LISPコンパイラとインタプリタの処理速度の理論的比較

伊藤貴康・田村 卓・和田慎一
（東北大学 工学部 通信工学科）

1．まえがき

リスト処理言語LISPは，1960年にMcCarthy他[1]によって実現されて以来，人工知能や記号処理の分野で広く利用されている．LISPの処理系は，通常，インタプリタ形式で実現され，処理系の高速化のためにコンパイラを備えていることが多い．LISP関数をコンパイルして実行するとインタプリタ実行に比べて10～100倍の高速処理が達成できることが，McCarthy他[1]によって指適されている．また，第2回LISPコンテストの結果[2]によると，同一システム上のインタプリタとコンパイラ・オブジェクトの実行速度比は数倍～30倍程度となっている．（この速度比の低下は，実用的なインタプリタの場合，多くのシステム関数が用意されていることによる．）また，マイクロプログラミング方式によるLISP処理系におけるLISP関数のマイクロ化LISPインタプリタとマイクロ化直接実行（コンパイラ・オブジェクト実行に相当）との速度比が20倍以上という結果も報告されている([4],[5],[6]).

本論文では，Pure LISPを対象として，LISPコンパイラとインタプリタの処理速度比較をスタック・マシンのモデルを設定して行ない，インタプリタ実行に比べて，コンパイラ・オブジェクトの実行の方が20倍～30倍高速であることを示す．また，高度の並列性を仮定したマシン・モデルに基く速度向上の可能性についても言及した．[なお，本論文の解析法は，特定のLISP関数の実行ステップ計算に文献[4]において用いたのと同様の考え方に基いていることを指適しておく．]

2．LISPコンパイラとインタプリタの処理速度の概念的比較と理論的解析の方針

2.1 処理速度の概念的比較

Pure LISPインタプリタの定義を図1に与えた．Pure LISPのインタプリタ関数evalquoteは，関数fnと引数リストxを入力として実行が開始され，関数適用applyと関数評価evalを互に呼び合いながら処理を行う．evalquoteの処理内容は，図1の破線のように分類できる．

一方，LISP関数のコンパイラ・オブジェクトの実行では，（既に文献[4]において説明したことであるが），以下の理由により実行の高速化が達成できる：

①構文解析の削減：コンパイル実行では，コンパイル時に関数や式の分類などの構文

```
evalquote[fn;x] <=
  apply[fn;x;NIL]

apply[fn;x;a] <=
  [atom[fn]                                   基本関数の処理
      -> [eq[fn;CAR]   -> caar[x];
          eq[fn;CDR]   -> cdar[x];
          eq[fn;CONS]  -> cons[car[x];cadr[x]];
          eq[fn;ATOM]  -> atom[car[x]];
          eq[fn;EQ]    -> eq[car[x];cadr[x]];
          T            -> apply[eval[fn;a];x;a]];
                          関数適用   関数の評価
  eq[car[fn];LAMBDA]   -> eval[caddr[fn];pairlis[cadr[fn];x;a]];
                          関数本体の評価        変数束縛
  eq[car[fn];LABEL]    -> apply[caddr[fn];x;
                          関数適用 cons[cons[cadr[fn];caddr[fn]];a]]
  関数タイプの分類                                変数束縛

eval[e;a] <=
  [atom[e]                    -> cdr[assoc[e;a]];  ----変数値の探索
   atom[car[e]]
      -> [eq[car[e];QUOTE]    -> cadr[e];  ---------- QUOTEの処理
          eq[car[e];COND]     -> evcon[cdr[e];a];----- CONDの処理
          T                   -> apply[car[e];evlis[cdr[e];a];a]];
   T                          -> apply[car[e];evlis[cdr[e];a];a]]
   式の分類                      関数適用          引数評価

assoc[x;a] <=
  [equal[caar[a];x] -> car[a];
   T                -> assoc[x;cdr[a]]]


evcon[c;a] <=
  [eval[caar[c];a] -> eval[cadar[c];a];
   T               -> evcon[cdr[c];a]]


evlis[m;a] <=
  [null[m] -> NIL;
   T       -> cons[eval[car[m];a];evlis[cdr[m];a]]]


pairlis[x;y;a] <=
  [null[x] -> a;
   T       -> cons[cons[car[x];car[y]];pairlis[cdr[x];cdr[y];a]]]
```

図1. Pure LISPインタプリタの定義

解析を行ってしまうので，オブジェクト・コード実行時における関数や式の分類などの構文解析が省略できる．

②変数束縛の削除：コンパイル実行では，値の受渡しをスタックやレジスタを用いて行うので，変数束縛の処理が不要となり，引数評価も簡単化される．

③関数適用および関数評価のオーバーヘッドの削減：a-list の変数値探索や中間結果の管理がコンパイル実行では不要になり，インタプリタにおける apply と eval の相互呼出しによるオーバーヘッドが削除される．

以上のような理由に基く，コンパイラ・オブジェクトの実行の高速性を理論的に示すのが本論文の目的である．

2.2 理論的解析の方針

処理速度の理論的な解析は，スタック・マシンのモデルを設定し，このマシン・モデルによる実行ステップを比較することによって行う．本論文の議論は次のように進められる．

(1) スタック・マシン・モデルの説明（スタック・マシン命令とその実行ステップの設定）

(2) Pure LISP に対するコンパイラ・オブジェクトコード生成規則の提示

(3) Pure LISP インタプリタの実行ステップ計算式の提示

(4) コンパイラ・オブジェクト実行ステップとインタプリタ実行ステップの理論的比較 [最悪時で15倍以上，実際的な観点からは，19倍～30倍の高速性が達成できることが示される．]

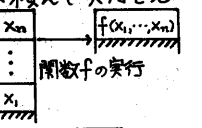
(5) Pure LISP 関数のコンパイラによるオブジェクトコード生成ステップの計算式とその上限値の計算

(6) 高度の並列性を持つマシン・モデルによる処理速度向上の可能性の検討 [理想的なデータフローマシンあるいは関数型並列マシンのモデルによる LISP 関数の処理の解析の1例と考えてよい．]

3．スタックマシンモデルとその命令

スタックマシンのモデルを Pure LISP の処理を考えて設定した．その命令とスタックの動きを図2に示した．S式データをスタック上で処理できるようにモデルを構成した．各命令の実行ステップの値は，文献[4,5,6]の経験を基に設定した．CONS は実行ステップが大きいので K としたが，文献[4,5,6]の経験では K=3 と考えてよい．(ただし，ガーベージコレクションは無いとしている．) 以降の具体的な実行ステップ評価にはこれらの数値を用いている．

