ソフトウェア工学　28−1
（１９８３．２．９）

# 例題系列に基くプログラムの解析
## ——LISPプログラムからの例題系列の生成——

伊藤貴康・田村卓
（東北大学工学部）

## 1．まえがき

プログラムの計算の振舞いを適当な例題系列を生成することによって理解する事は，日頃，我々がよる活用する手法である．自然数nの階乗を計算する関数f(n)のLISP流の再帰的関数定義は次のように与えられる：

f(n)⇐ if n=0 then 1 else f(n-1)×n

f(n)による計算を次のような例題系列およびドット系列記法によって表現し，理解することができる．

f(n):{ n=0→1, n=1⇒1×1, n=2⇒1×1×2,
n=3→1×1×2×3, ・・・ }

f(n):{(0,1),(1,1),(2,2),(3,6),(4,24),・・・}

f(n)=1×2×3×・・・×(n−1)×n

プログラムから有限個の分り易い例題系列を生成する事，および，作成済みのプログラムの理解や検証が例題系列によって行える事はIncremental Programming環境において有用かつ重要であると考える．

本論文では，LISPプログラムからの例題系列の生成とそれに基くプログラム解析について，筆者らの研究室で行っている研究の一部を紹介する．

## 2．プログラムからの例題系列生成法

プログラムから，その計算の振舞いを分り易く表現する例題系列を生成するためには，いくつかの機能を準備する必要がある．例題系列の生成は，与えられたプログラムをスキーマ（schema）と考えて評価・展開し，その結果を変形・簡単化して望ましい例題系列を生成することであると考えてよい．次のような機能が必要とされる．

(1) 記号的評価（symbolic evaluator）

プログラムや式を記号的あるいはスキーマ的に評価・展開する機能である．

(2) 簡単化（simplification）

プログラム記述の中の制御構造（条件文など）および演算に関する簡単化を行う機能である．

(3) 部分評価と値の設定

条件文の条件を評価し，値を求め，その値を条件の成立する分岐に対しては設定するような機能．

(4) 例題系列の生成

例題系列の生成の仕方にも，先述のf(n)の例から知られるように種々の形がある．例えば，

<入出力値対応系列>

{(0,1),(1,1),(2,2),(3,6),(4,24),・・・}

<出力値系列>

{1, 1, 2, 6, 24, ・・・}

<入力条件・出力計算式対応系列>

{n=0⇒1, n=1⇒1×1, n=2⇒1×1×2, ・・・}

<ドット系列記法の式>

1×2×3×・・・×n

ユーザの選択に応じて，このような諸種の系列を生成できる機能が必要である．

[記号的評価について]

LISPプログラムの場合における記号的評価は，再帰的関数の評価展開と考えてよい．次の再帰的関数を例に考えてみよう．

f(x)⇐ if p(x) then a(x) else f(b(x))

このような関数の評価展開に次の2つの考え方がある．

① 項書き換えシステム（TRS）に基く展開

与えられた再帰的関数定義を

f(x)⇒ if p(x) then a(x) else f(b(x))

なるTerm Rewriting System（TRS）と考えて展開する方法である．例えば

f(x)⇒ if p(x) then a(x) else f(b(x))
⇒ if p(x) then a(x) else
[if p(b(x)) then a(b(x)) else f(b(b(x)))]
・・・・・・・

TRSに基く展開にも評価方式によって，深い展開・浅い展開などが考案されているが，それらについては別の機会に報告したい．上述のような展開・評価を何回か行ってから条件式の簡単化などに進むのが基本的な考え

方である.

②記号的インタプリタ(Symbolic Interpreter)に基く展開

プログラムに入力を与えながら,順次,解釈実行させることによって値の系列を生成できるから,インタプリタを値系列の生成に活用することが考えられる.すなわち,記号的入力が与えられたプログラムを,必要なレベルまで解釈実行する記号的インタプリタを用いることが考えられる.

LISPの場合にはそのインタプリタevalquoteが知られているので,対応する記号的インタプリタs-evalquoteが容易に構成でき,これを用いてLISP関数の展開評価が行える.

本論文では,この記号的インタプリタ方式(すなわち,Symbolic LISP interpreter:s-evalquote)に基くLISPプログラムからの例題系列の生成法とその実験結果について報告する.

