ALGOL Machine に関する一考察*

房 岡 璋**

Abstract

The machine outlined in this paper is one of the language acceptors that directly execute programs written in some problem oriented languages. The construction of this machine is conciderably different from the conventional Von-Neumann type computers and it operates with following three phases.

In the 1st phase, the microprogrammed processing routine for the specified language, especially ALGOL in this paper, is entered in the fixed memory.

The 2nd phase consists of the semantic-invariant transformations that transform source programs irrelevant to the data.

After this transformation phase, the machine enters the control mode of the execution phase. In this mode, the machine is considered to be the hardware version of ALGOL interpreter.

Since ALGOL sentences are generated by the context free grammar, the machine control in this 3rd phase may be accomplished with the push down mechanism. It is significant that the machine control is intergrated with the syntax analysis in this control mode.

1. はじめに

一般に，問題向き言語の処理は，異なった2つの過程をたどることによって行なわれる．コンパイルとインタープリテーションである．コンパイルの過程では，問題向き言語のプログラムをもっと容易にインタープリートし，実行できる中間形態に変形する．これは，データとは無関係な過程であり，当初のプログラムが持っていた字義どおりの意味を増減することはない．この変形された中間形態の記号列を逐一解釈し，データに変形を加えることにより，計算機の応答を形成する過程が，インタープリートである．場合によっては，単に，コンパイルを経ずに，インタープリートするだけのものもあるが，多くは，この2つの過程をたどることによって問題向き言語を処理する．そして，その中間語の形態としては，機械語，またはアセンブリ言語を用いるのが普通である．

これは，一方で計算機の機構とは無関係な問題向き言語があり，他方で，それとは無関係に計算機が設計され，その間をつなぐものとして，コンパイラを設計するという基本的な考え方に従っているからである．しかし，問題向き言語を使うことが当然とされている現在では，機械語の比重はうすれ，中間言語を必ずしもいままでの機械語とする必要もなくなってくる．それに応じて，上に述べた標準的な問題向き言語の処理方式を再検討する必要があろう．

さらに，機械語は，計算機のハードウェア的機能を表現しているものと考えられ，それは，その時代の回路技術にとって最適なものとして設計されている．したがって，集積回路などの開発によって，複雑な回路が高い信頼性をもって組みうる現在では，従来の計算機のシステム設計についても問いなおされるべきであり，それに応じて機械語も変わることになろう．

この論文の目的は，direct execution を行なうに適した言語の文法的な特徴を抽出し，それに対応する計算機を構成したうえで，ALGOL の場合に対し具体的な制御ルーティンを与え，通常の計算機やコンパイラとの特徴比較を行なうことである．

しかし，ALGOL machine といっても，単に ALGOL だけでなく，FORTRAN, COBOL などの問題向き言語をも処理できなければ現実的な意味はない．

* Design Concideration for ALGOL Machine, by Akira Fusaoka (Faculty of Engineering, Kyoto University)
** 京都大学・工学部

以下のシステムでは，これらの言語の共通部分について，ハードウェアによる装置として組み，マイクロプログラムでそれらを制御する．したがって，異なる言語を用いるときは，マイクロプログラムで組まれた制御ルーティンを取り換えるだけでよい．

Appendix では，ALGOL の部分集合に対して，その制御ルーティンを具体的に与える．本論は，ALGOL のすべての機能を前提としているから，全 ALGOL への拡張は容易である．

2. 中間言語の選択

言葉の厳密な意味からすれば，ALGOL machine は，ALGOL を，そのままの形で即座に実行しなければならない．そのこと自体は，ALGOL の機能を若干制限すればむずかしいことではない．しかし，たとえば，未出現の label を含む goto statement が現われた場合，時には，プログラム全体を調べる必要があり，効率の点で，ほとんど問題にならない．さらに，多様な入力言語を，一切の変更を加えずに，それらをすべて即座に実行できるように装置を設計するならば，システムが複雑になり，言語の構造を設計に反映させるという基本的な考え方を，かえって不明確にするおそれもある．

ここでは，問題向き言語で書かれた入力プログラムを，ほとんどその原型を保っている形態の中間言語に変形し，それをデータとともに実行する方式を考える．その場合，中間言語は，ソフトウェアと比較して融通性に乏しいハードウェアによる文法処理を行なう必要があるので，簡単で標準的な文法を持つものでなくてはならない．さらに，文脈の分析を行ないつつ実行するのであるから，その分析のプロセスは back tracking を含まないことが望ましい．そのためには，文法は deterministic であればよい．加えて，高速のインタープリテーションを許すためには，文法分析の 1 step ごとに新しい入力記号を読み込むことが望ましい．すなわち，real time definable である文法を持つべきである．結局，中間言語の文法に課せられる条件としては，つぎの 3 つがあることになる．

