GPMとそのプログラム

和田 英一[1,a)]

概要：Christopher Strachey 他が, 英国製商用計算機 Atlas などのプログラム言語 CPL の実装手法として, GPM(general purpose macrogenerator) を開発したのは 1960 年頃のことだ. いまでは古典になり, 知る人も少くなった GPM は, 例えばマクロ a を`$def,a,<b~1d>;` と定義し, `$a,c;` と呼び出すと, `~1` が第 1 引数なので, `bcd` が得られ, `1+`を`$def,1+,<$1,2,3,4,5,6,7,8,9,10,$def,1,<~>~1;;>;` と定義し, `$1+,4;` と呼び出すと, 局所定義のマクロ `1` が起動して, `5` が得られるものである. これだけでどの程度のことが出来るか.
GPM の論文にある例題の評価法を理解すべく, 処理系を書いたのはかなり前だが, 最近 その例題以外にも GPM 処理系で走らせるプログラムをいくつか書いてみた.
基底にある GPM には算術演算もないので, 例題にはその辺から書いてあり, それを活用すれば, 小さい整数を使う ある程度のプログラムの書けることが分った.
GPM による風変りなプログラミングのノウハウ, プログラミング支援環境, さらに GPM 処理系の実装法などについて述べる.

キーワード：マクロジェネレータ, 記号処理, 文字列処理, スタック実装

GPM の使い方

英国ケンブリッジ大学の EDSAC は, プログラム記憶方式の最初の実用計算機であった. これを基本として商用計算機 Titan や Atlas が開発され, それで使うプログラム言語として, 当時制定されたばかりの Algol 60 を手本にした CPL が設計, 開発された [0].
CPL のコンパイラを作るにあたり, 例えばスタック操作では似たようなアセンブリコード列を繰り返し書く場合が多いので, 多少のパラメータを取り入れ, アセンブリコード列を吐き出すシステムが有用であろうということになった.
その解決策として提案され, 開発された記号処理系がこのマクロ言語 GPM である [1].

[1] IIJ 技術研究所
[a)] eiiti.wada@nifty.com

図 0 CPL GPM のテープ [2]

Cristopher Strachey はそれをかなり一般的な仕様にしたので, 出来上がったものは, それなりに面白いものであった. (使い難いという説もあり)
例えば 1 個の引数をとるマクロ a を, `$def,a,<b~1d>;` と定義すると, `~1` が第 1 引数を表わし, `$a,c;` と呼び出すと, マクロの定義が評価され, `~1`

に実引数の c が代入されて, 文字列 bcd が返えるものである. (元の論文では, マクロ呼出しの先頭は section sign (§) だが, ここでは$を使う.)
今の例を Scheme で書いてみると
```
(define a (lambda (~1) (list 'b ~1 'd)))
(a 'c) ⟹ (b c d)
```
のようなのもである.
~0 はマクロ名を表わし, $def,a,<~0b~1d>; の定義なら, 呼出し$a,c; は abcd になる.
定義自体もマクロ呼出しであり, その際, 定義本体の~1 が評価されないよう, <と>で囲む.
上の$a,c; の例では, 呼出しのマクロ名も実引数も文字列であったが, そういう場所にもマクロ呼出しが書ける.
$a,$a,c;; と呼び出すと, まず第 1 引数が評価されて bcd になり, それで a が呼び出されるので, 最終結果は bbcdd になる.
マクロ名がマクロ呼出しの例では, マクロ bcd を, $def,bcd,<b~1c~2d>; と定義しておき, $$a,c;,e,f; と呼び出すと, まず$a,c; が呼び出されて, bcd になり, 次に$bcd,e,f; の呼出しになって, becfd が出来上がる.
素晴らしいのはマクロ呼出しのなかに局所定義が書けることである. $a,c,$def,a,<b~1d>;; のように書くと, 第 2 引数が a の定義になっており, 最後のセミコロン; でマクロ a の評価が始まる時には, a の定義は出来ているのである.
これを局所定義といい, このマクロ評価が終ると局所定義は消える.
同じマクロ名のマクロが複数定義されていると, 最後に定義されたものが使われる. その様子を示すのが次の例だ. (各行の左端の斜体数字は行番号)
0 $def,a,b;
1 $a,$def,a,c;,$def,a,d;; ⟹ d
2 $a; ⟹ b
行 *0* でマクロ a を b と定義する. 行 *1* でマクロ a を呼び出すが, a の局所定義を 2 個持っている. この場合, 後の方の$def,a,d; が有効であり, 評価は d である. この行の評価が終わると局所定義は削除され, 次の行 *2* の評価では最初の a の定義が復活し, この評価は b である.
この機能を使い条件式 (if (eq? α β) γ δ) が書ける. つまり
$$\alpha,$def,\alpha,<\delta>;,$def,\beta,<\gamma>;;$$
とすれば, $\alpha = \beta$ なら後方の β の定義が使われて γ が得られ, そうでないなら前方の α の定義が生き, δ となる.
```
$def,if,<$~1,$def,~1,~4;,$def,~2,~3;;>;
$if,x,y,t,f; ⟹ f
$if,x,x,t,f; ⟹ t
```
元の論文にある suc(後者) の定義は次のようだ.
```
$def,suc,<$1,2,3,4,5,6,7,8,9,10,
  $def,1,<~>~1;;>;
```
これを$suc,3; で呼び出すと, suc の本体でマクロ 1 が呼び出される. このマクロの定義は後の方にあり, 定義は引数が代入されて$def,1,~3; になる. 従ってマクロ呼出しの先頭を 0 として 3 番目の 4 が得られる. $suc,0; で呼び出すと, マクロ 1 の名前が取り出されて, 1 が得られる.
tilde(~) に続く引数は数字 1 文字なので, 最大が 9 であり, その時の suc で 10 が得られる. 元の論文には他の文字を使って 10 以上に対応させてもよいと書いてある. 例えば ASCII コードの第 3 列 0123456789:;<=>?を使い
```
$def,a,
  <~?~<~=~>~;~:~9~8~7~6~5~4~3~2~1~0>;
$a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p;
```
⟹pmnolkjihgfedcba (m と o の位置に注意:-)
suc を下請けに, 2 桁の十進数の後者を求めるマクロ successor がある. $successor,2,3;⟹2,4, $successor,2,9;⟹3,0 になるのもである.
```
$def,successor,<$~2,
  $def,~2,~1<<,>$suc,>~2<;>;,
  $def,9,<$suc,>~1<;<,>0>;;>;
```
当然ながら 1 の桁 (~2) が 9 かどうかと調べる必要がある. それが 9 なら 10 の桁に 1 を足し, 1 の桁は 0 とする. そうでないなら, 10 の桁はそのままで, 1 の桁に 1 を足す. 上の定義は
```
(define (suc a) (1+ a))
(define (successor a b)
 (if (= b 9) (list (suc a) 0)
  (list a (suc b))))
```

のようになっている.
それが分かっても bra(<) と cket(>) が増えてくると, いつ使われるかを知るのが難しくなってくる. 次に示すのは先ほどのマクロの, $と;, <と>の入れ子関係が分かるように, それぞれにラベルをつけたものだ.

```
$def,successor,<$~2,
a                0b
  $def,~2,~1<<,>$suc,>~2<;>;,
  c          12 2d     1   1d1c
  $def,9,<$suc,>~1<;<,>0>;;>;
  c       1d     1   1d2 2 1cb0a
```

このうち, ラベルが 0 の bra と cket は$def の呼出しで外される. 5 回出てくる仮引数は深さが 1 だから, successor の呼出しで代入される.
$successor,2,3; の例だと

```
$3,
$def,3,2<<,>$suc,>3<;>;,
$def,9,<$suc,>2<;<,>0>;
;
```

