番地割付の諸問題*

野　崎　昭　弘**

1. 序

どのようなコンパイラでも，数量を操作する便宜としていわゆる「記号番地」の使用をゆるしている．そこでコンパイラ作製者はまず基礎的な作業の一つとして，使用される記号に対して記憶場所を割り当てる方法を考えなければならない．その方法は，記号の使い方としていろいろな自由が許されることになると，それほど簡単なものではなくなって来る．そこで種々の工夫がこらされるわけであるが，現在周知となっている方法は比較的やさしい問題についてだけであって，まだ十分一般的な手法が開発・比較・整理されてはいない．

ここではそのような認識に出発して，番地割付に関する技術の分析を行なった．記号のつかい方にかかわる文法としては十分自由と考えられるアルゴル 60 をえらんだ***．

1.1 問題の意味

番地「割付」といっても，ここに述べる意味では次のような作業をも含んでいる．

（1） ある記号（いわゆる**名札**——identifier）によってあらわされる変数に，その**値**(value)をおくべき記憶場所を割り当てる．

（2） 名札に関する情報を，その名札に対応する記憶場所に関する情報（たとえば**番地**）に変換する．

それ以外の問題，たとえば入力情報の文法的誤りを検出する方法であるとか，二次記憶装置の利用法などの考察はここでは除外される．

本文の立場を示す，ありうるいくつかの手法を比較選択するときの**前提**は，次の三つの公準に要約される．

P I どのような手法でも，そのアルゴリズムの記述が十分でさえあれば，コーディングの手間は問題としない．

P II どのような手法でも，それを実現するために要する記憶場所の容量が少なくてすむものほどよい．

P III その手法にしたがったとき，実計算時間（run time）が長くなるようなものはさけるべきである．

P I において記述が「十分」といったのは明確な表現とはいいがたい．ここでは標準的な程度の記述法としてやはりアルゴル 60 を採用し，これによる記述は十分満足すべきものと考えることにした．ただし，たとえば次のような函数の使用が許されるものとしている．

integer procedure Extract (W, A); **comment** W なる語の，A 部分をぬきだし，それを整数の値としてもつ函数．「A 部分」の定義は本文で精確に行なわれなければならない；

procedure Imbed (W, A, B);

comment W なる語の A 部分に，整数 B をかきこむ procedure である

2. 用語について

はじめにいくつかの用語の説明をしておこう．

（1） 問題の処理手順は，前処理部分（process time）と実計算部分（run time）とにわけることができる．簡単のため前者を**処理段階**，後者を**計算段階**とよぶ．P III によれば，処理段階でできるような作業は計算段階に廻してはならない．

（2） ある名札を**式**（expression）や**命令**（statement）の中で使うことを**引用**という．名札がある量をあらわすものとして**宣言**（declare）されることを**定義**とよぶ．便宜上，整数・実数は機械語 1 語の情報としてあらわされるものとし，その場合名札 I に対応づけられる記憶細胞の番地を I* とかくことにする．

論理変数を取扱う場合は，1 語とは機械語よりもっと小さな単位を意味することになるであろう．したがって「番地」の意味も弾力的に解されねばならない．しかし本文では簡単のため，整数・実数の場合に準じて説明を行なうことにしたい．

（3） 名札の**スコープ**（有意味範囲，scope）とは，その名札がそこで表わしている量が意味をもつと考えられるすべての宣言，命令，式の集合である（**第 1 図**）．

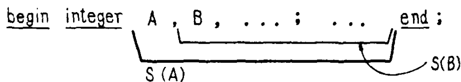
ブロック B で宣言された名札 I のスコープを

---

* Problems of Storage Allocation, by Akihiro Nozaki (Electric Communication Laboratory)

** 電気通信研究所

*** 機種についての制限は特にないが，記述は「定長」語構成のものに準じて行なっている．



第 1 図

S(I, B), または単に S(I)

