

SCHEME コンパイラ

元吉 文男

(電子技術総合研究所)

□ はじめに

SCHEME は LISP の方言であり、筆者のところでは既にそのインタープリタが動いている。ところが、このインタープリタは LISP で書かれており処理速度が遅い。前回の記号処理研究会で発表した「LINGOL コンパイラ」も出力は SCHEME のプログラムであり、このままでは「コンパイル」した意味がない。また、Prolog のプログラムを SCHEME に変換するプログラムも作成したが、これも二重にインタープリトされているために速度が遅い。

そこで SCHEME のコンパイラを作成した。SCHEME のコンパイラとしては既に RABBIT というものがあるが、ここで紹介するのは、それと同様のオプティマイズと更に違ったオプティマイズを行なっているものである。

□ SCHEME

SCHEME と LISP の違いを述べておく。

1.) 静的スコープ

LISP では変数のスコープルールが動的であったのに対し、SCHEME では静的である。そのために A-リスタの長さが、静的に完まらないうち、関数呼出しのたびに長くなることになり、関数引数を完全にサポートしても効率をそれほど落とさずに実行することができる。

そのため、関数自身を値とするプログラミングをすることにより、今までとは変った技法を使うことができる。図 1 はその例である。図 1.a は単に関数を用いる例であり、

図 1.b は値として複数の

ものを返させるのにコンティニューエーションとして利用したものである。また図 1.c は同じく関数をコンティニューエーションとして利用するが、それを Lazy Evaluation として使う例である。

なお関数の表記法については LISP では

```
(FUNCTION (LAMBDA (F) (F A B)))
```

と書いていたものを SCHEME では単に

```
(LAMBDA (F) (F A B))
```

と書くようになった。

2.) LABELS

SCHEME では LISP の LABEL を拡張して局所的な関数の定義を複数個行うことができるようになっていた。そのために、Algol 等の言語のようにブロック

```
(DE CONS (A B) (LAMBDA (F) (F A B)))  
(DE CAR (U) (U (LAMBDA (P Q) P)))  
(DE CDR (U) (U (LAMBDA (P Q) Q)))
```

図 1.a 関数引数の例

```
(DE FIB (N)  
  (LABELS  
    ((FIB1 (LAMBDA (N C)  
      (IF (LESSP N 1) (C 1 C)  
            (FIB1 (SUB1 N)  
                  (LAMBDA (A B)  
                    (C (PLUS A B) A)))))))  
    (FIB1 N (LAMBDA (A B) A))))
```

図 1.b 多値関数として使用した例

構造を持ったプログラムを書くことが可能となっている。また、LABELSで定義する関数は局所的であるので、名前を考えたときに衝突の心配をする必要がない。

また、1)で述べた静的スコープとあわせて、コンパイルするときグローバルなレベルでのオペティミズを容易に行うことが可能となっている。

LABELS の構文規則を図2に示しておく。

3.) IF

SCHEME では条件を表すのに COND を使用せずに IF を使用することになっている。

COND を使用したときにはマクロを使って IF を使う形に展開するようになっている。

IF は引数が三つあり、カー引数を評価してその値が NIL でなければカー引数を評価した値を答とし、NIL ならばカー引数を評価した値を答とする。この関数は値どりでなく、名前どりで引数を渡している。

4.) マクロ

SCHEME には LISP の FEXPR に相当する関数をユーザが定義する機能は備えていない。システムでもこれに相当する関数は QUOTE、LAMBDA、IF と後述する CATCH だけである。あとの関数はすべて値どりの関数か、マクロ呼出しの関数である。このようにしたために、インタプリタのプリミティブの数が少くなり、実装が容易になっていると同時に、コンパイル時のオペティミズについても場合分けの数が少なくて容易に行うことができる。

このため、SETQ、PROGN についてもこれをマクロ展開で行っているが、この例についてはコンパイルのところで説明する。

5.) CATCH

SCHEME の CATCH は LISP の CATCH と似た働きをするものであるが、その機能を説明しておく。CATCH は二引数であるが、カー

```
(CATCH RETURN
  (PROGN (SETQ A 1)
    (IF X (RETURN NIL) NIL)
    (PLUS A 2)))
```