スタックマシンにおいては，関数の実行に際し，スタックトップから連続した領域に置かれた引数に対して計算を行ない，答が求まったら引数はスタックから取り除き，結果をスタックに積んで実行を完了するとしている[右図参照]．例えば (LAMBDA (X Y)(CAR(CONS (CDR X) Y))) は，図3[次頁]の命令系列で実行できる．



スタック・マシン命令	命令の意味	実行ステップ(値)
〈基本操作〉		
CAR	car[S[1]] を S[1] に書き込む．	scar (1)
CDR	cdr[S[1]] を S[1] に書き込む．	scdr (1)
CONS	cons[S[2];S[1]]を，スタックの内容を1つPOP UPしてから S[1] に書き込む．	scons (K)
ATOM	atom[S[1]]の値を S[1] に書き込む．	satom (1)
EQ	eq[S[2];S[1]]の値を，スタックの内容を1つPOP UPしてから S[1] に書き込む．	seq (1)
〈その他のスタック操作〉		
PUSHC c	定数 c を PUSH する．	sconst (1)
PUSHS i	S[i] を PUSH する．	svar (1)
CUT i	S[2]…S[i+1] をスタックから取り去る．	scut (1)
〈分岐命令〉		
GOTO l	ラベル l の命令に ジャンプする．	sgoto (1)
JUMPF l	スタック内容を1つ POP UP し，その値が NIL のときにはラベル l の命令に分岐する．	sjump (1)
CALL s	サブルーチン s に ジャンプする．	scall (1)
RETURN	サブルーチンから元に戻る．	sret (1)

注)① S[i] はスタックの i 番目の内容を表わす．
② サブルーチンの返りアドレスは別に用意されたスタックに自動的に PUSH されると仮定している．
③ scar, scdr などは各命令の実行ステップであり，括弧内の値は本論文でステップ評価を用いるのに仮定した値を示す．

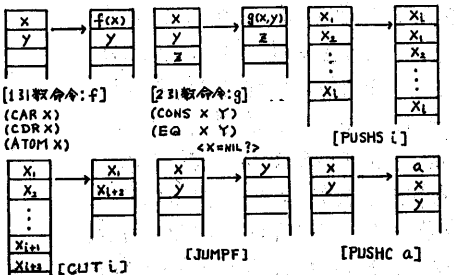


図2．スタックマシンモデルの命令語とスタック動作

	スタックの内容 1	2	3	4	5	実行ステップ数
	b	a	#		…	
PUSHS 2	a	b	a	#	…	svar(1)
CDR	*cdr	b	a	#	…	scdr(1)
PUSHS 2	b	*cdr	b	a	#	svar(1)
CONS	*cons	b	a	#	…	scons(k)
CAR	*car	b	a	#	…	scar(1)
CUT 2	*car	#			…	scut(1)
(LAMBDA(X Y) (CAR(CONS(CDR X) Y)) の 実行ステップ合計						5+k

[注]*cdr=cdr[a], *cons=cons[cdr[a]; b]
*car=car[cons[cdr[a]; b]]

図3. スタック命令プログラムの例

```
**********  compile  ************************
Function          I
  Definition      I          +----------------+
                  I     Fn:  I compexp[Body] I
(DE Fn            I          +----------------+
  (X1 ... Xn)     I             CUT     n
     Body    )    I             RETURN
**********  compexp  ************************
Constant          I
   T              I             PUSHC   T
   NIL            I             PUSHC   NIL
 (QUOTE Exp)      I             PUSHC   Exp
------------------I-------------------------------
Variable          I
   X              I             PUSHS  depth[X]
------------------I-------------------------------
Function          I          +----------------+
  Application     I          I compexp[E1]    I
                  I          +----------------+
(Fn E1 ... En)    I                 *
                  I                 *
                  I                 *
                  I          +----------------+
                  I          I compexp[En]    I
                  I          +----------------+
                  I          +----------------+
                  I          I compapply[Fn] I
                  I          +----------------+
------------------I-------------------------------
Conditonal        I          +----------------+
  Expression      I          I compexp[P1]    I
                  I          +----------------+
(COND (P1 E1)     I             JUMPF label 1
      (P2 E2)     I          +----------------+
        *         I          I compexp[E1]    I
        *         I          +----------------+
        *         I             GOTO    exit
      (Fn En))    I          +----------------+
                  I label 1: I compexp[P2]    I
                  I          +----------------+
                  I                 *
                  I                 *
                  I                 *
                  I          +----------------+
                  Ilabel n-1:I compexp[Fn]    I
                  I          +----------------+
                  I             JUMPF  label n
                  I          +----------------+
                  I          I compexp[En]    I
                  I          +----------------+
                  I             GOTO exit
                  I   label n:   PUSH   NIL
                  I      exit: (next code)
********** compapply ************************
Basic Function    I
    CAR           I                CAR
    CDR           I                CDR
    CONS          I                CONS
    ATOM          I                ATOM
    EQ            I                EQ
    NULL          I                NULL
------------------I-------------------------------
Defined           I
  Function: Fn    I                CALL    Fn
------------------I-------------------------------
Lambda            I          +----------------+
  Function        I          I compexp[Body] I
(LAMBDA           I          +----------------+
  (X1 ... Xn)     I             CUT     n
     Body    )    I
------------------I-------------------------------
Label Function    I             GOTO   skip
(LABEL  Fn        I +-----------------------------+
  (LAMBDA         I I compile[(DE Fn Args Body)] I
    Args          I +-----------------------------+
    Body  ))      I       skip:    CALL   Fn
------------------I-------------------------------
```

図4. Pure LISPに対するコード生成規則

4. Pure LISPコンパイラとそのコード生成規則

Pure LISP関数をスタック言語プログラムに変換するコンパイラのコード生成規則について説明する. 図4にコード生成規則を与えた. 生成規則は関数定義をコンパイルするcompile; 式をコンパイルするcompexp; 関数適用をコンパイルするcompapplyの3つの部分からなる. 各部分で, 次のような仕方でオブジェクトを生成する.

(1) compile: 関数本体の値を計算して(compexp[Body])スタックに積まれている引数を取り除いて(CUT n)呼び出した所へ戻る(RETURN)

(2) compexp: 式の値を計算してスタックに積む. 関数適用の場合には式の引数を順にスタックに積んで行き, その関数の計算を行なう(compapply[Fn]).

(3) compapply: スタックに積まれた引数を参照して計算を行ない, 引数の分はスタックから取り除いて(CUT n)結果はスタックトップに積んだ状態で終了する. (LABEL関数の場合には, 関数(Fn)の定義とみなして対応するオブジェクトを生成し, 更にそれを呼び出す形のオブジェクトを生成する.)