になり, 従って, 3 と 9 の定義はそれぞれ

```
2<,>$suc,3;
$suc,2;<,>0
```

になる.
次に 2 桁の数 a,b と 1 桁の数 c を足すマクロ sum.

```
$def,sum,<$s,~1,~2,0,
a          0b
 $def,s,<$~3,
 c       1d
  $def,~3,
  e
   <$s,>$successor,~1,~2;<,>$suc,~3;<;>;,
   2f   2g                 g2 2g       g2f2e
  $def,>~3<,~1<<,>>~2;;>;;>;
  e     1   1    23 32   ed1cb0a
```

これを Scheme 風に書くと次の通り.

```
(define (sum a b c)
 (define (s p q r)
  (if (= c r) (list p q)
   (let ((t (successor p q)))
    (s (car t) (cadr t) (suc r)))))
 (s a b 0))
```

この動き方は次のようだ. ~3 が 2 種類あってややこしいが, 最下行のは外部から呼んだ c であり, それ以外は s の第 3 引数である (Scheme のプログラムでは r の方). つまり$sum,3,4,2; と呼ぶと, すぐ$s,3,4,0; が呼ばれるが, 最下行の~3 は 2, それ以外は 0 である.
次は 0 を呼ぶのだが, 局所的に$def,0,<$s,3,5,1;>; と$def,2,<3,4>; が定義される. 0 の定義が使われ, $s,3,5,1; が呼ばれる. 今回は 0 と 2 が定義されたが, 次回は 1 と 2 が定義され, さらに 2 と 2 が定義され, 後のが使われることになって結果 3,6 が得られるのである.
しかし, Strachey の方法より, もう少し簡単に出来そうだと思って書いたのが

```
$def,sum,
<$~3,
$def,~3,
  <$sum,>$successor,~1,~2;<,>$1-,~3;<;>;,
$def,0,~1<<,>>~2;;>;
```

$sum,3,4,5; ⟹3,9
である. これは Scheme 風では

```
(define (sum a b c)
  (if (= c 0) (list a b)
    (let ((t (successor a b)))
    (sum (car t) (cadr t) (1- c)))))
```

とやったことになる.
この辺までで様子が分かったから, これからは情報科学の標準問題を GPM でやってみよう.

Hanoi の塔

```
$def,1-,<$-1,0,1,2,3,4,5,6,7,8,
a        0b
 $def,-1,<~>~1;;>;
 c        1 1   cb0a
$def,hanoi,<$~4,
a           0b
 $def,~4,<$hanoi,>~1<,>~3<,>~2<,$1-,>~4<;;
 c        1d       1   1 1   1 1   1 e    1   1ed
  +>~1<->~3<+
   1   1 1   1
  $hanoi,>~2<,>~1<,>~3<,$1-,>~4<;;>;,
  d       1   1 1   1 1   1 e    1   1ed1c
 $def,0,~1<->~3;;>;
 c         1 1   cb0a
```

実行例

```
$hanoi,a,b,c,0; ⟹a-c
$hanoi,a,b,c,1; ⟹a-b+a-c+b-c
$hanoi,a,b,c,2;
⟹a-c+a-b+c-b+a-c+b-a+b-c+a-c
```

Fibonacci 数

```
$def,1+,<$1,2,3,4,5,6,7,8,9,10,
a       0b
 $def,1,<~>~1;;>;
 c       1 1   cb0a
$def,1-,<$-1,0,1,2,3,4,5,6,7,8,
a       0b
 $def,-1,<~>~1;;>;
 c       1 1   cb0a
$def,+,<$~1,
a       0b
 $def,~1,<$1+,$+,$1-,>~1<;,>~2<;;>;,
 c       1d    e   f    1   1f 1   1ed1c
 $def,0,~2;;>;
 c         cb0a
$def,fib,<$~1,
a        0b
$def,~1,<$+,
c       1d
 $fib,$1-,>~1<;;,
 e    f    1   1fe
 $fib,$1-,$1-,>~1<;;;;>;,
 e    f    g    1   1gfed1c
$def,1,1;,
c       c
$def,0,0;;>;
c       cb0a
```

実行例

```
$fib,0;,$fib,1;,$fib,2;,$fib,6;
```

⟹0,1,1,8

階乗

階乗を計算するには乗算が要る.

```
(define (* x y)
  (if (= x 0) 0
    (+ (* (1- x) y) y)))
```

これを GPM に直して,

```
$def,*,<$~1,
a      0b
  $def,~1,<$+,$*,$1-,>~1<;,>~2<;,>~2<;>;,
  c       1d  e   f    1   1f 1   1e 1   1d1c
  $def,0,0;;>;
  c       cb0a
```

実行例

```
$*,0,5;$*,1,4;$*,2,3;$*,3,2;$*,4,1;$*,5,0;
```

⟹0,4,6,6,4,0

乗算が出来たから階乗に取りかかる.

```
$def,!,<$~1,
a      0b
  $def,~1,<$*,>~1<,$!,$1-,>~1<;;;>;,
  c       1d  1   1 e   f    1   1fed1c
  $def,0,1;;>;
  c       cb0a
```

実行例

```
$!,0;,$!,1;,$!,2;,$!,3; ⟹1,1,2,6
```

中央値

このマクロは実際には 3 つの値のすべての場合のソートと殆ど同じである. 大小比較のマクロ `lt` は, $-1 \leq a, b$ について, $a < b$ を返す. 変域が -1 まであるのは tarai 関数で使いたいからである.

```
(define (lt a b)
 (cond ((= a b) f)
       ((= a -1) t)
       (else (lt (1- a) b))))
```

が定義. Scheme と GPM の `med` は次の通り.

```
(define (med a b c)
 (if (< b a)
  (if (< c a)
   (if (< c b) b c)
   a)
   (list b a c))
  (if (< c b)
   (if (< c a) a c)
   b))
```

```
$def,lt,<$~1,
a       0b
  $def,~1,<$lt,$1-,>~1<;,>~2<;>;,
  c       1d    e    1   1e 1   1d1c
  $def,-1,t;,$def,~2,f;;>;
  c         c c         cb0a
$def,med,<$$lt,~2,~1;,
a        0bc        c
  $def,t,$$lt,~3,~1;,
  c       de         e
    $def,t,$$lt,~3,~2;,$def,t,~2;,
    e      fg         g g         g
                        $def,f,~3;;;,
                        g         gfe
    $def,f,~1;;;,
    e        edc
  $def,f,$$lt,~3,~2;,
  c       de         e
    $def,t,$$lt,~3,~1;,$def,t,~1;,
    e      fg         g g         g
                        $def,f,~3;;;,
                        g         gfe
    $def,f,~2;;;;>;
    e        edcb0a
```

実行例

```
$lt,-1,-1;,$lt,-1,0;,$lt,0,-1; ⟹f,t,f
$med,0,0,2;,$med,0,1,2;,$med,0,2,1;,
$med,0,2,2;,$med,1,0,1;,$med,1,0,2;,
$med,1,1,1;,$med,1,2,0;,$med,1,2,1;,
$med,2,0,0;,$med,2,0,1;,$med,2,1,0;,
$med,2,2,0;
```

```
⟹
0,1,1,
2,1,1,
1,1,1,
0,1,1,
2
```

GCD

除算は使わず, 減算だけで GCD を計算する.

```
(define (gcd a b)
 (cond ((= a b) a)
       ((< a b) (gcd a (- b a)))
       ((> a b) (gcd (- a b) b))))
```

