Vol. 26 No. 4 情報処理学会論文誌 July 1985

万能関数 APPLY を用いず EVAL のみを用いた LISP インタプリタ†

河村 知行††

新しく開発した LISP 処理系のインタプリタ (YAL: Yet Another LISP interpreter) について述べている．従来の LISP インタプリタが，万能関数 EVAL と APPLY により構成されていたのに対し，YAL では，従来の LISP システムとの互換性を保ったまま，万能関数 EVAL だけで，LISP インタプリタを実現している．万能関数 APPLY を除去した結果，①インタプリタの核部分を小さくすることが可能になり，②LAMBDA, LABEL, FUNARG 機能を，リスト処理の核部分から分離することができた．

1. はじめに

これまで LISP インタプリタは LISP 1.5[1),2)] の万能関数 EVAL・APPLY に代表されるようなインタプリタが使用されてきた．もちろん，実際のインプリメンテイションでは，処理効率の向上のために種々の改良が施されている[3),4)]．たとえば，万能関数 EVAL のなかに，万能関数 APPLY を埋め込むようなことも行われている．また，LISP 1.9[5)] のように，新しい関数属性を設けて，言語仕様まで含めた意味での改良もなされてきた．

しかし，これらの処理系（インタプリタ）は，万能関数 EVAL と APPLY の二つを使うという意味で，LISP 1.5 の処理系と本質的には同じ物である．(ただし，ここでは deepbinding や shallowbinding など，変数の束縛の問題には触れないことにする．また，本論文の説明は，LISP 1.5 流の ALIST を用いた deepbinding に沿って行う．) もちろん，LISP 1.5 のインタプリタを変造して，本質的に異なったインタプリタを作ることは簡単である．しかし，その新しいインタプリタに，積極的な意味での価値を見いだすのは，困難のように思われた．本論文で報告するのも，この変造 LISP インタプリタの一つである．しかし，この変造インタプリタは，従来の LISP との互換性を保ちながら，万能関数 APPLY が除去されており，積極的な意味での価値があるものと思われる．この変造 LISP インタプリタを YAL (Yet Another LISP interpreter) と呼ぶ．

以下，LISP の表記法などは，LISP 1.9 風に行うものとする．EXPR 属性と FEXPR 属性の関数の定義は，次のように記述するものとする．

(DE 関数名 変数リスト 本体)

(DF 関数名 (引数名 環境名) 本体)

この記法を採用したのは，LAMBDA 記法が表に現れていないため，2章以降に示すように，YAL に適した記法となっているためである．QUOTE については (QUOTE X) の略記法として，'X を用いる．真理値の真の値をもつ ATOM として T を用いる．[ ] は，スーパーカッコとして用いる．また，従来の LISP インタプリタの代表として，LISP 1.5 を取上げ，説明はこれとの比較により行うものとする．

2. YAL のアルゴリズム

図1に示すのが，PASCAL 風の言語により記述された LISP 1.5 の LISP インタプリタである．このインタプリタは次の4点に注意が払われている．

① 簡単のため，関数属性は，EXPR・FEXPR・SUBR・FSUBR だけとしている．

② ＊印の部分は，apply を呼ばず，直接 eval を呼び出して，高速化を図っている．

③ 読みやすさのため，テールリカージョンの最適化を行わず，すべて，再帰呼び出しで記述してある．

④ 簡単のため，エラー処理を記述していない．

図1において，関数 property・varlist・body・subr・fsbr は，おのおの ATOM の属性の取出し，関数の変数リストの取出し，関数の本体の取出し，SUBR 関数の識別，FSUBR 関数の識別のために導入された関数であり，ATOM の実現方法から，アルゴリズムを独立にするためのものである．apvalø・exprø・fexprø・subrø・fsubrø は，ATOM の属性値である．その他の関数は，LISP 1.5 と同じものを用いてい

† A LISP Interpreter by Universal Function EVAL Only, without APPLY by TOMOYUKI KAWAMURA (Department of Information Electronics, Tokuyama Technical College).