とかく，この定義は報告［1］のとは異なっているから注意を要する．

（4） 番地割付の対象になる量は,次の 4 種である．

（a） **単純変数** (simple variable)

（b） **添字付変数** (array)

（c） 添字付変数に関係するコントロールワード

（d） 定　数

(a)〜(c) を一括して**変数**とよぶ，変数はまた次のように分類される．

S：情報量（占有語数）が計算段階では変化せず，一定であると考えられるもの，たとえば単純変数．

D：情報量が計算段階でも変化するもの，たとえば上下限を一般の算術式で指定された（動的に宣言された）添字付変数．

前者を **S 型** (static), 後者を **D 型** (dynamic) の変数とよぶ．なお添字付変数の全体のサイズが一定であるかどうかを判定することは一律に容易とはいいがたい．それは機種やコンパイラへの要求にもよる．それ故 S 型と D 型との文法的区別も細かい点は主観の問題に属する*.

変数に関して次の分類も有用である．

G：**own** の宣言をうけているもの．非局所的 (non-local, global) 変数，**G 型**とよぶ．

L：**own** の宣言をうけていないもの．局所的 (local) な変数，**L 型**とよぶ．

L 型変数 I, I' のスコープに関して，次のことがいえる．

lemma 1. ブロック B に属する記号全体の集合をやはり B であらわすことにする．そのとき

$S(I, B)\subset B$

lemma 2. $S(I, B)=S(I', B')$

$\longleftrightarrow B=B'$ かつ $I=I'$

lemma 3. $S(I, B)\cap S(I', B')\neq\phi$

$\longrightarrow S(I, B)\subset S(I', B')$ または

$S(I, B)\supset S(I', B')$

$((B\cap B')\neq\phi\longrightarrow B\subset B'$ または $B\supset B'$ なること，および lemma 1 に注意)

lemma 4. 計算段階のある時点 t において実行中の命令または宣言を T とする．$T\epsilon S(I, B)$ のとき，I に関する宣言は既に最も新しい意味で実行されており，しかもそれ以後変更されていない．

（I の宣言はブロック B の冒頭でなされる．しかもブロックの外から中への飛越は起りえない*. cf[1] 4.3.4.）

（5） 添字付変数の**第 k 成分の長さ** c_k とは，第 k 番目の添字に関して宣言された上限 $\bar{b}_k$ と下限 $\underline{b_k}$ との差

$\bar{b}_k-\underline{b}_k+1$ のことである．

n 次元変数 A の各成分の長さを $c_1, \cdots\cdots, c_n$ とすれば，$c=\Pi_{k=1}^{n}c_k$ は A に属する要素の総数をあらわす．これを **A のサイズまたは長さ**という．

添字式の値 $i_1, \cdots\cdots, i_n$ から対応番地を求めるのは通例次の式による．

$$A[i_1, \cdots\cdots, i_n]^*=(\cdots(i_1\times c_2+i_2)\times c_3+\cdots)\times c_n+i_n+A_0 \qquad (1)$$

または $$A[i_1, \cdots\cdots, i_n]^*=A_0+\sum_{k=1}^{n}a_k i_k \qquad (2)$$

ここで $A_0=A[0, \cdots\cdots, 0]^*$. $\{a_k\}$ は $\{c_k\}$ からきまるある定数である．これらを添字計算の基本式という．以後簡単のため専ら (2) によって説明をすすめる．

A_0 および $\{a_k\}$ は A の宣言によって決定され，番地計算のためのコントロールワードとして一括保存される．A が引用されたとき，A のコントロールワードと $i_1, \cdots\cdots, i_n$ とをうけとって（2）式の計算をしてくれるサブルーチンを作っておくとよい．コントロールワードの**長さ**（情報量）は次元 n によってきまると考えられるから，これは S 型の変数である．その所在（先頭）番地を単に A^* とかく，処理段階では名札 A を A^* に翻訳することができればよい．

