19. 再帰呼出しの実現法†

疋 田 輝 雄††

1. 再帰呼出しとは

正整数 n の階乗 $n!$ の値を計算する関数を PASCAL で書いてみる.

```
function fac (n : integer) : integer;
begin if n=1 then fac: =1
                  else fac: =fac(n-1) *n
end
```

この関数定義は次の数学的な定義ときれいに対応している.

$$n!=\begin{cases}1 & n=1,\\ (n-1)!\times n & n>1.\end{cases}$$

関数 *fac* は実行部において, 実引数 $n-1$ で自分自身を1回呼出している. このように, 関数や手続きが自分自身を呼出すことを再帰呼出し recursive call, 再帰呼出しを含む関数 (手続き) を再帰的関数 (手続き) という. 呼出しは直接でなくても, いくつかの呼出しを経て結局自分自身を呼ぶ場合をも含む.

手続きや関数の再帰的呼出しは, 汎用目的のプログラミング言語においては, FORTRAN, COBOL, BASIC では禁止されている. しかし ALGOL 60 と LISP 以降, PASCAL, C, ADA, APL, SNOBOL 等, 再帰を許す言語の方が普通になってきている.

一方, 現実の (von Neumann 以来の) 計算機の機械命令語体系は, 再帰をそのままの形で表現するようにはなっていない. 再帰呼出しの実現とは, 再帰的手続きや関数からなるプログラムを, 再帰呼出しを含まない, いわゆる逐次的 iterative なプログラムに変換することである. これは人手でもできるが, 通常は各言語のコンパイラやインタプリタの仕事である.

再帰の重要性*と, これを備える言語の多さを考えるとき, 本項目の標題は単に「手続き・関数の呼出しの実現法」と直したくなる**.

本稿では再帰呼出しの一般的な実現法を扱うが, 手続きや呼出しの具体的な形によっては, よりうまい実現もありうる. これについては, 本特集中の「ソースプログラムの高速化」と「再帰呼出しの索表計算法」を見られたい. 本項目は「多重環境の制御実現法」および「非決定性アルゴリズム」とも関連が深い.

再帰呼出しの一般的な実現のポイントとなるのは, 局所的な変数の値の保持の方法である. 原理を一口で言うと, これらの情報を, 手続き呼出しのたびにスタックに積み, 復帰のたびにスタックから出せばよい. この原理を簡単な例を用いて次章で示す. しかし実現の細部においては, 各言語における関数・手続きの定義の方法や名前の有効範囲の規則によって異なってくる. すなわち静的に定義される ALGOL 系の言語と, LISP などプログラム実行時に動的に定義する言語とである. それぞれを第3章と第4章で扱う.

以上はすべてスタックを実現において用いるが, 最近まったく異なる, コンビネータを用いる方法が実用的な実現法として提唱されている. これについて短く第5章でふれる. なお, 論理型言語 PROLOG にもいわば再帰呼出しは存在するが, 本稿では割愛する[13].

再帰呼出しの実現に関しては, 結果値を返す関数と返さない手続きの区別は本質的ではないので, 本稿中で一方について言うときはつねに他方をも含意しているものとする.

2. 簡単な例——フィボナッチ数

先の関数 *fac* は再帰としては簡単に過ぎ, 逐次的なプログラムに書き直すことはごく容易である. もう少し複雑な, 実行部中で再帰呼出しが2回現われている例として, **プログラム(1)**の, n 番目のフィボナッチ数の値を計算する関数 *fib* を考える. このフィボナ

† Implementation of Recursive Procedures by Teruo HIKITA (Department of Mathematics, Tokyo Metropolitan University).
†† 東京都立大学理学部数学科

* 計算一般に対する数学的理論である Kleene の帰納的関数理論においては, 再帰は中心概念である. Church による λ-calculus においてはパラドキシカル・コンビネータ Y がその役割をする.
** 本項目の内容にもっと忠実な標題は,「変数束縛の実現法」あるいは「環境の実現法」であろう.

```
.........
function fib(n: integer): integer;
begin
    if (n=0) or (n=1) then
        fib: =1
    else
        fib: =fib(n-2)+fib(n-1)
end;
.........
begin {main program}
    .........
    X: =fib(N);
    .........
end.
```