(1) 標準的な形態を持つこと．
(2) deterministic であること．
(3) real time definable であること．

この 3 条件を満たすものとして，regular grammar が即座に考えられるが，既存の問題向き言語の文法は，ほとんど近似的に context free grammar (以下 CFG と略.) であり，中間言語の文法として，regular grammar を採用するならば，言語の構造を反映した計算機を設計することは困難になろう．ここでは，上の 3 条件を満たす CFG として，以下の文法を考える[7)12)]．

CFG を $G=(V, \Sigma, P, \sigma)$ とする．ただし，V は terminal symbol および nonterminal symbol の集合，Σ は terminal symbol の集合，P は書き換え規則の集合，σ は $V-\Sigma$ の元である．

Greibach は一般の CFG が，つぎのような形態の書き換え規則のみからなる文法に変形できることを示した．

$$Z \to a X^1 X^2 \cdots\cdots X^n \quad n \geqq 0 \qquad (1)$$

ただし，$Z \in V-\Sigma$，また $i=1, 2, \cdots\cdots n$ に対して，$X^i \in V-\Sigma$，または $X^i=\varepsilon$ である．ここで ε は長さ 0 の記号列を意味する．さらに $a \in \Sigma$ である．これを n-標準形という．書き換え規則の左辺にある記号 (上記 (1) の Z に対応する記号) を generatrix，右辺にある terminal symbol〔上記 (1) の a に対応する記号〕を handle と呼ぶ．deterministic で，かつ，real time definable である言語をこれより構成する．

上の書き換え規則 (1) を

$$(Z, a) \to X^1 X^2 \cdots\cdots X^n \qquad (2)$$

と表現する．このような変形を行なった書き換え規則に順次番号を付し区別することにすれば，書き換え規則の集合 P は

$$(Z_i, a_i) \to X_i^1 X_i^2 \cdots\cdots X_i^n \qquad (3)$$

ただし，$n \geqq 0$，$1 \leqq i \leqq m$，と表わせる．ここで，与えられた文法が，条件

$$\left.\begin{array}{l}(Z_i, a_i)=(Z_j, a_j), \text{ ならば} \\ X_i^1 \cdots\cdots X_i^n = X_j^1 \cdots\cdots X_j^n\end{array}\right\} \qquad (4)$$

を満たすとき，$G=(V, \Sigma, P, \sigma)$ を s-deterministic gammar と名づける．すなわち，s-deterministic であるならば，特定の generatrix-handle pair を持つ書き換え規則は，高々 1 個しか出現しない．これは，deterministic で real time definable である言語の一つのモデルになっており，それゆえ，高速のインタープリテーションを保証する．上で述べた s-deterministic grammar は $n=2$ のもっと特殊な場合に還元できる．なぜなら，$Z \to a Y^1 \cdots Y^{n-1} Y^n$ なる規則の代わりに，新しい記号 $S(Y^{n-1}, Y^n)$ を V に加えて，$Z \to a Y^1 Y^2 \cdots Y^{n-2} S(Y^{n-1}, Y^n)$ とする．さらに，このようにして導入された記号 $S(A, B)$ に対し，もとのままの P の書き換え規則のうち，generatrix に A を持つ規則を調べる．いま，その規則を，$A \to a U^1 \cdots\cdots U^k$ とす

れば，$k \leq n-2$ なる規則に対しては，$S(A,B) \rightarrow aU^1 \cdots\cdots U^k B$ なる新たな書き換え規則をつけ加える．$S(A,B)$ は新しい generatrix であるから，条件(4)を満足し，s-deterministic であるという性質を変えることはない．もし，$k=n-1$ ならば，$S(A,B) \rightarrow aU^1U^2\cdots\cdots U^{n-2}S(U^{n-1},B)$ を，$k=n$ ならば，$S(A,B) \rightarrow aU^1\cdots\cdots U^{n-3}S(U^{n-2},U^{n-1})S(U^n,B)$ をつけ加える．先と同様に，これらの操作も，s-deterministic なる性質を変えることはない．この手続きにより，n 標準形の s-deterministic grammar から，それと同値な $(n-1)$ 標準形の deterministic grammar を得ることができる．これらの操作の繰り返しにより，結局2標準形（すなわち，$n=2$）へ s-deterministic なる性質を変えずに達することができる．以下では，制御の簡単さという利点から，もっぱら，2標準形の s-deterministic grammar を取り扱う．すなわち，上のことにより，書き換え規則はすべて

