数　式　翻　訳*

戸島（とじま）　潤（ひろし）

recognizer によって scan されたあとの単純変数および定数からなる単純算術式を翻訳して object code を generate する technique については，種々の工夫がなされているし，また，それらはかなりよく知られている事柄でもある。例えば，次の様な方法は標準的なものと思われる。まず，operand と operator に対して stack を用意する。以下では，それらをそれぞれ A-stack, Ō-stack とよぶ。算術式には前と後に⊢と⊣が補われた上で左端から scan されて，おのおのの stack に入れられる。その際，Ō-stack の一番上の opreator と stack の入口にある operator（すなわち，次の step で stack に入れられる operator）の優先順序が比較されて，入口の operator の方が高ければ stack に入れることをつづけ，そうでなければ，次の様な処理が行われる。もし，前の step で stack の入口にあった operator が今度の step で Ō-stack の一番上の operator になったら，A-stack のうえの 2 つの operand と Ō-stack の一番上の operator に関して object code を作り，stack からは使用ずみの entry を除き，A-stack には *'ACC'* を入れる。これ以外の場合は，もし *ACC* に演算結果が残っていれば，それを作業用番地に移し，A-stack の *'ACC'* をその作業用番地でおきかえてから，上述したのと同じ様にして object code を作り出す。ただし，operator の可換性を考えれば，*ACC* に残っている前の step の演算結果が，今度の operator に関しては右側の operand になっており，しかも，operator が + または * であるときに限り，左側の operand を演算数とした object code を作ればよい。

ところで，多くの compiler 語が許している算術式はもっと複雑なものであるから，それらを翻訳して object code を generate するには，上述した方法を単純に適用するだけでは不十分である。ここでは，単純変数および定数からなる単純算術式をもっと一般化したときに，どの様に翻訳を進めたらよいかを上の production rule を少し拡

1969年 3 月24日受領

* この論文は筆者が用意している formula translation に関する monograph の 1 部分である。なお，この方面の分野に筆者が興味をもつに至った最初の機縁は古瀬大六教授によって与えられた。記して感謝する次第である。

張することによって考える。

最初に単純変数ばかりでなく，添字付変数が許された場合が扱われる。次に条件付算術式の翻訳について述べる。

添字付変数を含む算術式

算術式では単純変数と並んで添字付変数も自由に用いられるが，ALGŌL 語でも FŌRTRAN 語でも，添字付変数を用いるにはその前に宣言を行わなければならない。そのさい，ALGŌL 語では添字の次元には制限がなく，しかも，添字の動きうる範囲は整数全体であるが，FŌRTRAN の場合は次元は3次元までで，かつ，正の整数に限られている。ここでは，まず最初に，添字付変数としては添字の次元に制限はないが，その値は正の整数に限られるものを考えることにしよう。宣言の形は FŌRTRAN 語と同様であり，さらに，添字付変数が使われる前には必ず宣言が行われなければならないものとする。従って，いま，添字付変数，すなわち，配列の名前を A とすれば，A が使われる前には

DIMENSIŌN $A(d_1, d_2, \ldots, d_n)$

という形の宣言があらわれなければならない。ここで，$d_i(i=1, 2, \ldots, n)$ は正の整数である。この様に宣言されてからはじめて A を

$A(e_1, e_2, \ldots, e_n)$

のように refer することができる。ここで，$e_i(i=1, 2, \ldots, n)$ は正の整数値を結果する様な任意の算術式である。算術式 e_i は一般に変数を含む部分 v_i と定数のみの部分 c_i にわけることができる。すなわち

$$e_i = v_i + c_i$$

である。なお，FŌRTRAN 語には e_i のかき方に制限があるが，ここではその様な制限はおかないで，正の整数値がえられさえすれば，どんな数式でもよいことにする。さて，記号を次の様に定義しよう。

$p_1 = 1,$

$p_2 = d_1,$

$p_3 = d_1 d_2,$

.....

.....

$p_n=d_1d_2\ldots d_{n-2}d_{n-1}$,

$m=p_1+p_2+\ldots+p_{n-1}+p_n$.

これらはいずれも DIMENSIŌN statement があらわれたら直ちに計算できるものばかりである。なお，以下では記号番地 X の絶対番地を

$ad(X)$

とあらわすことにしよう。また，番地の割付けは ascending order で行われるものと仮定する。このとき，$ad(A(e_1, e_2, \ldots, e_n))$ と $ad(A(1, 1, \ldots, 1))$ の間にどの様な関係がなり立つであろうか。まず，記憶装置の中に，配列は番号の若い添字がもっとも早く動く様な形で割付けられるものとしよう。このことは，例えば，2 次元の配列であれば，配列は columnwise に展開されて記憶装置の中に割付けられることをいみする。そうすると

$$\begin{aligned}ad(A(e_1, e_2, \ldots, e_n)) &= ad(A(1, 1, \ldots, 1))+(e_1-1)+d_1(e_2-1)\\ &\quad +d_1d_2(e_3-1)+\ldots+d_1d_2\ldots d_{n-1}(e_n-1)\\ &= ad(A(1, 1, \ldots, 1))+\sum_{i=1}^{n}p_i(e_i-1)\\ &= ad(A(1, 1, \ldots, 1))-m+\sum_{i=1}^{n}p_ie_i\end{aligned}$$

がえられる。名前の symbol table への登録と番地の割付けが同時に行われるならば，$ad(A(1, 1, \ldots, 1))$ の値はすぐわかるから

$ad(A(1, 1, \ldots, 1))-m$

の値もただちに確定する。これを配列 A の基準番地とよぼう。そうすると，添字付変数の任意の要素の番地は，基準番地に $(p_1, p_2, \ldots, p_n)$ と $(e_1, e_2, \ldots, e_n)$ の内積

$$\sum_{i=1}^{n}p_ie_i$$

を加えたものによって求めることができる。又は，同じことであるが，e_i を v_i と c_i にわければ，基準番地を

$$\sum_{i=1}^{n}p_ic_i$$

によって修正して（すなわち，この値を基準番地に加える），この修正された基準番地に $(p_1, p_2, \ldots, p_n)$ と $(v_1, v_2, \ldots, v_n)$ の内積

$$\sum_{i=1}^{n}p_iv_i$$

を加えたものによって求めることができる。例えば

$A(2*I+1,\ 4*J+3,\ 6*K+5)$

の番地は次の様に計算される。

A の修正された基準番地$+2i+4jp_2+6kp_3$.

