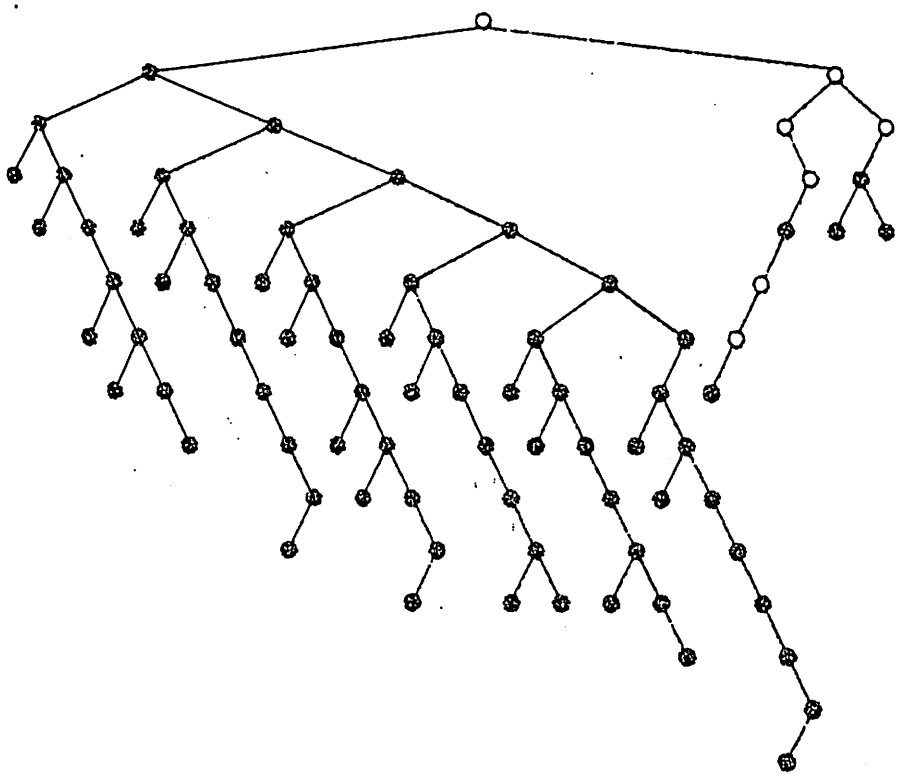
WARP

よい子の

はいぱー　りすぷ

Hyperlisp

はぎや　まさみ　著(ちょ)

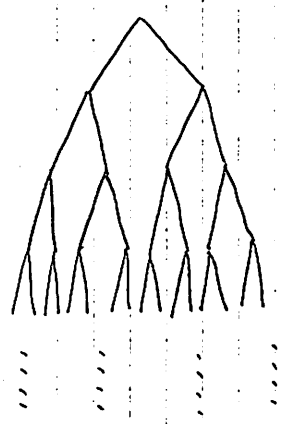


第1回

登場人物　先生：抽象的存在
A：ひねくれた人
K：何もわかっていない人
H：すぐなっとくする人

先生：今日は，少し毛色の変ったLispの話をしましょう。
H：いよいよHLISPの話ですね。
先生：いや～まいった。HLISPもなかなか興味深いけど，それはまたいつかということにしよう。
H：すると，LIPQのことかな。
先生：そういう変態的LISPは，ぼくは大好きだけれど，LIPQについては，残念ながら，あまり知らないのだよ。直接に，設計者にでも聞いてくれたまえ。今日は，何をかくそう，アトムのないLispの話なのだよ。
A：やれやれ。
K：？？？？。
H：なるほど。でも，アトムがなくて，どうしてLispができるんだろう。
先生：正確にいうと，Lispのアトムに対応するものはあるのだけど，それは，ちゃんと構造をもっていて，carもcdrもとれるのだよ。
A：そういうものは，アトムというべきでない。
先生：うん，そのとうりで，アトムと呼ぶべきでない。だから，前にアトムのないLispといったんだよ。でも，これからは，今いったアトムでないようなアトムのことを，アトムと呼ぶことにするよ。
K：アトムがあるといったり，ないといったり，それは矛盾しているのでは。
先生：まあ，これから説明するうちにわかってもらえるだろう。
A：（わかるもんか。）
先生：まず，次のような，無限にのびたバイナリー・トリーを考え

る。



H：理論ぽくなってきましたね。

A：コンピューターサイエンスで，無限などということを使うのはよくない。

先生：まあ，理論上のことだから，少しがまんして聞いて下さい。この無限木には，当然ながらnodeも無限にある。そのnodeたちのうちから，有限個選んで，その上に黒い石をおくとしよう。そうしてできた図形をS式と呼ぶ。

