ALGOL 60 の形式的な文法構造について*

有澤 誠**

Abstract

Naur described the syntax of ALGOL 60 using BNF notation. In this paper, the formal grammartical structure of this description is investigated. This description seems not to be convenient for computer's use, because there are a lot of redundant variables and rules which are intended to be man's use. All the variables can be classified into three groups, and we can represent the syntax structure using regular expressions in each group.

1. はじめに

最近，コンパイラなどの言語プロセッサの自動作成の問題の解決が急がれている。この問題に対する取り組み方はいろいろあるが，そのひとつに，プログラミング言語の文法をメタ言語で記述し，それを直接利用する考えかた (Syntax Directed Method) がある．メタ言語の代表的なものである BNF 記法は，この方向での研究にしばしば利用されている．

ALGOL 60[1] は，BNF 記法で文法が記述された最初の言語であり[2]，以後の BNF 記法の文法記述では多かれ少なかれ，これに影響されている面がある．

いっぽう，この十数年の間にめざましい発展をみせている数理言語学の分野[3] の中で，代表的な文法である CFG (Context Free Grammar) が，ちょうど BNF 記法に対応していることから，言語プロセッサの自動作成の問題に，数理言語学の諸定理を利用する期待ももたれるのである．

しかし，実際問題として，CFG の定義する言語のクラスは，プログラミング言語のレベルと比べてみるとき，制限が強すぎる面と弱すぎる面とを同時に含んでおり，その結果 BNF 記法による文法記述は，現在のところ非常に生かされているというわけではない．

この論文では，ALGOL 60 の BNF 記法による文法の記述を材料として，BNF 記法によって，どのような文法構造を表現しているかを，形式的な立場から調べていく．メタ言語としての BNF 記法の特徴は，この中で明らかにされていくと考えられる．

2. 用 語

ここでは，この論文でしばしば用いられる若干の用語だけを，簡単に説明する．厳密な定義は文献 3) などにゆずり，ここでは述べない．

「BNF 記法」は，つぎの例に示すような「規則」の集合である．

〈specifier〉 ::=〈type〉|**procedure**|**array**

::=の左側を，「規則の左辺」，右側を，「規則の右辺」と呼ぶ．| は選択を表わす．〈specifier〉や〈type〉のように，〈 〉で囲まれた項を「変数」と呼び，**procedure** や **array** のように〈 〉で囲まれていない項を「終端記号」と呼ぶ．変数の中のひとつは，とくに「開始記号」と呼ばれて他の変数と区別される．開始記号はどの規則の右辺にも現われない．つぎの例

〈unsigned integer〉 ::=〈digit〉|〈unsigned integer〉〈digit〉

のように，規則の左辺の変数が右辺にも現われる場合に，この規則を「再帰的な規則」，この変数を「再帰的な変数」と呼ぶ．

一般に，規則の左辺は，ただひとつの変数だけである．またどの変数も，ふたつ以上の規則の左辺には現われない．ある規則の左辺の変数は，その規則で「定義される」という．

ある規則の右辺の変数を，その変数を定義する規則の右辺の各項で置き換えて，その変数を消してしまう操作を「代入」と呼ぶ．代入は再帰的でない変数に関してのみ定義される．この代入の操作を繰り返してい

---

* On the Formal Grammatical Structure of ALGOL, 60 by Makoto ARISAWA (Information Systems Section, Electronic Computer Division, Electrotechnical Laboratory)

** 電気試験所・電子計算機部・情報システム研究室
(現：電子技術総合研究所・ソフトウェア部・言語処理研究室)

くと，1回ごとにひとつの変数が消されていき，したがって，その変数を定義する規則も意味をもたなくなるので，捨てる。代入を繰り返えすと，最終的には再帰的な変数だけが存在する形の状態になる．このときに残った変数を，「本質的な変数」と呼ぶ†.