3. 定数の取扱い

定数の取扱いは「定数表作成検索ルーチン」により簡単に処理できる．それによれば定数は記憶装置中の特定の場所に，表のようにまとめて記憶される．この表は処理段階で作られるが，それは同時に計算段階でもそのままの配置で利用できる**. 定数表のための記

* 参考文献 9）のように，添字付変数はすべて D 型として扱うのも一つの見識であろう．

* ブロックの入口が一つしかないことはアルゴル文法の欠陥としてしばしば指摘されるが，その制限をゆるめると影響はあまりにも広い．

** ツーパス以上のコンパイラでは，表の位置をずらして使うことも可能である．対応番地に一律に一定数を加えることをどこかですればよい．

憶場所として C_0 番地以前を予定したとき，定数表ルーチンのアルゴリスムは次のようにのべられる．

```
integer procedure Constant (a);
comment 与えられた定数 a の対応番地 a* を決定するルーチンである．処理段階中に，定数が引用される場所ごとに動作する．r 番地の内容を，ここでは CT[r] とかく，定数表の占める範囲を示すために，最初のアキ番地を示す変数 Cp が使われている．Cp の初期値は C0 である；
   begin integer r; r:=C0;
Pegel: if r=Cp then go to store;
      if CT[r]=a then go to address;
      r: =r-1; go to Pegel;
store: CT[r]:=a; Cp:=Cp-1;
address: Constant:=r end*)
```

4. S 型変数

本章ではS型の変数の取扱いを述べる．ただし procedure の中で宣言される変数は文法的に特殊な性質をもつので，後で別に取扱う．ここで単に変数といえば，主プログラムの中で宣言された変数のみをさす．

4.1 L 型 *S*-変数

L 型の *S*-変数に対しては，コンパイラはそれらが宣言された順に必要な記憶場所を割り当てていくことができる．その仕事は結局次の二つにしぼられるであろう．

（1） 定数時点で：名札 I が宣言されたところで，I の対応番地 I* をきめ，I と I* との対照表を作る．この対照表を **SVT** (static variables table) とよぶことにする．

（2） 引用時点で：I が引用されたとき，SVT を検索し，コンパイルに必要な情報を得る．

SVT の構造は，索引操作が可能でありさえすればどんなものでもよいのであるが，その各「行」の内容は次のような情報をもっていなければならない．

（a） I および I*

（b） I の性質（整数か実数か，など）に関する情報 D. I*, D は一定のビット数で表示できるが，I は一般には不定長である．簡単のためには有効字数を**末尾6文字**などと制限して**，I も一定ビット数内で表示できるようにするとよい．I* は必ずしも表の中に明示する必要はないので，検索の過程から自然と求まるようにしてもよい．その得失は PⅡ から判断すべきであるが機械語1語のビット数その他が関係する．

アキ番地． L 型 *S*-変数のために，記憶装置中の l_0 番地以後が予定されたとしよう．変数は l_0 番地以後に連続してスキ間なく割付けられることが望ましい．そのために，利用しうる最初のアキ番地 l_p (the address of the first usable storage cell, Pegel) は重要な量である．l_p の初期値は l_0 である．長さ c の L 型 *S*-変数 I が宣言されたとき，

$$I^* := l_p; \qquad (3)$$

と定められ，その後 l_p は n だけ増される．

$$l_p := l_p + n \qquad (4)$$

(3)，(4) を**割付の基本式**とよぶ．

ブロック構造． 次に引用時の処理手順中，アルゴルに特有のブロック構造の処理法について述べる．それは L 型変数のスコープをどのように規制するかという問題を考えればよいのであるが，一般的解決は [5] に極めて簡潔にのべられている*)．