## 3. LISPプログラムから例題系列の生成を行う記号的インタプリタ

LISPは,解釈実行を行うインタプリタevalquote[fn,x]を用いて,そのシンタックスとセマンティクスが簡明に定義された言語である.evalquoteは,関数(fn)と引数リスト(x)が具体的に与えられると評価が始まり,fn(x)を計算する.例題系列の生成という観点からは,記号的評価を行うインタプリタに変更する必要がある.記号的インタプリタの作成に当って,再帰的関数の評価を有限で打ち切る停止条件の指定,評価を行う関数の指定,LISPシステム関数の部分評価などを取り入れる必要がある.[以下の説明の理解には,LISPに関しての入門的知識が不可欠であるが,紙数の関係上省略した.]

Pure LISPを対象として,その記号的インタプリタと例題系列の生成について説明を行う.

### 3.1 LISP記号的インタプリタ:s-evalquote

具体的なLISP関数を扱えるようにするために,Pure LISPにSUBR関数,EXPR関数および数値データ,定数"T"と"NIL"を扱えるようにPure LISPのevalquoteを拡張してから,次のような評価を行う記号的インタプリタを作成した.

(1) 定数は評価しない
(2) 変数は未定義にならぬ限り評価する
(3) 条件式はすべての条件・結果の対に対して評価を行う
(4) SUBR関数の適用は引数リストに関してのみ評価を行う
(5) EXPR関数の適用は,その定義の本体を引数リストを評価した結果に適用したものである.

ただし,①SUBR関数に対する引数リストの値が定まると値を計算するものとする,②再帰的定義のEXPR関数に対しては,上述の評価法(5)だけであると評価が停止しないので停止条件を導入するものとする.

上述のような配慮の下に,evalquoteから構成されたs-evalquoteを図1(次頁)に与えた.

◆ s-evalquote[fn;args;ff;s] 記号的インタプリタであり,fnが関数,argsが引数,ffが展開評価を行う関数名のリスト,sが評価を行うべき回数(すなわち,停止条件)である.実際の評価は,s-applyおよびs-evalを用いて行われる.

◆ s-apply[fn;args;a;ff;s]
環境aの下で関数fnを引数リストargsに記号的に適用し,関数リストff,停止条件sの下で展開評価する.すなわち,

①fnがEXPR関数の場合,関数定義の本体を取り出し,s-applyに再び適用する.
②fnがSUBR関数の場合,引数リストの要素がすべて定数ならば実際に値を計算し,QUOTEを付け,そうでないならば単に関数適用の式を作る.
③fnがUNDEF関数の場合は単に関数適用の式を作る.(fnがSUBR関数あるいはEXPR関数でなく,fnが環境aに存在しないか,fn自身とbindされているときに,fnをUNDEF関数と呼ぶ)
④fnがLAMBDA関数またはLABEL関数の場合には,夫に,s-evalおよびs-applyをその本体に対して行う.

