Recursive Procedure の解析*

— 番地割付法統論 —

野　崎　昭　弘**

1. 序

筆者は，さきに発表した論文[1]において，番地割付の諸問題を考察し，特に local でしかも dynamic な変数の処理について，新しい方法（先行変数法）を提案した．ただ，1）では主プログラムの中での問題だけを主として取扱い，procedure body の中で定義される変数の処理については，ほとんど何も述べなかった．そこで今回はその点を補い，recursive call その他を許したとき，どのような問題が生ずるかを解析し，手法を選択するときの指針を考察した．

結論として，最も一般性のある方法は，ある意味で Naur らの方法（**go to** の administration）であることがわかった．しかし，先行変数法は最も簡単であり，run time での能率もよいので，できるだけ応（併）用することが望ましい．そこで応（併）用の範囲・条件についても論じた．

2. 問　　題

問題は local な変数の定義・引用のしかたである．念のために，今後留意すべき問題点を列挙しておこう．

2.1. recursive call が許される．

2.2. procedure からの脱出は，標準の exit によるだけでなく，go to statement による脱出もありうる —— abnormal exit の存在．

2.3. procedure body の中に，再度 procedure declaration が含まれていてよい．

2.4. actual parameter として，procedure name や label を用いることができる．

注意 formal parameter を経由する go to によれば，ある label へ，その label の scope の外から飛び越すということが起こりうる．それゆえ，**2.4.** の影響は重大である．（scope の概念の破壊）．

3. 先行変数法の一変形

次に，以下の考察の基礎となる，先行変数法の一変形を述べよう（これは 1）p. 316 **注意**に述べた“折衷案”の精密化にほかならない）．

各ブロックBに一つずつ単純変数 L_B を与えておく（procedure は，つねにブロックとみなす）．L_B の対応番地は，B の compile にはいった時点（もちろん処理段階）で，その他のS型変数とともに，絶対的にあるいは相対的に，定めることができる．

L_B は，L型D変数の番地割付のためのコントロール・ワードであって，“すでに使用されている記憶場所を除いた，最初のアキ番地”という意味をもっている（1）の記号では，L_p）．L_B を用いると，L型D変数の定義は，次のようにして実行できる．

I．ブロックBの入口で，次の作業を行なう．

1) Bがプログラム全体である場合：

$L_B:=$適当な定数，たとえば

＜プログラム終了位置＞$+1$

2) Bがある procedure 全体である場合：

$L_B:=L_{B'}$

ただし，B′はBを call した命令の所属するブロックである．この作業を行なうために，$L_{B'}$ の値は，call に際して，たとえば program parameter の一つとして，B側に渡される．

3) その他の場合：$L_B:=L_{B'}$

ただし，B′は，$B' \supsetneqq B$ で，しかもコの意味で最小のブロックである．（そのような B′ は compile の段階で unique に決定できる）

II．L_B に基づいて，L型D変数*の番地割付を，割付の基本式によって行なう（cf〔1〕）．

$$A\,[\underline{b_1}, \cdots\cdots, \underline{b_n}]^* := L_B;$$

$$L_B := L_B + \Pi_{k=1}^{n} c_k$$

* Recursive Procedure Analysis, by Akihiro Nozaki (University of Tokyo)

** 東京大学教養学部基礎科学科

* Lは local を，Dは dynamic を意味する．Sは static の意である．S, D は，compile の段階で情報量が定まるか否かによって区別する（cf〔1〕）．

ここで，Aは array name, $\underline{b_i}$ は lower boundary, A*は A に対応づけられる address をあらわす．

このような方法で，変数の定義が，L_B さえ正しく引用されれば，正しく行なわれる．また，D変数の引用は，コントロール・ワード（S変数）を経由して行なわれるのであるから，けっきょくS変数の引用が正しく行なわれればよい．次節からその問題の考察に移ろう．

注意 この方法で，主プログラム内の変数の定義・引用が正しく行なわれることは，1）に証明したとおりである．したがって，先行変数法の併用により，**少なくとも**，主プログラムの中の **go to**（あるいは label）の管理の手間は省くことができる．

2.1. （すなわち recursive call）を禁止した場合には，S変数の定義・引用法は，すでによく知られている（たとえば 2）第 4 節に正確な記述がある）．したがって，先行変数法を全面的に適用することができる．

4. 予備的考察

便宜上，主プログラムの中で宣言されている（直接引用できる）procedure のことを，一次の procedure と呼ぶことにしよう．n 次 procedure とは，$(n-1)$ 次の procedure の中で宣言されている，subprocedure のことである．

procedure P の次数を，deg (P) であらわす．主プログラム Π に対しては，deg (Π)=0 と定めておくとつごうがよい．

さて，recursive call を許すと，よく知られているとおり，procedure body で定義された変数の引用に際して，インデックス・レジスタなどによる修飾は不可避的である．やや詳しく述べるならば，手法の大要は次のとおり：