H：黒い石は，1つもおかなくてよいのですか。

先生：そうだ。石を1つもおかないS式を，0と呼ぶことにしよう。

K：0というのは，可換群における単位元を指しているのでは。

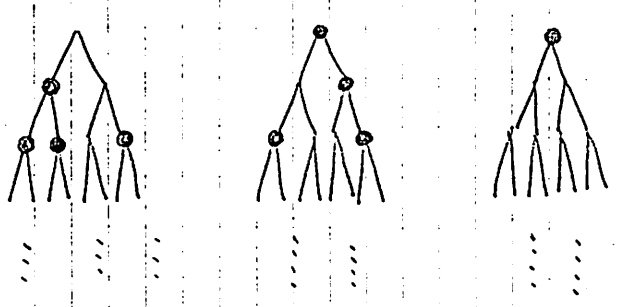
先生：むむっ，君は代数に強いらしいね。実は，この0は，非可換環における零元をさしているのだよ。

A：代数はちんぷんかんぷんだ。

先生：ごめん，ごめん。話をもとにもどそう。今，0を定義したね。つまり，

$= 0$

一般的なS式は，次のようになる。



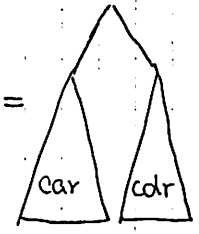
などなど。●が黒い石だ。⋮のところは，もう石がないとする。以下同じだ。さて，tree（木）のroot（根）というのは知っているかい。

A：上の絵では，一番上のnodeのことだろう。

先生：そのとうり。rootに石がおいてあるS式をatom（アトム）と呼ぶ。rootに石のないS式をmoleculeと呼ぶ。

H：なるほど。すると，atomはmoleculeと同じような構造をもつのですね。

先生：そうだ。では，carとcdrを定義しよう。xをS式としたとき，car[x]とは，xのrootの左の子をrootとする部分木とする。



その部分木には，黒石が有限個のっているわけだから，S式と考えられる。cdr[x]は，右の子から始まる部分木と定める。こう定義すれば，atomもmoleculeも，同じように，carやcdrがとれることがわかる。

A：でもやっぱり，atomという名はおかしい。

先生：まあ，がまんしてくれよ。さて，次の式はわかるかい。

car[0] = 0

cdr[0] = 0

H：0のrootの左の子をrootとある木には，右は1つものっていないから，結局0と同じで，だから，car[0]＝0となるのですね。

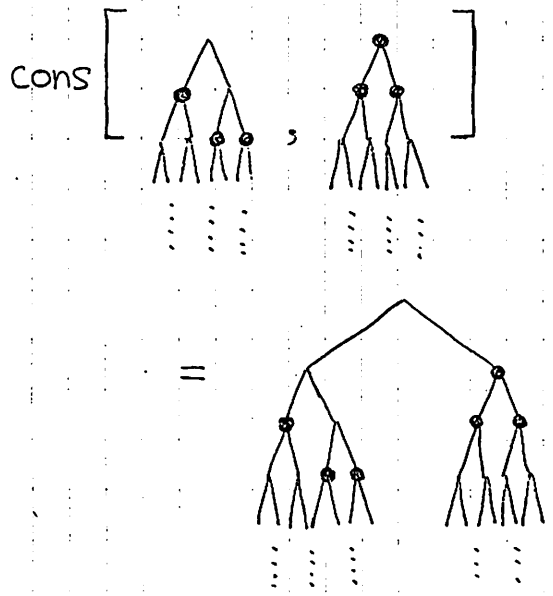
先生：そうだ。次に，Constructorを定義しよう。

K：constructorとは何だろう。

先生：LISPのconsのように，あるdata構造を作り出すprimitive（基本演算）のことをConstructorという。我々のLispには，2つのconstructorがある。これをconsとsnocという。

A：おいおい，snocというのは，consを逆にしたものかいな。

先生：そういうことです。悪趣味と思わないでね。さて，consを定義しよう。例からかくと，



つまり，consは，第1operand（引数）をrootの左の部分木，第2operandを右の部分木とする木を作って，rootには石をおかない。もう1つ例をあげれば，

cons[0, 0] = 0

これに対して，snoc では，2つS式から作った木のrootに，黒石をおいてできるS式を作り出す。たとえば，

Snoc[O, O] = 

つまり，cons は molecule を作り，snoc は，atom を作るわけだ。

H：0から始めて，cons と snoc を適当にくり返せば，任意のS式が表わせるのでは。

先生：よくわかるね。そのとうりです。まるで台本があるみたいだ。

A：やれやれ。

先生：さて，すべてのS式が，0とconsとsnocで組み立てられることがわかったから，一般のLispと同じようなdot notation や list notationを使うことを考えよう。まず，dot notation から，