[注1: このコード生成規則に基くコンパイラのLISPプログラムが第7節に与えられているから参照されたい.]

[注2: 図4でdepth[x]は, コード生成を適用していくにあたってスタックの深さの変化を記録していくことにより求めることができる.]

以上のコード生成規則から知られるように, この規則では関数引数, 計算関数, 自由変数の処理を扱えない. またLABEL関数の内部では, その関数自身の引数以外の変数は扱えない[すなわちname conflictがないと仮定]. すなわち, 本論文の処理速度比較は, このような制約の下に行っていることに注意せよ. また, 次節で述べるように, Pure LISPインタプリタの実行ステップ計算もこのコード生成規則を基に行っている.

5. Pure LISPインタプリタの実行ステップ計算法

Pure LISPインタプリタの定義を図1に与えたが, このインタプリタは, 図4のコード生成規則を用いてスタック命令のプログラムに変換されたのちに実行されるものと考え, その実行ステップ数もスタックマシンの実行ステップで計算するものとする. 例えば, evalのステップ数を与えるsevalの一部は次のように表わされる.

seval[e,a] = scut + sret ……実行終了時のステップ
+[atom[e]→(satom+svar)) ……atom[e]のステップ
+(scar+svar+svar ……cdr[…]とe, aのステップ
+scall+sassoc[e;a]……呼び出しとassocのステップ

Pure LISPインタプリタの実行ステップ計算式を図5に与えた．また図2に従ってステップ値を計算した結果を図6に与えた．図6を用いると，例えば，

eval[(CAR (QUOTE (A B)));(X.1)]

の実行ステップ数は次のように求まる．

seval[(CAR (QUOTE (A B)));(X.1)]
=31+sevlis[((QUOTE (A B)));(X.1)]
+sapply[CAR;((A B));(X.1)]
=31+(16+K+sevlis[NIL;(X.1)])+14
=31+(16+K+7)+14
=68+K

[注：以下の実行ステップの評価K当ってはevalトップのインタプリタを想定し，sevalを用いて解析を行っているから注意せよ．]

```
sevalquote[fn;x] = scut+sret
   +(scall+svar+svar+sconst+sapply[fn;x;NIL])


sapply[fn;x;a] = scut+sret
   +[atom[[fn]  -> (satom+svar+sjump)
     + [eq[fn;CAR] -> (seq+svar+sconst+sjump)
                      +(scar+scar+svar+sgoto);

        eq[fn;CDR] -> (seq+svar+sconst+sjump)
                      +(seq+svar+sconst+sjump)
                      +(scdr+svar+svar+sgoto);

        eq[fn;CONS] -> (seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(scons+scar+svar+scar
                         +sgoto)                    ;

        eq[fn;ATOM] -> (seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(satom+scar+svar+sgoto);

        eq[fn;EQ] ->   (seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(seq+scar+svar+scar
                         +scdr+svar+sgoto)      ;
        T            -> (seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(seq+svar+sconst+sjump)
                       +(scall+scall+svar+svar
                         +svar+svar
                         +sapply[eval[fn;a];x;a]
                         +seval[fn;a]+sgoto)    ]

     +sgoto                                     ;

   eq[car[fn];LAMBDA]
    -> (satom+svar+sjump)
      +(seq+scar+svar+sconst+sjump)
      +(scall+scar+scdr+scdr+svar+svar+svar
      +seval[caddr[fn];pairlis[cadr[fn];x;a]]
      +spairlis[cadr[fn];x;a]+sgoto);

   eq[car[fn];LABEL]  -> (satom+svar+sjump)
    -> (satom+svar+sjump)
      +(seq+scar+svar+sconst+sjump)
      +(seq+scar+svar+sconst+sjump)
      +(scall+scar+scdr+svar+svar+scons+scons
       +scar+scdr+svar+scar+scdr+scdr+svar
       +svar+sapply[caddr[fn];x;
            cons[cons[cadr[fn];caddr[fn]];a]]
         +sgoto)                                ]
```