†† 徳山工業高等専門学校情報電子工学科

```
FUNCTION apply(fn,args,alist:cellpt):cellpt;
BEGIN
 IF atom(fn) THEN
  CASE property(fn) OF
①  expr0: apply:=apply(body(fn),args,alist);
   subr0: apply:=do_subr(subr(fn),args,alist);
   ELSE : apply:=apply(cdr(sassoc(fn,alist)),args,alist);
  END
 ELSE IF car(fn)=label  THEN apply:=apply(caddr(fn),args,cons(cons(cadr(fn),caddr(fn)),alist))
㊁ ELSE IF car(fn)=funarg THEN apply:=apply(cadr(fn),args,caddr(fn))
 ELSE IF car(fn)=lambda THEN apply:=eval(caddr(fn),nconc(pair(cadr(fn),args),alist))
㊂ ELSE                      apply:=apply(eval(fn,alist),args,alist);
END;

FUNCTION eval(form,alist:cellpt):cellpt;
BEGIN
 IF atom(form) THEN
  IF      numberp(form)         THEN eval:=form
  ELSE IF property(form)=apval0 THEN eval:=apval(form)
  ELSE                               eval:=cdr(sassoc(form,alist))
 ELSE IF atom(car(form)) THEN
  CASE property(car(form)) OF
*  expr0 : eval:=eval(body(car(form)),nconc(pair(varlist(car(form)),evlis(cdr(form),alist)),alist));
㊀ *  fexpr0: eval:=eval(body(car(form)),nconc(pair(varlist(car(form)), list(cdr(form),alist)),alist));
   subr0 : eval:=do_subr(subr(car(form)),evlis(cdr(form),alist),alist);
   fsubr0: eval:=do_fsbr(fsbr(car(form)),      cdr(form)      ,alist);
   ELSE  : eval:=eval(cons(cdr(sassoc(car(form),alist)),cdr(form)),alist);
  END
 ELSE eval:=apply(car(form),evlis(cdr(form),alist),alist);
END;
```

図 1 従来の LISP インタプリタ
Fig. 1 Customary LISP interpreter.

```
FUNCTION eval(form,alist:cellpt):cellpt;
BEGIN
 IF atom(form) THEN
  IF      numberp(form)         THEN eval:=form
  ELSE IF property(form)=apval0 THEN eval:=apval(form)
  ELSE                               eval:=cdr(sassoc(form,alist))
 ELSE IF atom(car(form)) THEN
  CASE property(car(form)) OF
   expr0 : eval:=eval(body(car(form)),nconc(pair(varlist(car(form)),evlis(cdr(form),alist)),alist));
㊃  fexpr0: eval:=eval(body(car(form)),nconc(pair(varlist(car(form)), list(cdr(form),alist)),alist));
   subr0 : eval:=do_subr(subr(car(form)),evlis(cdr(form),alist),alist);
   fsubr0: eval:=do_fsbr(fsbr(car(form)),      cdr(form)      ,alist);
   ELSE  : eval:=eval(cons(cdr(sassoc(car(form),alist)),cdr(form)),alist);
  END
☆ ELSE eval:=eval(cons(eval(car(form),alist),cdr(form)),alist);
END;
```

図 2 新しい LISP インタプリタ (YAL)
Fig. 2 New LISP interpreter (YAL).

る.

図 2 に示すのが，本論文で述べる変造インタプリタ YAL である．YAL の万能関数 eval は☆印の箇所で LISP 1.5 インタプリタの万能関数 eval と異なっているだけである．要するに

apply(car(form), evlis(cdr(form), alist), alist)

を,

eval(cons(eval(car(form), alist), cdr(form)), alist)

に変更しただけである．これにより，YAL から，万能関数 apply が完全に除去されている．しかし，これでは，従来の LISP にあった LAMBDA・LABEL・FUNARG の処理ができなくなってしまう．次の章でその解決方法を述べる．

3. YAL による APPLY 機能の代替処理

図 1 の apply の機能を，①㊁㊂の三つの部分に分けて説明を行う．

3.1 ㊁の部分の代替処理

㊁の部分は，LAMBDA・LABEL・FUNARG の機能をユーザが定義することにより行われる．

最も単純な定義は，**図 3** である．

これらの定義は，LAMBDA と #LAMBDA, LABEL と #LABEL, FUNARG と #FUNARG が対になっており，YAL のなかで，関数 LAMBDA を評価したすぐ後に，関数 #LAMBDA が評価されるようになっている(他も同様)．また，#VARLIST・#BODY・#NAME・#FNC・#ALIST は，大域変数であり，関数 LAMBDA から関数 #LAMBDA に値を渡す役目をしている．これらの大域変数をユーザが他の目的に使用していてはならない．また，関数 CSET は，大域変数への代入のための関数である．