プログラム(1)

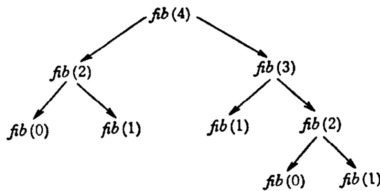

図-1 *fib*(4) が作る呼出しの木

```
var s: array [1..30] of
        record n, t: integer; ra: labelt end; {stack}
    p: 0..30;                                  {stack pointer}
    r: integer;                                {function result}
.........
L1:  if (s[p]. n=0) or (s[p].n=1) then
        begin r: =1;
            p: =p-1; goto s[p+1].ra    {pop and return}
        end
        else
        begin p: =p+1; s[p].n: =s[p-1].n-2; s[p]. ra: =L11;
            goto L1;                   {push and call}
L11:        s[p]. t: =r;
            p: =p+1; s[p].n: =s[p-1].n-1; s[p]. ra: =L12;
            goto L1;                   {push and call}
L12:        r: =s[p]. t+r;
            p: =p-1; goto s[p+1]. ra   {pop and return}
    end;
.........
begin {main program} p: =0;
    .........
    p: =p+1; s[p]. n: =N; s[p]. ra: =L21;
    goto L1;                           {push and call}
L21: X: =r;
    .........
end.
```

プログラム(2)

ッチ数の計算も，逐次的なうまい（実行効率の良い）プログラムは幾通りかある．しかしここでは，このプログラム－(1)に対して，どのような再帰呼出しにも通用するような一般性のある，逐次的なプログラムへの変換を試みる．

関数 *fib* を引数の値 4 で呼ぶとき，このプログラムによって計算はどのように進行していくであろうか．*fib*(4) を呼出すと，その実行途中でまず *fib*(2) を呼ぶ．この *fib*(2) はその途中で *fib*(0) を呼ぶが，これは値 1 をすぐに返すから，*fib*(2) は次に *fib*(1) を呼ぶ．これの値 1 を得て，*fib*(2) はその結果としての値 $1+1=2$ をはじめの *fib*(4) に返す．このようにして，実行部分のコードは同じで，引数(や局所変数)の値が互いに異なるような関数実行が，たくさんなされる．これらの呼出し関係は全体として **図-1** のようになる（矢印は呼出しを示す）．計算の全過程は，この木の根から出発して，全頂点を左下方優先の順序でたどることである．

再帰のない呼出しの場合との相違点は，引数（や局所変数）を，関数ごとではなく，関数の呼出しごとに用意しなければならないということである．その他に，呼出しのたびに制御の戻り先 return point, return address も保存しておかねばならないが，これは再帰のない場合でも同様であった．

今後本稿中では，ある手続き（関数）の引数，そこで宣言される変数，そしてコンパイラが生成する作業用の変数，これらを総称して単に局所変数 local variables ということにする．また，1 回分の呼出しに対応する局所変数の領域と，それに付加していくつかの制御用の情報（戻り先など）を含めたものを，ふつう，フレーム frame，起動レコード activation record，あるいはデータセグメント data segment などと称する．

フレームは，手続きの呼出しのたびに用意し，手続きからの復帰のたびに不要になる．このような状況のとき，これらフレームはスタック上に割付けるとよい．スタックとは 1 次元の形状の記憶領域で，データの出し入れがその一方の端からのみなされるものである．通常は配列で実現し，現在出し入れを行っている位置をポインタ（スタック・ポインタ）で指しておく．ハードウェアでスタックを備えている計算機もある．

再帰呼出しを含むプログラム(1)は呼出しと復帰とを次のように変換すれば逐次的になる．

呼出し (call)

1）呼ばれる側の関数のフレームの領域をスタックに新たに用意する (push)．引数には渡すべき値を入れる*．

2）スタックに戻り先も入れる．

3）呼ばれる関数の先頭へ飛ぶ．

---

* 引数の渡し方は，値呼出し call by value，または参照呼出し call by reference であるものとする．名前呼出し call by name のときはやや面倒である[1),10)]．

復帰 (return)

1) 現在のフレームを解放する (pop).