1) procedure の中で定義される変数については，その中だけの（他の procedure とは独立の）相対番地を与えておく．

たとえば s 語の変数が定義されているとすれば（今はS変数だけを考えているから，s は定数である），それらに 0, 1, ……, $s-1$ なる番地が与えられる．

2) これらの変数を引用する命令は，その相対番地に，ある修飾用レジスタによる修飾（簡単のため加法とする）を加えることによって，実効番地を定めるようにしておく（semi-static）．

3) これらの変数のために，記憶場所としては，D型変数のための領域を用いることにしよう．したがって，番地割付は，L_B を媒介にして行なわれる．修飾用レジスタの内容（相対番地の原点）は，L_B' に基づいて定められる．

変数の引用が正しく行なわれるためには，修飾用レジスタの内容の administration が正しく行なわれなければならない——**問題はここに集約される**．

修飾用レジスタは，一つでは済まされない．実は，少なくとも，次数ごとに一つずつないと，意味がない（またそれだけあれば十分なことが，以下の所論から明らかにされる）．そこで n 次 procedure で定義される変数のための修飾用レジスタを，M〔n〕であらわすことにしよう．ここで，次の注意が重要である．

変数を引用するレジスタは，それを引用している場所に関係なく，**それが定義された procedure の次数だけによって定まる**．たとえば変数 x が procedure P のあるブロックで定義されており，しかも，そのブロックに subprocedure Q が含まれていたとしよう．その場合，P, Q いずれにおいても，x を explicit に引用できる．しかし，それらの場合を通じて，x を修飾するレジスタはつねに M〔deg(P)〕である．

変数が，call by name のパラメータによって，implicit に引用される場合には，actual parameter を与えるときに，その絶対番地を与えるようにするとよい（formal parameter を引用する命令は，適当な場所に受け入れられたその絶対番地を，たとえば indirect に用いればよい）．そのようにすれば，compiler の負担はきわめて軽い．ただし，それが許されるかどうかには，ALGOL 文法の解釈問題がからんでくる（後に述べる）．

5. 予備的考察（続）

これまで直観にたよって議論を進めてきたが，さらに考察を進めるためには，問題の適当な形式化が望ましい．まず，そのための準備として，次のような記号系を導入しよう．

計算段階で，主プログラム Π から call が始まり，再び Π のどこかに link してくるまでの全過程を，たとえば次のようにあらわす．

$$\Pi \to P \to Q \to R \Rightarrow P * \Pi$$

ただし，P, Q, R はある procedure, → は call, ⇒ は go to による abnormal exit, * は normal exit をあらわす．一般に：

$$P_0 \Delta_0 P_1 \Delta_1 \cdots\cdots \Delta_{n-1} P_n \Delta_n P_0 \quad \cdots\cdots\cdots\cdots (*)$$

ただし，

1) P_i はある procedure, $P_0=\Pi$.

2) Δ_i は→, ⇒, ＊のいずれか.

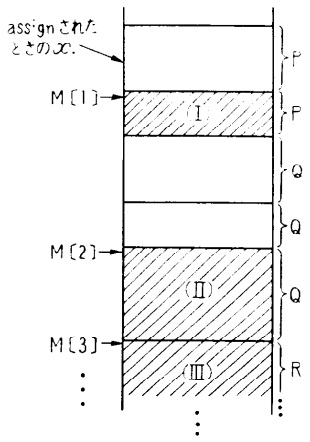
もちろん $\Delta_0=<\rightarrow>$, $\Delta_n=<\Rightarrow>$ または $<*>$.

また，あきらかに $\deg(P_1)=1$.

今後，このような系列（およびその部分列）を利用して，表現の明確化を図ることにしよう．まず簡単な応用から始める．

$\Pi \rightarrow P \rightarrow P \rightarrow Q \rightarrow Q \rightarrow Q \rightarrow R$　　$(P \supset Q \supset R)$

なる系列を考えよう．そして，現在 R が働いているものとする（計算段階）．そのとき，記憶場所の中での変数の配置は，**第1図**のようになっている．（斜線部分は，R において引用しうる変数の，current location である）．



第1図

M[i] はそれぞれの次数の相対番地の原点をあらわす．
同じ P, Q でもコントロールが進んだブロックの深さによって，割付けられる記憶容量が異る．

この例に示されるように，M[1], M[2], M[3] の相互の差は，コントロールがRに進むまでの経過（系列）に依存し，一定でない．したがって，図中のⅠ～Ⅲ部分の引用を単一の修飾用レジスタで制御するのは，ほとんど不可能である*.

次に，同じ系列を，別の角度から眺めてみよう．記号をあらためて，次のように書く．

$\Pi \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5 \rightarrow R$

Rの中に call by name の parameter あったとしよう．その parameter には，P_5 からの call に際して何かが assign されているはずであるが，それがまた call by name の formal parameter であることもあり得る．しかし，それにしても，系列の長さは有限であるから，assign の過程を逆にたどれば，どこかで具体的な（対応番地の確定している）variable —— プログラマによって宣言された変数か，call by value の formal parameter —— にぶつかるはずである．今それが，P_1 の中で定義された変数 x であったとしよう*.