ただし，ここで，$i,\ j,\ k$ は上の要素が refer されたときの $I,\ J,\ K$ の値である。この様に配列の要素の番地は基準番地（修正されていようといまいと）に，ある値（以下これを修飾値という）を加えたものによって求めることができるが，実は，その値は数式翻訳のときには一般に未知で，それが確定するのは翻訳されてできた object program が実行されるときである。従って，数式翻訳のときには修飾値を計算する算術式の object program を作っておき，その object program を実行して得られた結果を使って，基準番地から配列の要素の番地を求める様にしなければならない。すなわち，添字付変数を含む算術式を計算するための object program は算術式本来の演算を行う object program と添字付変数の番地を計算する object program の2つがからみ合ったものからなるのである。

さて，以上の多少制限された添字付変数の議論は ALGŌL の配列の場合に容易に拡張することができる。ALGŌL の source program では

array $A[s_1 : s_1',\ s_2 : s_2',\ \ldots,\ s_n : s_n']$

の様に宣言が行われる。ここで，$s_i,\ s_i'(i=1,\ 2,\ \ldots,\ \mathrm{n})$ は任意の整数で

$s_i \leqq s_i' \quad (i=1,\ 2,\ \ldots,\ \mathrm{n})$

でなければならない。この様に宣言が行われたあと，配列の要素は

$A[e_1,\ e_2,\ \ldots,\ e_\mathrm{n}]$

という形で refer される。ただし，ここで $e_i(i=1,\ 2,\ \ldots,\ \mathrm{n})$ は任意の整数値を結果する数式で

$s_i \leqq e_i \leqq s_i' \quad (i=1,\ 2,\ \ldots,\ \mathrm{n})$

となっていなければならない。いま

$d_i = s_i' - s_i + 1 \quad (i=1,\ 2,\ \ldots,\ \mathrm{n})$,

$p_1 = 1$,

$p_2 = d_1$,

$p_3 = d_1 d_2$,

.....

.....

$p_n = d_1 d_2 \ldots d_{n-2} d_{n-1}$,

$m = p_1 s_1 + p_2 s_2 + \ldots + p_n s_n$

とおく。そうすると，うえの array 宣言は，DIMENSIŌN statement でかくならば

DIMENSIŌN A $(d_1, d_2, \ldots, d_n)$

ということに等しい。ゆえに

$$\begin{aligned} ad(A[e_1, e_2, \ldots, e_n]) &= ad(A[s_1, s_2, \ldots, s_n]) + (e_1 - s_1) + d_1(e_2 - s_2) \\ &\quad + d_1 d_2 (e_3 - s_3) + \ldots + d_1 d_2 \ldots d_{n-1}(e_n - s_n) \\ &= ad(A[s_1, s_2, \ldots, s_n]) + \sum_{i=1}^{n} p_i (e_i - s_i) \\ &= ad(A[s_1, s_2, \ldots, s_n]) - m + \sum_{i=1}^{n} p_i e_i \end{aligned}$$

となる。ここで

$ad(A[s_1, s_2, \ldots, s_n]) - m$

は定数であるから，前の制限された場合と同様に，これを A の基準番地としよう。そうすると，ALGŌL の配列の任意の要素は基準番地に $(p_1, p_2, \ldots, p_n)$ と $(e_1, e_2, \ldots, e_n)$ の内積

$$\sum_{i=1}^{n} p_i e_i$$

を加えたものによって求めることができる。これが一般の場合の配列の要素の計算方法であるが，ただちにわかる様に，前の制限された場合の関係は $s_i = 1$ $(i = 1, 2, \ldots, n)$ とおくことによって求められる。なお，object program に関する前の注意はそのままこの場合にもあてはまる。

ここで，これまで述べてきたことを要約すると次の様になる。

1. 配列の parameter に関する情報は配列の宣言から得ることができる。
2. 従って，基準番地の値

   $ad(A(s_1, s_2, \ldots, s_n)) - m$

   も配列の宣言によって確定する。
3. 配列の要素に refer しているのに出会ったならば

   $$\sum_{i=1}^{n} s_i c_i$$

   の値が確定するから，必要によっては修正された基準番地の値

   $$ad(A(s_1, s_2, \ldots, s_n)) - m + \sum_{i=1}^{n} p_i c_i$$

を求めることもできる。

4. $\sum_{i=1}^{n} p_i v_i$

の値は，v_i の値が要素を refer しているときには一般に未定であるから，算術式を翻訳しているときに確定することはできない。

以上のことを計算機の中で行うときは次の様にすればよい。まず，配列の parameter に関する情報は constant table に入れることにする。以下ではこの table を c-table とよぶことにしよう。FŌRTRAN は3次元の添字までしか許されていないので，このときには c-table を使わなくても，symbol table だけで間に合わすことができるが，ALGŌL の様に次元に制限がないときには symbol table だけでは処理できないであろう。配列が n 次元ならば，以下の様に c-table の $2n+1$ 個の番地を使って配列の parameter に関する情報をあらわす。いま，c-table の中の，ある配列の parameter 情報に関係する部分だけに着目して，その部分の i 番目の番地を $A[i]$ で示せば，各番地にかき込まれる情報は次の通りである。

$$\begin{array}{ll} A[0] & n, \\ A[2i-1] & d_i, \\ A[2i] & s_i, \end{array} \qquad (i=1,\ 2,\ \ldots,\ \mathrm{n}).$$