減算 - と gcd を次のように定義する.

```
$def,-,<$~2,
a       0b
  $def,~2,<$-,$1-,>~1<;,$1-,>~2<;;>;,
  c         1d  e    1   1e e    1   1ed1c
  $def,0,~1;;>;
  c          cb0a
$def,gcd,<$~2,
a         0b
 $def,~2,
 c
  <$$lt,>~1<,>~2<;,
  1de    1   1 1   1e
  $def,f,
  e
   <$gcd,$-,>>~1<<,>>~2<<;,>>~2<<;>;,
   2f     g   21   12 21   12g 21   12f2e
  $def,t,
  e
   <$gcd,>>~1<<,$-,>>~2<<,>>~1<<;;>;;>;,
   2f     21   12 g   21   12 21   12gf2ed1c
 $def,~1,~1;;>;
 c          cb0a
```

実行例

```
$gcd,2,4;,$gcd,5,3;,$gcd,6,3; ⟹2,1,3
```

二進化

与えられた数を二進法表示にする.

```
(define (b x y)
 (if (< x 2) (if (= y 0) (list x)
   (append (b y 0) (list x)))
  (b (- x 2) (+ y 1))))
```

GPM 版は以下のようだ.

```
$def,b,<$$lt,~1,2;,
a       0bc         c
 $def,t,<$>~2<,
 c        1d1   1
  $def,>~2<,<$b,>>~2<<,0;>>~1<;,
  e     1   1 2f   21   12   f21   1e
   $def,0,>~1<;;>;,
   e       1   1ed1c
 $def,f,
 c
  <$b,$1-,$1-,>~1<;;,$1+,>~2<;;>;;>;
  1d   e    f    1   1fe e    1   1ed1cb0a
```

実行例

```
$b,0,0;,$b,3,0;,$b,6,0;,$b,9,0;
⟹0,11,110,1001
```

二項係数

二項係数 (b n m) は次のように計算した.

```
(define (b n m)
 (cond ((= m 0) 1)
       ((= m n) 1)
       (else (+ (b (- n 1) (- m 1))
         (b (- n 1) m)))))
```

これを GPM に変換する. 後半の binom が (do ((m 0 (+ m 1))) ((> m n)) (b n m)) として, Pascal 三角形の一段を計算する.

```
$def,b,<$~2,
a       0b
 $def,~2,<$+,$b,$1-,>~1<;,>~2<;,
 c         1d  e   f    1   1f 1   1e
  $b,$1-,>~1<;,$1-,>~2<;;;>;,
  e   f    1   1f f    1   1fed1c
 $def,0,1;,$def,~1,1;;>;
 c         c c          cb0a
$def,binom,<$bb,0,
a           0b
 $def,bb,<$~1,
 c         1d
  $def,~1,<$b,>>~1<<,>~1<;,
  e         2f   21   12 2   2f
  $bb,$1+,>~1<;;>;,
  f    g    2   2gf2e
  $def,>~1<,1;;>;;>;
  e     1   1   ed1cb0a
```

実行例

```
$b,4,2; ⟹6
$binom,3; ⟹1,3,3,1
```

素数テスト

9 までの世界で素数テストはどうかと思うが, これも練習問題のつもりでやってみる. n の素数テストのストラテジーはこうだ. 2 から n までの除数で n の剰余を求め, 剰余が 0 なら f, 除数が n なら t を返す.

```
(define (isprime? n)
 (define (p x)
  (cond ((= x n) #t)
        ((= (modulo n x) 0) #f)
        (else (p (+ x 1)))))
 (p 2))
```

その GPM 版は (r は剰余をとるマクロ)

```
$def,r,<$$lt,~1,~2;,
a       0bc         c
 $def,t,~1;,
 c         c
 $def,f,<$r,$-,>~1<,>~2<;,>~2<;>;;>;
 c       1d  e  1  1 1  1e 1  1d1cb0a
$def,p?,<$p,2,
a        0b
 $def,p,<$~1,
 c       1d
  $def,~1,<$$r,>>~1<<,>~1<;,
  e        2fg  21  12 2  2g
   $def,$r,>>~1<<,>~1<;,
   g     h  21  12 2  2h
    <$p,$1+,>>~1<<;;>;,
    3h  i    32  23ih3g
   $def,0,f;;>;,
   g         gf2e
  $def,>~1<,t;;>;;>;
  e    1  1  ed1cb0a
```

実行例

```
$r,9,5; ⟹4
```

```
$p?,2;,$p?,3;,$p?,4;,$p?,5;,$p?,6;,
$p?,7;,$p?,8;,$p?,9; ⟹
t,t,f,t,f,
t,f,f
```

tarai 関数

tarai 関数の定義は次の通り.

```
(define (tarai x y z)
  (if (<= x y) y
  (tarai (tarai (- x 1) y z)
         (tarai (- y 1) z x)
         (tarai (- z 1) x y))))
```

これを GPM で書くとこうなる.

```
$def,tarai,<$$lt,~2,~1;,
a           0bc         c
 $def,f,~2;,
 c         c
 $def,t,<$tarai,
 c       1d
  $tarai,$1-,>~1<;,>~2<,>~3<;,
  e       f   1  1f 1  1 1  1e
  $tarai,$1-,>~2<;,>~3<,>~1<;,
  e       f   1  1f 1  1 1  1e
  $tarai,$1-,>~3<;,>~1<,>~2<;;>;;>;
  e       f   1  1f 1  1 1  1ed1cb0a
```

実行例

```
$tarai,4,2,0; ⟹4
```

Ackermann 関数

Ackermann 関数の定義は

```
(define (a x y)
 (cond ((= x 0) (+ y 1))
       ((= y 0) (a (- x 1) 1))
       (else (a (- x 1) (a x (- y 1))))))
```

```
$def,a,<$~1,
a       0b
 $def,~1,<$>~2<,
 c        1d1  1
  $def,>~2<,<$a,$1-,>>~1<<;,
  e     1  1 2f  g   21  12g
   $a,>>~1<<,$1-,>>~2<<;;>;,
   g  21  12 h   21  12hgf2e
  $def,0,<$a,$1-,>>~1<<;,1;>;;>;,
  e       2f  g   21  12g  f2ed1c
 $def,0,<$1+,>~2<;>;;>;
 c       1d   1  1d1cb0a
```

実行例

```
$a,0,0; ⟹1
```

```
$a,0,2; ⟹3
```

```
$a,2,0; ⟹3
```

```
$a,2,2; ⟹7
```

論理関数

易しい論理計算をやってみる. まず Scheme で.

```
(define (and x y) (if x y #f))
(define (or x y) (if x #t y))
(define (not x) (if x #f #t))
(define (maj x y z) (or (or (and x y)
  (and y z)) (and z x)))
(define (xor x y) (or (and x (not y))
  (and (not x) y)))
```

(maj は majority(多数決), xor は排他的論理和.)

続いて GPM. (not は\で表わす.)

```
$def,&,<$~1,$def,~1,f;,$def,t,~2;;>;
a       0b   c        c c         cb0a
$def,|,<$~1,$def,~1,t;,$def,f,~2;;>;
a       0b   c        c c         cb0a
$def,\,<$~1,$def,f,t;,$def,t,f;;>;
a       0b   c        c c        cb0a
$def,maj,<$|,$|,$&,~1,~2;,
a        0b  c  d       d
```

```
  $&,~2,~3;;,$&,~3,~1;;>;
  d       dc c       cb0a
$def,xor,<$|,$&,~1,$\,~2;;,
a        0b  c     d    dc
  $&,$\,~1;,~2;;>;
  c  d    d   cb0a
```

実行例

```
$&,f,f;,$&,f,t;,$&,t,f;,$&,t,t;
⟹f,f,f,t
$|,f,f;,$|,f,t;,$|,t,f;,$|,t,t;
⟹f,t,t,t
$\,f;,$\,t; ⟹t,f
$maj,f,f,f;,$maj,f,f,t;,
$maj,f,t,t;,$maj,t,t,t;
⟹
f,f,
t,t
$xor,f,f;,$xor,f,t;,$xor,t,f;,$xor,t,t;
⟹f,t,t,f
```

Hilbert 曲線

GPM の元々の目的はプログラム生成なのだから, そういう例も書いてみたい.