さっきの方針では，x は絶対番地によって引用するのであった．その場合，その絶対番地が修飾なしに順送りされたとすると，R の中で x が（implicit に）用いられたとき，用いられる番地は第1図の矢印で図示される領域に属しており，current location（Ⅰ）の外部にある．

一方，x が R から explicit に引用できる位置にあるとき（必ずしもそうとは限らないが），引用を explicit に行なうと，Ⅰ．の中のどこかが使用されることになる．

このように，引用のしかたによって，対応番地が異なる点に，疑念をもたれる方もあるかも知れない．そこで，第2の方法として，絶対番地の代わりに，相対番地と次数とを与える方法も，考慮すべきかも知れない．

この点について，ALGOL 文法には正確な指定がないようであるが，‘引用のしかたで対応番地が変わるのは当然’というのが私の解釈である．その理由は，次の二つである．

1) current location（Mの内容）は，系列に依存して変わる．したがって，第2の方法によると，x の対応番地も系列に依存して変わる——しかし私の感覚では，ある具体的な変数を actual parameter として assign したときには，その意味（対応番地）はその時点で確定すべきであって，その後の系列には依存しないことが望ましい．

2) 第2の方法だと，x が未定義であることがあり得る．たとえば，x が P の innermost block で定義される変数で，$P_2(=P)$ はそこから call されるのであるが，P_2 から $P_3(=Q)$ への call は，そのブロックにはいらないうちに起こったとしよう．そのような場合は，第2の方法によっているときは，誤りとみなさざるを得ない（無意味なことしかできない）．先の方法によれば，そのような問題は絶対に起こらない．

---

* 実質的に修飾用レジスタなしでやるのと変わりない．ただし，ソフトウェアで多数のレジスタを設けるのなら，話は別である．

* 定義という言葉は，ここでいう‘具体的な’変数すべてに対して用いる．

今後，なお論点はあろうが，一応以上の解釈に従って，議論を進めることにする．

以上で，記号の意味内容に馴れられたことと思うので，次節から公理論的取扱いに移ろう．

6. 系列の代数学

$\sum$はある集合族である．$\sum$は半順序 $\supset$ をもっている．

deg(P) は，$P\varepsilon\sum$ に対して定義された，ある整数値函数である．

公理 1.　$\sum$は $\supset$ に関する最大元 II をもち，しかも

$$\deg(\Pi)=0$$

公理 2.　$P, Q\varepsilon\sum$. $P\cap Q\neq\phi$, かつ

$\deg(P)\geq\deg(Q)$ ならば $P\subset Q$

Cor 1.　$P\supset Q$ ならば $\deg(P)\leqq\deg(Q)$

Cor 2.　$P\cap Q\neq\phi$,　$\deg(P)=\deg(Q)$ ならば

$P=Q$

定義　$\Gamma=\{\rightarrow, \Rightarrow, *\}$

定義　$\sum$ sequence とは，次の手順によって得られる系列のことである．

1)　$P\varepsilon\sum$ なら，P は $\sum$ sequence である．

2)　α, β が $\sum$ sequence で，$\Delta\varepsilon\Gamma$ ならば，$\alpha\Delta\beta$ も $\sum$ sequence である．

今後の研究対象は，$\sum$ sequence のすべてではない．$\sum$ sequence のうち，次の公理をみたすものである．

公理 3.　$P\Rightarrow Q$ なる任意の部分列について，

$P\supsetneqq Q$.

公理 4.　$P\rightarrow Q$ なる任意の部分列について：

1°)　$\dim(Q)<\dim(P)$ ならば $Q\supset P$

2°)　$\dim(Q)=\dim(P)$ ならば，

任意の $R\supsetneqq P$ につき，$R\supsetneqq Q$.

3°)　$\dim(Q)>\dim(P)$ ならば，

$P\supset Q$，しかも $\dim(Q)=\dim(P)+1$.

Cor 1.　$P\rightarrow Q$ ならば $\dim(Q)\leqq\dim(P)+1$.