LAMBDA の場合の処理過程を次に示す．

((LAMBDA(X Y)(DIFFERENCE X Y)) ▽9 ▽1)

を実行すると

#VARLIST ← (X Y)

```
(DF LAMBDA (S A) (PROG () (CSET '#VARLIST (CAR S))
                          (CSET '#BODY (CAR (CDR S)))
                          (RETURN '#LAMBDA)]
(DF #LAMBDA (S A) (EVAL #BODY (PAIRASS #VARLIST (EVLIS S A) A)]

(DF LABEL (S A) (PROG () (CSET '#NAME (CAR S))
                         (CSET '#BODY (CAR (CDR S)))
                         (RETURN '#LABEL)]
(DF #LABEL (S A) (EVAL (CONS #BODY S) (CONS (CONS #NAME #BODY) A)]

(DF FUNCTION (S A) (LIST 'FUNARG (CAR S) A)]
(DF FUNARG (S A) (PROG () (CSET '#FNC (CAR S))
                          (CSET '#ALIST (CAR·(CDR S)))
                          (RETURN '#FUNARG)]
(DF #FUNARG (S A) (EVAL (CONS #FNC S) #ALIST)]

(DE EVLIS (X A)
 (COND (X (CONS (EVAL (CAR X) A) (EVLIS (CDR X) A)))
       (T NIL)]

(DE PAIRASS (FP AP A)
 (COND (FP (CONS (CONS (CAR FP) (CAR AP)) (PAIRASS (CDR FP) (CDR AP) A)))
       (T  A)]
```

図 3 LAMBDA, LABEL, FUNARG の定義
Fig. 3 Definitions of LAMBDA, LABEL, FUNARG.

```
([LABEL F (LAMBDA (X) (COND ((ZEROP X) 1) (T (TIMES X (F (SUB1 X)] (F 3))
```

(a)

```
(DE P (X Y Z) (X Y))
(DE Q (X) (LIST X Z))
(DE R (Z) (P (FUNCTION Q) 'YYY 'ZZZ))
(R 123)
```

(b)

図 4 プログラム例
Fig. 4 Example of programs.

```
(DF #LABEL (S A) (EVALREV (CONS (CONS #NAME #BODY) A) (CONS #BODY (QELIS S A)))]

(DF #FUNARG (S A) (EVALREV #ALIST (CONS #FNC (QELIS S A)))]

(DE EVALREV (A X) (EVAL X A)]

(DE QELIS (X A)
 (COND (X (CONS (LIST 'QUOTE (EVAL (CAR X) A)) (QELIS (CDR X) A)))
       (T NIL)]
```

図 5 LABEL, FUNARG の定義の修正
Fig. 5 Revised definitions of LABEL, FUNARG.

#BODY ←(DIFFERENCE X Y)

の代入が起こり，もとのS式は，次のS式に形を変える．

(#LAMBDA ▽9 ▽1)

次にこのS式が評価されることにより，そのなかで，

(EVAL #BODY (PAIRASS #VARLIST
(EVLIS S A) A))

が評価されることになる．#BODY と #VARLIST の値は，上の代入の値が残っているので，これは結局，

(((X. 9) (Y. 1)). Aの値)

の環境のもとで，

(DIFFERENCE X Y)

を EVAL（評価）することになり，正しく値8が求まる．

図3の定義により，LAMBDA 機能は完全に働くのであるが，LABEL と FUNARG に関しては問題が残る．それは，図3による LABEL, FUNARG 機能では，実引数を評価する環境が，LISP 1.5 と異なるために起こる．たとえば，**図4**(a)のプログラムは，([LABEL F…] (F 3)) の階乗の計算を正しく行ってしまう．LISP 1.5 では，(F 3) の計算はできないのが本当である．なぜなら，LISP 1.5 では，(F 3) の評価を行った後で，[LABEL F…] の評価が行われるからである．また図4(b)のプログラムは，(YYY 123) を値とするはずが，Y の値が ALIST に存在しないというエラーになってしまう．