「(x.y) は cons[x,y] を表す」

これは，普通のLISP と同じだ。もう1つ，snoc があるから，

「[x.y] は snoc[x,y] を表す」

と定めよう。これで，原理的には，すべてのS式が，0と2つのdot notationで表わせる。

次に，list notation を定めよう。

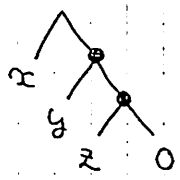
(x.(y.(w.0))) → (x, y, z)

[x.[y.[w.0]]] → [x, y, z]

となる。大体，普通のLISPと同じだ。ただ，NILの代わりに，0を使う。また，listのelementは，必ず，，で区切るとする。もちろん

$$[\,] = () = 0$$

ところが，一般のS式は必ず上のようになるとは限らない。たとえば，



つまり，

$$(x.[y.[z.0]])$$

このときは，次のようにする。

$$(x, 'y, 'z)$$

つまり，(…)は，consでつながったlistを意味するが，その中で，Snocでつながっている部分は，'をつけて示す。同様に，

$$[x.(y.0)] \rightarrow [x, 'y]$$
$$\text{or}$$
$$\rightarrow ('x, y)$$

K：さっぱりわからん。
先生：ゆっくり考えて下さい。
H：このLispでは，S式のcdrをどんどんとってゆけば，必ず0に至るから，list notationだけで，dot notationを使わずに，すべてのS式が表わせるわけですね。

先生：そうだ。さて，以下では，

$$1 \underset{def}{=} \mathrm{snoc}[0,0]$$

とおく。

K：この1は，群の単位元か，monoidの単位元を指すのだろうか。

先生：前にも同様のことをいったけれど，この1は，非可換の単位元です。

A：また始った!?

先生：いや〜，ごめん。次に，literal というものを定めよう。普通のLispには，literal atomというものがあるけれども，我々のLispにも，これを擬似的にS式として実現しようというわけだ。それがliteralだ。

A：苦しまぎれに，何か変なことを始めるぞ。

先生：（無視して）aはascii 141（8進）だね。bは142。そこで，abというliteralは，

$$[[1,1,0,0,0,0,1],[1,1,0,0,0,1,0]]$$

とencodeする．つまり，以下で，abというのをS式として扱った場合は，上のS式を表わしていると思うわけだ。encodeのし方は，上から類推してくれ。

H：つまり，ascii codeを0，1からなるlistとして表して，それからさらに，listを作るわけですね。できたものは，ちょうど我々のatomになっていますね。snocでlistを作っているから。

先生：そうだ。これからは，上のようなliteralを自由にlistやdot notationの中に使うことにするよ。

さて，いよいよ，これから，evaluationについての話に入ろう。

A：やっと中味のあることが始ったか。

先生：普通のLispでは，伝統的には，まずm式が評価の対象として存在し，これをm-S変換という形で，S式上にencodeして，それを，interpreterが評価して価を出すようになっている。

これに対して，我々のLispでは，m式というものはなくて，S式を直接に評価の対象とする。

H：なるほど。

先生：次のような略記法を導入しよう。

$$[f, x, y, \cdots, z] \rightarrow f[x, y, \cdots, z]$$

$$(f, x, y, \cdots, z) \rightarrow f(x, y, \cdots, z)$$

H：何か，関数部を外に出すという感じですね。

先生：うむ。（一瞬の沈黙）　まず，例から始めよう。

cons[a,b]　　（[cons, a, b] の略
cons, a, b は literal ）

を evaluate すると，

(a.b)

が値として返る。

car[(a.b)]

はaとなる。

car(cdr[(a, b)])

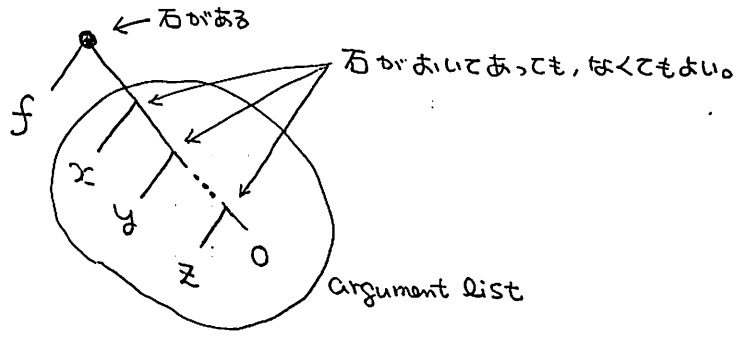
はbとなる。

K：だんだんわからなくなってきた。（実は最初からわかっていない。）

先生：call by name と call by value ということは知っているね。

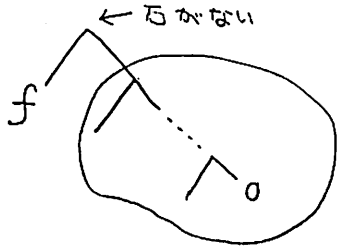
H：call by name は，引数を評価せずに，関数にそのままわたし，call by value は，引数を評価してからわたすのでしょう。