情報処理学会誌 [3] やブログにも書いたことのある Hilbert 曲線である.

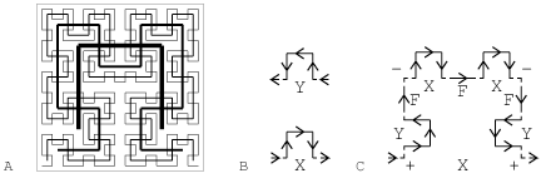

図 1 Hilbert 曲線

まず Scheme で書いてみる. GPM では状態を変数に置いておくことが出来ず, 常に引数で持ち回らなければならない.

GPM では絵は描けないので, PostScript の命令を出力することにする. `l 0 rlineto` などがそれで, これは現在地から x 方向に `l`, y 方向に `0` だけ線を引けという命令である.

一歩進む手続き `f` の引数 `d` は, 0,1,2,3 で右, 上, 左, 下を示す. `p` は上の図の `X` を, `q` は `Y` を計算する.

mod 4 の 1 の加減算には, `^`と`_`を使う.

```
(define (f d)
 (display (list-ref '("l 0 rlineto "
 "0 l rlineto " "l neg 0 rlineto "
 "0 l neg rlineto ") d)))
(define (^ x) (modulo (+ x 1) 4))
(define (_ x) (modulo (+ x 3) 4))
(define (p n d) (if (> n 0)
  (begin (q (- n 1) (^ d)) (f (^ d))
   (p (- n 1) d) (f d) (p (- n 1) d)
   (f (_ d)) (q (- n 1) (_ d)))))
(define (q n d) (if (> n 0)
  (begin (p (- n 1) (_ d)) (f (_ d))
   (q (- n 1) d) (f d) (q (- n 1) d)
   (f (^ d)) (p (- n 1) (^ d)))))
```

上手くいくことを確認してから, GPM で次のように書く.

```
$def,1-,<$-1,0,1,2,3,4,5,6,7,8,
a       0b
  $def,-1,<~>~1;;>;
  c       1 1  cb0a
$def,^,<$1,2,3,0,$def,1,<~>~1;;>;
a       0b        c       1 1  cb0a
$def,_,<$3,0,1,2,$def,3,<~>~1;;>;
a       0b        c       1 1  cb0a
$def,f,<$~1,
a       0b
 $def,0,<l 0 rlineto >;,
 c       1             1c
 $def,1,<0 l rlineto >;,
 c       1             1c
 $def,2,<l neg 0 rlineto >;,
 c       1                1c
 $def,3,<0 l neg rlineto >;;>;
 c       1                1cb0a
$def,p,<$~1,
a       0b
 $def,~1,
 c
 <$q,$1-,>~1<;,$^,>~2<;;$f,$^,>~2<;;
 1d  e    1  1e e  1  1edd  e  1  1ed
  $p,$1-,>~1<;,>~2<;$f,>~2<;
  d  e    1  1e 1  1dd  1  1d
  $p,$1-,>~1<;,>~2<;$f,$_,>~2<;;
  d  e    1  1e 1  1dd  e  1  1ed
  $q,$1-,>~1<;,$_,>~2<;;>;,
  d  e    1  1e e  1  1ed1c
 $def,0,;;>;
 c       cb0a
$def,q,<$~1,
a       0b
 $def,~1,
 c
 <$p,$1-,>~1<;,$_,>~2<;;$f,$_,>~2<;;
 1d  e    1  1e e  1  1edd  e  1  1ed
```

```
  $q,$1-,>~1<;,>~2<;$f,>~2<;
  d   e    1   1e 1   1dd   1   1d
  $q,$1-,>~1<;,>~2<;$f,$^,>~2<;;
  d   e    1   1e 1   1dd   e   1   1ed
  $p,$1-,>~1<;,$^,>~2<;;>;,
  d   e    1   1e e   1   1ed1c
 $def,0,;;>;
 c       cb0a
```

実行例

```
/l 20 def 100 100 moveto
$p,3,0;
stroke
```

$\Longrightarrow$

```
/l 20 def 100 100 moveto
0 l rlineto l 0 rlineto
0 l neg rlineto l 0 rlineto
l 0 rlineto 0 l rlineto
```

(この後同様な 28 行省略)

```
0 l neg rlineto
stroke
```

これを PostScript に食べさせると下のような図が得られる.

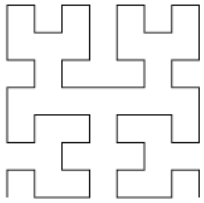

図 2 Hilbert 曲線 (出力)

マクロの書き方指南

上のマクロの例の殆どは次ように出来ている.

tarai 関数のように, 比較条件 (<= x y) は$$le,~1,~2; と始め, これが t か f を返すので, この後の仕事を$def,t,... や$def,f,... のところに書く. 今のシステムにはマクロ lt しかないので, le は引数の順を反対にし, t と f の定義も逆にしてある.

Ackermann 関数は

```
(cond ((= x 0) ...)
      ((= y 0) ...)
      (else ...))
```

だから, まず x の値を調べるため

```
$def,a,<$x,
       $def,x,<x!=0 の処理>;,
       $def,0,<x==0 の処理>;
;>;
```

と書く. x==0 の処理 は y+1 なので$1+,y; でよい. x!=0 の処理はさらに y の値を調べるので

```
$y,
$def,y,<y!=0 の処理>;,
$def,0,<y==0 の処理>;
;
```

この y!=0 の処理 は

```
$a,$1-,x;,$a,x,$1-,y;;;
```

y==0 の処理 は

```
$a,$1-,x;,1;
```

従って全体はとりあえず

```
$def,a,<$x,
       $def,x,<
$y,
$def,y,<$a,$1-,x;,$a,x,$1-,y;;;>;,
$def,0,<$a,$1-,x;,1;>;
;
>;,
       $def,0,<$1+,y;>;
;>;
```

と骨組みが出来る. それぞれの段階で$と;, <と>の構造を確認する. 確からしくなったらエディタを使って然るべき場所に挿入し, 整形する.

```
$def,a,<$x,
a       0b
 $def,x,<
 c       1
  $y,
  d
  $def,y,<$a,$1-,x;,$a,x,$1-,y;;;>;,
  e       2f   g     g g     h     hgf2e
  $def,0,<$a,$1-,x;,1;>;
  e       2f   g     g   f2e
  ;
  d
  >;,
  1c
  $def,0,<$1+,y;>;
  c       1d     d1c
;>;
b0a
```

再びエディタを使い, x を~1 で, y を~2 で置き換える. (emacs の query-replace を使う.)

```
$def,a,<$~1,
 $def,~1,<
  $~2,
  $def,~2,<$a,$1-,~1;,$a,~1,$1-,~2;;;>;,
  $def,0,<$a,$1-,~1;,1;>;
  ;
  >;,
  $def,0,<$1+,~2;>;
;>;
```

最後に tilde(~) の変数の場所が深さ 1 に揃うように, 前後の cket(>) と bra(<) を挿入する. 出来上がったプログラムは次だ.

```
$def,a,<$~1,
 $def,~1,<
  $>~2<,
  $def,>~2<,<$a,$1-,>>~1<<;,
   $a,>>~1<<,$1-,>>~2<<;;;>;,
  $def,0,<$a,$1-,>>~1<<;,1;>;
  ;
  >;,
  $def,0,<$1+,>~2<;>;
;>;
```