これらは，図3の関数 #LABEL·#FUNARG を，**図5**のように置きかえることで解決される．図5の中

の QELIS は，実引数であるリストの各要素を評価して，それを QUOTE したリストを値とする．たとえば，次のようになる．

(QELIS ᵛ((CAR ᵛ(A B)) (CDR ᵛ(C D))) NIL)
⇒(ᵛA ᵛ(D))

この関数 QELIS を用いることが可能な背景には，「万能関数 APPLY の LABEL･FUNARG で処理される関数の属性は，FEXPR や FSUBR ではなく，EXPR や SUBR である」ということがある．

関数 EVALREV は，関数 EVAL の引数の順番を一時的に，逆転させるためのものである．これは，関数 QELIS の実行により，#NAME や #BODY や #ALIST の値が，破壊される前にその値を取出すためのものである．

以上のようにして，LAMBDA･LABEL･FUNARG の機能が処理されるのであるが，「図3や図5のようなユーザ定義は，必要な場合にのみ行えばよい」というのが，YAL の大きな特徴である．実際，多くの問題で，図3や図5のような定義は，必要ないと思われる．

3.2 ㋒の部分の代替処理

㋒の部分は，S式が，

((COND ((P X) ADD1) (T SUB1)) XXX)

のようなときに必要になる．これは，YAL の☆印の部分で処理される．ただし，LISP 1.5 と異なり，YAL では (COND ((P X) ADD1) (T SUB1)) の部分が評価された後で，XXX の評価が行われるので，COND 式の評価により，XXX の値が変わらないことが望ましい．すなわち，関数呼び出し(P X)により，大域変数の値が変更され，その結果，XXX の値が変わってしまう場合は，LISP 1.5 との互換性はなくなってしまう．しかし，そのようなことは非常に少ないと考えられる．

3.3 ㋑の部分の代替処理

㋑の部分は，apply の何度かの再帰的な呼び出し(㋥㋒による）により，万能関数 apply の引数 fn が，ATOM になったときに処理される部分である．この「apply の何度かの再帰的な呼び出し」はすべて，「eval の何度かの再帰的な呼び出し」として処理される．また，この㋑の部分に対応する処理は，YAL の図2㋐の部分で処理される．

4. YAL の特徴

YAL は，LISP インタプリタの変造物であるが，次のような特徴をもつ．

① 万能関数 APPLY の除去により，LISP インタプリタの核部分が，従来のものより小さくなった．

② LAMBDA･LABEL･FUNARG 機能が，LISP インタプリタの核部分から分離できた．

③ プログラムの流れが，左から右へと移る原則が，関数部分にも拡張された．

④ LISP 1.5 では許されなかった記述が，自然な拡張として可能になった．

以下に，各項目について説明を行う．

4.1 インタプリタの核部分の小型化

LISP インタプリタは，いままでも，万能関数 EVAL と APPLY により，インタプリタの核部分が簡単に記述できるという長所があった．しかし，①この YAL では，APPLY を除去することにより，インタプリタの核部分をさらに小さくして単純なものにすることができた．これにより，LISP の処理系の理解が容易になる．とくに，教材として LISP 処理系を扱う場合に，効果があると思われる．②図1の㋑の部分は，実際のインプリメントでは，万能関数 eval の㋺の部分と意味的に重複することが多く，改良やメンテナンスの際に，二重の負担となることがあったが，万能関数 apply の除去によりそれが解消された．③インタプリタの核部分が小さくなることにより，この部分のマイクロプログラム化が容易になり，汎用目的のミニコン等でも，この部分を一つの機械語として実現することが容易になると考えられる．

4.2 付加機能の分離

万能関数 APPLY がなくなることにより，LAMBDA･LABEL･FUNARG 機能をユーザ定義の関数として，インタプリタの核部分と分離している．

LISP は，ラムダ計算にその基礎を置くといいながら，実際の処理系では，図1の＊印のように，直接処理してしまうのが普通である．すなわち，LAMBDA･LABEL･FUNARG 機能は，付加機能として，インタプリタの核部分と分離したほうが，それらの各意味を明確にすることができる．YAL はそれを可能にしている．