この c-table へのかき込みの algorithm は次の通りである。

for $i:=1$ step 1 until n do
begin $L:=s_i$;
　　$U:=s_i'$;
　　$A[2i]:=L$;
　　$A[2i-1]:=U-L+1$　end;
$A[0]:=n$;

この様にして作られた c-table を使えば，配列の要素の番地を計算する algorithm は以下の様になる。

BASE $:=ad(A(s_1,\ s_2,\ \ldots,\ s_n))$;
INDEX $:=0$;
for $i:=A[0]$ step -1 until 1 do
begin $j:=2*i$;
　　$E:=e_i$;

$Z:=E-A[j]$;

if $Z<0 \vee Z>A[j-1]$ then go to ŌUT ŌF BŌUND;

INDEX:=INDEX+Z;

if $i \neq 1$ then INDEX:=INDEX+$A[j-3]$ end;

$ad(A(e_1, e_2, \ldots, e_n)):=$BASE+INDEX;

また，c-table にかき込む情報を少しかえて次の様にすることもできる。

$A[0]$　$ad(A(s_1, s_2, \ldots, s_n))-m$,

$A[2i-1]$　p_{i+1},　　$(i=1, 2, \ldots, n)$.

$A[2i]$　s_i,

このときの algorithm は次の通りである。

かき込み：

$F:=1$;

for $i:=1$ step 1 until n do

begin $L:=s_i$;

$U:=s_i'$;

$A[2i]:=L$;

$A[2i-1]:=F:=F*(U-L+1)$ end;

基準番地の計算：

SUM:=$A[2]$;

for $i:=2$ step 1 until n do

SUM:=SUM+$A[2i-3]*A[2i]$;

ABASE:=$ad(A(s_1, s_2, \ldots, s_n))-$SUM;

配列の要素の番地計算：

BASE:=ABASE;

Z:=e_1−A[2];

if $Z<0 \vee Z>A[1]$ then go to ŌUT ŌF ŌBUND;

INDEX:=e_1;

for $i:=$ 2 step 1 until n do

begin $Z:=e_i-A[2i]$;

$V:=A[2i-1]/A[2i-3]+A[2i]$;

if $Z<0 \vee Z>V$ then go to ŌUT ŌF BŌUND;

INDEX:=INDEX+e_i*A[$2i-3$] end;

$ad(A(e_1, e_2, \ldots, e_n))$:=BASE+INDEX;

なお，簡単のために基準番地の計算を独立にやらないで，配列の要素の番地計算を行うこともできる。そうするには，BASE:=ABASE をのぞき，INDEX:=e_1 を INDEX:=e_1-A[2] とし，INDEX:=INDEX+e_i*A[$2i-3$] を INDEX:=INDEX+Z*A[$2i-3$] とすればよい。また，このときには A[0] に $ad(A(s_1, s_2, \ldots, s_n))$ を入れる。なお，配列の次元は配列の宣言文に comma が何個あるかによってきめることができる。すなわち，配列の次元は comma の数に 1 を加えたものに等しい。また，配列の要素の総数は A[$2n-1$] に入っているから，配列のための番地の reserve はこれに従って行うことができる。上の algorithm には，$\sum_{i=1}^{n} p_i c_i$ による基準番地の修正が含まれていないが，それを行うことも容易である。これによって，object program は短くなり，従って，object program の実行に要する時間は短縮するが，逆に翻訳のための手続が複雑化するので，全体として必ずしもよい効果をもたらすとは限らない。

さて，次に object code の production rule について述べよう。この場合には通常の operator の他に新たに comma，subscript と他のものを区切る左右の括弧が operator として加わってくる。その様な括弧として FŌRTRAN では数式の中の部分的な evaluation を示す括弧と同じものが使われるが，名前にすぐつづいてあらわれない括弧は数式に関するものであるから，その区別を容易につけることができる。ALGŌL では subscript に関する括弧には角括弧が使われるから混同の心配は全くない。以下では区別の便宜上，括弧の記法としては ALGŌL の場合と同様に丸括弧は数式に，角括弧は subscript に対して使うことにする。FŌRTRAN の場合でも同じ括弧を場合に応じて区別して考えるのであるから，実質的にはこれと同じことになることはいうまでもない。operator の優先順序は次の様になる。