$$Z_i \rightarrow a_i X_i^1 X_i^2 \qquad 1 \leq i \leq m$$

なる形をしている．

この文法のインタープリテーションはつぎのようにして，状態レジスタとスタックおよびテーブル M によって行なわれる．テーブル M は

$$((Z_i, a_i), (X_i^1, X_i^2), \Psi_i, \Phi_i) \qquad 1 \leq i \leq m$$

によって構成されている．ただし，Φ_i は書き換え規則

$$Z_i \rightarrow a_i X_i^1 X_i^2$$

の解釈ルーティンを示す．Ψ_i は jump 命令などによる制御の変更に関する情報である．さらに，M に

$$((\sharp, \sigma), (\phi, \phi), \phi, \text{stop})$$

なる行を加える．初期状態では，スタックには $\sharp$，状態レジスタには σ のみがはいっている．$S_1, S_2, \cdots\cdots S_q \in \Sigma$ とし，記号列 $S_1S_2\cdots\cdots S_q$ を実行する場合を考える．S_{i-1} の実行を終わったところで，状態レジスタの内容が Z_i であるとすると，第1列が (Z_i, S_i) である M の行が捜し出される．対応する (X_i^1, X_i^2) が

(i) (ϕ, ϕ) ならば，スタックの頭の内容を状態レジスタに移行し，スタックを pop up する．

(ii) (Y^1, ϕ) ならば，状態レジスタを Y^1 に書き換える．

(iii) (Y^1, Y^2) ならば，スタックを push down し，状態レジスタの内容を Y^1 に，スタックの頭の内容を Y^2 に書き換える．

いずれの場合も，解釈ルーティン Φ_i を実行する．さらに，$\Psi_i = \phi$ ならば，つぎの記号 S_{i+1} を読み込む．$\Psi_i \neq \phi$ ならば，Ψ_i で指定された記号列の位置にある記号を読みこみ，さらに適当な量だけスタックを pop up する（これについては，後に論ずる．）．

(Z_i, S_i) に対応する行がテーブル M 中に存在しなければエラーである．

3. ALGOL machine の機構

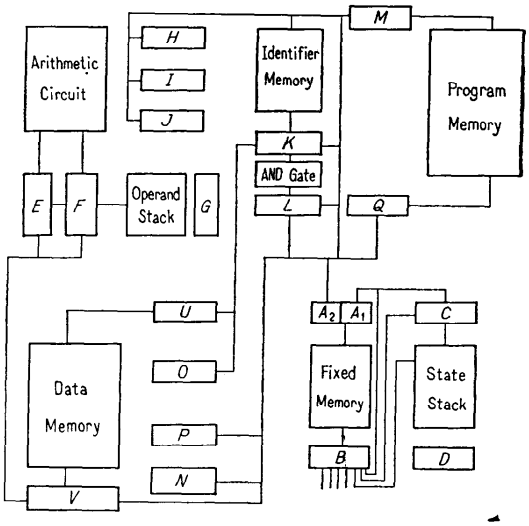
ALGOL machine の機構は概略つぎの7対 T

$$T = (\Xi, H, Z, \Theta, \Lambda, K)$$

で示される．ここに Ξ は制御ユニット，H は identifier メモリ，Λ は固定メモリ，Z は算術演算ユニット，Θ はプログラムメモリ，K はデータメモリである．全体としての構成を **Fig. 1** に示す（ただし，入出力ユニットは省略．）．

(1) 制御ユニット Ξ

Fig. 1 で，レジスタ A, B, C, D, N, P および状態スタック，固定記憶装置 Λ をまとめて制御ユニットとして考える．execution phase では，A_1 には現在の状態が，A_2 には入力記号がはいり，この対で固定メモリ Λ のアドレスを指定する．この事情については，



A. (A_1, A_2) Control Memory Address Register.
B. Control Register.
C. Current State Register.
D. State Stack Counter.
E. Operand Register.
F. Head of Operand Stack.
G. Operand Stack Counter.
H. Identifier Memory Address Register.
I. Register for Next Available Address.
J. Counter for Table Look Up.
K. Identifier Memory Register.
L. Window Register.
M. Program Memory Address Register.
N. Next Delimiter Counter.
O. Register for Next Available Address.
P. Block Level Counter.
Q. Program Memory Register.
U. Data Memory Address Register.
V. Data Memory Register.