図5　Pure LISPインタプリタの実行ステップ計算式

6．コンパイラ実行とインタプリタ実行の速度比較

図7K LISP関数に対して

①図4のコード生成規則によって生成したオブジェクトプログラムの実行ステップ sobject，および

②図6のインタプリタ実行ステップ数計算式によって求めた実行ステップ

の比較を与えた．対応するsobjectとsevalの式の右辺で引数が対応するsobjectとsevalを除いた部分を比較すると，K≦7.83の時，

『sobject×15≦seval』となっている．なお，関数適用に関しては，インタプリタでは関数適用のオーバーヘッドを併せて比較している事に留意せよ．

```
seval[e;a] = scut+sret

 + [atom[e]    ->  (satom+svar+sjump)
                   +(scdr+scall+svar+svar+sassoc[e;a]+sgoto);

    atom[car[e]] -> (satom+svar+sjump)+(scar+scdr+svar+sgoto)

     + [eq[car[e];QUOTE] -> (seq+scar+svar+sconst+sjump)
                            +(scar+scdr+svar+sgoto);

        eq[car[e];COND]  -> (seq+scar+svar+scons+sjump)
                            +(scar+scar+svar+sconst+sjump)
                            +(scall+scdr+svar+svar
                              +sevcon[cdr[e];a]+sgoto);

        T                -> (seq+scar+svar+sconst+sjump)
                            +(seq+scar+svar+sconst+sjump)
                            +(sconst+sjump) + (scall+scar+svar
                            +scall+scdr+svar+svar+svar
                            +sapply[car[e];evlis[cdr[e];a]]
                            +sevlis[cdr[e];a]+sgoto)]
      + sgoto                                           ;

    T          -> (satom+svar+sjump)+(satom+scar+svar+sjump)
                  +(sconst+sjump) + (scall+scar+svar+scall+scdr
                    +svar+svar+svar+sapply[car[e];evlis[cdr[e];a]]
                    +sevlis[cdr[e];a]+sgoto)                    ]

sassoc[e;a] = scut+sret

  +[eq[caar[a];e]  -> (seq+scar+scar+svar+svar+sjump)
                      +(scar+svar+sgoto);

    T          -> (seq+scar+scar+svar+svar+sjump)
                  +(sconst+sjump)
                  +(scall+svar+scdr+svar+sassoc[cdr[e];a]+sgoto)]


sevcon[c;a] = scut+sret

  +[eval[caar[c];a]
        -> (scall+scar+scar+svar+svar+seval[caar[c];a]+sjump)
          +(scall+sar+scdr+scar+svar++svar+seval[cadar[c];a]
            +sgoto)                                         ;

    T    -> (scall+scar+scar+svar+svar+seval[caar[c];a]+sjump)
          +(sconst+sjump)
          +(scall+scdr+svar+svar+sevcon[cdr[c];a]+sgoto)    ]


sevlis[m;a] = scut+sret

  +[null[m]  -> (snull+svar+sjump)+(scons+sgoto)            ;

    T        -> (snull+svar+sjump)+scons+sjump)
                +(scons+scall+scar+svar+svar+scall+scdr+svar
                  +svar+scall+scdr+svar+svar+seval[car[m];a]
                  +sevlis[cdr[n];a]+sgoto)                  ]


spairlis[x;y;a] = scut+sret

  +[null[x]  ->  (snull+svar+sjump)+(svar+sgoto)            ;

    T        ->  (snull+svar+sjump)+(svar+sjump)
                +(scons+scons+scar+evar+scar+svar
                  +scall+scdr+svar+scdr+svar+svar
                   +spairlis[cdr[x];cdr[y];a]+sgoto)        ]
```

```
sevalquote[fn;x]  =    6 +sapply[fn;x;NIL]

sapply[fn;x;a] =
  [atom[fn] ->
    [eq[fn;CAR]   -> 14;
     eq[fn;CDR]   -> 18;
     eq[fn;CONS]  -> 24+K;
     eq[fn;ATOM]  -> 26;
     eq[fn;EQ]    -> 33;
     T            -> 35 +seval[fn;a]
                        +sapply[eval[fn;a];x;a]];
   eq[car[fn];LAMBDA]
                  -> 22 +spairlis[cadr[fn];x;a]
                        +seval[caddr[fn];
                               pairlis[cadr[fn];x;a]];
   eq[car[fn];LABEL]
                  -> 30+2K
                        +sapply[caddr[fn];
                                x;
                                cons[cons[cadr[fn];
                                          caddr[fn]];
                                     a]]]

seval[e;a] =
  [atom[e]             -> 10 +sassoc[e;a];
   atom[car[e]] ->
    [eq[car[e];QUOTE] -> 19;
     eq[car[e];COND]  -> 25 +sevcon[cdr[e];a];
     T                -> 31 +sevlis[cdr[e];a]
                            +sapply[car[e];
                                    evlis[cdr[e];a];
                                    a];
   T                  -> 20 +sevlis[cdr[e];a]
                            +sapply[car[e];
                                    evlis[cdr[e];a];
                                    a]]

sassoc[e;a] =
  [eq[caar[a];e]     -> 11;
   T                 -> 15 +sassoc[e;cdr[a]]]

sevcon[c;a] =
  [eval[caar[c];a]   -> 15 +seval[caar[c];a]
                           +seval[cadar[c];a];
   T                 -> 15 +seval[caar[c];a]
                           +sevcon[cdr[e];a]]

sevlis[m;;a] =
  [null[m]       -> 7;
   T             -> 16+K +seval[car[m];a]

spairlis[x;y;a] =
  [null[x]       -> 7;
   T             -> 18+2K +spairlis[cdr[x];cdr[y];a]]
```

図6 Pure LISP インタプリタの実行ステップ数

定数: (QUOTE e)

Compiler	sobject[(QUOTE e)] = sconst = 1
Interpreter	seval [(QUOTE e); a] = 19

局所変数: x

Compiler	sobject[x] = svar = 1
Interpreter	seval[x; a] = 10+sassoc[x; a] ≥ 10+11 = 21

基本関数

CAR: (CAR e)

Compiler
sobject[(CAR e)] = scar + sobject[e] = 1 + sobject[e]

Interpreter
seval[(CAR e); a]
= 31 + sapply[CAR; (V); a] + sevlis[(e); a]
= 31 + 14 + (16 + K + seval[e; a] + sevlis[NIL; a])
= 31 + 14 + 16 + K + seval[e; a] + 7
= 68 + K + seval[e; a]

CDR: (CDR e)

Compiler
sobject[(CDR e)] = scdr + sobject[e] = 1 + sobject[e]

Interpreter
seval[(CDR e); a]
= 31 + sapply[CDR; (V); a] + sevlis[(e); a]
= 72 + K + seval[e; a]

CONS: (CONS e1 e2)

Compiler
sobject[(CONS e1 e2)] = scons + sobject[e1] + sobject[e2]
= K + sobject[e1] + sobject[e2]

Interpreter
seval[(CONS e e2); a]
= 31 + sapply[CONS; (V1 V2); a] + sevlis[(e1 e2); a]
= 31 + 24 + K + (16 + K + seval[e1; a] + sevlis[(e2); a])
= 31 + 24 + K + 16 + K + seval[e1; a] + (16 + K + seval[e2; a] + sevlis[NIL; a])
= 94 + 3K + seval[e1; a] + seval[e2; a]

ATOM: (ATOM e)

Compiler
sobject[(ATOM e)] = satom + sobject[e] = 1 + sobject[e]

Interpreter
seval[(ATOM e); a]
= 31 + sapply[ATOM; (V); a] + sevlis[(e); a]
= 80 + K + seval[e; a]

EQ: (EQ e1 e2)

Compiler
sobject[(EQ e1 e2)] = seq + sobject[e1] + sobject[e2]
= 1 + sobject[e1] + sobject[e2]

Interpreter
seval[(EQ e1 e2); a]
= 31 + sapply[EQ; (V1 V2); a] + sevlis[(e1 e2); a]
= 103 + 2K + seval[e1; a] + seval[e2; a]

関数の適用: (f e1 ··· en)

Compiler
sobject[(f e1 ··· en)]
= sobject[e1] + ··· + sobject[en] + sobj-apply[f]

Interpreter
seval[(f e1 ··· en); a]
= 31 + sevlis[(e1 ··· en); a] + sapply[f; (V1 ··· Vn); a]
= (16+K)n + 38 + seval[e1; a] + ··· + seval[en; a] + sapply[f; (V1 ··· Vn); a]

(∵) sevlis[(e1 ··· en); a] = (16 + K + seval[e1; a]) + sevlis[(e2 ··· en); a]
= (16+K+seval[e1; a]) + (16+K+seval[e2; a]) + sevlis[(e3 ··· en); a]
⋮
= (16+K+seval[e1; a]) + ··· + (16+K+seval[en; a]) + sevlis[NIL; a]
= (16+K)n + 7 + seval[e1; a] + ··· + seval[en; a]

関数適用の処理

束縛される関数: (DE f (x1 ··· xn) body)