「名札を定義された順に表（SVT）の中におさめ，引用の際は表を逆順に検索する．またブロックを閉じるときそのブロックに属する変数を表から抹消する」いわゆる push-down, pop-up の方法がそれである．（サブブロックの定義は親ブロックの定義に優先して適用される．cf[1]5.）表から抹消された変数に割当てられた記憶場所は，以後別の目的で使用してよい．すなわち **end** によってブロックが閉じられる場所で l_p の値をそのブロックに入ったときの値まで引戻すことができる．そこで新しいブロックに入るたびにその時の l_p の値をどこかにかきこんでおくとよい．対応する **end** が来たときその値はよみだされ，l_p は復旧される．ブロック構造が多重になる場合を考えると，l_p の値のかきこみ・よみだしは**穴蔵式** (last-in-first-out) でなければならない**．

プログラム例． 次に SVT 作成検索の一方法のアルゴリスムを示す．簡単のため，I と D とは機械語1語の中におさまるものとし，他にコンパイラ作製者が自由に使える1ビット d_0 が含ませられるとしよう．SVT の中の形を**第2図**のようにきめる．F, B, N 各

---

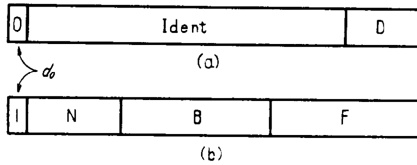
* プログラムの中まで斜体文字を使用するのは文法違反であるが，見易いかと思う．この場限りの便宜としてお許し頂きたい．

** ふつう冒頭の 5~6 文字のみ認識し，それ以後はよみとばす．しかし末尾 5~6 文字というのも一法と思う．

* 用語は本文のものにあわせてあり，原文そのままではない．別の着想については参考文献 6) 参照．ただし PⅢ からみて 5) がすぐれている．

** 括弧（**begin end**），というものは「最後に開いたものが最初に閉じる」という構造原理をもっている．

部分は，ある範囲内のアドレスを表示できるビット数があれば十分であるが，その意味は後で説明する．



第 2 図

r 番地の内容をここでは SVT[r] とかく．SVT は記憶場所の r_0 番地以後に作られるとして，現在 r_p-1 番地まで占有しているとする．

```
procedure Definition (W, n);
comment ある名札 I に関する第2図 (a) の形をした1語と，Iの長さ n をうけとり，SVT にかき加えるルーチンである．W および n の決定は宣言処理ルチーンの仕事であるが，われわれはそれには立入らない．n>1 のとき，このルーチンは第2図 (b) の形の語を作成して SVT に加える．F, B は使わない；
begin SVT[r_p]:=W; r_p:=r_p+1; l_p:=l_p+n;
if n≠1 then begin Imbed (SVT[r_p], do, 1);
Imbed (SVT[r_p], N, n−1); r_p:=r_p+1 end
  end definition;
integer procedure Refer (I) address:(I*);
integer I;
comment 名札Iにより SVT を検索し，I* と D とを求めるルーチンである；
begin integer r; r:=r_p−1; I*:=l_p−1;
LOOP:if Extract(SVT[r], d_0)=1
then I*:=I*−Extract(SVT[r], N)
else if Extract (SVT[r], Ident)=I
then go to END else I*:=I*−1;
  r:=r−1; go to LOOP;
END: Refer:=Extract (SVT[r], D) end;
procedure Block Begin;
commet l_p の値を穴蔵式に保存する (push down) ためのルーチンである．l_p は第2図 (b) B 部分にかきこまれて SVT に加えられ，その場所は stack に保存される．stack の前の内容は (b) F 部分に保存される；
begin SVT[r_p]:=0;
Imbed (SVT[r_p], d_0, 1); Imbed(SVT[r_p], B, l_p);
Imbed(SVT[r_p], F, stack);
stack:=r_p; r_p:=r_p+1 end block begin;
procedure Block End;
comment l_p の値を復旧し，一部変数を無効にするルーチンである(抹消する代り，検索開始位置 r_p を引き上げている)．また l_p の最高位置 l_max も求めている．l_max の初期値は l_0 である；
begin if l_p>l_max then l_max:=l_p;
  r_p:=stack; l_p:=Extract(SVT[stack], B);
  stack:=Extract(SVT[stack], F) end
```