これには次の Scheme のプログラムを使った.

```
(define (insertbracket file)
(let ((ip (open-input-file file))
      (op (open-output-file "output"))
      (lv -1))
(define (copy)
 (let ((ch (read-char ip)))
  (if (eof-object? ch)
   (close-output-port op)
   (begin
    (if (char=? ch #\~)
     (display
      (string-append
       (make-string lv #\>)
       "~" (string (read-char ip))
       (make-string lv #\<)) op)
     (let ((m (member ch '(#\< () #\>))))
      (if m (set! lv (+ lv (length m) -2)))
      (display ch op))) (copy)))))
(copy)))
```

二項係数の binom の計算では, do ループ bb で, m が局所変数, n が大域変数なので多少面倒である. m,n を使って素直に書くと

```
0 $def,bb,<$m,
1  $def,m,<$b,n,m;,$bb,$1+,m;;>;,
2  $def,n,1;;>;
```

これを$bb,0; で呼び出すことになる.

```
$def,binom,<$bb,0,
bb の定義
>;
```

が全体像だ. bb の定義の中で, 行 *0* と *2* の bra と cket の間の m は~1 に, n は>~1<に置き換え, 行 *1* の bra と cket の間の m は>~1<に, n は>>~1<<に置き換える. この程度だと手でやっても簡単だ.

全体の完成品は

```
$def,binom,<$bb,0,
$def,bb,<$~1,
 $def,~1,<$b,>>~1<<,>~1<;,
 $bb,$1+,>~1<;;>;,
 $def,>~1<,1;;>;
>;
```

となる.

ついでに, 対応する$と;, <と>にラベルをつけるプログラムは次だ. 本文にある色付きのラベル表示のプログラムは utilisp で書いた別のものである.

```
(define (gpmformcheck file)
 (let ((ip (open-input-file file))
       (op (open-output-file "output"))
       (c 0) (n 0) (outs ""))
 (define (getn n)
   (string-ref "0123456789" n))
 (define (getc c)
   (string-ref "abcdefghijklmn" c))
 (define (copy)
  (define (strap x) (set! outs
  (string-append outs (string x))))
  (let ((ch (read-char ip)))
   (if (eof-object? ch)
    (begin (close-output-port op) 'done)
   (begin (display ch op)
```

```
    (case ch
     ((#\newline) (display outs op)
      (newline op) (set! outs ""))
     ((#\$) (strap (getc c))
      (set! c (+ c 1)))
     ((#\;) (set! c (- c 1))
      (strap (getc c)))
     ((#\<) (strap (getn n))
      (set! n (+ n 1)))
     ((#\>) (set! n (- n 1))
      (strap (getn n)))
     (else (strap #\space)))
   (copy)))))
 (copy)))
```

GPM のその他の機能

これまでには, GPM の組込みマクロとして$def だけしか説明しなかったが, GPM には他にも組込みマクロが用意してあった.

- $val,$x$; x に定義されている値を, 評価せずに取り出す.
- $update,$x$,$y$; x に定義されている値を, y で置き換える. ただし, 実装によっては最初に定義された時の長さを超えるとエラーになる.
- $bin,$x$; x の文字列を二進数に変換する.
- $dec,$x$; x の二進数を文字列に変換する.
- $bar,$op$,$x$,$y$; op の (+,-,*,/,r) に従って, x と y の加減乗除算, 剰余の計算をする.

a,b を変数として使う val と update の例は

```
$def,a,0;$def,b,f;$val,a;$val,b;
$update,a,1;$update,b,t;$val,a;$val,b;
```

⟹0f1t

総和をとる fold-left の例は

```
$def,sum,<$s,0,0,
$def,s,<$~1,
 $def,~1,<$s,$1+,>~1<;,$+,$>$val,list;<,
  $def,>$val,1st;<,<~>>~1<;;,>~2<;;>;,
 $def,>$val,len;<,~2;;>;;>;
$def,list,<1,0,2,3>;
$def,len,4;$def,1st,1;
$sum; ⟹6
```

bin, dec, bar の例は

```
$dec,$bar,+,$bin,12;,$bin,23;;;,
$dec,$bar,-,$bin,55;,$bin,38;;;,
$dec,$bar,*,$bin,12;,$bin,14;;;,
$dec,$bar,/,$bin,80;,$bin,11;;;,
$dec,$bar,r,$bin,80;,$bin,11;;;
```

⟹

```
35,
17,
168,
7,
3
```

素数テストは

```
$def,1+,<$dec,$bar,+,$bin,~1;,$bin,1;;;>;
$def,r,<$dec,$bar,r,$bin,~1;,$bin,~2;;;>;
```

を追加するだけで

$p?,31;,$p?,49; ⟹t,f

と能力は抜群に向上するが, GPM を楽しむためには禁じ手としたい. (私の処理系には内緒で実装してある.)

ホワイトスペース問題

空白, 改行文字の扱いは議論のあるところだ. 元の論文には

It takes as its input a stream of characters and produces as its output another stream of characters which is produced from the input by direct copying except in the case of *macro-calls* in the input stream, which are "evaluated" in a way which is described below before they are put into the output stream.

と述べられているだけである.

これはもともとアセンブリ言語で書かれているプログラムの一部にマクロ呼出しがあり, 入力の大体は通過させ, マクロ呼出しのところだけを展開するという趣旨から見れば, 空白や改行はそのまま複写するのが当然である.

ところが, 例えば tarai の定義が仮に

```
$def,tarai,
<$if,$le,~1,~2;,~2,
$tarai,$tarai,$1-,~1;,~2,~3;,
```

␣␣$tarai,$1-,~2;,~3,~1;,
␣␣$tarai,$1-,~3;,~1,~2;;>;
であって, 改行や空白が入っていたとする.
$tarai,4,3,2;
を計算したとする.
(tarai 4 3 2)=
(tarai (tarai 3 3 2) (tarai 2 2 4)
  (tarai 1 4 2))=
(tarai 3 2 4)
なのだが, 下請け tarai の返した値の前に, 定義内のホワイトスペースが挿入され
$tarai,3,\n␣␣2,\n␣␣4;
になり, 引数の比較が不可能になる.
一方, ホワイトスペースが使えると, マクロ名に空白や改行を含むことも出来る. 例えば
$def,␣,<a~1c>;$␣,␣␣␣; ⟹a␣␣␣c
はマクロ名が空白 1 文字, 第 1 引数が空白 3 文字だ. こういうマクロは書かない方がよいのはもちろんだが.
注意して書けば, 定義もホワイトスペースには対処出来るが, 最終出力には定義内のホワイトスペースの残骸が散らばり, 始末に負えない.
Martin Richards の提案した対処法は grave accent(‘) を置くと, その文字を含めて行末までとそれに続くホワイトスペースを読み飛ばすものである.
これを使うとコメントも書くことが出来るし, 上のような定義に忍び込むホワイトスペースも除くことが出来る.
上の tarai の定義は
$def,tarai,‘
<$if,$le,~1,~2;,~2,‘
$tarai,$tarai,$1-,~1;,~2,~3;,‘
␣␣$tarai,$1-,~2;,~3,~1;,‘
␣␣$tarai,$1-,~3;,~1,~2;;>;
のように書く.

マクロ合成マクロ問題

Strachey の元の論文に
$def,fnprod,
  <$def,~1~2,<$>~1<,$>~2<,~1;;>;>;
というのがある.
$fnprod,log,sin;
と呼び出すと, sin の log がとれるマクロ logsin が作れるというものだ.
例えば
fnprod,1+,1+;
でマクロ 1+1+が$def,1+1+,<~1>; のように出来そうだが, マクロから出た途端に露と消える.
論文には fnprod で出来たマクロは引数リストの一部ではないから, マクロ呼出しが終わっても消えないと書いてあるが, 後述する CPL の GPM 処理系でも出来た新しい定義が定義のリンクに繋がらないので使えない. どこかおかしいのではないかと思うが, よくは分からない.