図3.a CATCH の使用例

```
(DE SAMEFRINGE (P Q)
  (LABELS
    ((FRINGE
      (LAMBDA (X L)
        (IF (ATOM X) (CONS X L)
          (FRINGE (CAR X)
            (LAMBDA ()
              (FRINGE (CDR X) L)))))))
    (SAME
      (LAMBDA (A B)
        (IF (ATOM A) (ATOM B)
          (IF (ATOM B) NIL
            (IF (EQ (CAR A) (CAR B))
              (SAME ((CDR A)) ((CDR B)))
              NIL))))))
    (SAME (FRINGE P (LAMBDA () T))
      (FRINGE Q (LAMBDA () T))))))
```

図1.C Lazy Evaluation として使用した例

```
(LABELS
  (fn1
    (LABELS
      ((fn1 (LAMBDA (...) ...)
        (fn2 (LAMBDA (...) ...)
          ...
          (fnz (LAMBDA (...) ...)
            body))
```

図2. LABELS の構文規則

```
(IF (SETQ REPEAT (CATCH M M))
  'TRUE
  'FALSE)
```

```
(REPEAT REPEAT)
TRUE
(REPEAT NIL)
FALSE
```

図3.b CATCH の使用例

引数は文字アトムでその値に CATCH から抜けるための一引数の関数が束縛されて、カニ引数をその環境のもとで評価するようになる。言葉では説明しにくいので例を図3に書いておく。図3.a は LISP で使うのと同様の使い方。RETURN という文字アトムが CATCH から抜けるための関数になり、X の値が NIL でないと CATCH の値は NIL となり、NIL のときは 0 が答となる。図3.b が LISP とは異なる使い方。CATCH から抜け出す関数である M 自身を CATCH の値として返して REPEAT に代入している。種で REPEAT という関数に自分自身を引数として与えると先ほどの CATCH のところに戻り、その値を REPEAT に代入する。最後に NIL を引数として REPEAT を呼ぶと今度は CATCH のところに戻り今度は REPEAT の値が NIL となる。

6.) その他

LISP では引数の評価は左から右に行われていたが、SCHEME ではどの順番に評価してもよいようにプログラムを書かなければならない。これにより、コンパイラの最適化を行うときに、順番の入れ替えができるので進んだ最適化ができる。

LISP の EVAL では関数部分は関数になるまで何度でも(0回も含む)評価するが、SCHEME では丁度1回だけ評価するようになる。このため、多くの LISP では可能であった関数名と変数名に同じ文字アトムを使用することができなくなっている。

□ SCHEME コンピュータ

ここでは SCHEME コンピュータの概観を説明する。コンピュータの出力するものは LISP のプログラムである

が、これは LISP のサブセットになり、使っている機能は PROGN, SETQ とプリミティブな関数呼び出しであり、あとは、テールトランスフォームの制御を行っているだけで、再帰呼出しは行っていないので、機械語とほとんど対応したものであり、すぐに機械語に落とすことが可能となっている。以後ではコンピュータの実際の説明にあたり、図4のプログラムを例として取り上げることにする。

```
(DE FIB (N)
  (LABELS
    ((FIB1 (LAMBDA (N C)
      (IF (LESSP N 1) (C 1 0)
          (FIB1 (SUB1 N)
                (LAMBDA (A B)
                  (C (PLUS A B) A)))))))
    (FIB1 N (LAMBDA (A B) A))))
```

図4. コンピュータのソースプログラム例

コンピュータの実行手順を次に示す。

1. マクロ展開および名前の付替え (renaming)
2. CPS (Continuation Passing Style) への変換
3. オプティマイズ
4. コード生成

以下では、これらの詳細を順を追って説明することにする。

□ マクロ展開および名前の付替え

まず与えられたコードを調べて、マクロ定義のあるものについてはその尾回を行ない、LAMBDA 変数、LABELS の関数名、CATCH の変数名については、新しいユニークな名前を作り出して、その置き換えを行なう。