先生：我々のLispでは，2つのcallingを，呼ぶ方で制御できるのだよ。つまり，



となっていれば，argument list は，評価されずに，そのまま，関数（この場合は f ）にわたされる。

しかし，上の car(cdr[(a, b)]) の例のように，root に石がないと，つまり，

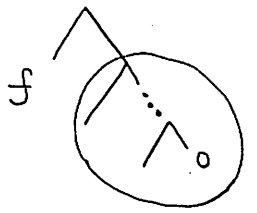


となっていると，引数がそれぞれ評価されて，結果として，argument list が作られ，関数にわたされる。上の例だと，まず，

cdr[(a,b)]

が評価されて，(b) となり，これから，((b)) という argument list が作られ，これが car へわたされる。そして，car は，((b)) という argument list から，第１argument である (b) をとり出し，この car をとって，値 b を返すわけだ。これに対し，その前の，car[(a.b)] だと，[(a.b)] そのものが，argument list として，car へわたされる。

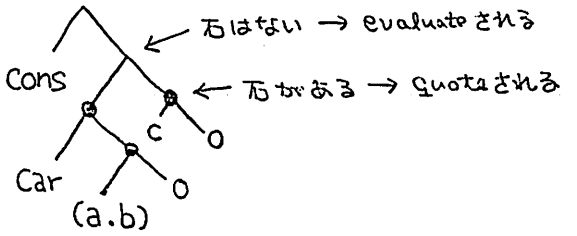
実は，



となっている場合は，もっと複雑なことがおこる。例として，

cons(car[(a.b)], 'c)

を考える。（（cons, [car, (a.b)], 'c）の略。）図で書くと，



A：ややこしい。もっとわかりやすく。

先生：この場合，car[(a.b)] は評価されるが，c は評価されない。つまり，'がLISPのQUOTEのように働くわけだ。答えは，

(a.c)

各引数ごとに，評価するかしないかを決めることができる。

A：わからん。

K：？？？？。

先生：この辺は，我々のLispのuniversal function，特に，evalとevlisをみないとわからないだろう。Algolicにかくと，次のようになる。

```
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
```

A：わかった。
H：なるほど。
K：わからない。
先生：この辺で，今日は終わることにしよう。演習として，次の式を評価してみて下さい。

```
car(cons[a,b])
cdr[0]
cdr[1]
snoc[0,0]
cons[car[c], cdr[b]]
```

解答：a，0，0，1，([car,c].[cdr,b])
（第１回おわり）

第2回

登場人物　先生：Syntax Sugar
H：もうわかっている人
A：世の中を絶望している人
I：何でもおもしろがる人
K：まだ来ていない。

先生：前回の復習から始めよう。

cons[car[x], cdr[z]]

は，evaluate すると何になるか。

A：([car, x]. [cdr, z])

だ。

先生：えらい。

I：これは，なかなか，おもしろいLISPだな。

先生：今日は，cond や lambda の話をしよう。

A：あたりまえだ。それがなければLISPじゃない。

先生：（無視する。）普通のLispだと，condはfexprとかfsubrとかいって，exprやsubrとは，evaluateのし方が違うのだけれど，このLispでは，そういう区別はないんだ。

A：区別がなければいいというものでもない。

H：わかった。このLisp独特のcall by nameを使ってしまえば，別にfexprやfsubrといったものを作らなくてもいいわけだ。

先生：そのとうりです。condの例をあげよう。あっと，その前にいっておかなくちゃいけないことがあるな。LISP1.5なんかでは，falseはNILであらわされて，その他のS式がtrueをあらわす。このLispでは，atomがtrueをあらわし，moleculeがfalseをあらわすことになっている。

H：う～ん，みごとだ。

I：ううっ。

先生：それから，1は，恒等関数をあらわす。
A：なに？
先生：たとえば，

```
1[a]
```

は，aとevaluateされる。これはよく使うので，次のように略記する。

```
1[ab] → "ab
```

H："abは評価するとabになるから，ちょうどQUOTEと同じように働くのですね。
A：" と ' の違いがわからん。
先生：よく考えて下さい。
H：'は，list記法の一部で，"とは全然違うのだよ。
A：わからん。（ここで，A君だけ勉強する。）わかった。
先生：ではcondの例にもどろう。

```
cond[(eq[a,b], "1),
     ("1, 0)]
```

あまり意味のない式だけれど，これを評価すると何になるか。
H：0です。まず，eq[a,b]が評価される。… あれ，これは何になるんだろう。
先生：わかっていないのによく答が出たなあ。そうか，eqはまだ説明していなかった。eqは，第1引数と第2引数を比較し，等しければ1，そうでなければ0を返す。ここで，1はatomだから，true，0はfalseを代表している。eq[a,b]は，0が返る。
H：つまりfalseだから，次のbranch ("1, 0)がとられて，"1が評価される。これは1になる。1はtrueだから，このbranchのbodyである0が評価されて，全体の値となる。あれ，0は評価すると何になるんだろう。