◆ s-eval[form;a;ff;s]
環境aの下で(関数リストff,停止条件sに関して)formを記号的に評価する.すなわち,

①formが,数値か,Tか;NILか;aの下でbindされていない変数のときはそのものを答とする.そうでな

```
s-evalquote[fn;args;ff;s] <=e=
  s-apply[fn;args;nil;ff;s]


s-apply[fn;args;a;ff;s] <=e=
  [atom[fn]
     -> [fn-type[fn;EXPR;a]
           -> [and[plusp[s];member[fn;ff]]
                 -> s-apply[fn-body[fn];args;a;ff;s];
               t -> cons[fn;args]];
         fn-type[fn;SUBR;a]
           -> [defined-args[args]
                 -> list[QUOTE;eval[cons[fn;args];a]];
               t -> cons[fn;args]];
         fn-type[fn;UNDEF;a] -> cons[fn;args];
         t -> s-apply[cdr[assoc[fn;a]];args;a;ff;s]];
   eq[car[fn];LAMBDA]
     -> s-eval[caddr[fn];pairlis[cadr[fn];args;a];ff;s];
   eq[car[fn];LABEL]
     -> s-apply[caddr[fn];
                args;
                cons[cons[cadr[fn];caddr[fn]];a];
                ff;
                s]]

s-eval[form;a;ff;s] <=e=
  [numberp[form] -> form;
   atom[form]
     -> [eq[form;t] -> t;
         null[form] -> nil;
         null[assoc[form;a]] -> form;
         t -> cdr[assoc[form;a]]];
   eq[car[form];QUOTE]
     -> [numberp[cadr[form]] -> cadr[form];
         eq[cadr[form];t] -> t;
         null[cadr[form]] -> nil;
         t -> form];
   eq[car[form];COND]
     -> s-evcon-2[s-evcon-1[cdr[form];a;ff;s]];
   atom[car[form]]
     -> [fn-type[car[form];EXPR;a]
           -> [and[plusp[s];member[car[form];ff]]
                 -> s-apply[fn-body[car[form]];
                            s-evlis[cdr[form];a;ff;subl[s]];
                            a;
                            ff;
                            subl[s]];
               t -> cons[car[form];s-evlis[cdr[form];a;ff;s]]];
         fn-type[car[form];SUBR;a]
           -> [defined-args[cdr[form]]
                 -> list[QUOTE;eval[form;a]];
               t -> cons[car[form];s-evlis[cdr[form];a;ff;s]]];
         fn-type[car[form];UNDEF;a]
           -> cons[car[form];s-evlis[cdr[form];a;ff;s]];
         t -> s-eval[cons[cdr[assoc[car[form];a]];
                         cdr[form];
                         a;
                         ff;
                         s]]];
   t -> s-apply[car[form];s-evlis[cdr[form];a;ff;s];a;ff;s]]

 s-evcon-2[c] <=e=
   [null[c] -> nil;
    and[null[cdr[c]];eq[caar[c];t]] -> cadar[c];
    t -> cons[COND;c]]


 s-evcon-1[c;a;ff;s] <=e=
   [null[c] -> nil;
    null[s-eval[caar[c];a;ff;s]] -> s-evcon-1[cdr[c];a;ff;s];
    defined[s-eval[caar[c];a;ff;s]]
      -> list[list[t;s-eval[cadar[c];a;ff;s]]];
    t -> cons[list[s-eval[caar[c];a;ff;s];
                   s-eval[cadar[c];a;ff;s]];
              s-evcon-1[cdr[c];a;ff;s]]]


 s-evlis[m;a;ff;s] <=e=
   [null[m] -> nil;
    t -> cons[s-eval[car[m];a;ff;s];s-evlis[cdr[m];a;ff;s]]]


 pairlis[x;y;a] <=e=
   [null[x] -> a;
    t -> cons[cons[car[x];car[y]];pairlis[cdr[x];cdr[y]]]]


fn-type[fn;type;a] <=e=
  [eq[type;UNDEF]
     -> or[null[assoc[fn;a]];eq[cdr[assoc[fn;a]];fn]];
   null[getd[fn]] -> nil;
   t -> eq[car[getd[fn]];type]]
```

図1. LISP Symbolic Evaluator
"s-evalquote"

```
fn-body[fn] <=e=
  cdr[getd[fn]]


defined-args[args] <=e=
  [null[args] -> t;
   defined[car[args]] -> defined-args[cdr[args]];
   t -> nil]


defined[e] <=e=
  [numberp[e] -> t;
   eq[e;t] -> t;
   null[e] -> t;
   atom[e] -> nil;
   t -> eq[car[e];QUOTE]]
```

いときは，formがatomならaから対応する値を取り出す．

② formが(QUOTE e)の場合，eが数値，T，NILのときはe，そうでないときはform自身を答．

③ formが条件式の場合，s-evcon1，s-evcon2を用いて処理

④ formが関数atom適用の場合，引数リストをs-evlisによって記号的に評価した後，関数タイプに応じて処理．

・・・・・・

◆停止条件について

evalquoteの定義を単純に記号的インタプリタに書き換えると，再帰的関数定義の場合，評価が無限に続くため，図1のs-evalquoteでは評価の対象とする関数リストと評価の実行回数を設定できるようにした．s-evalにおいてffに属する関数の適用の式の評価毎にsの値を1つ減らし，s=0のときに評価を打ち切るようにしている．