SCHEME では FEXPR に相当するものがないので MACRO を多用することになり、

マクロ展開は重要である。また PROGN がないために LET や DO という制御構造もマクロを使用して書かなければならない。LET については図5に展開の方法を示してある。DO については複雑なので省略してある。PROGN の展開を見るとわかるように評価の順番を付けたのは、applicative order に評価されて、引数が揃うまで関数を呼び出さないということを利用してある。

名前の置き換えを行うことにより、式を移動して LAMBDA 式の中に入れた時でも名前のスコープを考慮する必要がないのでオプティマイズが容易に行える。

またこのときには、AND 等で展開された IF については図6のオプティマイズを行うことにより、IF のオー引数をプレティケイトを持ってくることができるので、出力された LISP のコードを機械語にするときに、効率のよいコードが出力されることになる。

なお、このような名前の置き換えが可

```
(IF (LESSP N 1)
  (C 1 0)
  (FIB1 (SUB1 N)
    (LAMBDA (A B)
      (C (PLUS A B) A))))
continuation = K1
      |
      V
(LESSP
 (LAMBDA (Q1)
  (IF Q1
    (C K1 1 0)
    (SUB1
     (LAMBDA (Q2)
      (FIB1
       K1
       Q2
       (LAMBDA (K2 A1 B1)
        (PLUS2
         (LAMBDA (Q3) (C K2 Q3 A1))
         A
         B))))
      N)))
  1)
```

図7. CPS への変換の例.

```
(PROGN el) = el
(PROGN el . rest)
= ((LAMBDA (A B) (B))
   el
   (LAMBDA () (PROGN . rest)))

or = ((LAMBDA (N?) (PROGN . rest))
      el)
'N?' is a unique atom not in 'rest'

(AND el) = el
(AND el . rest)
= (IF el (AND . rest) NIL)

(OR el) = el
(OR el . rest) = (IF el T (OR . rest))

(LET ((v1 a1) (v2 a2) ...) e)
= ((LAMBDA (v1 v2 ...) e) a1 a2 ...)
```

図5. マクロ展開

```
(IF T b c) = b
(IF NIL b c) = c
(IF (IF a b c) d e)
= (IF a (IF b d e) (IF c d e))
= ((LAMBDA (?1 ?2)
   (IF a (IF b (?1) (?2))
        (IF c (?1) (?2))))
   (LAMBDA () d)
   (LAMBDA () e))

cf. (IF (AND a b) d e)
= (IF (IF a b NIL) d e)
= (IF a (IF b d e)
        (IF NIL d e))
= (IF a (IF b d e) e)
```

図6. IF オプティマイズ

能なのは、スコープが静的で各変数に対するアクセスをコードを見ただけで知ることができるからで、動的なスコープでは、他の関数からアクセスされる可能性もあるので、このような名前の置き換えを行うことはできない。

□ CPS (Continuation Passing Style) への変換

CPS (Continuation Passing Style) form というのは、関数を呼び出すときにその結果が出た後次に何をすることがということも引数として渡す形である。図7にCPS form への変換する例を示してある

が、上の (IF ...) という式の値を K1 というコンティニュエーションで、CPS form に変換するとその下の (LESSP ...) という式になる。この読み方は LESSP に N と 1 を与えてその結果を引数として、LESSP のオI引数である (LAMBDA (Q1) ...) を apply すると読む。つまり、関数呼出しのとき、引数を1つ増してそこに1引数の関数を書いて、呼出された関数の値を引数として、引数の関数を apply するようにする。

このようにして CPS form にすると関数呼出しはその値を返してもしる必要がなく、呼出しはいつも tail transfer になり、必要な情報はいつも引数に含まれているので、コードを生成するときはその式の通りに生成すればよい。

図8に CPS form に変換するための規則を書いておくが、図中での & という記号は、この記号の左側の式を右側の式をコンティニュエーションとする式に変換する操作である。

LAMBDA optimization

((LAMBDA () e)) = e

LAMBDA substitution

1. (ai is a constant)
and (vi is not SETQed)
2. (ai is a variable
and is not SETQed in e)
and (vi is not SETQed)
3. (ai is a LAMBDA closure)
and (vi is occurred
only once in e)
4. (ai has no side effects)
and (vi is not referenced)