2) 保存してあった戻り先へ飛ぶ.

プログラム(1)中の呼出しと復帰をすべてこれらの規則に従って書き換えると**プログラム**(2)が得られる. ただし, この(偽 PASCAL の) プログラムでは, 戻り先を示すため *labelt* というデータ型を使用しているが, これは PASCAL には存在しえないものである (機械語ではこれは番地であり, データとして扱うことに問題はない).

FORTRAN の言語仕様を拡張して再帰呼出しを書けるようにし, 本章の方法によってプリプロセスして通常の FORTRAN プログラムに変換するシステムもいくつか使用されている[3),11)].

3. 静的な手続き定義をもつ言語[1),6),10),14)]

前章のフィボナッチ数のプログラムにおいては, 関数の実行部から参照する変数は, 局所的なものだけであった. 言い換えると, 再帰をスタックによって実現したとき, 参照する変数は最新のフレーム内にあった. しかし一般的には, 実行中に大域的な変数を, つまりスタックの底の方のフレーム内の変数をも参照する. この参照をどのように実現するかは, 各言語における手続き・関数の定義の方式や名前の有効範囲の規則に依存する.

ALGOL 60 系の言語—PL/I, PASCAL, ADA—では手続きは静的に定義される. すなわち, 実行中に動的に生成されることはなく, プログラム・テキスト上で定義され意味が決まっている. なおこの章では, 再帰の実現は, 対象の言語の性質上, コンパイラによって機械語を生成するという形で考える (本質的なことではない).

1つのプログラム・テキスト中で, 手続き全体はいわゆる入れ子構造をなす*. 1つの手続きの実行部からは, テキスト上それを包含している手続きたち (それ自身を含めて) において宣言された変数だけ参照できる. **プログラム**(3)において, 手続き $p2$ の実行部では, 手続き $p1$ と $p2$ で宣言された変数は使用できるが, $p3$ と $q2$ の変数は参照できない.

図-2 は, プログラム実行時において, 手続きが順に, $p1$, $p2$, $q2$, $p2$, $p3$ と呼出した時点でのスタックの状況を示す. この時点で手続き $p3$ の実行部が参照できる変数は, スタック中の①, ④, ⑤の部分である. ②, ③の部分は参照してはならない. 一般に, プログラム実行時のある時点で参照できる変数の全体を環境 environment という.

```
procedure p1;
    var x1, y1;
    procedure p2;
        var x2, y2;
        procedure p3;
            var x3, y3;
        begin ……… end;
    begin ……… end;
    procedure q2;
        var z1;
    begin ……… end;
begin ……… end;
```

プログラム(3)

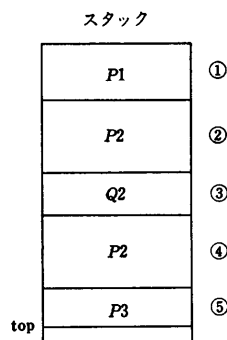

図-2 局所変数のスタックへの動的割付け

以上のような変数アクセスの状況すなわち環境を一般的に実現するのに, 大きく言って2通りの方法がある.

3.1 ディスプレイ法

これは Dijkstra[6)] による.

手続きの入れ子の深さをレベルと称することにし, 主プログラムのレベルを 1, それから順に 2, 3, … と深くなっていくものとする. 各手続きのレベルはプログラム・テキストを見ればわかるから, (実行しなくても) コンパイル時に決定できる.

プログラム実行中において, スタック中のフレームのうちで参照できるものは, 現在実行中の手続きのレベルと等しいか小さいような各レベルに対して, ちょうど1つずつある. そこで, 現在参照できる各フレームの先頭の, スタック内での位置 (base address) を1個所にまとめて保持し, これを**ディスプレイ** display と称する. 変数は, その宣言された手続きのレベルを

* ここでは, 簡単のため, ALGOL 60, PL/I, ADA にあるいわゆるブロックは考えず, 手続きと関数とからなるブロック構造を考える. なお, 言語Cでは, 関数の入れ子構造は許されないので, 再帰呼出しの実現法は, 前章の方法でほぼ充分である.