Fig. 1 The Information Flow of ALGOL Machine

後に詳しく述べる．N は delimiter カウンタで，通常の計算機の逐次制御カウンタ (SCC) に対応する．すなわち，つぎに実行する delimiter のはいっている番地を指定する．P は block の level を計算する．状態スタックは高速の記憶装置をDレジスタによってスタックに編成して用いる．このように，context free grammar を文法構造として持つ言語をインタープリートする場合，制御ユニットがすでに infinite のメモリを持たなければならない．

(2) プログラムメモリ θ

Fig. 1 のプログラムメモリ，レジスタ M, Q がプログラムメモリユニットΘとして取り抜われる．プログラムメモリは，通常の8ビットを1バイトとするランダムアクセスのコアメモリであるが，アドレス修飾の必要がほとんどないため，書き込み速度の遅い半固定記憶装置で構成される．

(3) identifier メモリユニット H

identifier からアドレスの変換は種々の方法が考えられ，それに応じて計算機の構成もかなり変わってくるが，ここでは，execution phase に declaration があるたびに，新しい identifier を identifier メモリ内に記憶し，プログラム中の identifier にシステムが出会うたびにこのメモリを順次調べ，対応するデータメモリ内のアドレスを得ることにする．さらに，block が終了すれば，その block 内の identifier はこのメモリから取り去られる．execution phase では，このメモリを **Fig. 2** のような語構成で用いている．

Name of Identifier	Block Level	Type	Address of Program Memory

Fig. 2 The Format of Words in Identifier Memory

(4) データメモリ

プログラムメモリが，8ビットを1バイトとするシステムの半固定記憶装置であったのに対して，データメモリは，48ビットを1語とするコアメモリで構成される．Uはデータメモリに対するアドレスレジスタであり，Vは，データメモリレジスタであって，読み出された，またはメモリに記憶されるべきデータがはいる．

(5) 算術演算ユニット Z

Fig. 2 で，演算回路，E, F レジスタ，オペランドスタックが算術演算ユニットとして取り抜われる．ただし，オペランドスタックは，高速メモリをTレジスタによってスタックに編成して用いる．transformation phase において，算術式はすべてポーランド記法に変形されるため，このようにオペランドをスタック形式で処理する機構を持たせる必要がある．データがバイト構成で表現されているとき，すなわち，variable size のデータが許されるとき，それをスタック形式で取り抜うのはきわめてむずかしいので，この計算機ではプログラムはバイト構成で，データは語構成で処理される．このように，このシステムでは，プログラムとデータは完全に区別して取り抜われる．これは従来の計算機には見られない特徴である．

実際の算術演算は，EおよびFレジスタで行なわれる．オペランドスタックや E, F レジスタの内容はデータまたはアドレスである．算術式中に identifier が現われると，逐一 identifier メモリの検索を行なわなくてはならず，これが，この計算機の大きな欠陥である．実際の設計では，算術演算ユニットに独立した制御装置をつけ加えて，算術演算と identifier の検索を併行して行なうべきであろう．多少システムは複雑になるが，算術演算の部分は異なったプログラミング言語でもほとんど共通しており，したがって，この制御装置は各言語に対して同じものとして設計できるうえ，1つの算術式中に行なわれるべき算術演算と identifier の出現は，ほぼ同じ回数だけあることからも有効な方法として考えられる．

4. transformation phase

上記の構成を持つ計算機は3つの phase を経て，ALGOL プログラムを処理する (**Fig. 3**).