((LAMBDA (... vi ...) e)
... ai ...)
= ((LAMBDA (... ...) e-sub)
... ...)
where e-sub = subst(ai,vi,e)

図9 LAMBDA オプティマイズ

```
( ?1,?2,...
are unique (GENESYMed) atoms )

e&cont = (cont e)
"e" is an atom or a constant

(IF p c a)&cont
= p&(LAMBDA (?1)
  (IF ?1 c&cont a&cont))

(CATCH id e)&cont
= ((LAMBDA (id) e&cont)
  (LAMBDA (?1 ?2) (cont ?2)))

(LAMBDA vs e)&cont
= (cont (LAMBDA (?1 . vs) e&?1))

(LABELS ((f1 (LAMBDA vs1 e1))
...
ex)
= (LABELS
  ((f1 (LAMBDA (?1 . vs1) e1&?1))
  ...
  ex&cont))

(fn al a2 ...) &cont
= fn&(LAMBDA (?0)
  al&(LAMBDA (?1)
    a2&(LAMBDA (?2)
      ...
      an&(LAMBDA (?n)
        (?0 cont ?1 ?2 ... ?n))...))

(fn al a2 ...) &cont
= al&(LAMBDA (?1)
  a2&(LAMBDA (?2)
    ...
    an&(LAMBDA (?n)
      (cont (fn ?1 ?2 ... ?n))...))
if fn is a primitive function
```

図8 CPS form への変換規則

図10に図4の式を CPS form に変換したものを示す。なおこの図では図10と違っておりミティブな関数である LESSP などは CPS form に直さずに、直接にコードを生成するようにしてある。また、ここでは RETURN という関数が使われているが、これはあくがコンティニュエーションであることをコンパイラのデバッグ時にわかり易くするためであり、この RETURN は見えな"ものだと思えばよい。すなわち

(RETURN C A) = (C A)

である。

```

(LAMBDA (C_1 N_2)
  (LABELS
    ((FIB1_3
      (LAMBDA (C_4 N_2_5 C_6)
        ((LAMBDA (K_7)
          (IF
            (LESSP N_2_5 1)
            (C_6 K_7 1 0)
            (FIB1_3
              K_7
              (SUB1 N_2_5)
              (LAMBDA (C_10 A_11 B_12)
                (C_6
                  C_10
                  (PLUS2 A_11 B_12)
                  A_11))))))
          C_4))))
    (FIB1_3
      C_1
      N_2
      (LAMBDA (C_14 A_15 B_16) (RETURN C_14 A_15))))))

```

図10. 図4の式を rename して CPS form にしたもの

□ オプティマイズ

前節で CPS form に変換された式からコード生成する前に、CPS form に対してオプティマイズを行う。IF に関しては CPS form に直す前にオプティマイズを行っていたが、ここではラムダ変数に対して除去できるものについて分析して変数の数を減らす操作を行う。またこの分析過程において、式中に表れるラムダ式について実際に関数 closure を作る必要のないものについては、コード生成のときにそのコードを生成しないようにマークをつける作業も行っている。

ラムダ変数を減らす操作は図9に示してあるように、関数部分にラムダ式があったときに、そのラムダ本体の仮引数のうち直接実引数で置き替えようというものである。現在のところは、図9にある4つの場合について行っているが、より精密な条件を考えた場合の数を増やすことも可能である。

このオプティマイズを行うために、まず分析を行うが、調べたことは式の中に表れる変数の参照の様子である。その変数が関数として作られたか、単に変数として作られたか、また変数として使われた場合には代入がなされたかどうか、さらに、その変数がラムダ式の中で自由変数として使われ、そのラムダ式を close する必要がある場合にはそのことも情報として蓄えておく。またこれらの情報はオプティマイズを1度行うと変化することがあるので、そのための変更に対してのオーバヘッドを減らすために参照が行われたたびにカウンタを増して、オプティマイズで変化が起ったときにはその値を減じて0になるときの性質が消えたと仮定している。図10の式について分析を行った結果を図11に示しておく。

```

A_15 ... VARIABLE
C_14 ... FUNCTION
N_2 ... VARIABLE
C_1 ... VARIABLE
C_4 ... VARIABLE
B_12 ... VARIABLE
A_11 ... VARIABLE
REF 2
PLUS2 ... PRIMITIVE
C_10 ... VARIABLE
SUB1 ... PRIMITIVE
FIB1_3 ... LABELS FUNCTION
REF 2
K_7 ... VARIABLE
REF 2
C_6 ... CLOSED FUNCTION
REF 2
N_2_5 ... VARIABLE
REF 2
LESSP ... PRIMITIVE