優先順序	operator
0	([
1	)] ⊢ ⊣ ,
2	=
3	+ −
4	∼

5　　*　/

6　　↑

ここで，[は (と同じ様に必ず stack に入れ，] は stack の一番上の operator が [のときに，[と共に cancel することにする。comma は subscript expression と適当な p_i との乗算と，その結果を前の subscript expression の evaluation に加算するという operation をいみする。しかも，comma が stack の入口にあれば，ACC にある結果を作業用番地に save しなければならない。] は本来の右括弧の働き（すなわち，これで subscript expression が終るということを示すこと）の他に comma と同様に乗算，加算と save することの機能をもつと考えなければならない。なお，object code の generation のためには，A-stack にそれと同じ大きさの table（これを R-table という）を附随させて，A-stack の operand が修飾値によって modify される場合は，修飾値が格納される作業用番地を A-stack の operand の位置に対応する R-table の位置に入れる様にすると便利である。いいかえると，] によって $\sum_{i=1}^{n} p_i e_i$ を求める object program の generation が完了すると共に，その結果を作業用番地に save する object code が作られるが，R-table にはその作業用番地が入れられるのである。R-table をつくらないで，直接 index register に修飾値の番地を入れることも考えられるが，実はこれは index register が複数個あっても，かえって面倒な事態をひきおこす可能性をもつ。なぜならば，1 つの数式に添字付変数がいくつもでてきた場合に，すべての添字付変数の修飾値の番地を index register に入れることは，index register の数が添字付変数より少なければできないから，このときは作業用番地を必要に応じて適当に index register に入れかえて行く様にしなければならなくなる。この手続をできるだけ無駄なく行うことはかなり難しい問題で，通常 index register allocation というのはこうした問題に他ならない。ここでは，この問題に立ち入ることをさけるため，index register がひとつしか利用できない場合を想定した R-table の方法を述べることにしたのである。

さて，以上の考え方に従って

$$a[i+j,\ k]+b[i*j+m]*c[j,\ k,\ l]$$

という算術式の object code がどの様に作られて行くかを見よう。まず，左端から始めて最初の comma までをよみ込んだときの stack 等の状態は次の様になってい

る。[1]

A-stack	R-table	Ō-stack	H-table	E-cell	HENT
a		⊢	1	,	1
i		[	0		
j		+	3		

従って，ここで，$i+j$ を計算する object code[2] をつくる。

LAD i

ADD j

comma が E-cell にあるから結果を作業用番地に save する。

TFR ω_1

つづいて，右括弧までよんだときには次の様になる。

A-stack	R-table	Ō-stack	H-table	E-cell	HENT
a		⊢	1	]	1
ω_1		[	0		
k		,	1		

以下では，x に関係する p_i が格納されている c-table の番地を $p_i(x)$ であらわすことにしよう。そうすると，object code の generation は次の通りである。

LAD k

MUL $p_2(a)$

ADD ω_1

TFR ω_2

次に第2の j のあとの＋までよみ込む。

A-stack	R-table	Ō-stack	H-table	E-cell	HENT
a	ω_2	⊢	1	+	3

---

(1) ここで，H-table には Ō-stack に入っている各 operator の優先順序を示す数を入れ，E-cell には Ō-stack の入口にある operator を入れ，HENT には E-cell の operator の優先順序を示す数を入れる。

(2) 以下では，次の様な sym bolic code を使う。
add; *ADD n*　$ACC+(n)\rightarrow ACC$
subtruct; *SUB n*　$ACC-(n)\rightarrow ACC$
multiply; *MUL n*　$ACC*(n)\rightarrow ACC$
divide; *DIV n*　$ACC/(n)\rightarrow ACC$
load; *LAD n*　$(n)\rightarrow ACC$
transfer (store); *TFR n*　$ACC\rightarrow n$
index modify は *ADD* n* の様に番地部の前に＊をつけることによってあらわす。なお，index (*XR*) は1個しかないものとして，それに数値を格納するときには *LDR n*　$n\rightarrow XR$ という命令を用いる。

b	+	3
i	[	0
j	*	5

そこで

LAD i

MUL j

が作られたあと，] までよむと stack 等の状態は

A-stack	R-table	Ō-stack	H-table	E-cell	HENT
a	ω_2	⊢	1	]	1
b		+	3		
ACC		[	0		
m		+	3		

となる。すなわち，前に stack の入口にあった operator が今後は stack の一番上の operator になったので，数式翻訳における通常の production rule によって，すぐ

ADD m

がつくられる。E-cell に] があるから

TFR ω_3

がつくられて，ω_3 は b に対応する R-table の位置に格納される。さらに，k のあとの comma までよみ込む。

A-stack	R-table	Ō-stack	H-table	E-cell	HENT
a	ω_2	⊢	1	,	1
b	ω_3	+	3		
c		*	5		
j		[	0		
k		,	1		

object code は

LAD k

MUL $p_2(c)$

ADD j

TFR ω_4

である。最後に右端までよみ込む。

A-stack	R-table	Ō-stack	H-table	E-cell	HENT
a	ω_2	⊢	1	⌋	1
b	ω_3	+	3		
c		*	5		
ω_4		⌊	0		
l		,	1		

object code は

LAD l

MUL $p_3(c)$

ADD ω_4

TFR ω_5

である。そうすると，stack 等は次の様になる。

A-steck	R-table	Ō-stack	H-table	E-cell	HENT
a	ω_2	⊢	1	⊣	1
b	ω_3	+	3		
c	ω_5	*	5		

この段階にいたってはじめて算術式を計算するための object code をつくり出すことができる。そのさい，R-table がうまっている A-stack の operand に対しては index modify をかけた object code を作る。

LDR ω_3

LAD $*$ b

LDR ω_4

MUL $*$ c

LDR ω_2

ADD $*$ a

結局，subscript expression から修飾値を計算する object program が本来の数式を計算する object program に superimpose して全体の object program が組み立てられることになる。そして，算術式は，それが subscript expression としてあらわれようと，本来の算術式としてあらわれようと，数式翻訳における一般の production

rule に従って object code を generate しなければならない。しかし，本来の算術式の計算だけならば object program で中間結果を *ACC* から save しなくてもよい場合でも，添字の計算のためにその中間結果を *ACC* から作業用番地に移す必要が起ることがある。こうしたことの一切を考慮に入れた処理手順を作らなければ添字付変数を含んだ算術式の翻訳を行うことはできないのである。だが，stack を有効に使って翻訳を進めて行く基本的な technique には変りがない。例えば，もう1つ次の様な数式を翻訳してみよう。