4.2 G 型 *S*-変数

前節で「push-down, pop-up の方法」が使えた理由に注意しておこう．

(1) コンパイラは原プログラムを頭から逐次読んでゆくものと仮定している．そこでブロックの出入に関して **last-in-first-out が厳格に成り立つ**．

(2) L 型の変数のみを対象にしている．

(2) からして，G 型変数は別の記憶場所をあてた方が都合がよいことがわかる．専用記憶場所（たとえば g_0 番地以後）を使うことにきめてしまえば，その取扱いはきわめて容易であり，特に L 型と比べて新しいアルゴリスムは必要としない．また push-down, pop-up の操作も必要ない．記憶場所の使い方についての反省は後で述べる．

5. D 型 変 数

D 型変数とは計算段階で長さが定まらぬ量であった．

以下主プログラムで定義される D 型変数について述べる．かりに L 型 *D*-変数のためには L_0 番地以後の G 型 *D*-変数のためには G_0 番地以前の記憶場所が予定されているとしよう．それぞれの最初のアキ番地を L_p, G_p とする．そのとき**割付の基本式**は次のようにあらわされる．まず L 型のばあい；

$$A[\underline{b}_1, \cdots\cdots, \underline{b}_n]^*:=L_p; \quad (5)$$

$$L_p:=L_p+\Pi_{k=1}^{n}c_k \quad (6)$$

G 型の場合；

$$A[\bar{b}_1, \cdots\cdots, \bar{b}_m]^*:=G_p; \quad (7)$$

$$G_p:=G_p-\Pi_{k=1}^{m} c_k \quad (8)$$

したがって，L 型に準じていえば，L_p さえ正しく定められれば，A_0 は次の式から決定することができる．

$$A_0:=L_p-\sum a_k b_k \quad (9)$$

5.1 L 型 *D*-変数

L 型 *D*-変数の取扱い中本質的な部分は，L_p の操

作法である．Sattley は L_p の値を，計算段階でブロックに出入するたびに穴蔵式に復旧または記録する方法を提案した*)．それは LS-変数における l_p の取扱い法を，計算段階で L_p にあてはめる，ということである．しかし前の方法が有効であるためには 4. 2-1 の前提を要したのであるが，計算段階では **go to** によるブロック脱出ということがあって，それは一般には成り立たない．そこで P. Naur などは計算段階で飛越命令の監視を行なうルーチンを作り，どのレベルまでのブロックが脱出されるのか調べて，L_p を正しく復旧できるようにした．

しかし，そのような監視ルーチンを使うことは，P Ⅲ からして好ましくない．以下にのべる先行変数による方法はその点すぐれている．

先行変数． L 型 D-変数 A の先行変数 $F(A)$ とは次のような変数のことである．

（1） $S(A)\subsetneqq S(A')$ なる如き L 型 D-変数 A' のない場合；

$F(A)=L$．ただし L はコンパイラが創作する特別の変数である．このような A をプログラムリーダーという．

（2） $S(A)\subsetneqq S(A')$ なる如き L 型 D-変数 A' がある場合．そのような A' でスコープ最小のもの A'' が存在する(cf. lemma 2, 3)．この場合，$F(A)=A''$ A'' が A より上位のブロックに属するとき，A をブロックリーダーという．

先行変数の方法によれば，割付の基本式は次のとおりあでる．

$$A[\underline{b}_1, \cdots\cdots, \underline{b}_n]^*:=\overline{F(A)}+1 \qquad (10)$$