Compiler
sobj-apply[f] = scall + sobject[body] + scut + sreturn = 3 + sobject[body]

Interpreter
sapply[f; (V1 ··· Vn); a]
= 35 + sapply[(LAMBDA (x1 ··· xn) body); (V1 ··· Vn); a] + seval[f; a]
= 35 + (22 + seval[body; ((x1.V1) ··· (xn.Vn)). a])
+ spairlis[(x1 ··· xn); (V1 ··· Vn); a]) + (10 + sassoc[f; a])
= 67 + spairlis[(x1 ··· xn); (V1 ··· Vn); a] + sassoc[f; a] + seval[body; ((x1.V1) ··· (xn.Vn)). a]

LAMBDA関数: (LAMBDA (x1 ··· xn) body)

Compiler
sobj-apply[(LAMBDA (x1 ··· xn) body)] = sobject[body] + scut = 1 + sobject[body]

Interpreter

LABEL関数: (LABEL f (LAMBDA (x1 ··· xn) body))

Compiler
sobj-apply[(LABEL f (LAMBDA (x1 ··· xn) body))]
= sgoto + sobj-apply[f] = 4 + sobject[body]

Interpreter
sapply[(LABEL f (LAMBDA (x1 ··· xn) body)); (V1 ··· Vn); a]
= 30 + 2K + sapply[(LAMBDA (x1 ··· xn) body); (V1 ··· Vn); ((f. (LAMBDA (x1 ··· xn) body)). a)])
= 52 + 2K + spairlis[(x1 ··· xn); (V1 ··· Vn); ((f. (LAMBDA (x1 ··· xn) body)). a)]
+ seval[body; ((x1.V1) ··· (xn.Vn)) (f. (LAMBDA (x1 ··· xn) body)). a)]

条件式: (COND (p1 e1) ··· (pn en))

≪仮定≫ p1, ···, pi-1 が不成立で, pi が成立とする.

Compiler
sobject[(COND (p1 e1) ··· (pn en))]
= (sobject[p1] + sjump) + ··· + (sobject[pi-1] + sjump) + (sobject[pi] + sjump) + sobject[ei] + sgoto
= i+1 + sobject[p1] + ··· + sobject[pi] + sobject[ei]

Interpreter
seval[(COND (p1 e1) ··· (pn en)); a]
= 25 + sevcon[((p1. e1) ··· (pn. en)); a]
= 25 + (15 + seval[p1; a]) + sevcon[((p2 e2) ··· (pn en)); a]
= 25 + (15 + seval[p1; a]) + ··· + (15 + seval[pi-1; a]) + sevcon[((pi ei) ··· (pn en)); a]
= 25 + 15(i-1) + (seval[p1; a] + ··· + seval[pi-1; a]) + 15 + seval[pi; a] + seval[ei; a]
= 15i + 25 + seval[p1, a] + ··· + seval[pi; a] + seval[ei; a]

[注] seval 以外の関数を帰納法が適用できる形に必要に応じて展開した.

図7. LISP関数に対するコンパイラ・オブジェクトとインタプリタの実行ステップの比較

構造的帰納法Kより次の事が言える：

《最悪値評価》『Pure LISPコンパイラのオブジェクトの実行速度はインタプリタ実行より15倍以上高速である。』

[注：15倍になるのは，条件式においてしが無限大となった時である。なお，K≧7.83のときはCONSの比率が15以下となるが，第3章において述べたように実際的な観点からはK=3～5と考えて良いからCONSの比率は20倍以上となる。]

《実際的な最悪値評価》

上述の議論で最悪値を与える条件式について検討してみる。p_iが更に条件式の形になっていることがなく、consのステップ値Kも5以下とすると，次の事が言える。

『Pure LISPコンパイラのオブジェクトの実行速度はインタプリタ実行に比べて19倍以上高速である。』

この理由を簡単に説明しておこう。

(COND $(p_1\ e_1)\cdots(p_n\ e_n)$)において$p_j$ $(1\leq j\leq n)$が条件式でないと仮定する。このとき、$p_1,\cdots p_{i-1}$に包まれてNILを返すような式のなかで、インタプリタとコンパイルコードのステップ比が19に近い値をとるのは、a-listの最左端にある変数(1:21)、定数(QUOTE NIL)の場合のみである。(QUOTE NIL)は意味をもたないので考えないものとし、変数もたかだか1回しか出現しないとして妥当である。他の場合は$seval[p_j;a]$ $(1\leq j\leq i-1)$の値が$sobject[p_j]$の値の19倍より余裕(+4以上)をもつことが図7よりわかるので、この関係を図7の条件式の部分に代入することにより証明ができる。

$$\begin{aligned}&seval[p_1;a]+\cdots+seval[p_i;a]\\&\geq(19\cdot sobject[p_1]+4)+\cdots+(19\cdot sobject[p_i]+4)\\&=4i+19\cdot sobject[p_1]+\cdots+19\cdot sobject[p_i]\end{aligned}$$

7. コンパイルコスト

4節で、Pure LISPに対するスタック言語のオブジェクトコード生成規則を与えたが、その規則に従ったコード生成を行うコンパイラのLISPによる記述例を図8に与えた。関数compileは、関数名fn，引数リストvars，関数本体の式bodyを入力として、スタック言語のオブジェクトコードを生成する。bodyのコンパイルでは，prupによって各仮引数の位置を記録し，countによって仮引数の数を数えてスタックの深さを計算した後、compexpを呼ぶ。各部分のコンパイルの際には、それぞれ、vpl，depthという変数によって引き渡され，局所変数値を取り出すときのオフセット計算の際に用いられる。

コンパイルコストの計算は，インタプリタの実行ステップ数計算と同様に行うことができる。ただし，コンパイラそのものは，ターゲットマシンの機械語でコーディングされてよい訳であるから、その事を考慮して、コンパイラ中に現われる関数のいくつかに対して，次のようなコスト評価を与えた事に注意せよ。

list … 1（最も外側の出現の場合は0），
append … 引数の個数，count …リストの要素の数，
assoc … 必要な等価判定の数，
gensym, numberp … 1

図8に与えたコンパイラは、リスト構造の形でオブジェクトコードを生成するが，コードを直接メモリ上に書き込むものと考えれば，list，appendのコストは小さく見積ってよい。

[コンパイルコストの評価]