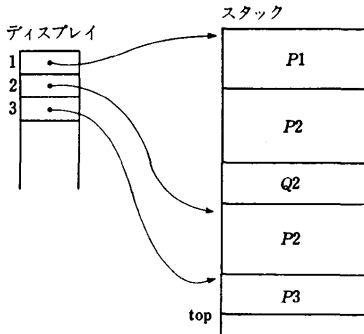

図-3 ディスプレイ

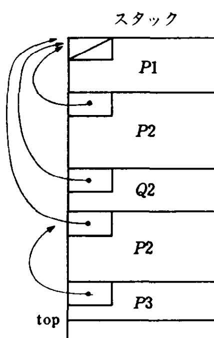

図-4 静的チェイン

i とすると，スタック中での位置は，

$$display\,[i]+offset$$

で表わされる (**図-3**)．ここで *offset* は，その変数の，フレーム内での相対位置であり，コンパイル時に決定できることが多い．

ディスプレイの大きさはプログラムの入れ子の深さと一致する．ディスプレイの内容は，手続きの呼出しと復帰の際に更新ないし復活しなければならない．

ディスプレイを用いる方法の特徴は，次に述べる静的チェインを用いる方法と比べて，大域的変数へのアクセスが速いことである．ディスプレイを汎用レジスタ上に実現すれば一層速い．反面，呼出しと復帰時における手間が大きい．

3.2 静的チェイン法

第2の方法は，フレーム間にチェインすなわちポインタをもつというものである．スタック内の各フレームにおいて，そのプログラム・テキスト上の親，つまりそのフレームの手続きを含むもっとも内側の手続きに対応するフレームで（これは複数個ありうるが），もっとも新しくスタック上に作られたものへのポインタをもつようにする．これを**静的チェイン** static chain という (**図-4**)．

プログラムの実行中に大域的な変数への参照が生じたときは，現在のフレームのレベルとその変数の宣言された手続きのレベルとの差の分だけチェインを溯って，その大域的変数のあるフレーム（の先頭位置）を見つける（ようなコードを生成する）．

この方法はディスプレイ法に比べて，変数のアクセスの時間はかかるが，呼出しと復帰時における手間は小さい．

4. 動的な関数定義をもつ言語[2),9)]

LISP, APL, SNOBOL においては関数はプログラム実行時に動的に定義され，また変数の意味も動的に決定される．この章では話を LISP に限る．また再帰呼出しを実現するシステムはインタプリタであると想定する（本質的ではない）．

4.1 深い変数束縛

LISP 1.5[9)], INTERLISP で用いられている方法である．プログラム実行中に，変数とその値は組として，a-list (association list) と称されるものに順に積まれていく．この a-list がスタックの役割をする．

プログラム(4)では，関数 f, g, h が定義されている*．仮引数は，それぞれ，x, x, y である．関数 h の実行部の中に変数 x が現れるものとする．この x の意味は，静的な名前の有効範囲規則をもつ言語と異なって，実際に h が実行されたときにしか決まらない．

```
(defun f (x) ( … (g x) … ))
(defun g (x) ( … (h x) … ))
(defun h (y) (setq x (add1 y)))
```

プログラム(4)

すなわち，プログラム実行中に，関数 f が g を呼び，さらに g が h を呼んだ時点を考える．a-list には変数とその値の組が順に積まれていくので，この時点で **図-5** のようになっている．このとき関数 h の実行部内の変数 x の意味は，その名前を仮引数としてもつ関数のうちで，もっとも最近に呼ばれてまだ完了していないものに属するものとして定まる（この場合は g の x)．

このことをスタックにおいて考えると，変数の値は，単に変数名を a-list の現在の top から順に捜し

* これは MACLISP 風の関数定義である．システムにより，*defun* のかわりに *de* あるいは *define* を用いる．

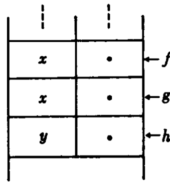

図-5 深い変数束縛 (1)

```
label [f; λ[[x; y];
         .........
         label [g; λ[[x]; .........]] [・]
         .........
         label [h; λ[[z]; .........]] [・]
         .........
    ]] [・; ・]
```

プログラム(5)