```

図11. 図10の分析結果

このようにして分析された結果をもとにしてオプティマイズを行うが、そのときに1度オプティマイズを行うと以前にはオプティマイズできなかったものが、できるようになる可能性もあるので、内部で変化があったところに対してはもう1度オプティマイズを試みる。前にも述べたようにオプティマイズを行って変数が消去されると変数の参照の非経が変化するので、そのUPDATE もこのときに行う。図10に対してラムダ式のオプティマイズを行った結果が図12であり、図10から K_7 という変数が消えている。

```
(LAMBDA (C_1 N_2)
  (LABELS
    ((FIB1_3
      (LAMBDA (C_4 N_2_5 C_6)
        (IF
          (LESSP N_2_5 1)
          (C_6 C_4 1 0)
          (FIB1_3
            C_4
            (SUB1 N_2_5)
            (LAMBDA (C_10 A_11 B_12)
              (C_6
                C_10
                (PLUS2 A_11 B_12)
                A_11)))))))
    C_1
    N_2
    (LAMBDA (C_14 A_15 B_16) (RETURN C_14 A_15)))) )
```

図12 図10をオプティマイズしたもの

次にグローバルなオプティマイズを行うが、今までの例を使用して説明することにする。ここではラムダ式が呼ばれたときの引数を調べて、省略できるものを探す操作を行う。

図12にはラムダ式が3個あるが、そのうちの1つは LABELS の変数である FIB1_3 という関数として使われている。残りの2つには名前がないので参照するときのために、上の方の (LAMBDA (C_10 ...) ...) に ?FN_21 と、下の方の (LAMBDA (C_14 ...) ...) に ?FN_18 と名前を付けて話を進めることにする。

まず、図12の式でプリミティブな関数以外の関数呼出しについて調べると、FIB1_3 が2回、C_6 が2回、C_14 が1回呼ばれていることがわかる。ここでは FIB1_3 だけが値が関数であることがわかっており、その定義は LABELS の中の定義式である。そこで仮引数として渡されるものを調べてみると図13の FIB1_3: と書いてあるようになる。この読み方は C_4 には C_4 が C_1 が渡され、N_2_5 には (SUB1 N_2_5) が N_2 が渡され、C_6 には ?FN_21 か ?FN_18 が渡されると

```
?FN_18
= (LAMBDA (C_14 A_15 B_16) (C_14 A_15))
?FN_21
= (LAMBDA (C_10 A_11 B_12) (IF ...))

FIB1_3: C_4      N_2_5      C_6
        C_4 (SUB1 N_2_5) ?FN_21
        C_1      N_2      ?FN_18
?FN_21: C_10     A_11     B_12
        C_4      I       0
        C_10 (PLUS2 ...) A_11
?FN_18: C_14     A_15     B_16
        C_4      I       0
        C_10 (PLUS2 ...) A_11

C_4 = C_4 or C_1 ==> C_4 = C_1
C_10 = C_4 or C_10 ==> C_10 = C_4 = C_1
C_14 = C_4 or C_10 ==> C_14 = C_1
```

図13 グローバルオプティマイズ

読む。すると、C-4 という変数の値は C-4 が C-1 が渡されるが、最初の C-4 は自分自身であるので結局 C-4 の値としては C-1 以外の値をとらないことがわかる。しかも C-4、C-1 とともに SETQ が行われていないので、図12の式中の C-4 を C-1 に書き替えて、C-4 を消去してもよいことがわかる。