ただし $\overline{F(A)}=F(A)[\bar{b}_1', \cdots\cdots, \bar{b}'_n{}']^*$（最高占有番地）

または $A_0:=\overline{F(A)}+1-\sum a_k\underline{b}_k \qquad (11)$

$F(A)=L$ のときは $\overline{F(A)}+1=L_0$ とみなす．

この計算のためには A のコントロールワードの中に$\bar{A}+1$ および $F(A)^*$ を含ませておくとよい．

この方法の正当なことは次のようにしてたしかめられる．

lemma 5．いま，ある LD-変数 A の定義にとりかかるところとする．その時，$F(A)=L$ であるかまたは，

（a） $S(A)\subsetneqq S(F(A))$，しかも

（b） L 型 D-変数の集合 $\{V; S(A)\subsetneqq S(V)\}$ は⊂に関して有限**全順序**集合をなす．

（c） 上記 V はすべて定義済で，$\bar{V}$ は定まっている．

（d） $\overline{F(A)}+1$ 番地以後の記憶場所は自由に使ってよい．

［証］ $F(A)\neq L$ のとき，まず (a) は明らか．(b) は lemma 2, 3 からわかる．(c) も帰納的に考えて明らか．(d) は，この時点で有効な変数 V はすべて $S(A)\subsetneqq S(V)$ を満足することと (b) とを併せていえる．

定理 いま，ある LD-変数 A の定義にとりかかるところとする．その時有意味な LD-変数はすべて番地 L_0 と $\overline{F(A)}$ との間の記憶場所を与えられている．またその間の記憶場所は有意味な変数のみによって占められている．

理由はわれわれの方法そのものからすでに明らかであろう．

われわれの方法の根拠は，変数やレーベルのスコープがブロックに区切られることにある(cf. lemma 4)．したがって内部的にブロックまたは変数を設定する場合，それらも含めて一括して取扱いができるようにするためには，それらに対してもスコープに関する上記性質が満足されるようにしなければならない．そうしたとして，上記定理をふつうの表現にあらためれば：

「あるブロックを脱出したとき，その中の変数のために設定した作業番地はすべて解放してよい」

注意 記憶容量の節約をはかるためには，次のような折衷案も可能である．

（1） ブロックリーダーのみを先行変数方式で取扱う．

（2） それ以外の場合は式 (5) (6) による．このようにするとコントロールワードを (2) の場合節約できる．また処理速度も向上しよう．

5.2 G 型 D-変数

G 型 D-変数で厄介なことは，Sattley が注意しているように，c_k のどれか一つが変更された場合にも，新しい十分な記憶容量の確保を考慮するばかりでなく，変数要素の**配置がえ**を必要とすることである．Ingerman はその配置がえを行なうプログラムの例を示した*．それは ARSHIFT とよばれる函数である．

（1） パラメータ：その添字付変数の次数，$\{\underline{b}_k, \bar{b}_k\}$

* 参考文献 7) 参照．また Naur 等の方法については 9) を参照．

* 参考文献 8) 参照．

および $A[\underline{b}_1, \cdots\cdots, \underline{b}_n]$ それぞれに関する新，旧二とおりの値．

（2） ARSHIFT の値：$\{\underline{b}_k, {}_k\bar{b}\}$ が全く変更されないとき，値 **false** を，それ以外の場合 (3) の効果が起って値 **true** をとる．

（3） 効果：もとの配列法に従っている要素のうち意味のあるものをぬきだし，新しい位置におく．

記憶場所のとり方としては次のような方法が考えられる．まず単位語数から成る記憶場所をいくつか用意する（それらは**頁**とよばれる）．各頁には利用の可否，および**次の頁**の所在を示す指標がついている．割付は頁単位で行なわれる．引用の際，対応番地を計算するプログラムはまず頁数を求め，それからその中にある要素の正確な番地を求めるようにしなければならない．