今後，上記公理をみたす $\sum$ sequence のことを，単に sequence（あるいは**系列**）と呼ぶことにする．

これらの公理系をみたす実例としては，次のようなものがあげられる．

例 1　$\sum$ を主プログラムの中のすべてのブロックの集合，deg(B) をそのうえで定義される‘ブロックの次数’とする．

$\rightarrow$ をブロックへの **enter**，$*$ を **end** を経由する標準の **exit**，$\Rightarrow$ を **go to** による **exit** とする．

そのとき，計算段階でのブロックの推移は，$\sum$ sequence によってあらわされるが，それは上記公理系をみたす（identifier の scope を考えれば，明らか）．

例 2　$\sum$ をすべての procedure の集合とし，その他の記号の意味は前節のとおりとする．その場合，**2. 4.**（すなわち，actual parameter として procedure name や designational expression を用いること）を禁止するかぎり，前節に述べた系列（$*$）は，上記公理系をみたす（scope の概念が保存されているから，明らか）．

注意　系列についての公理 1～4 は，前節に述べた具体的な系列（$*$）の性質の，すべてをあらわしてはいない．

たとえば，

$P\rightarrow Q*R$ なら $P=R$

のごとき基本性質（本来の exit の定義）は，仮定されていない．しかし，1～4 だけでも基本的なことは導けるので，まずここから出発しよう，というのである．

条件‘**2. 4.** 禁止’は必要である．これをゆるめた場合は，後に論ずる．まず，見通しをよくする意味で，この条件の枠内の問題を論じておこう．

lemma 1　$P_0\rightarrow P_1\rightarrow P_2\rightarrow\cdots\cdots\rightarrow P_j$ において，

$\deg(P_0)\leqq k<\deg(P_j)$ ならば，

$0\leqq i<j$, $\deg(P_i)=k$

なるごとき i が，少なくとも一つある．

（証）　$0\leqq s\leqq j$ なる整数 s について，次の条件を考える．

“$0\leqq h\leqq s$ なる任意の h につき，$\deg(P_h)\leqq k$”

明らかに，0 はこの条件をみたし，j はみたさない．そこで，この条件をみたす最大の整数を t とすると，

$0\leqq t<j$　で，しかも

$\deg(P_t)\leqq k$, $\deg(P_{t+1})\gneqq k$

しかるに**公理 4**（Cor 1）によれば，

$\deg(P_t)\geq\deg(P_{t+1})-1>k-1$

$\therefore\ \deg(P_t)=k$　　（証終）

lemma 2　前 lemma と同じ条件の下で，

$\deg(P_t)=k$, $P_t\supset P_j$ なるごとき t がある．

$(0\leqq t<j)$

（証）　$\deg(P_i)=k$ なる $i(<j)$ のうち，最大のものを t とする．そうすると，$t<h\leqq j$ なら

$\deg(P_h)>k$

である．なぜなら，$\deg(P_h)\neq k$ は t の定義から明らかであり，また，もし $\deg(P_h)<k$ とすると，

$P_h \rightarrow \cdots\cdots \rightarrow P_j$

に lemma 1 を適用して，

$h<h'<j$, $\deg(P_{h'})=k$ なる h' が少なくとも一つあることになる．しかし，それは t の定義（最大）に反する．

さて，$t<s\leqq j$, $P_t \supset P_s$ なる s が仮にあったとして，そのような s のうち，最小のものを r としよう．すぐにわかることは，$r>t+1$ である．

なぜなら，$\deg(P_t)<\deg(P_{t+1})$ から，**公理4**（3°）により，$P_t \supset P_{t+1}$ である．

次に，$P_{r-1} \rightarrow P_r$ $(r-1>t)$ に公理4を適用して考えてみる．

1°）の場合：$P_{r-1} \subset P_r$．$\therefore$ $P_t \cap P_r \supset P_{r-1} \neq \phi$ しかるに（1）から，$\deg(P_t)<\deg(P_r)$

よって公理2により，$P_t \supset P_r$

2°）の場合：$R_t \supset R_{r-1}$ であるから，公理の主張から $R_t \supset R_r$ が得られる．

3°）の場合：$P_r \subset P_{r-1} \subset P_t$

けっきょく，いずれの場合にも $P_r \subset P_t$ となり，P_r の定義と矛盾する．したがって；

'すべての $t<r\leqq j$ について，$P_t \supset P_r$.'

これは十分な結果である． （証終）

lemma 3 $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \cdots\cdots \rightarrow P_j \Rightarrow P_{j+1}$

において，$P_0=\Pi$ なら，

$P_i=P_{j+1}$, $0\leqq i<j$

なるごとき i が少なくとも一つある．

（証） **公理3**によれば $P_j \subsetneqq P_{j+1}$，したがって公理から，

$\deg(P_j)>\deg(P_{j+1})$

$\therefore$ $\deg(P_0)\leqq k<\deg(P_j)$

それゆえ lemma 2 により，

$P_i \supset P_j$, $\deg(P_i)=k(=\deg(P_{j+1}))$,

$0<i<j$ なるごとき i がある．ところが，

$P_i \cap P_{j+1} \supset P_j \neq \phi$ であるから，公理2（Cor 2）と合わせて $P_i=P_{j+1}$ を得る． （証終）

定義 次の形の系列は，**閉じている**という．

$P \rightarrow P_1 \rightarrow P_2 \rightarrow \cdots\cdots \rightarrow P_j \Rightarrow P$