3.2 式の簡単化および例題系列生成

与えられた停止条件の下でs-evalquoteによってPure LISPのプログラムを記号的に実行し，展開形を求めることができる．しかし簡明な例題系列を得るには，条件式や算術演算・リスト処理基本演算に関する簡単化や式の変形を行うことが要求される．現在行っている簡単化は次の通りである：

① 条件式に関する簡単化

if-then-elseの条件式に関する簡単化および式の変形［文献5( )参照］

② 基本演算に関する簡単化

・四則演算に関する簡単化

・リスト処理演算(cons, car, cdr, list,

append）に関する簡単化

また条件式の条件が単純な場合は，条件を満す値が一意的に求まる場合もあり，その場合には具体的な値をその条件に対応する式に代入して簡単化および評価を行う．[これを値の設定操作と言う．]

例題系列の生成の形式としては，

① 式の生成
② 入出力対応系列の生成
③ ドット系列記法の生成
④ 出力値系列の生成

が実現されている．式の簡単化を行う場合と簡単化を行わぬ場合があり，利用者が選択して例題系列を生成できるようになっている．このために図2のような関数群が用意されている．

| 出力形式 | 代数的・記号的簡単化 | |
|---|---|---|
| | 無 | 有 |
| 展開式 | i-expand-fn | e-expand-fn |
| 入出力対応系列 | i-val-table | e-val-table |
| ドット系列記法 | i-dot-seq | e-dot-seq |
| 出力値系列 | i-val-seq | e-val-seq |

図2 例題系列生成の形式

## 4. 例題系列に基くプログラムの解析例

記号的インタプリタ S-evalquote と簡単化および例題系列生成機能を用いて実際の LISP プログラムを解析し，例題系列を生成した例を示す．

### 4.1 階乗関数 fact[x] の展開例

自然数 x の階乗を求める関数 fact[x] は次のように定義される：

fact[x] = if x=0 then 1 else fact[x-1]*x

この関数を図2の8種類の例題系列生成法によって処理した結果を図3に示した．fact[x] の場合，展開式を求めた時点で，各条件が等号判定となり，簡単に，等号成立時の x が求まるのでその値を結果の式に代入して値系列などを計算することができる．図3では展開を3回行うように指示している．与えられた関数に対して何回展開を行えばよいかは難しい問題である．[例題系列からのプログラム合成アルゴリズムが知られている場合には，再合成できるレベルまでは展開を進めてやるというのが1つの目やすである．]

```
fact[x] <=e=
  [x=0 -> 1;
   t -> fact[x-1]*x]
-----
? [I-EXPAND-FN 'FACT '(FACT) 3]
fact[x] =>
  [x=0 -> 1;
   x-1=0 -> 1*x;
   x-2=0 -> 1*(x-1)*x;
   x-3=0
     -> 1*(x-2)*(x-1)*x;
   t -> fact[x-1]*(x-3)*(x-2)*(x-1)*x]
-----
? [E-EXPAND-FN 'FACT '(FACT) 3]
fact[x] =>
  [x=0 -> 1;
   x=1 -> 1*x;
   x=2 -> 1*(x-1)*x;
   x=3
     -> 1*(x-2)*(x-1)*x;
   t -> fact[x-1]*(x-3)*(x-2)*(x-1)*x]
-----
? [I-VAL-TABLE 'FACT '(FACT) 3]
fact[x] =>
  [x=0 -> 1;
   x=1 -> 1*1;
   x=2 -> 1*1*2;
   x=3 -> 1*1*2*3]
-----
? [E-VAL-TABLE 'FACT '(FACT) 3]
fact[x] =>
  [x=0 -> 1;
   x=1 -> 1;
   x=2 -> 2;
   x=3 -> 6]
-----
? [I-DOT-SEQ 'FACT '(FACT) 3]
fact[x] =>
  1*1*2*...*(x-3)*(x-2)*(x-1)*x
-----
? [E-DOT-SEQ 'FACT '(FACT) 3]
fact[x] =>
  1*2*...*(x-3)*(x-2)*(x-1)*x
-----
? [I-VAL-SEQ 'FACT '(FACT) 3]
fact[x] =>
( 1 , 1*1 , 1*1*2 , 1*1*2*3 ,  ... )
-----
? [E-VAL-SEQ 'FACT '(FACT) 3]
fact[x] =>
( 1 , 1 , 2 , 6 ,  ... )
-----
```