$a[i]*b[j]+c[k]$

まず，object code が次の様に作られる。

LAD　i
TFR　ω_1
LAD　j
TFR　ω_2
LAR　ω_1
LAD $*$ a
LDR　ω_2
MUL $*$ b

このあと c のあとの [までよみ込んだところでは stack 等は次の様になる。

A-stack	R-table	Ō-stack	H-table	E-cell	HENT
ACC		⊢	1	[	1
c		+	3		

ここで，前に E-cell にあった operator が今度は stack の一番上の operator にならないことがわかるから

TFR　ω_3

を作らなければならない。あとは

LAD　k
TFR　ω_4

が作られて

A-stack	R-table	Ō-stack	H-table	E-cell	HENT
ω_3		⊢	1	⊣	1
c	ω_4	+	3		

となるから

LAD ω_3

LDR ω_4

ADD $*$ *c*

が出て翻訳は終了する。もしうえの算術式の変数がすべて単純変数ならば $a*b$ の演算結果を save する必要はないが，添字付変数のときはこのように $c[k]$ の計算のために，前の演算結果を save しなければならない。しかしながら，その判断は，通常の数式翻訳の production rule と同様に，前に E-cell に入っていた operator が今度は stack の一番上の operator となっているかどうかによって下せばよいから，その点では production rule に変更があるわけではない。R-table に作業用番地を格納するのは E-cell に] が入ってきたときに行えばよい。A-stack の内容が変ったら，それに応じて R-table の内容も適当に変更しなければならない。また，c-table の配列に関する情報の位置が翻訳では必要となるから，配列の名前は c-table の基準番地が入っている番地でおきかえておいた方がよい。そうすれば，object code で配列の基準番地を使うときには，c-table からそれを求めることができる。また，この様にしておけば p_i と乗算するときも，[の次に comma が何個きたかを count して，その数に応じて，c-table の p_i が入っている番地を確定すればよいことになる。なお，上の object program には添字が定義された範囲からはみ出していないかどうかを check する機能が含まれていないが，上述した algorithm に示されているその様な機能をもつ object program を作り出すことは容易であろう。

以上のべて来たことでわかる通り，配列の要素の位置を計算するには，subscript expression を評価して，次にそれに subscript に固有な operation (p_i との乗算など) を施すが，これをひとつながりの手続とはしないで，前者は通常の数式翻訳に他ならないのだから，それを数式翻訳の手続にまかせる様にした方がよい。すなわち，算術式がどこにあらわれようと，数式翻訳は1個所で行う様にすると，手続全体が過度に複雑化するのを避けることができる。

条 件 付 算 術 式

ALGŌL の文法では単純算術式の他に条件付算術式も許している。これは

if *BE* then *SE* else *E*

という形の算術式である。ここで，*BE* は Boolean expression，*SE* は単純算術式，*E* は一般の算術式（すなわち，単純算術式と条件付算術式）である。このいみは *BE* が真なら算術式は *SE* となり，*BE* が偽なら算術式は *E* になるということである。ここで注意しなければならないのは，*BE* が Boolean expression ということである。Boolean expression にも算術式と同様に単純 Boolean expression と条件付 Boolean expression の両方があるから，このことは *BE* が条件付 Boolean expression でもありうるということを示している。条件付 Boolean expression は一般に

if *BE* then *SBE* else *BE*

という形をしているから，上のことは条件付算術式が

if if *BE* then *SBE* else *BE* then *SE* else *E*

という形になる可能性があることをいみしている。ここで *SBE* は単純 Boolean expression である。ここでは Boolean expression の翻訳についてはふれないで，条件付算術式をどの様に翻訳するかということだけに問題を限定する。すなわち，if, then, else などが operator として加わってきた特別な構造をもつ算術式の翻訳方法を考えるのである。そこで，以下では次の様に仮定して話を進めることにしよう。Boolean expression は object code にただちに翻訳することができ，さらに，それを実行したとき，Boolean expression が真ならば *ACC* に 1 が入り，偽ならば 0 が入る様な object code が作られるものとする。これらの object code が実際にどの様に組み立てられるかということは，以下の議論には何ら関係がないので，ここでは具体的な object code は示さないで，とりあえず

$\boxed{BE}$

とかき，これで上述の様に，*BE* が真なら *ACC* は 1，偽なら 0 となる object code を示すものとしよう。そして，以下では，これは k 個の object code からできているものと考える。個々の Boolean expression によって当然 object code の数は変ってくるわけだから，この k は任意の正の整数である。

条件付算術式は条件によって異った算術式を考えるわけであるから，その object program は必然的に条件によって control を任意の場所に移す命令を含むことになる。ここでは，その様な命令として 2 種類の jump 命令を考える。

conditional jump; *JUZ n*　　if *ACC*=0 then go to *n*;

unconditional jump; *JUM n*　　go to *n*;

また，以下では program step counter (*PSC*) を考える。これは次に作られる object code が計算機の何番地に格納されるか（すなわち，絶対番地は何番地か），または program の先頭から数えて何番目の object code にあたるか（すなわち，相対番地は何番地か）ということを示す counter で，適当な初期数から始めて，object code が1つ作られる毎に1つづつふやして行くものである。条件付算術式が翻訳されるに先立って *PSC* には n が set されているものとする。すなわち，object code は n 番地から格納されて行く状態になっているものとする。