$P \neq P_i$ $(1\leqq i\leqq j)$

定義 任意の系列において，その中の閉じている部分列を，その終端 P でおきかえることを，

'⇒ の消去' という．

たとえば

$P \rightarrow Q \rightarrow P \rightarrow Q \rightarrow R \Rightarrow P$ （1）

において，⇒ を消去すると，

$P \rightarrow Q \rightarrow P$ （2）

が得られる．

このことは，内容的には，exit ⇒ が，第2の P（current level）に向かって行なわれることを主張しているのである．（ついでながら，（1）式と（2）式とで，関係する M の内容は一致していなければならない．その意味で，（1）式と（2）式とは '等価' である）．

定義 $P \rightarrow Q * P$ なる（部分）列もまた，閉じているといい，これを P でおきかえることを，'＊の消去' という．

lemma 4 σ が公理系を満足する系列であれば，その中のあるまたはを消去して得られる系列もまた公理系を満足する．（証明は，不必要であろう）

定義 次の性質をみたす sequence を，complete admissible sequence という．

1） その中のすべての ⇒ および ＊ を，左から順次一つずつ，消去することができる．

2） しかもその結果 Π のみから成る sequence が得られる．

重要なのは，complete admissble sequence あるいはその部分列に，消去の操作を施して得られる系列（admissible sequence）の諸性質である．たとえば，'＊の消去' の定義からすぐわかるように，

$P \rightarrow Q * R$ ならば $P=R$

が，任意の admissble sequence において成り立つ．

計算段階に実際あらわれ得る系列（前節の（＊））は，complete admissible なることによって特徴づけられる．それは，次の二つのことに注意すれば，明白であろう．前節に述べた，実際の系列（＊）において：

A） $P \rightarrow Q * R$ ならば，$P=R$

これは，本来の＊の定義である．

B） $\Pi \rightarrow P_1 \rightarrow P_2 \rightarrow \cdots\cdots \rightarrow P_j \Rightarrow P_{j+1}$ は，P_{j+1} を終端とする閉じた部分列を含む．

このことは，以前の lemma から証明できる．

（証） lemma 3 から，$0\leqq i<j$, $P_i=P_{j+1}$ なる i が少なくとも一つある．そのような i のうち，最大のものを h とすれば，

$P_h \rightarrow P_{h+1} \rightarrow \cdots\cdots \rightarrow P_j \Rightarrow P_{j+1}$

は閉じている． （証終）

B の証明に，前の lemma（したがって **2.4.** の禁止）を必要としたことに，注意すべきである． **2.4.** を許した場合，closedness（つまり exit の性質）がどのように formulate されるかは，自明ではない．

次節において，この場合（**2.4.** の禁止）における M〔*n*〕の管理法を述べる．それは最も一般の場合の管理法の，基礎となるものである．

7. 簡単な場合

方法の骨子は，**第3節**で述べた，L_B を用いる先行変数法である．ただ，次の点だけつけ加える．

（I） deg(P)=*m*, deg(Q)=*n*, *m*≥*n* とする．

その場合，

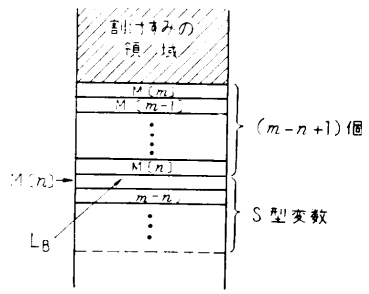
P→Q

の実行に際して，

M〔*n*〕, M〔*n*+1〕, ……, M〔*m*〕　　（1）

の内容を保存しておく．詳しく書けば次のとおり．

```
procedure Entry (L, m, n);
value L, m, n;
comment m次の procedure から，n次の procedure を call するときに用いられる．L は，call 側のブロックの，LB の値である．;
begin integer i;
    if m<n then go to Block Entrance;
    for i:=0 step 1 until (m-n) do
        Memory〔L+i〕:=M〔m-i〕;
```



第2図

M〔*n*〕は相対番地の原点をあらわす．
L_B はこの procedure に attach されている，第3節で述べた変数．

```
    L:=L+(m-n+1);
Block Entrance:
    M〔n〕:=Memory〔L〕:=L;
    Memory〔M〔n〕+1〕:=(m-n) end
```

なお，Memory〔*l*〕は *l* 番地の記憶レジスタをあらわす．情報の配置については，**第2図**参照．

（II） 同じ場合（*m*≥*n*），

Q＊P

の実行に際して，（1）の値を復旧する．

```
procedure Exit (n);
value n;
comment n次 procedure からの，normal exit に際して用いられる;
begen integer i;
    if Memory〔M〔n〕+1〕<0 then
        go to end;
    for i:=0 step 1 until Memory〔M〔n〕+1〕
    do M〔n+i〕:=
        Memory〔M〔n〕-i-1〕;
    end end
```

注意　この手法は，Watt〔3〕に述べられている．しかし Watt は，その正当性について，何の証明も行なっていない．彼は，条件 '**2.4.** の禁止' を見落していたようである．