ていき，最初に見つかった，つまり top にもっとも近いものの値が現時点の値となる．これを**深い変数束縛** deep binding と称する．

例をもう1つ挙げる．プログラム(5)のような関数定義がされているとする*．プログラム実行中に，関数 f が g を呼び，g から復帰した後に f は h を呼ぶ．その後に h は f を呼び，f は再び h を呼んだものとする．このとき a-list は**図-6** のように変化する**．*label* による定義の場合は関数名とその定義内容との組も a-list 中に積まれるが，変数も関数もともに，a-list の top から順にその名を捜すことによってその値（関数定義）が得られる．

深い変数束縛の方法においては，再帰の深さが大きいとき，a-list が長くなり，大域的な変数（底の方にある変数）にアクセスするのに長い線形探索を強いられることが欠点である．

4.2 浅い変数束縛

深い変数束縛の欠点に対処する方法はいろいろ考えられているが[2]，LISP 1.6 や MACLISP で採用されているのが**浅い変数束縛** shallow binding と称するものである．これは，変数(LISP のアトム)の領域においては (value cell なるものの中に) つねに現在の値をもち，古い値はスタックに積んでおくというものである．(**図-7** は 図-6 に対応する．図中のスタックの中の名前は実際には，変数の領域中の対応する場所へのポインタである．)

* これは M 式という表現によるプログラムで，実際の実用的なシステムでは，(プログラム(4)のような) S式による表現を用いる．また *label* は実用上は用いることは少ない．

** λ式による関数定義は，関数のその場限りの使用である．*label* によってその関数に名前をつけることができるが，これは臨時的で，その関数名の有効範囲はこの λ 式の中だけである (*defun* などと異なる)．したがって，プログラム(5)で，関数 g の中から関数 h は呼出せない．もしも呼出せるとすると，図-6 の変数束縛はうまく働かない．

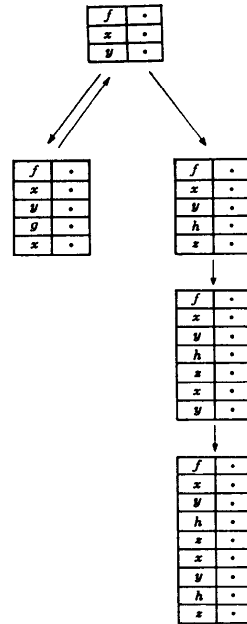

図-6 深い変数束縛 (2)

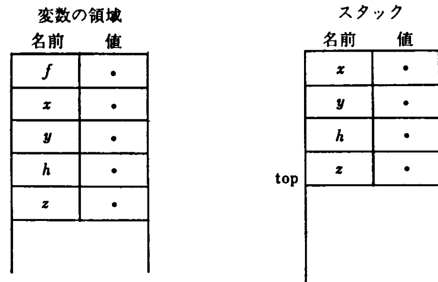

図-7 浅い変数束縛

この方法では変数へのアクセスは速い．しかし関数の呼出しと復帰時の値の更新に手間がかかる．両方法の効率の比較研究が必要である[7]．

現在実用的な LISP システムにおいては，浅い変数束縛の方式の方が主流となりつつあるようである．なお，LISP 1.5 の a-list 上でも浅い変数束縛は工夫すれば実現可能であるが[4]，実用的とは言い難い．

5. コンビネータを用いる方法*

前節までのような変数の環境を用いない，根本的に

* この節の解説はごく概略だけであり，しかも λ-calculus や combinatory logic (の初歩) の知識を必要とする．

異なる方法で，純 LISP や Backus による FP など関数型プログラミング言語に使用できるものが Turner によって提案された[12]．これは大略，以下のような方法である．第1段階（コンパイル）として，プログラムをコンビネータ（と定数，定関数）からなる式に変換する．この際，仮引数の変数を見かけ上なくしてしまえる．この式は木として表現できるが，再帰を含むときは一般のグラフとなる．次に第2段階（実行）として，入力値を与えてこのグラフを還元（実行）していってプログラムの結果の値を得る．

コンビネータ combinator とは自由変数のない λ 式のことである．とくに次の3つが基本である*.