4.3 制御の流れの是正

「LISP を関数型言語としてとらえて，その引数は並列に計算されるべきである.」との主張もあるが，現実の LISP インタプリタは逐次処理により，引数を計算している．そして，その計算は，左側の引数から右側へと行われるのが普通である．しかし，従来の

LISP インタプリタでは，関数部分の評価は，引数の評価の後で行われる．たとえば，

((COND ((P X) F1) (T F2)) (G1 X) (G2 X) (G3 X))

なるS式を計算するとしよう．この式のなかの四つのS式は，(④①②③) の順に，各S式が評価され，最後に関数の計算が行われる．一方，YAL では，(①②③④) の順に各S式の評価が行われ，最後に関数の計算が行われる．YAL のほうが自然だと考えられる．

4.4 自然な拡張

LISP 1.5 では，S式の関数部分に計算式を置く場合，この計算式の値が，FEXPR, FSUBR 属性になることが許されなかった．これは，4.3節にも述べたように，S式の中の各S式の評価の順序が不自然なために起きている．例として，次のS式を考えよう．

((COND ((P X) 'PLUS) (T 'TIMES)) 3 5)

このS式は，LISP 1.5 ではエラーとなってしまう．一方，YAL では，プログラマの意図したとおり，3+5 または 3×5 の値を正しく計算する．このように，LISP 1.5 では禁止されていた記述を，YAL では自然な拡張として可能にしている．

5. 評　価

従来の LISP インタプリタ(図1)と YAL (図2)で同じ問題を計算させれば，YAL の性能評価を行うことができる．ただ，計算する問題をどのように記述するかによって，結果が大きく異なってくる．すなわち，LISP らしく LAMBDA 記法などを多用したプログラムでは，関数 LAMBDA, #LAMBDA などの実行が多くなり，そのために，YAL の実行速度は遅くなる．一方，DE, DF による関数定義だけを用いたプログラムで，LAMBDA, LABEL, FUNARG 機能をいっさい用いず，図2の☆印の部分の実行がない場合には，従来の LISP インタプリタとまったく同じ実行速度が得られることになる．実用的プログラムでは，後者に近い記述が多いと考えられるので YAL の性能は，従来の LISP インタプリタと同程度と考えられる．ただし，従来の LISP システムの SUBR 組み込み関数のなかで，万能関数 APPLY を使用して組み込み関数を実現しているものは問題である．万能関数 APPLY そのものがないのだから，機械語サブルーチンとしての APPLY を呼び出すことはできない．しかし，それらをユーザ定義とすることは，従来どおり可能である．関数 MAPCAR を例にとれば，

```
(DE MAPCAR (X FN)
  (COND ((NULL X) NIL)
    (T (CONS (FN (CAR X))
             (MAPCAR (CDR X) FN]
```

として定義できる．

当然，このような EXPR 属性の関数は，SUBR 属性の関数に比べれば，実行速度は遅くなる．これは，YAL の欠点である．このような関数を高速化するためには，結局，万能関数 APPLY を機械語で記述することが必要になる．しかし，その場合にも，インタプリタの核部分と，付加機能の部分を分離するということは，達成されているのである．

6. ま と め

新しい LISP インタプリタ YAL について論じた．今後は，実用的な問題を YAL に適用して，その性能を評価していくことが課題であろう．また，性能の問題は別にして，LISP 型のリスト処理系の教育用プロトタイプとして，YAL は，そのインタプリタの核部分の小ささに意義があると考えられる．

参 考 文 献

1) McCarthy, J. et al.: *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, Mass. (1962).
2) 中西正和：LISP 入門，p. 194, 近代科学社，東京 (1977).
3) 菱沼，山下，中西：LISP インタプリタにおけるスタック技法と a リストの抑制法，情報処理，Vol. 17, No. 11, pp. 1002-1008 (1976).
4) 黒川利明：LISP のデータ表現，情報処理，Vol. 17, No. 2, pp. 127-132 (1976).
5) 電子技術総合研究所：LISP 1.9 User's Manual (Mar. 1976).

(昭和59年4月23日受付)
(昭和60年1月17日採録)