GPM に出来ないこと

これまで多くの例を GPM で書いてきたが, これなら出来そうだと思って探したわずかな例である. 実際問題として, pureGPM では面白いことは殆ど何も出来ないといってよい.

- 変数がない. $val, $update を使えば別だが, データはすべて引数で持ち回らなければならない. 関数型プログラミングの気分である.
- 数値演算がない. 実用的には$bin, $bar, $dec を使う.
- 文字列演算がない. 文字列を連結することは出来るが, 切り取り, その部分を使うことは出来ない. 2 桁の加算の例にあったように, 35 なら 3,5 のように用意する.
- データ構造がない. 従ってソートのようなことが出来ない. Martin Richards の bgpm には list があるようだが, どう使うのかな.
- データ読込み機能がない. これも関数型風だ.
- 呼び出しているマクロの外側のマクロの引数を見ることは出来ない. それが使いたければ内側のマクロの定義にコピーしておく. そこにデータ構造がないので, 不定個数の引数が扱えない.

このようにないない尽しだが, 頭の体操としては結構楽しかった. $, ,, ;, ~, <, >以外の文字が, 空白改行も含めて自由に使えるのは素晴らしい経験であった.

GPM 処理系の実装

Strachey の論文は, Part 1 の GPM の概略の説明と, Part 2 の CPL で記述した処理系での解説で構成されている. 私は Part 1 を読み, 自分の解釈に従って Scheme で実装した処理系 (Scheme GPM) と, CPL の記述を Scheme に直した処理系 (CPL GPM) を持っている. これらについて次に述べる. なお Martin Richards が BCPL で実装した処理系が存在する [4]. また Fortran で書いたものもあるらしいが, ネット上では見つからない [5].

Scheme で実装した処理系

前述の GPM の例では, 再帰処理が面白いが, それには処理系も再帰的に働かなければならない. Scheme はそれに適しているから, それでの実装は簡単である.

基本的には, Strachey が warning character(要注意文字) という, $, ,, ;, <, >, ~ の 6 文字以外は (空白, 改行文字も含めて) そのまま出力にコピーし, 要注意文字はそれぞれについて必要な処理をするのである.

- $ マクロの呼出しが始まるので, 引数用のリストを用意し, , か; が来るまで文字列の読込み処理を続ける.
- , 処理した文字列を引数リストに追加する.
- ; 処理した文字列を引数リストに追加し, マクロを呼出す. 第 0 引数のマクロ名を定義のリストから探す. def なら定義の処理 (定義を定義リストに追加する) に進む. それ以外なら定義の本体の文字列処理を開始する. マクロ本体の処理が終ったら結果の文字列を出力に追加する. マクロ呼出し中に追加した局所定義を削除する.
- < 対応する>までの文字列を (両端の<, >は外して) 出力に追加する.
- ~ 続く 1 文字 (数字) を読み, その数字番目の引数を (文字列処理せず) そのまま出力に追加する.

Scheme で書いた, GPM のとりあえず使える処理系を次に示す. 注意すべきは行 *01* にある cs で, 各レベルでの局所定義の数を記憶する. マクロ呼出しから戻る時に, トップにある数だけ定義を消去する. 下に示すプログラムリストの行 *01* の env, cs は$def,a,b; を読み終わった時, ((a b)), (1) になっている. さらに$a,$def,a,c;,$def,a,d;; を読み終わった時, ((a d) (a c) (a b)), (2 1) になっている. $a のマクロの実行が終わった時, 先頭の 2 個の定義を削除する.

readstring の mc は, ,, ; が要注意文字ではないマクロ呼出しの外の判定に使う (行 *30*). ~の判定には cs を使う (行 *35*).

```
(define (gpm str)
 (let ((env '()) (ch '()) (cs (list 0))
(outport (open-output-file "output"))
 (define (g str args)
(let ((outs "") (ix 0))
; getch は str から次の 1 字を ch に読み込む.
(define (getch)
 (if (< ix (string-length str))
  (begin (set! ch (string-ref str ix))
   (set! ix (+ ix 1)))
  (set! ch #\delete)))
; readquote は<, >の文字列を読む.
(define (readquote)
 (let ((outs ""))
 (define (rq) (getch)
  (if (char=? ch #\>) outs
   (begin (set! outs
    (if (char=? ch #\<)
     (string-append outs
       "<" (readquote) ">")
     (string-append outs (string ch))))
    (rq))))
 (rq)))
; readstring は str をスキャンし, 結果を outs
; に作る.
(define (readstring mc)
 (let ((outs ""))
 (define (strap x)
  (set! outs (string-append outs x)))
 (define (rs)
  (if (= ix (string-length str)) outs
   (begin (getch)
```

```
28   (case ch
29    ((#\; #\,)
30     (if mc (begin (strap (string ch))
31       (rs))
32      (cons ch outs)))
33    ((#\<) (strap (readquote)) (rs))
34    ((#\~)
35     (if (= (length cs) 1) (strap "~")
36      (begin (getch) (strap
37       (list-ref args
38        (- (char->integer ch) 48)))))
39     (rs))
40    ((#\$) (set! cs (cons 0 cs))
41     (strap (readmacrocall)) (rs))
42    (else (if (char<? ch #\delete)
43     (strap (string ch))) (rs))))))
44  (rs)))
; readmacrocall は$を見つけると引数を読み込み
; actuals に繋げる.
45 (define (readmacrocall)
46  (let ((actuals '()))
47  (define (rm)
48   (let* ((result (readstring #f))
49     (ch (car result)) (s (cdr result)))
50    (cond ((char=? ch #\,)
51      (set! actuals (cons s actuals))
52      (rm))
53     ((char=? ch #\;)
54      (set! actuals (cons s actuals))
55      (macrocall (reverse actuals))))))
56   (rm)))
; セミコロンが来ると, 引数のリストを持って
; macrocall へ. def なら定義を繋げ, 空文字列を
; 返す. それ以外のマクロなら, 定義を探し, その本
; 体の文字列と実引数の組で行 3 の手続き g を再帰
; 呼出しし, 本体文字列のスキャンを開始する.
57 (define (macrocall actuals)
58  (display "macro-call ")
59  (display actuals) (newline)
60  (cond ((string=? (car actuals) "def")
61    (set! env (cons (cdr actuals) env))
62    (set! cs (cdr cs))
63    (set-car! cs (+ (car cs) 1)) "")
64   (else (let*
65    ((def (assoc (car actuals) env))
66     (result (g (cadr def) actuals)))
67   (display "output ") (display result)
68   (newline) result))))
69 (let ((result (readstring #t)))
70  (set! env (list-tail env (car cs)))
71  (set! cs (cdr cs))
72  result)))
; gpm の本体. 一番外のスキャンの結果を出力する.
73 (display (g str '()) outport)
74 (close-output-port outport)))
```

この gpm は文字列 str を受け取るから, gpm のプログラムをファイル xxx.gpm から読み込むには

```
(define (file2str file)
 (read-string (char-set)
 (open-input-file file)))
(gpm (file2str "xxx.gpm"))
```

とする.

CPL の GPM

GPM は 1964 年の初め頃からケンブリッジ大学の Mathematical Laboratory で使われていたというから, Strachey が処理系を書いたのは 1963 年頃と思われる. Strachey は 1916 年生まれだから, 47 歳くらいの時にこういうプログラムを書いていたかと想像すると微笑ましい.

CPL で書いた GPM 処理系は, GPM の唯一の形式記述である. 解釈が怪しくなった時, 頼りになるのはこの処理系である.