プログラムを構成する要素を次のように考えてコンパイルコストを求める。

- 関数定義の仮引数の数：n
- すべての仮引数の数：n_v（nを含む）
- 本体に現われる定数・変数の数：n'
- CONDの数：n_c，条件式の最大長：l_c
- 関数アトムの数：n_f，取り得る引数の最大数：m_f
- LAMBDAの数：n_λ，取り得る引数の最大数：m_λ
- LABELの数：n_ℓ，取り得る引数の最大数：m_ℓ

$$\begin{aligned}&scompile[f;(x_1\cdots x_n);body]\\&\quad=26+(19+2K)n\\&\qquad+scompexp[body;((x_n.n-1)\cdots(x_1.0));n]\\&scompexp[body;((x_n.n-1)\cdots(x_1.0));n]\\&\quad\leq(26+n_v)\cdot n' &&\cdots\text{定数・変数}\\&\qquad+(32+43l_c)\cdot n_c &&\cdots\text{条件式}\\&\qquad+(74+21m_f)\cdot n_f &&\cdots\text{アトム関数の適用}\\&\qquad+(81+21m_\ell)\cdot n_\ell &&\cdots\text{label関数の適用}\\&\qquad+(83+(40+2K)\cdot m_\lambda)\cdot n_\lambda &&\cdots\lambda\text{関数の適用}\end{aligned}$$

なお、定数・変数のコンパイルコストは、関数定義の第1引数x_1のコンパイルコストが最大となるのでそのコストを用いて計算を行った。

```
(DE APPEND (X Y)
  (COND ((NULL X) Y)
        (T (CONS (CAR X) (APPEND (CDR X) Y)))))
```

で定義されるAPPENDの場合，K=3とすると，

[APPENDのコンパイルコスト] ≦ 942

```
compile[fn;vars;body] <= append[
      list[fn];compexp[body;prup[vars;0;NIL];count[vars]];
      list[list[CUT;count[vars]]];
      list[list[RETURN]]]

prup[vars;n;vpl] <=
 [null[vars] -> vpl;
          t -> cons[cons[car[vars];n];
                    prup[cdr[vars];add1[n];vpl]]]

compexp[exp;vpl;depth] <=
 [atom[exp] -> [null[exp]    -> list[list[PUSHC;NIL]];
                eq[exp;t]    -> list[list[PUSHC;T]];
                numberp[exp] -> list[list[PUSHC;exp]];
                t -> list[list[PUSHS;difference[depth;
                                  cdr[assoc[exp;vpl]]]]]];
  eq[car[exp];QUOTE] -> list[list[PUSHC;cadr[exp]]];
  eq[car[exp];COND] -> compcond[cdr[exp];vpl;
                                depth;gensym[]];
  t  -> compapply[car[exp];complist[cdr[exp];vpl;depth];
                  vpl;depth]]

compcond[u;vpl;depth;glob] <=
  [null[u] -> list[list[PUSHC;NIL];glob];
   t -> append[compclause[car[u];vpl;depth;gensym[];glob];
               compcond[cdr[u];vpl;depth;glob]]]

compclause[c;vpl;depth;loc;glob] <= append[
 compexp[car[c];vpl;depth];list[list[JUMPF;loc]];
 compexp[cadr[c];vpl;depth];
 list[list[GOTO;glob]];list[loc]]

compapply[fn;args;vpl;depth] <=
  [atom[fn]
     -> [eq[fn;CAR]  -> append[args;list[list[CAR]]];
         eq[fn;CDR]  -> append[args;list[list[CDR]]];
         eq[fn;CONS] -> append[args;list[list[CONS]]];
         eq[fn;ATOM] -> append[args;list[list[ATOM]]];
         eq[fn;EQ]   -> append[args;list[list[EQ]]];
         eq[fn;NULL] -> append[args;list[list[NULL]]];
         t -> append[args;list[list[CALL;fn]]]];
   eq[car[fn];LABEL]
     -> lambda[[g];append[args;
                          list[list[GOTO;g]];
                          compile[cadr[fn];
                                  cadaddr[fn];
                                  caddaddr[fn]];
                          list[g];
                          list[list[CALL;cadr[fn]]]]]
                [gensym[]];
   eq[car[fn];LAMBDA]
     -> append[args;
               compexp[caddr[fn];
                       prup[cadr[fn];depth;vpl];
                       plus[count[cadr[fn]];depth]]]
               list[list[CUT;count[cadr[fn]]]]]]

complist[m;vpl;depth] <=
  [null[m] -> NIL;
   t -> append[compexp[car[m];vpl;depth];
               complist[cdr[m];vpl;add1[depth]]]]
```

図8. Pure LISP コンパイラの記述

8. 並列LISPインタプリタ: parallel evalquote

Pure LISP関数を高度の並列性を持つマシンモデルによって実行し，高速化を行うことも考えられる。本節では，並列実行の要素をPure LISPインタプリタに導入した場合の実行ステップ数の評価を行う。[注：データフローマシンの抽象的モデルによる実行と考えてもよい。]

並列実行を行うPure LISPインタプリタ#evalquoteを図9に与えた。#evalquote, #apply, #evalは見かけ上通常のインタプリタと同じであるが，次のような動作を行うものと考える。

①タイプの分類は並列に行うものとし，名分岐にかかるコストに順番による差異は生じない。

②#eq, #cons, #conc2は2つの引数を並列に評価し，両方が求った段階でeq, cons, nconcを行う。

③#の付いたインタプリタ関数は，すべての引数を並列に評価し，但が求った段階で図9に与えた定義に従って評価を続ける。

④㊞の付いた関数は，引数のリストの各要素について並列に関数の適用を行う。

⑤㊞matchは第2引数のリストの各要素と，第1引数とのマッチングをとる。

⑥#scan-selectは，第1引数のリストの各要素のNILチェックを左から右へ繰り返し行い，初めてnon-NIL定数が得られた時点で，対応する位置にある第2引数のリストの要素の評価結果を取り出す。各要素はすべて並列に評価される。

以上のような動作を行う#evalquoteについて，実行ステップ数を計算したものを図10に与える。

タイプチェックにおける並列性と引数リストの評価における並列性を考慮し，㊞matchのステップ数を#m, #scan-selectのステップ数を#ssとおけば，evalquoteの場合と同様な手法で，実行ステップ数が計算できる。

インタプリタにおける並列実行によって，インタプリタとコンパイラの処理速度の差の大きな要因である①関数タイプや式の分類 ②変数束縛 ③a-listによる中間結果の管理等に要するステップ数が大幅に減少する。従って，parallel evalquoteによって逐次マシンモデルにおけるコンパイラ処理速度と同様なだけでなく関数評価のステップ数が重い関数群から構成されるプログラムの場合，並列実行によって逐次マシン上のコンパイラモードよりも高速性が期待できる事になる。また，コンパイラオブジェクト自身も並列実行を考えた場合でも逐次実行の場合ほどでは速度比が大きくならないことが予想される。