今の分析の結果 C-6 には ?FN-21 が ?FN-18 が渡されるがこの2つともラムダ式であるので、今の議論と同様のことが出来る。C-6 が呼ばれるところは2か所あり、そのカ1引数には C-4 が C-10 が、カ2引数には 1 が (PLUS2 ...) が、カ3引数には 0 が A-11 が渡される。そこで ?FN-21 の方を見るとカ1引数が C-10 であり、渡される実引数が C-4 が C-10 であるので先程と同様に C-10 を C-4 で書き替えて C-10 は消去することが出来る。すると ?FN-18 のところのカ1引数である C-14 には C-4 と C-10 が渡されていたが、C-10 は C-4 に書き替えられていたので結局 C-14 には C-4

が渡されることになり、C-14 を C-4 で書き替えて C-14 を消去してもよいことがわかる。

以上の結果をもとに、図12の式を書き替えた結果を図14に示す。

□ コード生成

ここでは前節でオプティマイズされた結果から、これまでの分析の情報を考慮に入れてコード生成を行う操作について述べる。

生成されるコードは LISP の subset とするが、これは直接機械語を生成することも可能であるが、コンパイラのデバッグのためには LISP のプログラムの方が便利であること、また他の機種に移すにも中間コード的なものを用いた方が移植には便利であるし、既に LISP のコンパイラを備えたシステムではそのまま機械語に落とすことができる。このため、LISP のプログラムといっても LISP の構っている機能のほとんどは利用していない。使っているのはプリミティブな関数呼出し、暗黙 PROGN、COND などであり、あとはグローバル変数を使用して引数の受渡しを行い、コンパイルされた LISP プログラムの引数の数はいつも0で、しかも、再帰呼出しは行っていない。この再帰呼出しというのは狭い意味でのもので、tail recursive になっているものは tail transfer として goto と同じことなので、ここでは再帰呼出しとは呼んでいない。生成されたプログラムで別のプログラムを呼ぶには、今の tail transfer を用いるが、*FN* という変数に次に呼ぶべき関数名と環境の組を代入して一度関数から出て、コントロールプログラムから求めた関数を呼ぶのと2通りある。この後者の場合は、コンパイルされたプログラムからインタプリタされた関数を呼ぶ場合もあるのでコントロールプログラムを経由している。

実際のコード生成は、close する必要があるラムダ式の1つに対応して1つの

```
(LAMBDA (C_1 N_2)
  (LABELS
    ((FIB1_3
      (LAMBDA (N_2_5 C_6)
        (IF (LESSP N_2_5 C_6)
          (C_6 1 0)
          (FIB1_3
            (SUB1 N_2_5)
            (LAMBDA (A_11 B_12)
              (C_6
                (PLUS2 A_11 B_12)
                A_11)))))))
    (FIB1_3
      N_2
      (LAMBDA (A_15 B_16) (RETURN C_1 A_15))))))
```

図14. 図12にグローバルオプティマイズを行った結果

LISP プログラムを生成している。このとき、前節で分析した情報を利用して close する必要のないラムダ式については、その振舞いを知ることができたので、別の LISP 関数ではなく、1 つにまとめたコードを生成する。

コード生成に際しては、インタープリタが与えられた式をインタープリットするのとはほぼ同じに式をたどっていき、インタープリタがすること（コードとして出力して行く）という方法をとっている。このとき、参照されない変数への代入を検出して、副次作用のないときにはそのコードを生成しないというオプティマイズも行っている。

図 15 に今までの例題として使ってきた関数 FIB からコード生成した結果を示す。

```
(DE FIB NIL
  (SETQ *ENV* (CONS *ARG1* *ENV*))
  (SETQ *ARG3* *ARG2*)
  (SETQ *ARG4* (CONS 'CBETA (CONS '?FN_18 *ENV*)))
  (?FN_20))

(DE ?FN_20 NIL
  (SETQ *ENV* (CONS *ARG4* *ENV*))
  (COND
    ((LESSP *ARG3* 1)
     (SETQ *FN* (NTHENV 0))
     (SETQ *ARG1* 1)
     (SETQ *ARG2* 0))
    (T (SETQ *ARG3* (SUB1 *ARG3*))
        (SETQ *ARG4* (CONS 'CBETA (CONS '?FN_21 *ENV*)))
        (POPENV 1)
        (?FN_20))))

(DE ?FN_21 NIL
  (SETQ *FN* (NTHENV 0))
  (SETQ *1 *ARG1*)
  (SETQ *ARG1* (PLUS2 *ARG1* *ARG2*))
  (SETQ *ARG2* *1))

(DE ?FN_18 NIL
  (SETQ *FN* (NTHENV 0)))
```