先生：これもまだいってなかったなあ。0を評価すると，実は0なんだよ。そのようにおぼえておいてちょうだい。

次に，nullとatomという関数を定義する。nullは，第1引数が0なら1，そうでなければ0を返す。atomは，第1引数がatomならば1，そうでなければ0を返す。次の式を評価して下さい。

```
cond[ (null[1], cons[a, b]);
      ("1, "oui) ]
```

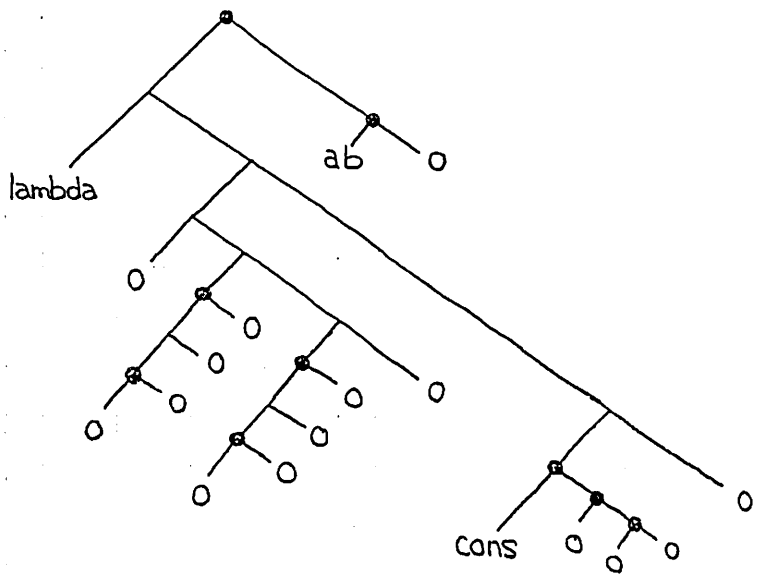
I：答えは，ouiでしょ。
A：あれ，；も使っていいのか。
先生：；と，は同じ働きです。見やすいように書きましょう。ここでさらに，次のような略記を導入する。

$$(x, y) \quad \rightarrow \quad x : y$$

I：ほほん。
A：それでは，ambiguityが生ずる気がする。
先生：まあ，その辺のごちゃごちゃした話は，今は無視しよう。次はどうなるか。

```
cond[ atom[atom] : "atom;
      "1 : "molecule ]
```

I：atom。
先生：さていよいよlambdaのはなしだ。まず例をあげよう。



これを評価すると

(ab.ab)

になる。

一同：ちんぷんかんぷん。

（ここで，とっぜんK氏登場。）

K：やあ，おくれてすいません。I/Oに行っていたので。今日，原稿料が出たんですよ。

先生：これはあとにして，さあlambdaの説明をしよう。

K：lambdaって何だろう。

先生：（無視しながら，）このLispには変数というものがないんだよ。

一同：ええー。

K：（遅れて）ええー。

先生：束縛変数というのは，場所だけを示すもので，べつに，たとえばxがyとかわっても変わりがないことはわかるね。つまり，LISPで，

(LAMBDA (X) (CONS X X))

と,

(LAMBDA (Y) (CONS Y Y))