図3 階乗関数 fact[x] の展開例

### 4.2 Fibonacci 関数の展開例

自然数 x に対する Fibonacci 関数 fib(x) は図4のように展開できる．この

```
fib[n] <=e=
  [n=0 -> 1;
   n=1 -> 1;
   t -> fib[n-1]+fib[n-2]]


? [I-VAL-TABLE 'FIB '(FIB) 2]
fib[n] =>
  [n=0 -> 1;
   n=1 -> 1;
   n=2 -> 1+1;
   n=3 -> 1+1+1;
   n=4
     -> fib[1]+fib[0]+1+(1+1);
   n=5
     -> fib[2]+fib[1]+(fib[1]+fib[0])+(fib[1]+fib[0]+1)]
-----
? [E-VAL-TABLE 'FIB '(FIB) 2]
fib[n] =>
  [n=0 -> 1;
   n=1 -> 1;
   n=2 -> 2;
   n=3 -> 3;
   n=4 -> 5;
   n=5 -> 8]
-----
? [E-VAL-SEQ 'FIB '(FIB) 2]
fib[n] =>
( 1 , 1 , 2 , 3 , 5 , 8 ,  ... )
-----
```

図4 Fibonacci 関数の展開例

例における展開の回数は図5から知られるように evaluation tree の深さ2までである。

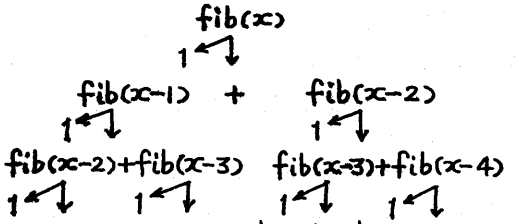


図5 Fibonacci関数の解析

## 4.3 簡単な記号処理関数の展開例

リストxのreverseを行う関数reva[x]の展開例を図6に与えた。reva[x]の展開式では条件式の各条件がnullによるNIL判定となるが、NIL判定はS式の構造に関する判定なのでXの値が一意には定まらない。そこで任意のS式を表わすXn (n=1,2,3,---)を用いて値の設定を行っている。単に値の設定を行ったものにはappend関数が残っているが、更にappendに関する簡単化を行うことにより、図6のような入出力対応系列が求まる。なお、図中で、(car[x])はlist[car[x]]の意味である。

maplist[fn;x]の展開例を図7に示した。この結果から、maplistは、リストxのcdr部に次々と関数fnを適用する関数であることが容易に知られる。

```
reva[x] <=e=
  [null[x] -> nil;
   t -> append[reva[cdr[x]];(car[x])]]
-----
? [I-EXPAND-FN 'REVA '(REVA) 3]
reva[x] =>
  [null[x] -> nil;
   null[cdr[x]] -> append[nil;(car[x])];
   null[cddr[x]]
     -> append[append[nil;(cadr[x])];(car[x])];
   null[cdddr[x]]
     -> append[append[append[nil;(caddr[x])];
                      (cadr[x])];
               (car[x])];
   t -> append[append[append[append[reva[cddddr[x]];
                                    (cadddr[x])];
                             (caddr[x])];
                      (cadr[x])];
               (car[x])]]
-----
? [I-VAL-TABLE 'REVA '(REVA) 3]
reva[x] =>
  [x=NIL -> nil;
   x=(X2) -> append[nil;(X2)];
   x=(X3 X2)
     -> append[append[nil;(X2)];(X3)];
   x=(X4 X3 X2)
     -> append[append[append[nil;(X2)];
                      (X3)];
               (X4)]]
-----
? [E-VAL-TABLE 'REVA '(REVA) 3]
reva[x] =>
  [x=NIL -> nil;
   x=(X2) -> (X2);
   x=(X3 X2) -> (X2 X3);
   x=(X4 X3 X2) -> (X2 X3 X4)]
-----
```