3. ALGOL 60 の個々の規則の特徴

ALGOL 60 の BNF 記法による記述は，〈program〉を開始記号として，111 個の変数を持ち，109 個の規則からなる．〈code〉と〈any sequence of basic symbols not containing 'or'〉との 2 つの変数を定義する規則がない．Naur のレポート[2]では，〈actual parameter〉，〈actual parameter list〉，〈actual parameter part〉，〈if clance〉，〈letter string〉，〈parameter delimiter〉，および〈unconditional statement〉の 7 つの変数を定義する規則は，それぞれ 2 回ずつ，別々の場所に書かれている．

いっぽう，定義だけされて，どの規則の右辺にも引用されない変数としては，開始記号〈program〉のほかにも，〈basic symbol〉と〈number〉とがある．ただし，basic symbol ということばは，さきに未定義の変数としてあげた〈any sequence of…〉の中に現われている．

これらの変数と規則とを，ひとつずつ独立にみてみると，つぎのようなことがわかる．

まず，それぞれの変数がどのくらい，規則の右辺で引用されているかをみる．何回も引用されている変数の表わしている概念は，回数が少ないものの場合に比べて，ALGOL 60 の文法構造の特徴づけに寄与するところが多いと考えられる．

それぞれの変数が，規則の右辺に引用される回数と，それぞれの変数を，右辺に引用している規則の数とを**表 1** に示す．当然，後者の値は前者の値を越えることはない．

再帰的な変数では，その変数を定義する規則の右辺で，少なくとも 1 回は自分自身を引用している．

この値を調べてみると，3 個以上の規則に引用されている変数は，全体の 2 割弱であり，5 個以上となると，1 割にも満たない．大部分の変数が表わしている概念は，局所的なものであると考えられる．

つぎに，それぞれの規則について，その右辺の選択の数を調べてみる．この値が大きい規則で定義されている変数は，広い範囲をまとめた概念を表わしているものと考えることができる．

再帰的な規則では，少なくとも 2 つの選択が右辺になければならない．

表 1 のこの値をみると，3 以上の選択を持つものが約 4 割近い．しかしいっぽう，この値が 1 である変数も，3 割ちかくある．この値が 1 である変数は，構造の上からは，まったく冗長な変数であると考えられる．

上記の 2 つの要因を合わせてみて，どちらの値も大きいような変数は，〈letter〉や〈statement〉など数少ない．それとは反対に，どちらの値も 1 であるような変数が 2 割近くもあるのである．

4. 再帰的な規則の型

ALGOL 60 の方法の規則に，代入の操作を繰り返していって，本質的な変数を残してみる．最後には，33 個の初めから再帰的な変数と開始記号とが残る（ただし，〈code〉と〈any sequence of…〉とは，終端記号と同等にとり扱った．）．そこで，この 33 個の再規的な変数と，それらを定義する規則とを調べる．

まず，個々の規則について，パターン分けをしてみる．右辺のひとつの項に，左辺の変数が 2 つ以上現われる規則はひとつだけである．

〈open string〉 ::=〈proper string〉|'〈open string〉'|〈open string〉〈open string〉

この規則では，'と' との使用がネスト構造になることを表わしている．

残りの 32 の規則では，いずれも，左辺の変数は，右辺の項の右はし，または左はしにのみ現われる形である．これらの規則は，ほぼつぎの 3 つの型に分類される．

〔A型〕 〈A〉 ::=*a*|〈A〉*a*

〔B型〕 〈A〉 ::=*a*|*b*〈A〉

---

† 再帰的な変数は必ず本質的な変数であるが，逆は成立しない．すなわち，はじめの状態で再帰的ではなかった変数が，代入の操作の途中で再帰的になることがある．たとえばつぎの例

〈A〉 ::=*a*|〈B〉|*a*〈A〉
〈B〉 ::=*b*|〈C〉〈D〉
〈C〉 ::=*ab*|〈D〉
〈D〉 ::=〈C〉〈C〉

について，この状態では〈A〉だけが再帰的な変数である．ところが，〈B〉と〈C〉とを代入で消去してしまうと

〈A〉 ::=*a*|*b*|*ab*〈D〉|〈D〉〈D〉|*a*〈A〉
〈D〉 ::=*abab*|*ab*〈D〉|〈D〉*ab*|〈D〉〈D〉

となって．〈D〉が再帰的な規則で定義され，結局本質的な変数は〈A〉と〈D〉となる．

また，〈B〉と〈D〉とを代入の操作で消去すると，

〈A〉 ::=*a*|*b*|〈C〉*ab*|〈C〉〈C〉〈C〉|*a*〈A〉
〈C〉 ::=*ab*|〈C〉〈C〉

となって，〈A〉と〈C〉とが本質的な変数になる．

表 1 Characteristics of Variables (1)