とは同じものである。

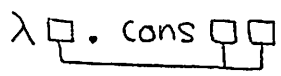
K：XとYが違うから違うのでは。

H：関数として比較しているんだよ。

A：理論的すぎてよくわからないな。

先生：つまり，上でXやYで場所だけを示すのだから，もっと，「場所」というものを直接に示すようにした方がよい。

H：たとえば，Bourbakiでは，

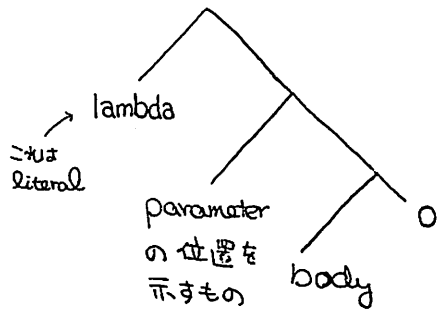


という風に書きますね。

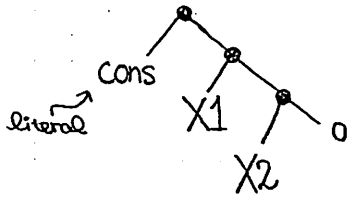
先生：Bourbakiをよく知っているね。

K：Bourbakiは，ぼくも愛読しています。とくに代数のところがいいなあ。

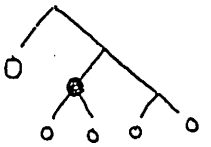
先生：（無視して）我々のLispでは，lambda式は，次のようになる。



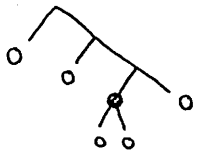
さて，どうやって，位置（つまり場所）をあらわすか。たとえば，



というbody を考えよう。X1 で示された場所は,



という S式で示される。つまり, ● が場所を示すんだ。X2 は,



で示される。

H：よくわかるなあ。

先生：さて, body の中の位置は, こうして示されることがわかった。

H：つまり, skeleton といった感じですね。

先生：そうだ。むつかしいことばを知っているね。

K：skeletonって何だろう。

A：骨組みというような意味だよ。

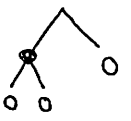
先生：次に, bodyの中の場所と同じように, argument list の中の場所を表すことを考える。

A：argument listの中の場所とは, どういうことさ。

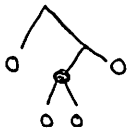
先生：つまり, apply には, 実引数の listがわたされるけれども, たとえば, 第1引数というものは, この list の car として identify される。

H：第2引数は cadr ですね。（cdr の car）

先生：だから，第１引数は，

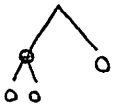


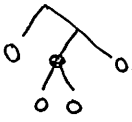
で示せる。第２引数は，



で示せる。

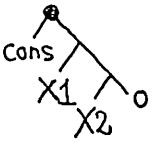
A：あれ，またわかんないぞ。

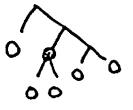
先生：  で，黒石があるのは，このcarだろう。だから，第１引数を示しているんだ。

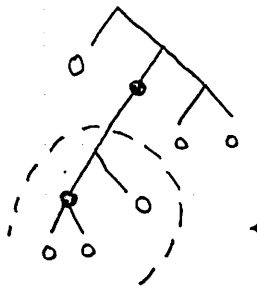
 は，黒石がcadrにある。実引数のlistのcadrは，第２引数だから，左のS式は，第２引数を示す。

H：なるほど。

先生：さて，こんどは，bodyのところへもどろう。



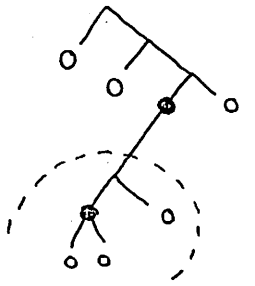
で，X1は，  で示した。で，このX1のところへ，実引数のlistの中の第１引数が入るときを考える。このとき，黒石のcarのところに，さっきやった，第１引数の場所の意味をつなげる。つまり，



← X1の場所を示す

← 第1引数がそこへ入ることを示す

X2のところへも，同じように，第1引数が入るとすると，

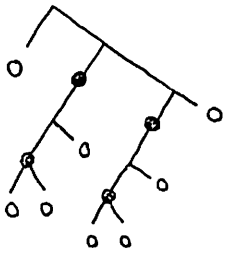


← X2の場所を示す

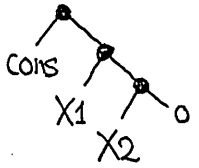
← 第1引数がそこへ入ることを示す

となる。で，この2つを重ねあわせる。

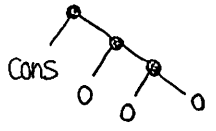
H：つまり，黒石のところを重ねあわせるのですね。次のようになりますね。



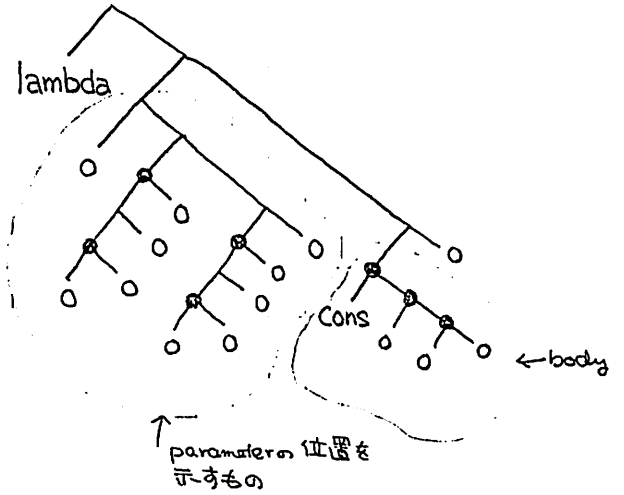
先生：できたね。これがさっきいった「paramaterの位置を示すもの」だ。さて，bodyは，



だったけれど，この仮に書いてあったX1，X2を，0にする。



これが正式なbodyだ。さて，上の「parameterの位置を示すもの」と，上の，からのbodyをlambdaというliteralとつなげて，



となる。

K：あれ，同じものが，さっき出てきたぞ。

先生：これは，どういう関数を表しているだろう。

H：consの両方のoperandに，第1引数が入るのだから，普通のLISPでいうと，

(LAMBDA (X) (CONS X X))

ということですね。

先生：そのとうり。

A：しかし，こんなもの，人間がそのまま書いていたら，頭がおかしくなるに違いない。

先生：そこで，syntax sugarというものがあるんだよ。

K：それは，どこのお店ですか。　A：ばかっ!