図6. reverse関数 reva[x] の展開

```
maplist[fn;x] <=e=
  [null[x] -> nil;
   t -> (fn[x] . maplist[fn;cdr[x]])]
-----
? [I-EXPAND-FN 'MAPLIST '(MAPLIST) 3]
maplist[fn;x] =>
  [null[x] -> nil;
   null[cdr[x]] -> (fn[x]);
   null[cddr[x]] -> (fn[x] fn[cdr[x]]);
   null[cdddr[x]]
     -> (fn[x] fn[cdr[x]] fn[cddr[x]]);
   t -> (fn[x] fn[cdr[x]] fn[cddr[x]] fn[cddddr[x]] . maplist[fn;
                                                        cddddr[x]])
]
-----
? [I-VAL-TABLE 'MAPLIST '(MAPLIST) 3]
maplist[fn;x] =>
  [x=NIL -> nil;
   x=(X2) -> (fn[(X2)]);
   x=(X3 X2)
     -> (fn[(X3 X2)] fn[(X2)]);
   x=(X4 X3 X2)
     -> (fn[(X4 X3 X2)] fn[(X3 X2)] fn[(X2)])]
-----
```

図7. 関数 maplist[fn;x] の展開

## 4.4 インタプリタ関数 eval*[e;a] の展開について

LISPインタプリタは、evalquote*[fn;args]で与えられるが、実際の評価はapply*、eval*が相互に呼び合うことによって行われる。

apply*[fn;args;a]は環境aのもとで引数リストargsに関数fnを適用する関数である。また、eval*[e;a]は環境aのもとで式eを評価する関数である。

以上の様なapply*, eval*の概念的な理解を持った上で、eval*[e;a]の動きを理解することを考える。しかるに、eval*[e;a]は図8の如くassoc*, evcon*, evlis*を用

```
eval*[e;a] <=e=
  [atom[e] -> cdr[assoc*[e;a]];
   atom[car[e]]
     -> [eq[car[e];QUOTE] -> cadr[e];
         eq[car[e];COND] -> evcon*[cdr[e];a];
         t -> apply*[car[e];evlis*[cdr[e];a];a]];
   t -> apply*[car[e];evlis*[cdr[e];a];a]]


assoc*[x;a] <=e=
  [equal[caar[a];x] -> car[a];
   t -> assoc*[x;cdr[a]]]


evcon*[c;a] <=e=
  [eval*[caar[c];a] -> eval*[cadar[c];a];
   t -> evcon*[cdr[c];a]]


evlis*[m;a] <=e=
  [null[m] -> nil;
   t -> cons[eval*[car[m];a];evlis*[cdr[m];a]]]
```

図8. LISPインタプリタ関数 eval*[e;a] の定義

```
? [I-EXPAND 'ASSOC* '(E A) '(ASSOC*) 3]
assoc*[e;a] =>
  [caar[a]=e -> car[a];
   caadr[a]=e -> cadr[a];
   caaddr[a]=e -> caddr[a];
   caadddr[a]=e
     -> cadddr[a];
   t -> assoc*[e;cddddr[a]]]
-----
? [I-EXPAND 'EVCON* '((CDR E) A) '(EVCON*) 3]
evcon*[cdr[e];a] =>
  [eval*[caadr[e];a] -> eval*[cadadr[e];a];
   eval*[caaddr[e];a]
     -> eval*[cadaddr[e];a];
   eval*[caadddr[e];a]
     -> eval*[cadadddr[e];a];
   eval*[caaddddr[e];a]
     -> eval*[cadaddddr[e];a];
   t -> evcon*[cddddr[e];a]]
-----
? [I-EXPAND 'EVLIS* '((CDR E) A) '(EVLIS*) 3]
evlis*[cdr[e];a] =>
  [null[cdr[e]] -> nil;
   null[cddr[e]] -> cons[eval*[cadr[e];a];nil];
   null[cdddr[e]]
     -> cons[eval*[cadr[e];a];
             cons[eval*[caddr[e];a];nil]];
   null[cddddr[e]]
     -> cons[eval*[cadr[e];a];
             cons[eval*[caddr[e];a];
                  cons[eval*[cadddr[e];a];nil]]];
   t -> cons[eval*[cadr[e];a];
             cons[eval*[caddr[e];a];
                  cons[eval*[cadddr[e];a];
                       cons[eval*[caddddr[e];
                                  a];
                            evlis*[cddddr[e];
                                   a]]]]]]
```