この方法は，一見繁雑のようであるが，I，II の場合に → と＊との管理を行なうだけで，めんどうな go to administration や，能率の悪い label administration を**行なわずにすむ**．次にそのことを証明しよう．

complete admissible sequence

$$\mathrm{II}\,\Delta_1 P_1 \Delta_2 P_2 P_2 \cdots\cdots \Delta_n P_n \Delta_{n+1} \mathrm{II} \cdots\cdots \quad (2)$$

を考える．

1°）現在 P_i が process されているとしよう．そのとき用いられる変数は，$P \supset P_i$ なる P で定義された変数に限る（必要条件）．例外は formal parameter であるが，それは絶対番地で process されるから，M〔*n*〕に関する以下の議論とは無関係である．したがって M〔*n*〕（*n*≦deg(P_i)）の値が正しくさえあれば，問題はない．要するに，

P＊Q や P⇒Q

に際して，M〔*n*〕（*n*≦deg(Q)）の値が正しく復旧されればよい．このことをきちんと述べてみよう．

M〔*n*〕は，（2）における添字（時間ノ）の関数であるから，その点を明示するために，M〔*n*, *t*〕と書くことにしよう．そうすると，証明の目標は，次のように表現される．

（A） $P_{j-1} \to P_j * P_{j+1}$ のとき（$P_{j-1}=P_{j+1}$），

$$\left.\begin{aligned} &M〔1, j-1〕=M〔1, j+1〕\\ &M〔2, j-1〕=M〔2, j+1〕\\ &\qquad\vdots\\ &M〔k, j-1〕=M〔k, j+1〕 \end{aligned}\right\} \quad (3)$$

ただし *k*=deg〔P_{j-1}〕

（B） $P_i \to P_{i+1} \to \cdots\cdots \to P_j \Rightarrow P_{j+1}$　　（4）

が閉じているとき ($P_i=P_{j+1}$),
(3) 式が, ($j-1$) を i におきかえて, $k=\deg(P_i)$ として成り立つ.

(Aの証明) $k'=\deg(P_j)$ とおくと, 明らかに

$$M[1, j-1]=M[1, j]=M[1, j+1]$$
$$\vdots$$
$$M[k'-1, j-1]=M[k'-1, j]=M[k'-1, j+1]$$

ゆえに $k<k'$ のとき, (3) 式が成り立つのは明らか. $k\geq k'$ のときは, 少なくとも $M[k', j-1]$ は破壊されてしまうが, それ以前に

$$M[k', j-1], \cdots\cdots, M[k, j-1]$$

は procedure Entry によって待避させられ, * の際に復旧させられる. ゆえに, この場合にも (3) は成り立つ.

(Bの証明)

lemma 5 任意の閉じた系列 (4) について,

$$\deg(P_i)<\deg(P_h) \text{ (ただし } i<h\leq j)$$

(証) $P_i=P_{j+1}\supsetneqq P_j$ (公理3) だから,

$\deg(P_i)<\deg(P_j)$ は明らか.

$\deg(P_i)\geq\deg(P_h)$ なる $h(i<h<j)$ があったとする. そうすると, lemma 2. から,

$$h\leq s<j,\ \deg(P_s)=\deg(P_i),\ P_s\supset P_j$$

なるごとき s がある. そうすると,

$$P_i\cap P_s\supset P_j\neq\phi$$

であるから, 公理2 (Cor 2) から $P_i=P_s$.
これは $P_i\neq P_h$ $(i<h\leq j)$ という, closed の定義に反する. (証終)

この lemma からわかるように, $M[n]$ の値は, $n\leq\deg(P_l)$ なるかぎり, (4) の全過程を通じて, 保存されている. (Bの証明終)

あらかじめ断っておくべきことであったが,

$$P\rightarrow Q$$

に際して, $M[n]$ $(n\leq\deg(P), n<\deg(Q))$ の値は影響をうけない. (A), (B) の証明に際しては, このことだけを用いている. また, この性質は, 消去の操作を何回繰り返しても保存される. それゆえ (A), (B) の証明も, 消去の操作を何回繰り返しても有効である. (このことは, たとえば $P\rightarrow Q$ のとき, いつでも

$$M[\deg(Q)]$$

だけを待避させる方法だと, 成立しない)

8. 一般の場合

次に, **2. 4.** を許した場合について, 考える.

まず, parameter として, label が用いられた場合を考察しよう. (一般に, designational expression が用いられた場合を考察すべきであるが, 式の評価 (evaluation) はできたものとして, 一つの label が確定したところから出発しよう).

もう一度, **第5節**で用いた系列

$$\Pi\rightarrow P\rightarrow P\rightarrow Q\rightarrow Q\rightarrow Q\rightarrow R \qquad (1)$$

について考えてみる.