variable	①	②	③	④	⑤	
⟨actual parameter⟩	1	2	5		12	
⟨actual parameter list⟩	2	2	2	⟨A⟩::=a|⟨A⟩ba	13	
⟨actual parameter part⟩	2	2	2		14	
⟨adding operator⟩	1	2	2		1	
⟨ang sequence of basic symbols not containing 'or'⟩	1	1	/		0	
⟨arithmetic expression⟩	8	12	2	⟨A⟩::=a|bac⟨A⟩	⑩	
⟨arithmetic operator⟩	1	1	6		1	
⟨array declaration⟩	1	1	2		16	
⟨array identifier⟩	3	4	1		3	
⟨array list⟩	2	3	2	⟨A⟩::=a|⟨A⟩ba	15	
⟨array segment⟩	2	3	2	⟨A⟩::=ab|ac⟨A⟩	14	
⟨assignment statement⟩	1	1	2		17	
⟨basic statement⟩	2	2	2	⟨A⟩::=a|b⟨A⟩	19	
⟨basic symbol⟩	0	0	4		4	
⟨block⟩	3	3	2	⟨A⟩::=a|b⟨A⟩		57
⟨block head⟩	2	2	2	⟨A⟩::=ab|⟨A⟩cb		55
⟨Boolean expression⟩	6	6	2	⟨A⟩::=a|bac⟨A⟩	⑩	
⟨Boolean factor⟩	2	3	2	⟨A⟩::=a|⟨A⟩ba	23	
⟨Boolean primary⟩	1	2	5		21	
⟨Boolean secondary⟩	1	2	2		22	
⟨Boolean term⟩	2	3	2	⟨A⟩::=a|⟨A⟩ba	24	
⟨bound pair⟩	1	2	1		12	
⟨bound pair list⟩	2	2	2	⟨A⟩::=a|⟨A⟩ba	13	
⟨bracket⟩	1	1	8		1	
⟨code⟩	1	1	/		0	
⟨compound statement⟩	3	3	2	⟨A⟩::=a|b⟨A⟩		53
⟨compound tail⟩	3	3	2	⟨A⟩::=ab|ac⟨A⟩		52
⟨conditional statement⟩	2	2	4	⟨A⟩::=a|ab|c|d⟨A⟩		㊿
⟨decimal fraction⟩	1	2	1		3	
⟨decimal number⟩	1	2	3		4	
⟨declaration⟩	1	2	4			54
⟨declarator⟩	1	1	7		1	
⟨delimiter⟩	1	1	5		3	
⟨designational expression⟩	5	6	2	⟨A⟩::=a|bac⟨A⟩	⑩	
⟨digit⟩	3	4	10		1	
⟨dummy statement⟩	1	1	1		2	
⟨empty⟩	6	6	1		1	
⟨exponent part⟩	1	2	1		4	
⟨expression⟩	1	1	3		11	
⟨factor⟩	2	3	2	⟨A⟩::=a|⟨A⟩ba	17	
⟨for clause⟩	1	1	1		15	
⟨for list⟩	2	2	2	⟨A⟩::=a|⟨A⟩ba	12	
⟨for list element⟩	1	2	3		11	
⟨formal parameter⟩	1	2	1		3	
⟨formal parameter list⟩	2	2	2	⟨A⟩::=a|⟨A⟩ba	4	
⟨formal parameter part⟩	1	1	2		5	
⟨for statement⟩	3	3	2	⟨A⟩::=a|b⟨A⟩		㊿
⟨function designator⟩	2	2	1		15	
⟨goto statement⟩	1	1	1		11	
⟨identifier⟩	8	10	3	⟨A⟩::=a|⟨A⟩a|⟨A⟩b	2	
⟨identifier list⟩	3	4	2	⟨A⟩::=a|⟨A⟩ba	3	
⟨if clause⟩	5	5	1		11	
⟨if statement⟩	1	2	1			51
⟨implication⟩	2	3	2	⟨A⟩::=a|⟨A⟩ba	25	
⟨integer⟩	1	1	3		3	