さて，条件付算術式で，if, then, else がどの様な働きをしているかを考えてみよう。if は，あきらかに，次に続いてくる *BE* に対して左括弧の役割を果している。then は前の *BE* に対しては右括弧の役割を果すが，次に続いてくる *SBE* または *SE* に対しては左括弧の役割を果たす。else も then と同様に前の *SBE* または *SE* に対しては右括弧，次に続く *BE* または *E* に対しては左括弧となる。すなわち，then と else は二重の性格をもっている。これまでは，左括弧の優先順序は0，右括弧は1であった。そして，左括弧は比較をしないで無条件に stack に入れ，一方，右括弧は比較のみを行い，左括弧と出合えば両者を cancel した。そこで，then も else もこの原則に従うものと考えなければならない。すなわち，これらが stack の入口にあって，stack の一番上の operator と比較を行うときには，右括弧の優先順序をもつものとし，ひと度 stack に入ってしまったら，今度は左括弧の優先順序をもつものとするのである。then が if と出合ったときは，左括弧である if は消去されて，then が新しい左括弧として stack に入る。同様な理由によって，else が then と出合ったときは then が消されて else が stack に入る。else はそのうしろに続く *BE* または *E* が終了したことを示す operator（すなわち，else に対応する右括弧）と出合ったら消される。ところで，その様な operator が then であることがあることに注意しなければならない。例えば，上にあげた *BE* 自身が条件付 Boolean expresoion である場合を考えてみればこのことはあきらかであろう。そこで，then の優先順序が else よりも高ければ，else が stack の一番上にあり，then が stack の入口にあるときは then がただ stack に入るだけになってしまう。しかし，else の優先順序を then と等しいか，または，then よりも低くしておけば，else と then が出合ったとき，else は stack から出ることになって好都合である。この様な考察によって，operator の優先順序は次の様になる。

優先順序	operator
0	(
0	if
0 (stack priority)/ 1 (compare priority)	then
1 (stack priority)/ 2 (compare priority)	else
1	) ⊢ ⊣
2	:=
3	+ −
4	～
5	* /
6	↑

then と else は右括弧として働くときには compare priority を使い，stack に入ってからは stack priority を使う。else は stack から出たら消去し，if, then は対応する右括弧にあたる，then, else がそれぞれきたら消去する。これで，if, then, else の括弧としての扱いがきまった。次に，条件に応じて control を移すには次の様にする。*BE* が偽なら *ACC* に 0 が入っている筈であるから，*BE* に関する翻訳が終了して if が then におきかえられたときに，番地部をあけたままの conditional jump の object code

JUZ □

を作る。この命令によって，else 以後の *BE* または *E* が作る object code の先頭に飛び越したいのであるが，then と else の間の *SBE* または *SE* がどの位の長さの object program になるか不明なので，この段階ではこの様に番地部を未定のままにしておく他ないのである。その代り，Ō-stack と同じ大きさの table を Ō-stack に附属させて，then に対応する位置に *JUZ* が格納された番地を *PSC* をみて入れておく。then と else が出合ったならば，then と else との間の *SBE* または *SE* はすでに翻訳されているから，*JUZ* の未定番地部を *PSC* をみてうめることができる。また，else に続く *BE* または *E* の object program は *ACC* が 0 のときにのみ実行されればよいのだから，then のうしろの *SBE* または *SE* の object program（これは *ACC* が 1 のときに実行される）の最後を *JUM* にして，else のうしろの *BE* または *E* の object program を skip する様にしなければならない。しかし，その object program の命

令の数もこの段階では不明であるから，*JUM* の番地部はあけておく。そして，else に対応する table の位置に *JUM* の格納されている番地を入れておく。この未定の番地部がうめられるのは else が消去されるときである。*JUZ*, *JUM* の未定の番地部をうめるときには then, else に対応する table のそれぞれの位置に格納されている番地を利用すればよい。以上のことをわかり易く示せば次の様になる。

$n)$ | *BE* |
$n+k)$ *JUZ* | |
$n+k+1)$ | |

if を then でおきかえたとき。
(then に $n+k$ を附属させる。
このとき，*PSC* は $n+k+1$ になっている。)

$n)$ | *BE* |
$n+k)$ *JUZ* $n+k+l_1+1$
⋮ } l_1 個
$n+k+l_1)$ *JUM* | |
$n+k+l_1+1)$ | |

then を else でおきかえたとき。
(else に $n+k+l_1$ を附属させる。
このとき，*PSC* は $n+k+l_1+1$ になっている。)

$n)$ | *BE* |
$n+k)$ *JUZ* $n+k+l_1+1$
⋮ } l_1 個
$n+k+l_1)$ *JUM* $n+k+l_1+l_2$
⋮ } l_2 個
$n+k+l_1+l_2)$ | |

else が消去されたとき。
(このとき，*PSC* は $n+k+l_1+l_2$ になっている。)

ここで l_1 は then と else の間の *SBE* または *SE* の object program の命令の数，

l_2 は else に続く BE または E の object program の命令の数とする。n) などは object code が格納されている番地を示す。この様な production rule に従って

$a+$(if $b>c$ then $d+e$ else $d-e$)

という条件付算術式を翻訳してみよう。まず，then までよみ込んだところでは stack 等は次の様になっている。

A-stack	Ō-stack	E-cell	*PSC*
a	⊢	then	n
b	+		
c	(		
	if		
	>		

relational operator および logical operator の優先順序は) ⊢ ⊣ よりも高く + − よりも低いものとする。そうすると，ここで