CPL で書いた GPM のプログラムは, Goto 文有害論 [6] が登場する前なので, Goto 文だらけだが, Goto のラベルから始まる部分を Scheme の手続きにし, Goto 文の代わりにその手続きを呼ぶという形にして書き直した. x,y:=y,x のような CPL 独特の同時代入が頻出するので, 注意が必要だ.

GPM はマクロの中でマクロを呼ぶから, 再帰呼出し対策をしなければならず, それを 1 個のスタックで実装する. スタックの要素は文字と, 文字列の

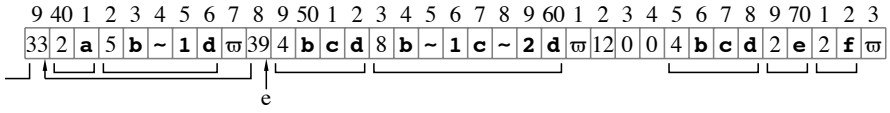

図 3 CPL GPM のスタックの例

長さを示す整数と, スタック内のアドレスを示すリンクである. その他, 定義の本体の終りを示すマーカーもスタックに乗る.

図 3 はそのスタックの例で,

0 `$def,a,<b~1d>;`

1 `$a,c;`

2 `$def,bcd,<b~1c~2d>;`

3 `$$a,c;,e,f;`

を最後の; まで読み終わった時の様子である. タイプライター体は文字を示す.

スタックの 0 から 38 までは, def のような組込みマクロの情報が入っている. 39 からが a の定義で, 39 にある 33 はこの前の定義が 33 にあることを示す. 次の 2a がマクロ名, 5b~1c がマクロの本体, 47 の ϖ が定義の終りを示すマーカーだ. 2a, 5b~1c から察せられるように, 文字列は文字列長を示す数値を先頭に持つ. 本体は<, >で囲まれていたが, 一度評価されたので, 外側の<, >は外れている.

48 からは bcd の定義で, 同様な構造である. この辺りは, 行 *1* の`$a,c;` を評価した時の作業場所であったが, 評価が終り出力されたので, その痕跡はない.

62 からが行 *3* の`$$a,c;,e,f;` の評価中のところで, マクロ名内の`$a,c;` はすでに bcd に置き換わり, 実引数の e も f も読まれ, 最後の; によってマーカー ϖ も入っている. CPL の処理系ではマーカーには適当な負数を使う.

次は, e=48 から始まる定義のリンクを辿ってマクロ名 bcd の定義を探すことだ. これはすぐに見つかり, 54 から始まる本体を評価しつつスタックに積むか出力するかする. その評価中に仮引数に出会うと, 65 から並んでいる実引数の方をコピーする.

図 4 は

```
$def,1+,<$1,2,$def,1,<~>~1;;>;
$1+,$1+,0;;
```

を評価する時のスタックのトレースである.

左端の斜体 2 桁がトレースの行番号, その右が読み込んだ文字 (またはマーカー ϖ), 続いてその時点でのスタックを制御する変数, c, f, h, p, s. さらにその右がスタックの内容である. *09* 行目までは先頭がスタックの 39 からを, *10* 行目からは 39 から 63 までは変わらないので, 64 からを示す. (上下の見出しを参照.)

変数 c はスキャンする文字の読込み位置を示す. c= 0 の時は入力端末から読み, ≠ 0 の時はスタックのその位置から読む. h= 0 は出力を出力端末へ書き, ≠ 0 の時は呼出したマクロのスタックでの位置を示す.

f は呼び出されたマクロのスタック上の位置を記憶する. マクロ呼出しが次々と中に入ると f のリンクが形成される. semicolon(;) がきて実行が始まると, f の値は p が引き継ぐ.

s はスタックの最初の空き地を示す. 今は s=39. *00* 行目 $を読むとスタックに h,f,0,,0 を積み, f=s+1=40, h=s+3=42, s=43 になり, 文字列処理 (評価) モードになる. *01* 行目から d, e, f はそのまま積まれ, s=46. *04* 行目 , を読み, 42 に def の長さ 4, 46 に 0 を入れる. s=47. 1, +, , の読込みも同様. *08* 行目 <を読み, 対応する>の直前までをスタックに積む. *09* 行目 ; を読み, def の処理を行う. つまり 39 に定義のリンク 33 を入れ, 40 から 42 が見出し 1+, 43 から 62 までが本体. 63 にマーカー. s=64 となる. これ以降 63 までは不変. ここで文字列処理モードからコピーモードに戻る. f と h はスタックの 41, 40 で 0 と 0 に復活する.

次に改行を読むが, h=0 なのでスタックには積まず, そのまま出力される.

10 行目 次の入力行の$を読み, *00* 行目と同様. 1, +, , の処理も同様. *14* 行目 $を読み, 71,72 に h と f を積む. f=72, h=74 になる.

```
      c f h p s  39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
00 $  0 0 0 0 39  0| 0  0  0
01 d  0 4042 0 43  0| 0  0  0 d
02 e  0 4042 0 44  0| 0  0  0 d e
03 f  0 4042 0 45  0| 0  0  0 d e f
04 ,  0 4042 0 46  0| 0  0  4 d e f 0
05 1  0 4046 0 47  0| 0  0  4 d e f 0 1
06 +  0 4046 0 48  0| 0  0  4 d e f 0 1 +
07 ,  0 4046 0 49  0| 0  0  4 d e f 3 1 + 0
08 <  0 4049 0 50  0| 0  0  4 d e f 3 1 + 0 $ 1 , 2 , $ d e f , 1 , < ~ > ~ 1 ; ;
09 ;  0 4049 0 69 33| 3  1 +20 $ 1 , 2 , $ d e f , 1 , < ~ > ~ 1 ; ; ϖ
10 $  0 0 0 0 64  0| 0  0  0
11 1  0 6567 0 68  0| 0  0  0 1
12 +  0 6567 0 69  0| 0  0  0 1 +
13 ,  0 6567 0 70  0| 0  0  3 1 + 0
14 $  0 6570 0 71  0| 0  0  3 1 + 0 7065 0 0
15 1  0 7274 0 75  0| 0  0  3 1 + 0 7065 0 0 1
16 +  0 7274 0 76  0| 0  0  3 1 + 0 7065 0 0 1 +
17 ,  0 7274 0 77  0| 0  0  3 1 + 0 7065 0 3 1 + 0
18 0  0 7277 0 78  0| 0  0  3 1 + 0 7065 0 3 1 + 0 0
19 ;  0 7277 0 79  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ
20 $ 4565707280  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 0
21 1 4681837284  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 0 1
22 , 4781837285  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 2 1 0
23 2 4881857286  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 2 1 0 2
24 , 4981857287  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 2 1 2 2 0
25 $ 5081877288  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 2 1 2 2 0 8781 0 0
26 d 5189917292  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 2 1 2 2 0 8781 0 0 d
27 e 5289917293  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 2 1 2 2 0 8781 0 0 d e
28 f 5389917294  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 2 1 2 2 0 8781 0 0 d e f
29 , 5489917295  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 2 1 2 2 0 8781 0 4 d e f 0
30 1 5589957296  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 2 1 2 2 0 8781 0 4 d e f 0 1
31 , 5689957297  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 2 1 2 2 0 8781 0 4 d e f 2 1 0
32 < 5789977298  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 2 1 2 2 0 8781 0 4 d e f 2 1 0 ~
33 ~ 6089977299  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 2 1 2 2 0 8781 0 4 d e f 2 1 0 ~ 0
34 ; 628997721 00 0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 7065 0 2 1 2 2 0 39 2 1 3 ~ 0 ϖ
35 ; 6381877295  0| 0  0  3 1 +25 9 0 0 3 1 + 2 0 ϖ 167263 2 1 2 2 8 39 2 1 3 ~ 0 ϖ ϖ
36 ~ 9365708196  0| 0  0  3 1 +25 9 0 0 3 1 + 2 0 ϖ 167263 2 1 2 2 8 39 2 1 3 ~ 0 ϖ ϖ 1
37 ϖ 9565708197  0| 0  0  3 1 + 9 9 0 0 3 1 + 2 0 ϖ 1
38 ϖ 6465707281  0| 0  0  3 1 + 0 1
39 ;  0 6570 0 72 9| 0  0  3 1 + 2 1 ϖ
40 $ 45 0  0 6573 9| 0  0  3 1 + 2 1 ϖ 0 0 0 0
41 1 4674766577  9| 0  0  3 1 + 2 1 ϖ 0 0 0 0 1
42 , 4774766578  9| 0  0  3 1 + 2 1 ϖ 0 0 0 2 1 0
43 2 4874786579  9| 0  0  3 1 + 2 1 ϖ 0 0 0 2 1 0 2
44 , 4974786580  9| 0  0  3 1 + 2 1 ϖ 0 0 0 2 1 2 2 0
45 $ 5074806581  9| 0  0  3 1 + 2 1 ϖ 0 0 0 2 1 2 2 0 8074 0 0
46 d 5182846585  9| 0  0  3 1 + 2 1 ϖ 0 0 0 2 1 2 2 0 8074 0 0 d
47 e 5282846586  9| 0  0  3 1 + 2 1 ϖ 0 0 0 2 1 2 2 0 8074 0 0 d e
48 f 5382846587  9| 0  0  3 1 + 2 1 ϖ 0 0 0 2 1 2 2 0 8074 0 0 d e f
49 , 5482846588  9| 0  0  3 1 + 2 1 ϖ 0 0 0 2 1 2 2 0 8074 0 4 d e f 0
50 1 5582886589  9| 0  0  3 1 + 2 1 ϖ 0 0 0 2 1 2 2 0 8074 0 4 d e f 0 1
51 , 5682886590  9| 0  0  3 1 + 2 1 ϖ 0 0 0 2 1 2 2 0 8074 0 4 d e f 2 1 0
52 < 5782906591  9| 0  0  3 1 + 2 1 ϖ 0 0 0 2 1 2 2 0 8074 0 4 d e f 2 1 0 ~
53 ~ 6082906592  9| 0  0  3 1 + 2 1 ϖ 0 0 0 2 1 2 2 0 8074 0 4 d e f 2 1 0 ~ 1
54 ; 6282906593  9| 0  0  3 1 + 2 1 ϖ 0 0 0 2 1 2 2 0 39 2 1 3 ~ 1 ϖ
55 ; 6374806588  9| 0  0  3 1 + 2 1 ϖ 166563 2 1 2 2 8 39 2 1 3 ~ 1 ϖ ϖ
56 ϖ 88 0  0 7489 9| 0  0  3 1 + 2 1 ϖ
57 ϖ 64 0  0 6573 |
      c f h p s  44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89
```