表 1 Characteristics of Variables (2)

variable	①	②	③	④	⑤		
⟨label⟩	6	6	2		3		
⟨left part⟩	1	2	2			15	
⟨left part list⟩	2	3	2	⟨A⟩::=a|⟨A⟩a		16	
⟨letter⟩	3	5	52		1		
⟨letter string⟩	2	2	2	⟨A⟩::=a|⟨A⟩a	2		
⟨localor or own type⟩	2	2	2		2		
⟨logical operator⟩	1	1	5		1		
⟨logical value⟩	2	2	2		1		
⟨lower bound⟩	1	1	1			11	
⟨multiplying operator⟩	1	1	3		1		
⟨number⟩	0	0	3		6		
⟨open string⟩	2	4	3	⟨A⟩::=a|b⟨A⟩c|⟨A⟩⟨A⟩		⑩	
⟨operator⟩	1	1	4		2		
⟨parameter delimiter⟩	2	2	2		3		
⟨primary⟩	1	2	4			16	
⟨procedure body⟩	1	2	2				52
⟨procedure declaration⟩	1	1	2				53
⟨procedure heading⟩	1	2	1		6		
⟨procedure identifier⟩	5	5	1		3		
⟨procedure statement⟩	1	1	1			15	
⟨program⟩	0	0	2				58
⟨proper string⟩	1	1	2		2		
⟨relation⟩	1	1	1			20	
⟨relational operator⟩	2	2	6		1		
⟨separator⟩	1	1	11		1		
⟨sequential operator⟩	1	1	6		1		
⟨simple arithmetic expression⟩	3	5	3	⟨A⟩::=a|ba|⟨A⟩ba		19	
⟨simple Boolean⟩	2	3	2	⟨A⟩::=a|⟨A⟩ba		26	
⟨simple designational expression⟩	1	2	3			13	
⟨simple variable⟩	2	3	1		4		
⟨specification part⟩	2	2	3	⟨A⟩::=a|b|⟨A⟩b	4		
⟨specificator⟩	1	1	3		1		
⟨specifier⟩	1	2	8		2		
⟨statement⟩	4	5	3				51
⟨string⟩	1	1	1			11	
⟨subscripted variable⟩	1	1	1			13	
⟨subscript expression⟩	2	3	1			11	
⟨subscript list⟩	2	2	2	⟨A⟩::=a|⟨A⟩ba		12	
⟨switch declaration⟩	1	1	1			12	
⟨switch designator⟩	1	1	1			12	
⟨switch identifier⟩	3	3	1		3		
⟨switch list⟩	2	2	2	⟨A⟩::=a|⟨A⟩ba		11	
⟨term⟩	2	4	1	⟨A⟩::=a|⟨A⟩ba		18	
⟨type⟩	3	6	3		1		
⟨type declaration⟩	1	1	1		6		
⟨type list⟩	2	2	2	⟨A⟩::=a|ab⟨.1⟩	5		
⟨unconditional statement⟩	2	2	3				⑤⓪
⟨unlabelled basic statement⟩	1	1	4			18	
⟨unlabelled block⟩	1	1	1				56
⟨unlabelled compound⟩	1	1	1				53
⟨unsigned integer⟩	5	8	2	⟨A⟩::=a|⟨A⟩a	2		
⟨unsigned number⟩	2	4	3		5		
⟨upper bound⟩	1	1	1			11	
⟨value part⟩	1	1	2		4		
⟨variable⟩	4	4	2			14	
⟨variable identifier⟩	1	1	1		3		

① number of rules in which the variable is refered
② number of reference of the variable
③ number of selection of right side of the rule
④ type of recursive rule
⑤ level number