n) $\boxed{b>c}$

が作られ，if が then と出合うから，if は then におきかえられるとともに

$n+k$) *JUZ* $\boxed{}$

が出る。従って，stack 等は

A-stack	Ō-stack	E-cell	*PSC*
a	⊢		$n+k+1$
	+		
	(		

となる。ここで，A-stack には *'ACC'* をおかないことに注意しなければならない。else までよみ込めば

A-stack	Ō-stack	E-cell	*PSC*
a	⊢	else	$n+k+1$
d	+		
e	(		
	then, $n+k$		

となるから，通常の数式翻訳の production rule に従って

$n+k+1$) *LAD* d

$n+k+2)$ *ADD* e

が作られて，stack 等は

A-stack	Ō-stack	E-cell	*PSC*
a,	⊢	else	$n+k+3$
ACC	+		
	(		
	then, $n+k$		

と変る。ここで，then は else におきかわる。そこで

$n+k+3)$ *JUM* □

が作られるとともに，未定番地部は

$n+k)$ *JUZ* $n+k+4$

とうまり，stack 等は

A-stack	Ō-stack	E-cell	*PSC*
a	⊢		$n+k+4$
ACC	+		
	(		
	else, $n+k+3$		

となる。ここで，*'ACC'* を消去する。もし，A-stack の一番上が *'ACC'* でなかったら，一番上の operand を *ACC* に格納する object code を作らなければならない。このあと，右括弧までよめば stack 等は

A-stack	Ō-stack	E-cell	*PSC*
a	⊢	)	$n+k+4$
d	+		
e	(		
	else, $n+k+3$		
	−		

となって

$n+k+4)$ *LAD* d

$n+k+5)$ *SUB* e

が作られ，*'ACC'* が A-stack に入る。そして，else は stack から消去されるから（消

去する前に，A-stack の一番上が '*ACC*' でないなら，*ACC* に一番上の operand を格納する命令を作る。）

$n+k+3)$ *JUM* $n+k+6$

が作られる。そこで，stack 等は

A-stack	Ō-stack	E-cell	*PSC*
a	⊢	⊣	$n+k+6$
ACC	+		

と変るから

$n+k+6)$ *ADD* a

が作られて翻訳は終了する。この様にしてできた object program をまとめて示すと次の様になる。

$a+$(if $b>c$ then $d+e$ else $d-e$) のobject program

$n)$	$\boxed{b>c}$	$b>c$ なら $1 \rightarrow ACC$ $b \leqq c$ なら $0 \rightarrow ACC$
$n+k)$	*JUZ* $n+k+4$	$ACC=0$ なら $n+k+4$ へ
$n+k+1)$	*LAD* d	$d+e \rightarrow ACC$
$n+k+2)$	*ADD* e	
$n+k+3)$	*JUM* $n+k+6$	無条件で $n+k+6$ へ
$n+k+4)$	*LAD* d	$d-e \rightarrow ACC$
$n+k+5)$	*SUB* e	
$n+k+6)$	*ADD* a	$ACC+a \rightarrow ACC$

この例であきらかな様に，then と else の間の *SE* の結果と else につづく *E* の結果はいずれも *ACC* に格納されていなければならない。以上のことを production rule としてまとめてみよう。

1. if が stack の一番上にあり，then が stack の入口にあるとき，if を then でおきかえ

   JUZ $\boxed{\quad}$

   を object code として作る。

2. then が stack の一番上にあり，else が stack の入口にあるとき，算術式の結果が *ACC* に残る様な object code が作られていなければ，A-stack の一番上の

operand を *ACC* に格納する命令と

JUM |　　|

を object code として作り，*PSC* の番地を *JUZ* の未定の番地部にうめる。

3. else よりも優先順序が低い operator がきたら else を消去する。このとき，算術式の結果が *ACC* に残る様な object code が作られていなければ，A-stack の一番上の operand を *ACC* に格納する命令を作り，*PSC* の番地を *JUM* の未定の番地部にうめる。

さて，次の条件付算術式の object program を作ってみよう。

$a+$(if if $b>c$ then $d=e$ else $d<e$ then f else if $b<c$ then $d*e$ else e/d)

ここで

if $b>c$ then $d=e$ else $d<e$

は条件付 Boolean expression である。従って，$d=e$ の = は assignment をあらわす operator ではなく，logical operator としての等号であるから，$d=e$ は「d が e と等しい」という命題をいみすることに注意しておこう。そこで，最初の if は算術式の if であり，2番目の if は Boolean expression の if である。まず，最初の then までよみこんだときの stack 等の状態を示す。

A-stack	Ō-stack	E-cell	*PSC*
a	⊢	then	*n*
b	+		
c	(		
	if		
	if		
	>		

ここで，object code として

n) |　*b>c*　|

n+k) *JUZ* |　　|

が作られて，stack 等は

A-stack	Ō-stack	E-cell	*PSC*
a	⊢		*n+k+1*
	+		

(

if

then, $n+k$

となる。そこで，第1の else までをよみ込めば

A-stack	$\bar{O}$-stack	E-cell	*PSC*
a	⊢	else	$n+k+1$
d	+		
e	(		
	if		
	then, $n+k$		
	=		

となる。従って，object code としては

$n+k+1)$ $\boxed{d=e}$

$n+k+k'+1)$ *JUM* $\boxed{}$

が作られ，*JUZ* の番地部は

$n+k)$ *JUZ* $n+k+k'+2$

とうめられる。k' は $\boxed{d=e}$ の長さである。stack 等は

A-stack	$\bar{O}$-stack	E-cell	*PSC*
a	⊢		$n+k+k'+2$
	+		
	(		
	if		
	else, $n+k+k'+1$		

と変っている。さらに，第2の then までよみ込む。

A-stack	$\bar{O}$-stack	E-cell	*PSC*
a	⊢	then	$n+k+k'+2$
d	+		
e	(		
	if		
	else, $n+k+k'+1$		

object code として

$n+k+k'+2)$ $\boxed{d<e}$

$n+k+k'+k''+2)$ JUZ $\boxed{\quad}$

が出る。k'' は $\boxed{d<e}$ の長さである。また，ここで else が消去されて

$n+k+k'+1)$ JUM $n+k+k'+k''+2$

となる。つづいて，if が then におきかえられて

A-stack	$\overline{\text{O}}$-stack	E-cell	PSC
a	$\vdash$		$n+k+k'+k''+3$
	$+$		
	$($		
	then, $n+k+k'+k''+2$		