図9 assoc*, evcon*, evlis*の展開

いて定義される。eval*[e;a]の定義にはeval*は直接的な再帰呼び出しとしては現われない。しかし、assoc*は単純な再帰関数ではあるが、evcon*, evlis*によってeval*は間接的に再帰呼び出しがされている。更にeval*を既知関数と考えれば、evcon*, evlis*も単純な再帰関数とみなされる。そこで、eval*[e;a]から、evcon*, evlis*, assoc*を消去し、eval*およびapply*のみによって理解することを考えて展開を行う事が考えられる。

このような観点から、assoc*, evcon*, evlis*をそれぞれ自分自身について3回展開評価した結果を図9に与えた。また、その結果をeval*の定義に埋め込んで得られる結果を図10に与えた。図10によってeval*, apply*の意味を上述のように解しながら、eval*の振舞いを理解することができる。

```
eval*[e;a] =>
  [atom[e] -> cdr[[caar[a]=e -> car[a];
                  caadr[a]=e -> cadr[a];
                  caaddr[a]=e -> caddr[a];
                  caadddr[a]=e
                    -> cadddr[a];
                  t -> assoc*[e;cddddr[a]]]];
   atom[car[e]]
     -> [car[e]=QUOTE -> cadr[e];
         car[e]=COND -> [eval*[caadr[e];a] -> eval*[cadadr[e];a];
                         eval*[caaddr[e];a]
                           -> eval*[cadaddr[e];a];
                         eval*[caadddr[e];a]
                           -> eval*[cadadddr[e];a];
                         eval*[caaddddr[e];a]
                           -> eval*[cadaddddr[e];a];
                         t -> evcon*[cddddr[e];a]];
         t -> apply*[car[e];
                     [null[cdr[e]] -> nil;
                      null[cddr[e]] -> cons[eval*[cadr[e];a];nil];
                      null[cdddr[e]]
                        -> cons[eval*[cadr[e];a];
                                cons[eval*[caddr[e];a];nil]];
                      null[cddddr[e]]
                        -> cons[eval*[cadr[e];a];
                                cons[eval*[caddr[e];a];
                                     cons[eval*[cadddr[e];a];nil]]];
                      t -> cons[eval*[cadr[e];a];
                                cons[eval*[caddr[e];a];
                                     cons[eval*[cadddr[e];a];
                                          cons[eval*[caddddr[e];
                                                     a];
                                               evlis*[cddddr[e];
                                                      a]]]]]];
                     a];
   t -> apply*[car[e];
               [null[cdr[e]] -> nil;
                null[cddr[e]] -> cons[eval*[cadr[e];a];nil];
                null[cdddr[e]]
                  -> cons[eval*[cadr[e];a];
                          cons[eval*[caddr[e];a];nil]];
                null[cddddr[e]]
                  -> cons[eval*[cadr[e];a];
                          cons[eval*[caddr[e];a];
                               cons[eval*[cadddr[e];a];nil]]];
                t -> cons[eval*[cadr[e];a];
                          cons[eval*[caddr[e];a];
                               cons[eval*[cadddr[e];a];
                                    cons[eval*[caddddr[e];
                                               a];
                                         evlis*[cddddr[e];
                                                a]]]]];
               a]]
```

図10 eval*[e;a]の展開例

## 6. あとがき

プログラムから例題系列を生成することは、プログラムの解析・理解という観点から有益と考えられるが、筆者らの知る限り、十分な研究がなされているとは言えない。本研究における試みがこのような研究に刺激を与えればと考える。[他の言語のインタプリタも同様な考え方で記号的インタプリタに改造できる本論文の方法が適用できると考える。]

S-evalquoteはpure LISPに留まっているが、LISP 1.5レベルに拡張する事、その他いろいろな基本関数に対する簡単化などを取り入れることを検討している。TRSに基く例題系列生成法、EXPOLに基く合成システムとの融合などについても研究を行っており、別の機会に報告したい。


文献 (1) J. McCarthy, 他: LISP 1.5, MIT
(2) 伊藤: ソフトウェア工学基礎論, 丸善 (昭56年)
(3) 伊藤・田村: 本学会第24回全国大会論文 213-8 (昭57年3月)