〔C型〕 〈A〉 ::=a|〈A〉ba

〈A〉および〈A〉 ::=a|ab

A型は，同じ項の羅列を表わす．たとえば

〈unsigned integer〉::=〈digit〉|〈unsigned integer〉〈digit〉

がこの型である．3つの規則がこの型に属する．

B型は，同じ項の羅列の最後に，別の項があるものである．たとえば

〈basic statement〉::=〈unlabelled basic statement〉|〈label〉:〈basic statement〉

がこの型である．この規則は，label の部分を任意回繰り返したあとに，statement の本体がくることを示している．B型の規則は4つあって，すべてこの label の繰り返しのあとに statement 本体がくるパターンである．

C型は，項の羅列の際に，正切り記号を使用することを表わす．たとえば

〈identifier list〉::=〈identifier〉|〈identifier list〉,〈identifier〉

がこの型である．一般に，〈…list〉の形の変数の定義には，この型が使われることが多く，例外は〈left part list〉(A型) くらいである．C型に属する規則は最も多く，15の規則がこの型である．このうちで〈type list〉を除く 14 の変数に対する規則は〈A〉 := a|〈A〉ba の形，いわゆる左再帰形になっている．

残りの 10 規則についてみても

〈A〉 ::=a|bac〈A〉

〈A〉 ::=ab|ac〈A〉

などのように，A～C の3つの型の変型と考えられるものばかりである．

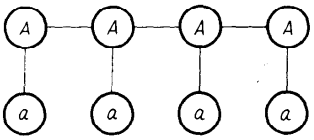

図 1 Tree structure for the rule 〈A〉 ::=a|a〈A〉

一般に，これらの型の規則は，**図1**に示すような木構造を持つ．以下，A型の規則について話しをすすめていくが，B型，C型の場合にも，本質は変わらない．図1の木構造の代わりに，**図2**のような木構造を考えてみても，構造的には同じである．むしろ，木としての構造は図2のほうが一般的で，それをリスト構造などを用いて実現するときに，図1のような形になるとも考えられる．図2を直接表現する規則は，たとえば，正規表現[4] 用いて

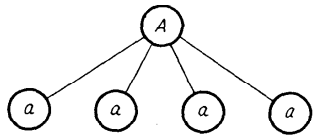

図 2 Tree structure for the rule 〈A〉=aa^*

〈A〉=aa^*

のように書くことができる．ただし，これは規則を局所的にみた場合であって，われわれが問題にしているALGOL 60 の場合，実際には a の部分にも再帰的な変数が含まれていることがあるので，全体が正規表現で記述できるわけではない．

5. 変数間の関係と層別

すべての規則が正規表現による記述に置き換えられないとしても，終端記号に近い部分については，正規表現による記述が，かなり使える．そこで，変数の間での関係をみながら，どのへんまで，正規表現で記述できるかについて考えてみる．たとえば，前節で述べたA型の再帰的な規則を一般化して

〈A〉 ::=〈B〉|〈A〉〈B〉

について，〈B〉の部分が正規表現 u で記述されるならば

〈A〉=uu^*

のように，〈A〉もまた正規表現で記述される．そこで変数を，終端記号だけから定義されるものから始めて，それまでに正規表現を用いて定義できた変数を用いて定義できるものを順々に取り出していくことによって，正規表現による記述の限界がわかる．

具体的には，つぎのような方法で，各変数にレベル番号と呼ぶ番号を割り当てることを行なう．まず，終端記号には，レベル番号0を与えておく．つぎに，各変数のレベル番号は，その変数を定義する規則の右辺に現われる変数，または終端記号のレベル番号の最大値に1を加えたものを与える．とくに，再帰的な変数のレベル番号は，再帰的な規則の形が，前節で述べたA～C型のように，項のはしにのみ定義される変数が現われる形 (one sided linear 類似形) のときに限ってレベル番号を定義する．このとき，右辺の中で，再帰的な変数を除いた変数のレベル番号の中で，最大値をとりそれに1を加えたものとする．

この操作を行なうと，47 個の変数に，最高 6 までのレベル番号をふったところまでで，それ以上，レベル番号をふることができなくなる．この 47 個の変数で表わされる範囲が，正規表現を用いて記述できる部分である．この中には，〈identifier〉，〈unsigned integer〉，〈letter string〉，〈identifier list〉，〈formal parameter list〉，〈specification part〉，〈type list〉の 7 つの再帰的変数が含まれる．