図 4 CPL GPM のスタックのトレース

19 行目 ; を読み, マクロ処理を始める. f を p にコピー. f は一段外のスタックを指すようになる (65). マクロ名 1+を定義のリンクを辿って探すと, *09* 行目の 40(31+) が見つかる. 従って本体を 44 から読み始める. (c=45 なのはスタックの c の位置から読み, c を既に 1 増やしたからだ.) *20* 行目の $は 44 のもので, 続けて 1, ,, 2, , を読み, *25* 行目で局所定義の$を読む. *32* 行目の<で~を読んだのが 98 に積まれている~だ. *33* 行目 裸の~を読んだので, その次の文字 1 を読み, 74 からの実引数の 1 番目, 0 をスタックに積む. *34* 行目 ; を読んだので, マクロ 1 の定義 21 3~0 ϖ が 89 以降に出来る. 88 に 39 とあるのは定義のリンクである.

35 行目の; は, こうした定義の環境の中で, マクロ 1 を起動する. つまり定義のリンクを探すと 90 が見つかり, 92 からの本体を読み始める. *36* 行目 ~ だから, その次の 0 を読み, 84 の 1 をスタックに積む.

37 行目 94 のマーカー ϖ を読み, ここで$1+のマクロ処理が終り, 結果の 1 が *20* 行の$を積んだところまで戻ってきた. *38* 行目 さらに 79 のマーカー ϖ を読み, $1+,0; の処理が終り, *14* 行目で$を積んだところ (71) まで 1 が戻ってきた.

次に c=0 で読む; は入力の最後にあったものである. これで外側の$1+の処理が出来るようになった. *40* 行目から再び 1+の定義の読込みが始まり, 同じように進行する. ただ今度は最後の 2 は *55* 行目で (h=0 なので) スタックには積まず, 出力してしまう. $1+,$1+,0;; ⟹2

もう一度 f と p の関係を確認しておく.

01 行目 古い f=0 をスタック 41 に積み, f は s+1 の 40 になる. def の処理には p を使うことはなく, 処理の終りで f は昔の値に戻る.

19 行目 この; で p=72 になり, p+2 からのマクロ名を定義リンクから探すことになる.

CPL による GPM の処理系は, かなり複雑なのだが, マクロ合成マクロが分からないのを除くとエラーは見当たらない. さらに実行中のエラーの検出が的確かつ親切に報告されるようになっており, 楽しむために作った Scheme の処理系に較べるとはるかに実用的に構成されている.

後書き

Christopher Strachey という名前を初めて見たのは 1959 年にパリで開催された IFIP 会議の報告書で, そこに彼の *Time sharing in large, fast computers* という論文があった [7].

Strachey に出逢ったのは, 1971 年 4 月 英国 Warwick 大学で開かれた IFIP WG2.2 の会議でであった. この会議には主査の Mike Woodger(NPL) を始め, Edsger Dijkstra[6], Willem van der Poel, Simula を開発した Ole-Johan Dahl など有名な計算機科学者が大勢いた. その会議中 パーティーが主催者の John Buxton[0] の超古い家で開催され, みなが車に分乗して行くことになった. その時昔の小さい Mini に乗せてくれたのが大柄な Strachey であった. 「Eiiti は後に入れ」と私を後部座席に押し込み, 助手席に乗った Brian Randell と機関銃のような英語で話し合っていた.

Strachey はその後数年して他界し, Tony Hoare がその Oxford の教授職についた.

参考文献

[0] D.W.Barron, J.N.Buxton, D.F.Hartley, E.Nixon, C.Strachey, *The main features of CPL*, Computer Journal, Vol.6, 134-143.(1963).

[1] C. Strachey, *A general purpose macrogenerator*, Computer Journal, Vol.8, No.3, 225-241(1965).

[2] Andrew Herbert, *Strachey's General Purpose Macrogenerator*, Computer Resurrection, Number 66, Spring 2014, http://www.cs.man.ac.uk/CCS/res/res66.htm#e.

[3] 和田英一, 関数画家, 情報処理学会誌, Vol.45,No.10,(2005 年 10 月), pp.1163-1171.

[4] http://www.cl.cam.ac.uk/~mr10/BCPL/bcplprogs/bgpm/bgpm.b

[5] Ju-Tung Hsu, Albert Newhouse, *Strachey's General Purpose Macrogenerator in Fortran*, CULLEN COLL OF ENGINEERING HOUSTON TEX., University of Houston Defense Technical Information Center, 1970 - 51.

[6] Edsger W. Dijkstra, *Go To Statement Considered Harmful*, Comm.ACM, Vol.11,No.3,pp.147-148(1968).

[7] C. Strachey, *Time sharing in large, fast computers*, Proc. IFIP Congress, 1959, 336-341.