最初の language phase では，制御ユニットの固定記憶装置に ALGOL に対応するマイクロプログラム

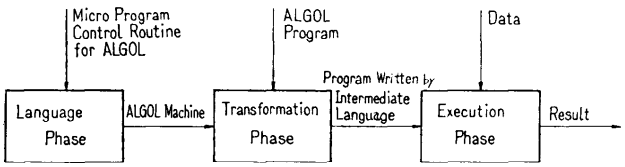


Fig. 3 Processing ALGOL Programs in Three Phases

で書かれた制御ルーティンを記憶させる．つぎの transformation phase において，ALGOL で書かれたプログラムを入力しつつ，中間言語に変形し，それが終了した時点で execution phase に移行する．execution phase では，データとともに，中間言語に変形された ALGOL プログラムを実行する．

ALGOL プログラムの変形を行なう tranformation phase の制御ルーティンは，やはりマイクロプログラムで書かれ，以下の操作を含む．

(1) プログラムを計算機の各部から内部表現に変換しつつ入力する．

(2) label の explicit declaration を導入する．さらに block head に相対な statement の nesting level (以下 S_d と略) を決定する．これは delimiter "begin" が出現すると S_d を1にセットし，新しい statement がはいるたびに $S_d=S_d+1$ とし，statement から出るときは $S_d=S_d-1$ とすることによって達成される．

(3) block の静的な nesting level (以下 S_n と略) を導入する．S_n は execution phase において identifier メモリの操作，とくに，identifier の検索を行なう際のスタティックチェインに利用する．

(4) expression の前後に左端および右端を示す標識 "⊢" および "⊣" をつける．さらに各 expression をオペレータ後出しのポーランド記法に変換する．

(5) implicit control transfer の行先を具体的に指定する．これは，たとえば，conditional statement の場合 "then" の前に "else" の番地を，"else" の前にこの statement の終了番地を与えることによって，実行時に不必要な scan の繰り返しを避けるためである．

(6) プログラム中に現われる数値は適当な標識に置き換え，数値はデータメモリ内に記憶する．さらに，その対照表を identifier メモリ内に記憶する．これはプログラムメモリがバイト構成であり，しかも，数式を実行する際にアキュムレータスタックを用いることによる．同様に transformation phase にそう入されるプログラムメモリのアドレスもデータメモリ内に記憶され，プログラムメモリにはその標識がはいる．

(7) 文法的な誤りの点検．以上の操作を行なうことの付帯的な結果として，かなりの文法的な誤りを調べることができる．

(i) プログラムの内部表現

ALGOL delimiter は全部で 50 個あまりあり，さらに transformation phase によってつけ加えられる各種の標識 (数値に対応するものを除く) を含めても5ビットで表現できる．identifier には7ビットを用意する．これによって 124 個の identifier を用いることができる．identifier が block に local であること，また array の使用などを考えれば個数としては十分であると考えられるが，プログラマに不当な制限を与えることになるかも知れない．プログラム中に出現する数値に対する標識，および transformation phase にそう入される番地に対する標識は，数値および番地であることの指定をも含めて7ビットを用いる．したがって，delimiter, identifier, 数値および番地に対する標識は **Fig. 4** のように区別して8ビット記憶装置内におさめられる．

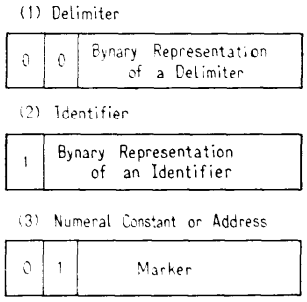


Fig. 4 Inner Representation of Program

(ii) データの内部表現

数式がポーランド記法で表現されているから，アキュムレータスタックを用いなければならない．したがって，バイト構成の計算機にすべて普通であるような variable size のデータを許すならば，その取扱いは非常に複雑になる．この計算機では，データは，48 ビットを1語とする語構成の記憶装置に記憶される．浮動小数点表示と固定小数点表示の数は混同して扱われる．そのために，小数点の位置は最下位の桁におかれるべきである．"logical value" の "truth" および "false" はそれぞれ "1" と "0" に対応させることによって処理する．type の整合は，assignment statement の際，一度だけ行なわれる．

(iii) label の処理

このシステムでは，identifier が identifier メモリに記憶されるのは，インタープリターによって declara-

tion が処理されるときである．したがって，label が未定義のままに goto statement によって refer された場合には，対応する implicit label declaration が現われるまで scan を続けなければならない．これを避けるために，transformation phase の段階で，explicit label declaration を導入する．label は block に対して local であるから，label の出現したプログラムの位置を囲む最小の block において，explicit に declaration がなされればよい．この label declaration に際して，与えなければならないのは label の出現したプログラム中の位置およびその位置の statement nesting level S_d である．explicit label declaration は **Fig. 5** に示された方法による．ただし，Fig. 5 で