$I \equiv \lambda x.\ x,\quad K \equiv \lambda x\,y.\ x,\quad S \equiv \lambda x\,y\,z.\ (x\,z)(y\,z).$

combinatory logic における基本的な事実として，任意の（自由変数を含まない）λ 式（つまり関数型プログラム）は，上の3つのコンビネータの積で書ける．積とは関数の実引数に対する適用 application である．例として，次の関数定義

fac *n*=**if** *n*=1 **then** 1 **else** *fac*(*n*−1) ***n**

は，関数型プログラム風な

fac=λ*n*.(*cond* (*eq* *n* 1)
1(*times*(*fac*(*minus* *n* 1))*n*))

から出発して，

fac=*S* (*C* (*B* *cond* (*eq* 1)) 1)
(*S* *times* (*B* *fac* (*C* *minus* 1)))

のようにコンビネータからなる式に直せる（この式中に *n* が現れないことに注意せよ.）

この式を，適用を頂点，コンビネータと定数，定関数を葉としてポインタでつないで木を作る．上の式のように *fac* という再帰があるときは，式中の *fac* は根へのポインタに直す．

このグラフに実引数を与えて，適用を1つずつ実行していく．これをグラフの還元 reduction という．還元の規則は次の通りである．

$Ia \rightarrow a,\quad Kab \rightarrow a,\quad Sabc \rightarrow (ac)(bc).$

この方法の利点は，高階の関数（関数を引数とする関数など）を自然に表現できることである．

6. その他の諸点

第3,4章において，手続きの呼出し・復帰以外に，手続きの外へ飛び出す goto 文（いわゆる long jump, global exit）やインタラプトにおいても，もう少し複雑な環境の更新が必要である．

言語によっては，引数として手続きや関数を渡すことができる（PASCAL や LISP）．この場合の環境はスタックでは実現できず，木構造が必要になる．LISP 系の言語に対しては，文献 2),5),8) を見られたい．第5章の方法ではこの問題は起こらない．

* Turner の実際のインプリメントでは，実行効率上の観点から，後出の *B*, *C* など他のいくつかのコンビネータも用いる．ここで
$B \equiv \lambda x\,y\,z.\ x(y\,z),\quad C \equiv \lambda x\,y\,z.\ (x\,z)y.$

参 考 文 献

1) Aho, A. V. and Ullman, J. D.: Principles of Compiler Design, Addison-Wesley, Reading, Mass. (1977).
2) Allen, J. R.: Anatomy of LISP,McGraw-Hill, New York (1978).
3) Arisawa, M. and Iuchi, M.: Debugging methods in recursive structured FORTRAN, Softw. Pract. Exper, Vol. 10, pp. 29-43 (1980).
4) Baker, H. G., Jr.: Shallow binding in Lisp 1.5, Comm. ACM, Vol. 21, pp. 565-569 (1978).
5) Burge, W. H.: Recursive Programming Techniques, Addison-Wesley, Reading, Mass. (1975).
6) Dijkstra, E. W.: Recursive programming, Numer. Math., Vol. 2, pp. 312-318 (1960).
7) 黒川利明：変数の浅い束縛について，情報処理学会論文誌，Vol. 20, pp. 524-526 (1979).
8) Landin, P.: The mechanical evaluation of expressions, Computer J., Vol. 6, pp. 308-320 (1964).
9) McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. I.: LISP 1.5 Programmer's Manual, second ed., M. I. T. Press, Cambridge, Mass. (1965).
10) Randell, B. and Russel, L. J.: ALGOL 60 Implementation, Academic Press, London (1964).
11) Sassa, M.: A poor man's realization of recursive structured FORTRAN, SIGPLAN Notices, Vol. 16, No. 5, pp. 43-53 (1981).
12) Turner, D. A.: A new implementation technique for applicative languages, Softw. Pract. Exper., Vol. 9, pp. 31-49 (1979).
13) Warren, D. H. D., Pereira, L. M. and Pereira, F.: PROLOG—The language and its implementation compared with LISP, Proc. Symp. on Artificial Intelligence and Programming Languages, SIGPLAN Notices, Vol. 12, No. 8, pp. 109-115 (1977).
14) Wirth, N.: The design of a PASCAL compiler, Softw. Pract. Exper., Vol. 1, pp. 309-333 (1971).

(昭和57年12月3日受付)