今, Rの中で, 命令 **go to X** が用いられたとし, Xは formal parameter, その actual parameter は2番目のQで assign された label A だったとする. (それが3番目のQの, ある formal parameter を経由して, Xに assign されたのである). そのとき, exit は3番目の Q に向かってではなく, 2番目のQに向かって行なわれるべきである*. したがって, その結果をあらわす系列は,

$$\Pi\rightarrow P\rightarrow P\rightarrow Q\rightarrow Q \qquad (2)$$

となるべきである. (その理由は, call by name の変数について, 第5節で述べたとおりである).

注意 この解釈に従うかぎり, ISSP の方法 (label の administration) は, そのままでは応用できない. Rから explicit に label を用いて飛ぶか——その場合はつねに3番目の Q に向かって飛ぶ—— formal parameter によって飛ぶか—— original に assign された場所による——によって異なりうるので, それは label のところだけで眼を光らせていてもわからない.

(2) を実現するためには, formal parameter による go to の管理だけを行なえばよい. そのためには, compile の段階での作業は何もいらないので, 計算段階で次の作業を行なえばよい.

(I) label を actual parameter として assign するときに, その location だけでなく, その call の所属する procedure の次数 k, および

$$M[1], \cdots\cdots, M[k] \qquad (3)$$

の値を, 組にして (program parameter のように) 渡す. ただし, その label が formal parameter である場合には, すでに渡されている k の値, および (3) の値を, 順送りする.

注意 k は label に依存せず, call の場所 (命令の位置) によって定められる. それゆえ compiler の負担はきわめて軽い.

---

* もちろん '2番目' と '3番目' とのちがいは, {M[n]} の状態である. 飛越先の番地そのもののちがいではない.

（Ⅱ） formal parameter を経由する **go to** を実施する際に，まずその label に attach されている情報によって，{M[n]} の復旧を行なう．しかる後に飛越を行なう．

（3）のすべてを attach（時には順送り）する必要は必ずしもない．どこまで戻ればよいかがわかっていれば，必要な復旧は，procedure Exit の反復適用によって，行ない得るはずだからである（Exit は，飛越命令を含んでいない）．そこで，（3）の代わりに，系列の中での位置をあらわす情報（たとえば Watt のいわゆる dynamic level）を渡すことにしてもよい．

このような go to 管理が，それ以外の場合についての前節の方法の妥当性に影響しないことは，明らかであろう（つまり，併用が可能である）．

次に，parameter として，procedure name が用いられた場合の考察に進もう．

（1）において，Rが，Qの中の subprocedure で explicit に用いられていたとする．その場合，第5節に述べた意味で current な項は，太字で示せば，次のとおりである．

$$\Pi \rightarrow \mathbf{P} \rightarrow \mathbf{P} \rightarrow \mathbf{Q} \rightarrow \mathbf{Q} \rightarrow \mathbf{Q} \rightarrow \mathbf{R}$$

他方，Rが formal parameter を経由して call された，2番目の Q→ のときに assign されたものとする．その場合，以前述べた理由（特に第2の理由）から，current な項は次のようになるべきである．

$$\Pi \rightarrow \mathbf{P} \rightarrow \mathbf{P} \rightarrow \mathbf{Q} \rightarrow \mathbf{Q} \rightarrow \mathbf{Q} \rightarrow \mathbf{R}$$

したがって，Rの process 中に Q への飛越命令が出てきたら，それは第2の**Q**への link になり，その結果は（2）と同じ系列（{M[n]}）を与える．

ただし，Rからの normal link は，やはり第3の**Q**に行なわれると考えるべきであろう．第3の**Q**は，cancel されるわけではない．

このような {M[n]} 管理を実現するためには，計算段階で，次の作業を行なえばよい．

（Ⅰ′） procedure name を actual parameter として assign するときに，その location だけでなく，その call の所属する procedure の次数 k，および

$$M[1], \cdots\cdots, M[k] \qquad (4)$$

の値を，組にして渡す．

（Ⅱ′） j 次の procedure の中で，formal parameter による calling を行なうときには，

$$M[1], \cdots\cdots, M[j] \qquad (5)$$

の値を渡し（待避），しかる後（4）によって {M[n]} を変更し，しかる後に procedure ‘Entry’ を適用する．ただし，その際，call 側の procedure の次数としては，kを用いる．

（Ⅲ′） 上記の場合，normal exit に際しては，（5）によって {M[n]} の状態を復旧する．

これはかなりめんどうであるが，この種の call はまれにしか起こらないであろうから，すべての go to（あるいはすべての label）に administration を行なうよりは，時間のロスはずっと少ないと思う．

Ⅰ′～Ⅲ′を用いた場合，前節の方法にどのような影響が及ぶかは，考えておかなければなるまい．そのために，次の概念を導入する．

定義 $\mathbf{P} \rightarrow \mathbf{P}_1 \rightarrow \mathbf{P}_2 \rightarrow \cdots\cdots \rightarrow \mathbf{P}_i \rightarrow \mathbf{Q}$ が pseudo-closed であるとは，次の条件をみたすことをいう．