仮に#m=5, #ss=1（理想的には0）としてみると，sobjectとsevalのステップ数比較と同様の議論により，最悪の場合でもコンパイル実行の方が14倍速いことが言える。すなわち，高度の並列の導入は，インタプリタモードでの実行により効果的であることが予想される。

9. 結言

スタックマシンモデルを設定して、Pure LISPに対するコンパイラ・オブジェクト実行とインタプリタ実行の処理速度の理論的比較を実行ステップ数を基に行い、コンパイラ・オブジェクト実行がインタプリタ実行に比べ最悪値15倍以上、実際的観点からは19倍以上速いことを示した。また、コンパイルコストの評価式も与えた。

例えばappend[x;y]に対し、xの長さ10で評価を行うと次のようになる。(K=3とする)

インタプリタステップ	7,963	
コンパイラオブジェクト実行	157	(計)
コンパイルコストの上限値	942	1,099

実際のLISP関数の場合には20～30倍の速度向上は十分期待できると考えてよいであろう。

本論文の手法は、他のスタックマシンモデルの場合にも対応するステップを設定することによって適用できると考えられる。しかし、本論文ではオブジェクトコードの最適化を考えていない。レジスタで値の受け渡しを行い、マルチレジスタ方式のスタックマシンモデルを設定した場合にはオブジェクトコード最適化の効果が大きくなると考えられ、そのようなモデルでのevalquoteの解析は今後の課題である。[最適化の効果は全体としてインタプリタのコーディングに現われることが多い]

高速の並列性を導入したparallel evalquoteは理想的なデータ・フローマシンに基くLISPインタプリタの実現と考えてよいであろう。なお、parallel evalquoteとして

⊛evalquote[(f, x1); …;(fn xn)]

= ⟨#evalquote[f1; x1], …, #evalquote[fn; xn]⟩

を考えることができるが、n個のa-listの管理問題を正面から扱う困難さがあるため、紙数の関係もあり、本論文では論じなかった。この場合の詳細検討および、LISP 1.5のレベルでの実行ステップの理論的比較は、今後に残された課題である。

〈文献〉[1] McCarthy et al.: LISP 1.5 Programmer's Manual, MIT Press (1962), [2] 竹内: 第2回LISPコンテスト, 情処 Vol.20, No.3 (1979), [3] J.Allen: Anatomy of LISP, McGraw-Hill (1978), [4] 伊藤, 庄内, 岸本: マイクロプログラミング方式によるLISP処理系とその評価(I), 信学会計算機研 EC-81-68 (1981), [5] 伊藤, 岸本, 田村: マイクロプログラミング方式によるLISP処理系—AIMの基本構想とプログラミング環境—, 情処学会記号処理研 18-8 (1982) [6] 伊藤, 岸本, 庄内, 田村, 藤本: マイクロプログラミング方式によるLISP処理系とその高速化, 情処学会記号処理研 22-1 (1983)


```
#evalquote[fn;x] <= #apply[fn;x;NIL]

#apply[fn;x;a] <=
  [#atom[fn] -> [#eq[fn;CAR]  -> caar[x];
                 #eq[fn;CDR]  -> cdar[x];
                 #eq[fn;CONS] -> #cons[car[x];cadr[x]];
                 #eq[fn;ATOM] -> atom[car[x]];
                 #eq[fn;EQ]   -> #eq[car[x];cadr[x]];
                 T            -> apply[#eval[fn;a];x;a]];
   #eq[car[fn];LAMBDA]
      -> #eval[caddr[fn];#pairlis[cadr[fn];x;a]];
   #eq[car[fn];LABEL]
      -> #apply[caddr[fn];x;#cons[#cons[cadr[fn];caddr[fn]];a]]]

#eval[e;a] <=
  [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] <= #scan-select[⊛match[e;⊛car[a]];a]

#evcon[c;a] <= #scan-select[⊛eval[⊛car[c];a];
                            ⊛eval[⊛car[⊛cdr[c]];a]

#evlis[m;a] <= ⊛eval[m;a]

#pairlis[x;y;a] <= #conc2[⊛cons[x;y];a]

⊛car[(el ... en)] = (car[el] ... car[en])

⊛cdr[(el ... en)] = (cdr[el] ... cdr[en])

⊛cons[(x1 ... xn);(y1 ... yn)]
  = (#cons[x1;y1] ... #cons[xn;yn])

⊛eval[(el ... en);a]
  = (#eval[el;a] ... #eval[en;a])
```