となる。第2の else までよめば，これは

A-stack	$\overline{\text{O}}$-stack	E-cell	PSC
a	$\vdash$	else	$n+k+k'+k''+3$
f	$+$		
	$($		
	then, $n+k+k'+k''+2$		

となるから

$n+k+k'+k''+3)$ LAD f

$n+k+k'+k''+4)$ JUM $\boxed{\quad}$

が作られて，then が else におきかえられて

$n+k+k'+k''+2)$ JUZ $n+k+k'+k''+5$

と番地部がうめられる。そこで，stack 等は

A-stack	$\overline{\text{O}}$-stack	E-cell	PSC
a	$\vdash$		$n+k+k'+k''+5$
	$+$		
	$($		
	else, $n+k+k'+k''+4$		

と変る。第3の then をよめば

A-stack	$\bar{O}$-stack	E-cell	*PSC*
a	$\vdash$	else	$n+k+k'+k''+5$
b	$+$		
c	$($		
	else, $n+k+k'+k''+4$		
	$<$		

となるから

$n+k+k'+k''+5)$ $\boxed{b<c}$

$n+k+k'+k''+k'''+5)$ *JUZ* $\boxed{\quad}$

が作られて（k''' は $\boxed{b<c}$ の長さ），stack は次の様になる。

A-stack	$\bar{O}$-stack	E-cell	*PSC*
a	$\vdash$		$n+k+k'+k''+k'''+6$
	$+$		
	$($		
	else, $n+k+k'+k''+4$		
	then, $n+k+k'+k''+k'''+5$		

さらに，第3の else までよむ。

A-stack	$\bar{O}$-stack	E-cell	*PSC*
a	$\vdash$		$n+k+k'+k''+k'''+6$
d	$+$		
e	$($		
	else, $n+k+k'+k''+4$		
	then, $n+k+k'+k''+k'''+5$		
	$*$		

そこで

$n+k+k'+k''+k'''+6)$ *LAD* d

$n+k+k'+k''+k'''+7)$ *MUL* e

$n+k+k'+k''+k'''+8)$ *JUM* $\boxed{\quad}$

という object code が作られて，then が else におきかわり

$n+k+k'+k''+k'''+5)$ *JUZ* $n+k+k'+k''+k'''+9$

とうめられる。stack 等は

A-stack	$\bar{O}$-stack	E-cell	*PSC*
a	$\vdash$		$n+k+k'+k''+k'''+9$
	$+$		
	$($		
	else, $n+k+k'+k''+4$		
	else, $n+k+k'+k''+k'''+8$		

となる。右括弧までよめば

A-stack	$\bar{O}$-stack	E-cell	*PSC*
a	$\vdash$	$)$	$n+k+k'+k''+k'''+9$
e	$+$		
d	$($		
	else, $n+k+k'+k''+4$		
	else, $n+k+k'+k''+k'''+8$		
	$/$		

であるから

$n+k+k'+k''+k'''+9)$ LAD e

$n+k+k'+k''+k'''+10)$ DIV d

が作られるが，ここで2つある else が両方とも消去されるから

$n+k+k'+k''+k'''+8)$ JUM $n+k+k'+k''+k'''+11$

$n+k+k'+k''+4)$ JUM $n+k+k'+k''+k'''+11$

となる。そこで stack 等は

A-stack	$\bar{O}$-stack	E-cell	*PSC*
a	$\vdash$	$\dashv$	$n+k+k'+k''+k'''+11$
	$+$		

と変るから，最後の object code として

$n+k+k'+k''+k'''+11)$ ADD a

が作られて翻訳が完了する。これをまとめて示せば次の様になる。

$n)$ $\boxed{b>c}$

$n+k)$ JUZ $n+k+k'+2$

$n+k+1)$ $\boxed{d=e}$

$n+k+k'+1)$ $JUM \quad n+k+k'+k''+2$

$n+k+k'+2)$ $\boxed{d>e}$

$n+k+k'+k''+2)$ $JUZ \quad n+k+k'+k''+5$

$n+k+k'+k''+3)$ $LAD \quad f$

$n+k+k'+k''+4)$ $JUM \quad n+k+k'+k''+k'''+11$

$n+k+k'+k''+5)$ $\boxed{b<c}$

$n+k+k'+k''+k'''+5)$ $JUZ \quad n+k+k'+k''+k'''+9$

$n+k+k'+k''+k'''+6)$ $LAD \quad d$

$n+k+k'+k''+k'''+7)$ $MUL \quad e$

$n+k+k'+k''+k'''+8)$ $JUM \quad n+k+k'+k''+k'''+11$

$n+k+k'+k''+k'''+9)$ $LAD \quad e$

$n+k+k'+k''+k'''+10)$ $DIV \quad d$

$n+k+k'+k''+k'''+11)$ $ADD \quad a$

これで与えられた条件付算術式の完全な object program ができていることはあきらかであろう。このように，条件付算術式も operator の優先順序を考慮して翻訳を進めて行く点においては単純算術式の翻訳と全く同じである。ただ条件に応じて control を適当な場所に移したり，不要なところを skip したりしなければならない点が条件付算術式の object program と単純算術式の object program との違いである。しかし，jump 命令は右括弧となる operator がくる毎に適当に作り出せばよいのであるから，結局，条件付算術式も規則的に object program が作られることになる。従って，条件付算術式の production rule は通常の数式翻訳の production rule を多少拡張したものに他ならないことが理解されるであろう。