1） **Q**は，$\mathbf{P}_i$ の formal parameter を経由して，call された．

2） **Q** が explicit に assign されたのは，$\mathbf{P} \rightarrow \mathbf{P}_1$ においてである．

定義 $\mathbf{P} \rightarrow \mathbf{P}_1 \rightarrow \mathbf{P}_2 \rightarrow \cdots\cdots \rightarrow \mathbf{P}_i \Rightarrow \mathbf{Q}$ が pseudo-closed であるとは：

1） $\Rightarrow$**Q** は，$\mathbf{P}_i$ の formal parameter を経由する飛越である．

2） その parameter が original に assign されたのは，$\mathbf{P} \rightarrow \mathbf{P}_1$ においてである．

定義 pseudo-closed な部分列を，その両端をとった系列 $\mathbf{P} \rightarrow \mathbf{Q}$ または $\mathbf{P} \Rightarrow \mathbf{Q}$ でおきかえることを，implicit call または implicit exit の簡約化という．

（Ⅱ）によれば，implicit exit

$$\Pi \rightarrow \mathbf{P}_1 \rightarrow \mathbf{P}_2 \rightarrow \mathbf{P}_3 \cdots\cdots \rightarrow \mathbf{P}_n \Rightarrow \mathbf{P} \qquad (6)$$

は，その簡約化

$$\Pi \rightarrow \mathbf{P}_1 \rightarrow \mathbf{P}_2 \rightarrow \cdots\cdots \rightarrow \mathbf{P}_i \Rightarrow \mathbf{P} \qquad (7)$$

における explicit な exit（$\Rightarrow$ P）として，解釈され，実行される．（以後，$\Rightarrow$ はすべて explicit とみなす）．

$$\Pi \rightarrow \mathbf{P}_1 \rightarrow \mathbf{P}_2 \rightarrow \cdots\cdots \mathbf{P}_i \qquad (8)$$

が，implicit call を含む場合は，次のように解釈される．その中に含まれているすべての implicit call に注目し，それらを一つずつ，**右側から**簡約化する．（cf.（Ⅱ′）——簡約化の効果は，後のものが優先する）そうして得られる，それ以上簡約できない形

$$\Pi \rightarrow \mathbf{Q}_1 \rightarrow \mathbf{Q}_2 \rightarrow \cdots\cdots \rightarrow \mathbf{Q}_h \qquad (9)$$

を，既約表現という．（→ はすべて explicit call とみなしてよい）．さて，一般に，

（8）$\Rightarrow$**Q**

なる exit は，

（9）$\Rightarrow$**Q**

として，前節に述べた意味で解釈される．

幸いなことに，既約表現（9）は，公理 3，4 を満足する．したがって，**第7節**の方法が適用でき，I′～III′を併用しても混乱を生じない（悪影響がない）．

——これでわれわれの方法の妥当性が確定した．

要 約

（1） 先行変数法を基調とする，むだの少ない方法が述べられた．

（2） formal parameter による，go to および procedure call の管理は，不可欠であることがわかった．

（3） また，それ以外には，abnormal exit のための管理は何ら必要としないことがわかった．

注意 修飾用レジスタは，procedure の最高次数だけあればよい（次数とは，declaration の nesting の数であるから，それほど大きな数にはならないであろう．recursive call は何重に起ってもよい）．それにしてもレジスタの個数は有限であるから，あまり複雑なプログラムは実行できないこともあろう．それは一種のメモリー・リミットである．

たとえば，修飾用レジスタが四つしかなかったとしよう（B_1, B_2, B_3, B_4）．その場合，次の方法が便利であろう．

1) $M[1]=B_1$, $M[2]=B_2$, $M[3]=B_3$

修飾はハードウエアで行なう．

2) $M[n]$ ($n\geq 4$) は，普通の記憶レジスタで代用する．

$M[n]$ ($n\geq 4$) による修飾は，

LOAD M[*n*] TO B_4;

を行なった後，B_4 による修飾で実現する．

$n\geq 4$ なる procedure はそう多くあるまい．また，LOAD 命令も，そういちいちはさむ必要はないので，時間の点でも全体に影響を及ぼすことは少ないと思う．（[2] の命令 OP A B があれば，こういうところはもっと簡単に実現できる）．

参考文献

1) 野崎昭弘：番地割付の諸問題，情報処理 **3**, No. 6 (1962),
2) 井上謙蔵，高橋秀知，清水公子：ISSP ALGOL のコンパイラ，情報処理 5, No. 1 (1964).
3) J.M. Watt: The Realization of ALGOL Procedures and Designational Expressions, Computer Journal 5, No. 4 (1963).
4) J. Jensen, P. Mondrop, P. Naur: A Storage Allocation Scheme for ALGOL 60, Com. ACM 4, No. 10 (1961),