コンパイラでは，これらの部分は，文字系列処理(Lexical Analyser)で処理されることが多い．文字系列処理は，通常有限状態オートマトンによってモデル化されており[6]，有限オートマトンと正規表現による記述とがちょうど対応する関係にあることから，これらの部分の記述には，BNF 記法よりも正規表現のほうが，実際便利であろう．

残った変数については，変数〈A〉の定義に〈B〉が使われ，〈B〉の定義には〈C〉が，また〈C〉の定義には〈A〉が使われる，といったように，環状の関係になっている．したがって，そのままでは前のような順序づけはできない．

これらの変数を定義している規則をよく調べてみると，数式を基本とした規則が多いことに気づく．そこで，〈expression〉をひとまとまりの概念と考えて

〈expression〉：＝〈arithmetic expression〉|
〈Boolean expression〉|
〈designational expression〉

の右辺の 3 つの変数に，レベル番号 10 を与えて，前と同じ操作を行なう．なお，このとき，便宜上〈open string〉にも，レベル番号 10 を与えておくことにする．

この操作は，新たに 47 個の変数にレベル番号を与えることができ，残りは〈program〉，〈block〉，〈compound statement〉など 15 の変数になる．

この残りの 15 の変数についてみると，文を基本としたものが多いことがわかる．そこでもう一度

〈statement〉：＝〈unconditional statement〉|
〈conditional statement〉|
〈for statement〉

の右辺の 3 つの変数に，レベル番号 50 を与えて，同じ操作を繰り返す．この操作によって，〈program〉にレベル番号 58 が割り当てられて，すべての変数に対して，レベル番号が割り当てられたことになる．このレベル番号は，表 1 に示してある．

このレベル番号によって，変数は 3 つの層に分けら

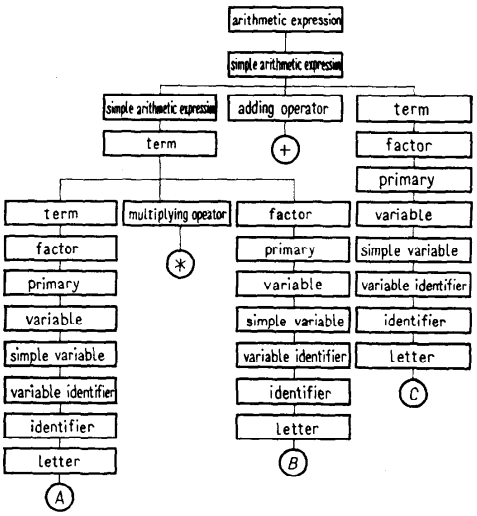

図 3 Formal syntax tree for the expression $A*B+C$

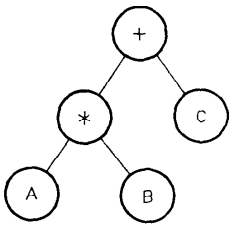

図 4 Basic syntax tree for the expression $A*B+C$

れた．第 1 の 1 から 6 までのレベル番号を持つ変数は，前述したように，正規表現で記述可能な部分で，文字系列処理で解析されるべき部分である．第 2 の，$_{1}0$ から 26 までのレベル番号を持つ変数は，数式に関連した部分である．そして残りの，50 番台のレベル番号をもつ変数は，さらに高いレベルの，文に関連した部分である．

2 番目の層について考える．単純な数式，$A*B+C$ の構文木をかいてみると，**図 3** のようになる．きわめてこみいった構造になっている．いっぽう，数式の木表現は，しばしば**図 4** のような形でも書かれる．図 4 の 2 分岐木構造[5]を，プリ・オーダにたどると，$+*ABC$ エンドオーダにたどると，$AB*C+$ となって，それぞれポーランド記法と逆ポーランド記法とによる表現になっている．またポストオーダ†† にたどると，$A*B+C$ となって，もとの表現が得られる．ただ