このように *GD*-変数を完全に許すことは極めて繁雑な手間を要する．そこで実用上差支えない程度に文法を制限する考え方もうまれる．たとえば

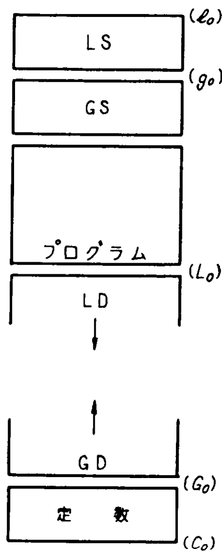
（a） D 型変数，または G 型変数の禁止．

（b） G 型 D 変数の禁止．

また定義を 1 回だけ計算段階で動的に行なうが，**以後変更しない**という解決策もある*)．これは GD 型変数に関しては有力な方法と思われる．(cf[10])

6. 記憶場所のレイアウト

これまで各型の変数の記憶場所を区別して説明してきたが，l_0, g_0, L_0, G_0 などを定数と考えるとそれは記憶場所の使い方として強い制限をおくやり方であるといえる．しかしツーパス以上のコンパイラでは，最初のパスでは対応番地を l_0, g_0 などに対する相対番地で表わしておき，あとから l_0, g_0 をきめて絶対番地を確定するという方法もとり得る．たとえば l_0, C_0 のみを固定して，次のような レイアウトも可能である．



第 3 図

（1） $g_0 := l_{\max}$; $g_p := g_p + l_{\max}$;

（2） <プログラム開始位置>$:= g_p$;

（3） $L_0 :=$ <プログラム終了位置>$+1$;

（4） $G_0 := C_p$

プログラムも相対番地でコンパイルしておき，最後のよみこみまでに入力ルーチンに情報 $l_{\max}$, g_p が伝えられるようにする．プログラム終了位置は入力ルーチンによって知られるから計算段階にはまにあうわけである（**第3図**）．このようにすれば，型によって記憶場所を浮動式にわける方法は，P II にも反しないということができる．

7. Procedure 本体における変数

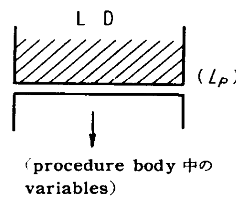
procedure の本体の中で定義される変数は，次のような特殊性をになっている．

（1） スコープなる概念の実際的な意味が明かでない．

（2） 回帰的作動 (recursive call) を許すときは，同じ変数が**二重に有意味**ということが起る．

D 型変数に対する割付は S 型変数を媒介として行なわれるので，S 型変数の定義・引用が正しく行なわれればよい．少なくとも一つの B-ボックスが専用に使えるものとしていえば，S 型変数の対応番地はその B ボックスに関して相対的に与えることが一つの解決策である（二重に有意味といっても，引用される値は一意に定まる．相対番地の起点が動的に変更できればよい．B ボックスにはその procedure 本体にとっての l_0 の値，すなわち**その procedure に入ったときの** L_p の値が与えられる*)（**第4図**）．

procedure から脱出したとき L_p, G_p を復旧させるためには，一般には Naur のルーチンを採用しなければならない．B ボックスが多数個つかえて，しかも多重修飾が可能であれば，先行変数の方法の一変形の適用もできるが，それは望みがたいであろう．procedures の出入りについては厳格な **last-in-first-out** の原則が成り立つものとして，Sattley の方法を適用することが実際的であろう．**go to** 監視の必要性を排除するため



第 4 図

* semi-dynamic ともいうべきか．

* この方法にしたがう変数を semi-static とよぶこともできよう．

には，文法的に次の規約を設ければよい．

procedure からの脱出は，次のような飛越命令によることはできない：**飛越先が条件節またはスイッチからなり，起りうる飛越先地点のレベルが不定のもの**．レベルとは（多重にありうる） procedures のレベルを意味する．

なお一つの procedure 本体内の，主プログラム内の変数の取扱いは，B-ボックスによる修飾以外の点で 6 節までに述べてきたところと変らない．また 1 個の B ボックスで間にあわせようとするときも，他に B-修飾を使う場合があればやはり二重修飾が可能でなければならない．