先生：殆々のLispでは，次のようなSyntaxを許すんだ。

Lambda([X], cons[X, X])

とかくと，処理系が先のように変換してくれるんだ。ここで，consはliteralだが，Lambda，Xはliteralではない。Lambdaは，このSyntaxの最初を示すKeywordで，Xの方は，擬似的な変数で，metaliteralという。KeywordもmetaliteralもX文字で始まる。だから，literalは，大文字で始めてはならない。

工：ぬぬぬ。

先生：従って，

Lambda([X], cons[X, X])[ab]

を評価すれば，

(ab.ab)

となる。

H：これは前にやったやつですね。

[Lambda([X]; cons[X, X]), ab]

と書くのと同じですね。

A：

Lambda([X]; cons(X, X))[ab]

と書いたらだめなんだろうか。

H：そうすると，cons(ab,ab)となって，abというものが評価されてしまうじゃないか。つまり，実引数が字づらでおきかわると思えばいいんだ。

A：あっそうか。じゃあ，

Lambda([X]; cons(X, X))[car[(a,b)]]

は,

(a.a)

となるんですね。

先生：君は，見かけよりも頭がよいらしいなあ。

H：引数がたくさんあるとどうなるのかな。

先生：たとえば,

Lambda([X,Y]; "(X.Y))[a,b]

は,

(a.b)

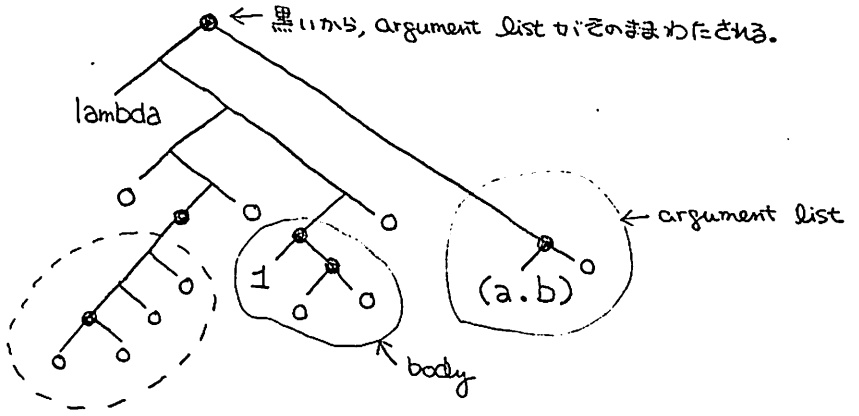
となる。

A：ははぁん？

H：つまり，"(X.Y) のX，Yに，それぞれa，bが代入され，"(a.b) となって，これが評価されて，(a.b) となったわけですね。

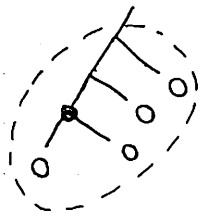
A：うっ，じゃあ，このLispでは，consやsnocという関数はなくても，上のように表わせるのか?

先生：そのとうりだ。もっとびっくりすることをやろう。



は何になるか。

H：( )の中が何か変だぞ。



は，●がcarのcarにあるから，実引数listのcaar，つまり，第1引数のcarを示す。bodyは，[1,X]という形，つまり，1[X]="Xだから，この場合は，第1引数(a.b)のcarであるaがXに入って，"aとなって，これが評価され，全体の値としてaが返るんですね。

先生：そのとうり。

A：げっ，これで，carという関数が表わせてしまった。

H：これをさっきのsyntax sugarで表すとどうなるんですか。

先生：

```
Lambda([[X]]; "X)[(a.b)]
```

となる。

A：

```
Lambda(((X)); "X)[(a.b)]
```

の方がいい気がする。

先生：どちらでも同じなんだ。formal paramaterの宣言部は，metaliteralのargument listでの位置を示すのに使うだけだから。

H：大体lambda式はわかりました。では，labelはあるんですか。

先生：当然だよ。たとえば，

```
Label(Append;
    Lambda([X, Y];
        cond[ null[X] : "Y;
              "1 : cons(car[X],
                        Append(cdr[X], 'Y)) ]))

[(a,b,c), (d,e,f)]
```