図 15 コード生成された FIB

なお、この生成されたコード中で NTHENV と POPENV という関数が使われているがこれは、(NTHENV n) は *ENV* というリストの n 番目の要素を取り出すものであり、(POPENV n) は *ENV* の CDR を n 回とってその値を *ENV* に代入するものである。

このコードから機械語を生成するのは容易に行えることは理解できることと思う。ここで使用している *ARG1*、*ENV* などというグローバル変数をレジスタに割り当てなければ出力されるコードの効率はかなり良いものになると思われる。

また、このコードを走らせるにはコントロールプログラムが必要であるが、その 1 例を図 16 に示しておく。この例では、*FN* の CAR が CBETA 以外だと、そこから抜け出すようになっているが、それは今の SCHEME インタプリタで使われている BETA という文字アトムならばインタープリタを呼ぶように変更することも容易にできる。またこのプログラムには LISP からコンパイルされた関数も呼べるようになっている (MAIN FIB 10) とすればコンパイルされた FIB を走らせることができる。

```

(DF MAIN (?X) (PROG (?Y)
  (SETQ *FN* (EVAL (CAR ?X)))
  (SETQ ?X (CDR ?X)) (SETQ ?Y (CDR *G-ARG-VARS*))
  (SETQ *ARG1* NIL)
  L (COND ((NULL ?X) (GO L0)))
    (SET (CAR ?Y) (EVAL (CAR ?X)))
    (SETQ ?X (CDR ?X)) (SETQ ?Y (CDR ?Y)) (GO L)
  L0 (COND ((NOT (EQCAR *FN* 'CBETA)) (GO L1)))
    (SETQ *ENV* (CDDR *FN*))
    (APPLY (CADR *FN*) NIL) (GO L0)
  L1 (RETURN *ARG1*)))

```

図 16 コントロールプログラム

□ おわりに

以上で SCHEME のコンパイラについて述べてきたが、オプティマイズに関してはスコープが静的であるという性質を利用したものが、LISP の場合には使えない方法もある。また SCHEME では LABELS というブロック構造を定義するものがあり、これは LISP の LABEL とは異なり、よく利用される。このために補助変数などをブロック構造の中で定義して使うようにすれば、名前の衝突を避けることもできるし、コンパイル時にはグローバルなオプティマイズを行うことも可能である。

これからの検討すべき点は、コンパイルされたコードで、SCHEME と同じく静的なブロック構造を持つ Algol 系の言語のようなフレームを利用した変数アクセス法を採用した場合効率の比較がある。またグローバルなオプティマイズを行うと変数の数は減るが、そのために関数 closure を作るときに自由変数の数が増える場合があるので、一概にグローバルオプティマイズを行うべきかどうかを考慮の必要がある。

□ 文献

- G. Sussman 他 "SCHEME: An Interpreter for Extended Lambda Calculus."
AI Lab Memo 349. MIT (Cambridge, Dec. 1975)
- G. Steele Jr. 他 "LAMBDA: The Ultimate Imperative."
AI Lab Memo 353. MIT (Cambridge, Mar. 1976)
- G. Steele Jr. "LAMBDA: The Ultimate Declarative."
AI Lab Memo 379. MIT (Cambridge, Nov. 1976)
- G. Steele Jr. 他 "The Revised Report on SCHEME."
AI Lab Memo 452 MIT (Cambridge, Jan. 1978)
- G. Steele Jr. "RABBIT: A Compiler for SCHEME."
AI Lab Memo 474 MIT (Cambridge May 1978)