†† 定義は文献 5) による．

し，ポストオーダにたどった場合には，演算子の優先順位に関して，正しいことを保証するためには，部分木を1回たどるたびにかっこをつけて，$((A*B)+C)$ のようにする必要がある．

この例でわかるように，数式の構造に関する問題点には，被演算数に，数式そのものをネストできる部分と，演算子の優先順位に関する部分とがあって，図3が複雑な木構造になるのは，この演算子の優先順位の問題が因をなしている．

BNF 記法の長所のひとつは，この演算子の優先順位のきまりを，気のきいた形で完全に表現できることである．しかし，その記述の構造をそのまま利用しようとすると，冗長性が大きすぎてしまう．これに対して，図4の木構造にも，演算子の優先順位はとりこまれている．ただ，この木構造を，演算子の優先順位を含めてそのまま表現することは，BNF 記法ではもちろんできないし，現在適当なメタ言語は知られていない．

数式の部分の処理については，上記のような事情から，BNF 記法による文法記述をそのまま使用するよりも，演算子プレシデンス法などの方法を利用したほうが能率がよいことになる．

この層に属する変数には，直接数式の構造を表わすもののほかに，〈for list〉や〈basic statement〉などのように，文の構成単位となるものもまじっている．これらの変数は，expression 関係の変数を終端記号とみなすと，正規表現を用いて記述しなおすことができる．終局，この層での複雑なネスト構造は，expression 関係だけである．**図5**には，この関係をグラフの形で示してある．変数〈A〉の定義に変数〈B〉を必要とするときに，〈B〉から〈A〉にむかう矢印をつける．これを expression に関連する 12 の再帰的な変数の間でかいてある．

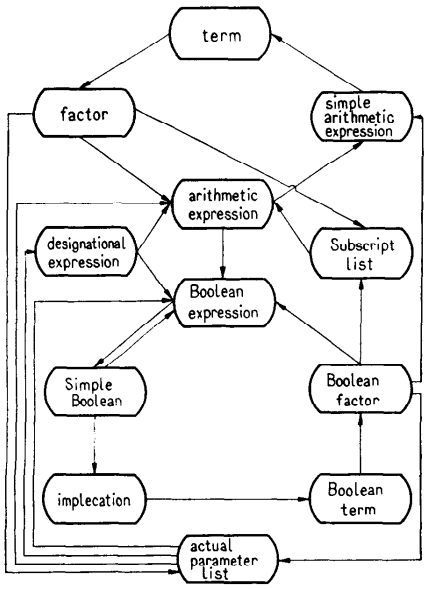

図 5 Relation between some essential variables

3番目の，最も高いレベルの層についても，変数の関係は複雑である．ここでも〈statement〉を終端記号とみなすと，正規表現を用いて記述しなおすことができる．いま簡単のために〈label〉:，〈if clanse〉，〈for clanse〉，〈basic statement〉，および〈procedure heading〉をそれぞれ L，IC，FC，BS および PH で表わす．また〈statement〉を〈$s.$〉，〈declaration〉を〈$d.$〉とかく．

〈$s.$〉: = L*FC〈$s.$〉|L*BS|L*〈compound $s.$〉|
L*〈block〉|L*IC BS|
L*IC〈compound $s.$〉|L*IC〈block〉|
L*IC BS **else** 〈$s.$〉|
L*IC〈compound $s.$〉**else**〈$s.$〉|
L*IC〈block〉**else**〈$s.$〉|L*IC L*FC〈$s.$〉

〈block〉: = L***begin**〈$d.$〉[;〈$d.$〉]*[〈$s.$〉;]*
〈$s.$〉**end**

〈compound $s.$〉: = L***begin**[〈$s.$〉;]*〈$s.$〉**end**

〈$d.$〉: =〈type $d.$〉|〈array $d.$〉|〈switch $d.$〉|
procedure PH〈$s.$〉|**procedure** PH
〈code〉|〈type〉**procedure** PH〈$s.$〉|
〈type〉 **procedure** PH〈code〉