をevaluateすると,

```
(a, b, c, d, e, f)
```

となる。

H：Label(…,…)は，どう変換されるのかな。

先生：まあ，大体Lambda(…,…)と同じだよ。でも，label式は，どうせ君たちは，あまり使わないだろうから，この辺にしよう。

H：labelがなくて，どうしてrecursionができるのかな。

先生：LISPのDEFINEと同じような機能があるんだよ。つまり，任意のatomには，関数定義を与えることができる。たとえば，

```
#append[X,Y]
= cond[ null[X] : "Y;
        "1 : cons(car[X],
                  append(cdr[X], 'Y)) ];
```

とtop levelで入力すると，appendが定義される。

実は，formal parameterの宣言のところには，＝をかいてもよい。上のappendは，

(＝ ↑ equal)

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

とした方がよい。

H：つまり，＝の両側は，argument list の中で，同じところを占めるのか。

A：おもろいな。ところでI君は？

K：ボルツへ宿命の対決に行ったよ。

先生：大体，入門といったところは終ったな。

H：このLispは，どこかで動いているのですか。

先生：なにをかくそう，UNIXの上にimplementされているのだ。

一同：ええっ―!?

先生：top levelは，#で始まれば，function definitionと思い，そうでなければ，evaluateして値をprintする。各入力の終わりにはSemicolon（かcomma）をつけること。では，今日はここまでにします。

K：ところで，四角いかっこと丸いかっこはどうちがうんですか。

先生：（ついに怒って）形がちがうんだよ。

（第2回おわり）

最終回

登場人物　　H：話す人

まず，login します。%が出たらば，hy とうって return をおします。すると，Hyperlisp(2.1)に入ります。何も prompt を出しません。ためしに，cons[a, b]; return とうちこみましょう。

```
% hy
cons[a, b];
(a . b)
```

すると，上のように(a.b) と答えが出ます。上で，入力は，＿＿を引いてあります。おなじみの ff を定義しましょう。

```
#ff[X = (X1)]
= cond[ atom[X] : "X ;
        "1 : ff[X1] ];
ff
```

定義がうまくいくと，systemは，関数名を print します。では，

```
ff[(((1, 0, (1))))]
1
ff[(([1, 0, (1)]))]
[1, 0, (1)]
```

Systemからぬけるときは control/d を入れます。

```
control/d
eof
%
```

もっとくわしく知りたい人は，manualを読みましょう。

% roff manual

で出てきます。
（最終回おわり）

あとがき

本稿は，佐藤雅彦先生のdesignしたHyperlispのTutorialで，「先生」，H，A，K，Eという人物の対話という形式をとっている。「先生」を除いて，登場人物はすべて，実在する人物をモデルとしている。

本稿は，今年の7月ごろ書かれ，非公式に回し読みをされ，かなりの反響をよんだ。特に，K氏の特異的存在と，formalistである「先生」，Hと，practicianであるAの対決が興味をひいたらしい。

最近になって，清書をしろという声が多方面から出され，このような形となった。清書の段階で，Aの発言の一部を削除したが，あまりにどぎついと思われたからである。

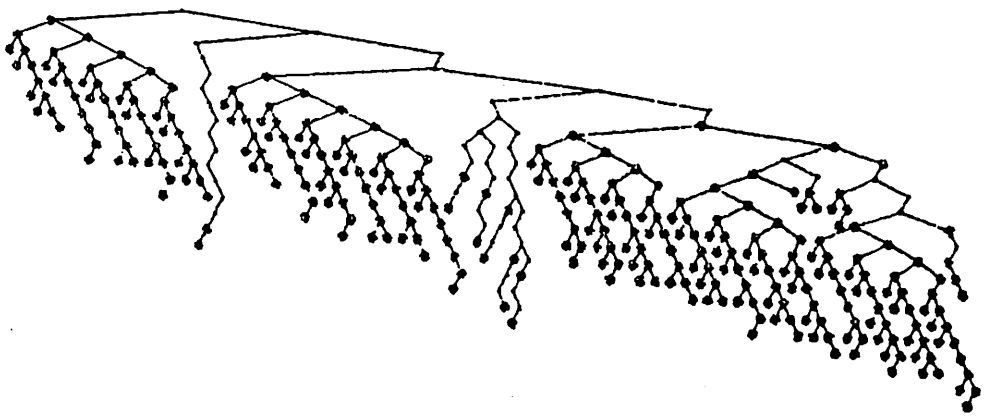
Hyperlispについてさらに知りたい人は，次の文献を見て欲しい。

M. Sato Theory of Symbolic Expressions, 東京大学理学部情報科学科 TR80-16

M. Hagiya Hyperlisp2.1 Manual

謝辞

本書は，何から何まで著者一人でやったので，感謝する人はいない。



検印廃止

よいこの
Hyperlisp

1980年11月10日 初版1刷発行	著者　はぎやまさみ 発行者　同上