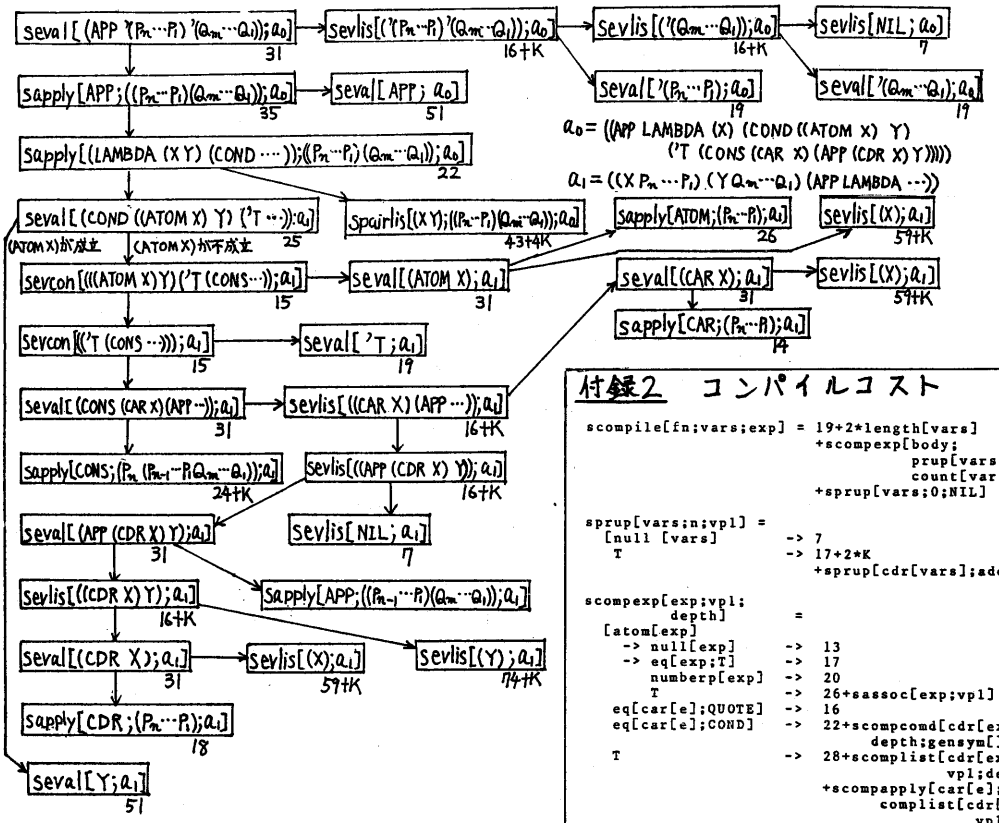
図9 並列LISPインタプリタ #evalquote

```
#sevalquote[fn;x] =  3 +#sapply[fn;x;NIL]

#sapply[fn;x;a] =
  [atom[fn] ->
    [eq[fn;CAR]  -> 12;
     eq[fn;CDR]  -> 12;
     eq[fn;CONS] -> 12+K;
     eq[fn;ATOM] -> 12;
     eq[fn;EQ]   -> 13;
     T           -> 11 +#seval[fn;a]
                       +#sapply[eval[fn;a];x;a]];
   eq[car[fn];LAMBDA]
               -> 11 +#spairlis[cadr[fn];x;a]
                     +#seval[caddr[fn];
                             pairlis[cadr[fn];x;a]];
   eq[car[fn];LABEL]
               -> 11+2K
                     +#sapply[caddr[fn];
                              x;
                              cons[cons[cadr[fn];
                                        caddr[fn]];
                                   a]]]

#seval[e;a] =
  [atom[e]          ->  6 +#sassoc[e;a];
   atom[car[e]] ->
    [eq[car[e];QUOTE] -> 14;
     eq[car[e];COND]  -> 14 +#sevcon[cdr[e];a];
     T                -> 13 +#sevlis[cdr[e];a]
                            +#sapply[car[e];
                                     evlis[cdr[e];a];
                                     a];
   T                  ->  6 +#sevlis[cdr[e];a]
                            +#sapply[car[e];
                                     evlis[cdr[e];a];
                                     a]]

#sassoc[e;a] =        3 +#ss + #m

#sevcon[c;a] =
  [eval[caar[c];a] -> #ss +max[#seval[cdar[c];a]+4;
                               #seval[cadar[c];a]+5];
   T               -> #sevcon[cdr[c];a]]

#sevlis[m;a] =
  [null[m]          ->  0;
   T                -> max[#seval[car[m];a];
                           #sevlis[cdr[m];a]]]

#spairlis[x;y;a] =
  [null[x]          ->  1;
   T                ->  3+K]
```

図10 並列LISPインタプリタの実行ステップ数

付録1　APPENDのインタープリタおよびコンパイラ・オブジェクトの実行ステップ数

インタープリタ実行ステップ数計算のTree



Xが長さnのリストのとき

spairlis[X;Y;a] = (18+2K)n+7

Vがa-listのi番目にあるとき

seval[V;a] = 15i+21

sevlis[(V);a] = 15i+44+K

APPの定義：

```
app[x;y]  <= [atom[x] -> y;
              T -> cons[car[x];app[cdr[x];y]]]
```

インタープリタによる実行ステップ数：

(723+12K)n+358+5K

APPをコンパイルしたオブジェクトは右の通り。

オブジェクトの実行ステップ数：

15n+7

```
APP   PUSHS   2
      ATON
      JUMPF   L2
      PUSHS   1
      GOTO    L1
L2    PUSHC   T
      JUMPF   L3
      PUSHS   2
      CAR
      PUSHS   3
      CDR
      PUSHS   3
      CALL    APP
      CONS
      GOTO    L1
L3    PUSHC   NIL
L1    CUT     2
      RETURN
```

注1) 関数APPの定義は a-listのトップから取り出すステップ数で取り出せるものとしている。

注2) 'e は (QUOTE e) を表わす。

付録2　コンパイルコスト

```
scompile[fn;vars;exp] = 19+2*length[vars]
                        +scompexp[body;
                                  prup[vars;0:NIL];
                                  count[vars]]
                        +sprup[vars;0;NIL]

sprup[vars;n;vpl] =
  [null [vars]          -> 7                                    ;
   T                    -> 17+2*K
                           +sprup[cdr[vars];add1[n];vpl]]

scompexp[exp;vpl;
         depth]       =
  [atom[exp]
     -> null[exp]      ->  13                                   ;
     -> eq[exp;T]      ->  17                                   ;
        numberp[exp]   ->  20                                   ;
        T              ->  26+sassoc[exp;vpl]                 ];
   eq[car[e];QUOTE]    ->  16                                   ;
   eq[car[e];COND]     ->  22+scompcomd[cdr[exp];vpl;
                                depth;gensym[]]                 ;
   T                   ->  28+scomplist[cdr[exp];
                                        vpl;depth]
                           +scompapply[car[e];
                                  complist[cdr[exp];
                                           vpl;depth];
                                  vpl;
                                  depth]                       ]

scompapply[fn;args;
           vpl;depth]   =
  [atom[fn]
     -> [eq[fn;CAR]    -> 16                                    ;
         eq[fn;CDR]    -> 20                                    ;
         eq[fn;CONS]   -> 24                                    ;
         eq[fn;ATOM]   -> 28                                    ;
         eq[fn;EQ]     -> 32                                    ;
         eq[fn;NULL]   -> 36                                    ;
         T             -> 39                                   ];
   eq[car[fn];LABEL]   -> 46+scompile[cadr[fn];
                                      cadaddr[fn];
                                      caddaddr[fn]]             ;
   eq[car[fn];LAMBDA]-> 41+2*length[cadr[fn]]
                        +scompile[caddr[fn];
                         prup[cadr[fn];depth;vpl];
                         plus[count[cadr[fn];depth]]
                        +sprup[cadr[fn];depth;vpl]  ]

scomplist[m;vpl
          depth]      =
  [null[m]             -> 7                                     ;
   T                   -> 21
                          +scompexp[car[m];vpl;depth]
                          +scomplist[cdr[m];vpl
                                     add1[depth]]    ]

scompcond[u;vpl;
          depth;glob] =
  [null[u]             -> 10                                    ;
   T                   -> 23+scompclause[car[u];vpl;
                                depth;gensym[];glob]
                          +sompcond[cdr[u];vpl;
                                    depth;glob]       ]

scompclause[c;vpl;
            depth;glob] =  20
                           +scompexp[car[c];vpl;depth]
                          +scompexp[cadr[c];vpl;depth]]
```