8. アルゴリスム

最後に本文に詳述しなかった先行変数方式の手順を示そう．まず計算段階における定義実行の手法について述べる．添字付変数のためのコントロールワードは次のように構成されているものとする．

（1） 形式指定語　1 語，添字付変数 A の次元 n，ブロックリーダか否か，ブロックリダーであるときは $F(A)^*$ の値，次に述べる内容指定部分の先頭の所在番地を含む．それぞれ dim 部分 bl 部分 (bl=1 はブロックリダーであることを示す)，FV 部分，CW 部分と決めた場所に記録する（A^* はこれを指示する）．

（2） 内容指定部分　$n+2$ 語，最初の 1 語に A_0，次に $\bar{A}+1$ をおく．続く n 語に a_1, ……, a_n の値をおく．

D-変数の宣言処理ルーチンは次の作業をしなければならない．

（i） $\underline{b}_k$, $\bar{b}_k$ の計算

（ii） $\{a_k\}$, $c=\Pi c_k$ などの計算

（iii） コントロールワードの決定，記憶．

以下に (iii) を実行するプログラムを示す．コンパイラは (i), (ii) をコンパイルし，このプログラムへのエントリを作成挿入する．

```
procedure D definition (Y, a, b, c);
integer array a; integer b, c;
comment パラメータは Y に A*, a に {a_k},
b=Σa_k b_k, c=Πc_k を与える．r 番地の内容をここ
  では Memory[r] とかく；
begin integer r;
for: r=0 step 1 until Extract(Y, dim)
do Memory[Extract(Y, CW)+r+2]:=a[r+2];
if Extract(Y, bl)=1 then
I_p: =Memory[Extract(Memory[Extract(Y,FV)],
  CW+1];
Memory[Extract(Y, CW)]:=L_p-b;
L_p:=Memory[Extract(Y, CW)+1]:=L_p+C
  end D definition;
```

なおコントロールワードを 2 段にわけるのは，次のような宣言がありうるからである．

real array A, B, C, ……, D[……]; 内容指定部分は S 型変数であるが，形式指定語は GS 変数または定数として扱ってよい．これらコントロールワードによって引用時点で番地計算をするのは，次のプログラムによる．

```
procedure Address(Y, i); value Y;
begin integer k, s; s:=0;
for k:=0 step 1 until Extract(Y, dim)
do s:=s+i[k+1]×Memory[Extract(Y, CW)
      +k+2];
Address:=s+Memory[Extract(Y, CW)]end
  the: end
```

参考文献

1) Report on the Algorithmic Language ALGOL 60: P. Naur: Com. ACM 3, No. 5, 1960; 邦訳: 淵一博「算法言語 ALGOL 60 に関する報告」情報処理 Nov. 1960, Dec. 1960, Feb. 1961.
2) MI ライブラリ: 室賀三郎, 戸田　巌編: 電通研（研実報別冊 4 号）
3) MI AUTO CODE I: 池野信一: 電通研（経資第 1077 号）
4) 電子計算機のプログラミング: 高須　達他: 日刊工業新聞社.
5) ALGOL 文法と Array Declaration: 井上謙蔵, 高橋秀知: 情報処理学会(36年度全国大会予稿集)
6) Compiling Technique に関するレポート (1): 藤野喜一, 小島惇: 情報処理学会アルゴル研究会.
7) Allocation of Storage for Arrays in ALGOL 60: K. Sattley: Com. ACM, Jan. 1961.
8) Dynamic Declarations: P.Z. Ingerman: Ibid.
9) A Strage Allocation Scheme for ALGOL 60: J. Jensen, P. Mondrup, P. Naur: Com. ACM, Oct. 1961.
10) Gebrauchsanleitung für den ALCOR-Übersezer der Ermeth: Institut für angewandte Mathematik der ETH.
11) Die Strukturanalyse von Formelübersetzern: P. Lucas: Elektron. Rechenanlagen 3[4] 1961.

（昭和 37 年 8 月 3 日受付）