X1, append[X2, Y]) ];

#reverse[X = [X1.X2]]
= cond[ null[X] : 0;
        "1 : append(reverse[X2], '(X1)) ];

#s[X] = "[X.X];

#add[X = [X1], Y]
= cond[ null[X] : "Y;
        "1 : s(add[X1, Y]) ];

#fib[X = [X1 = [X2]]]
= cond[ null[X] : 0;
        eq[X, 1] : "1;
        "1 : add(fib[X1], fib[X2]) ];
```

14. reference

[1] E. Goto : Monocopy and Associative Algorithms in an Extended Lisp, TR74-03 Information Science Laboratories, Faculty of Scienc, University of Tokyo

[2] M. Sato : Theory of Symbolic Expressions, TR80-16, Department of Information Science, Faculty of Science, University of Tokyo

[3] 萩谷昌己 : よいうのHyperlisp (not published)

[1] は monocopy について.
[2] に, Hyperlispの厳密な定義がある.
[3] は, Hyperlispのtutorial.

[4] M. Hagiya : Hyperlisp2.1 Manual

[4] はUnixのpackに入っています.