〈program〉 =〈block〉|〈compound $s.$〉

のようになる．ここで〈d〉に関する部分は，見にくくなるのを避けるために，独立した規則の形のまま残してあるけれども，代入によって消去できることは，容易にわかる．この層では，〈statement〉，〈block〉，〈compound statement〉の3つの変数の間だけで，複雑なネスト構造が吸収できることになる．

6. おわりに

ALGOL 60 の BNF 記法による文法記述を調べた結果，つぎのようなことがわかった．

まず，この記述は，文法の厳密な定義を人間に対して理解させるように構成されており，かなり多くの局所的な冗長な変数を導入している．したがって，コン

ピュータが，この記述をそのまま利用して，構文解析を行おうとする場合には，能率の面ではあまりよい結果を期待することができない．

つぎに，ここで使用されている変数とそれを定義する規則とは，ほぼ3つの層に分けることができる．それぞれの層の中では，かなりの部分は正規表現による記述が可能であって〈expression〉の部分と，〈statement〉の部分とに，複雑な構造が集中している．

筆者らの研究グループでは，機能別にモジュール分けされた，モデルコンパイラの実験と試作を行なった[8]．対象とした言語 ESDL (ETL's System Description Language) は，ALGOL 60 とほぼ類似の基本構造を持つ言語である．このコンパイラの analyser の部分が，文字系列解析 (Lexical Analyser)，数式解析 (Expression Analyser)，構造解析 (Structure Analyser) および有効範囲解析 (Scope Analyser) の4つのモジュールに自然に分かれたことも，この層構造との対応と考えることができる．

文献 7) の中には，syntax directed compiler の試みがいくつも紹介されている．そのひとつは，Tixier による RCF 言語 (BNF 記法中に正規表現を直接とりこんだメタ言語によって文法を記述している) であり，また別の例は，Cheatham による syntax directed compiler で，そこでは analyser の構成方法によって，BNF 記法による規則のレベル分けをすることが試みられている．

このふたつの例についても，BNF 記法による規則の中で，プログラム全体の文法構造に関するものを，ほかの部分と切り離していく考え方がとられている．どちらの場合にも，syntax-directed らしさは，大局的な構造の解析のほうに求めているように考えられる．

一般にコンパイラの自動作成については，ALGOL 60 の BNF 記法による記述のもつ，3つの層と，それでは表現できなかった有効範囲 (scope) の4つの部分を，コンピュータの処理むきに記述することのできるメタ言語を開発することが，まず，なされるべきことであると思われる．そのようなメタ言語への示唆は，BNF 記法だけでなく，正規表現にもあるように，筆者は考えている．

謝 辞

この研究の機会を与えていただき，数々の有益な助言をいただいた野田克彦部長，西野博二室長，相磯秀夫室長，淵一博主任研究官に感謝する．また，いっしょに討論していただいた鳥居宏次博士，斉藤信男氏，古川康一氏，杉藤芳雄氏，宮川正弘氏，佐藤光昭氏，資料の整理を手伝っていただいた堀口統子嬢，伊墻迪子嬢，長野和子嬢に，心からお礼を申しあげる．

参 考 文 献

1) 森口繁一：ALGOL 入門．JUSE 出版社 (1962)
2) Naur, P.: Revised Report on the ALGOL 60 (1962). (Rutishauser, H.: Description of ALGOL 60 Springer Verlag (1967) 付録より)
3) Ginsberg, S.: The Mathematical Theory of Context Free Languages. McGraw-Hill (1966)
4) Hennie, F. C.: Finite-State Models for Logical Machines. John Wiley (1968).
5) Knuth, D. E.: The Art of Computer Programming, Vol. 1. Fundamental Algorithms. Addison-Wesley (1968).
6) Hopgood, F.: Compiling Technique. Elsevier (1969).
7) J. Feldman, D. Gries: Translator Writing Systems. CACM 11 (1968). pp. 77-113. およびこの論文で引用している諸文献
8) 有沢誠，斉藤信男，佐藤光昭，杉藤芳雄，宮川正弘：ESDL コンパイラについて，電気試験所彙報 Vol. 34. No. 5-6. (1970) ESDL 特集号，および同誌掲載の ESDL 関係の諸文献．

(昭和45年6月9日受付)