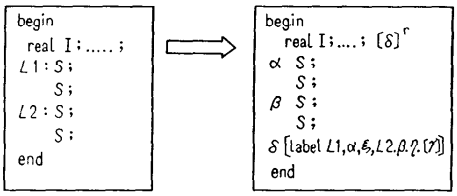


Fig. 5 Explicit Label Declaration $\alpha, \beta, \iota, \delta$ are adresses in Program memory, and ξ, η are statement nesting levels

α, β はそれぞれ L_1, L_2 の label がつけられた statement の最初の delimiter がはいっている番地，δ は "end" の一つ前の delimiter がはいっている番地のつぎの番地であって，ここより explicit label declaration が始まる．γ は label 以外の declaration の終了した地点のつぎの番地であり，ここにアドレス δ に対応する標識が記憶される．ξ, η はそれぞれ L_1, L_2 が出現したプログラムの位置の statemente nesting level S_d である．ここに述べられた処理の方法を実際行なうためには，transformation phase においてスタックを用いる必要がある．

(iv) スタティック チェイン

identifier は block に local であるから，プログラムのある位置で用いられた identifier は，その位置を textually に囲むどれかの block で declare されている．したがって，identifier の検索は block の textually なリスト構造をたどることによって行なわれる．しかし，ALGOL の場合，実行時には，この block の静的なリスト構造は保存されない．それゆえ，transformation phase において，あらかじめこれを与えておく必要がある．transformation phase で block head に，textually にそれが直属している親 block の declaration の終了番地を書き込む．execution phase では，declaration の終了番地に identifier メモリの current available address を記憶する．block にはいると，その親 block の declaration の終了番地の内容（すなわち，親 block の identifier が入れられた identifier メモリの最後の番地）を調べ，それを，identifier メモリの current avairable adress に記憶する．これによって block の identifier の検索が終了し，所要の identifier の declaration が見い出されなかったとき，親 block を調べることができる．

5. execution phase

transformation phase における処理が終わると execution phase に移行する．execution phase は，データを計算機の内部に入力し，変形された ALGOL プログラムを実行する過程であって，演算の流れの制御，identifier に関する操作，プログラムおよびデータメモリの制御，算術演算などよりなる．

(1) push down 制御

普通の計算機の制御は

(i) 命令の逐次的な実行，

(ii) jump 命令による sequentiality の変更，

(iii) インデックスレジスタによる命令の修飾，の3種類の操作による構成に従う．これは，機械語がきわめて単純な文法構造を持っていることの反映であって，いま考えている計算機では，制御はこれよりもかなり複雑になる．入力記号 (delimiter) は逐次的に実行され，control transfer があれば，この sequentiality が破壊される．この点については，普通の計算機と同様であるが，修飾に関する考え方が基本的に異なる．すなわち，単に delimiter だけでは，実行すべき演算を一義的に決定することはできない．delimiter はその領域，すなわち，それが出現した時点までのプログラムの構造に関係して意味を持つ．したがって，その時点までのプログラムの構造をなんらかのかたちで記憶しておき，それによって常に delimiter を修飾することにより，所要の演算を実行することができる．

context free grammar を文法構造として持つ言語では，プログラムの構造をスタックで取り扱うことができる．さきに述べたように，この計算機で用いられている s-deterministic grammar では，generatrix-handle pair で計算機の動作を決定できる．handle はもとより delimiter に対応し，generatrix は handle

を修飾するものであり，スタックで操作される．

また，制御の流れの変更も，単にプログラムのある地点に飛び移ることを意味しているだけでなくて，プログラムの構造の位置をも変化させなくてはならない．このように，ALGOL machine では，制御自身が，infinite の記憶装置を持つ機構で構成される必要がある．ここでは，このような制御方式を push down 制御と呼んでいる．

(2) control formula

さきにも述べたように，s-deterministic grammar の構造分析は書き換え規則によって自然に決定される．Appendix B で示された ALGOL の中間言語の文法は，それゆえ，そのままのかたちで，それを解析する計算機の制御規則にもなっているのである．実際は，この中間言語はインタープリートされるため，計算機全体の制御をこのように簡単に表現することはできない．しかし，基本的には，文法の構造分析を行ない，その過程で決定された演算を実行するのであるから，Appendix B の書き換え規則に jump 命令その他による control transfer をつけ加えれば，この計算機の execution phase における制御の流れを表現する規則を比較的簡単に書き表わすことができる．それを **Table 1** に示す．

この制御規則を Fig. 1 で示された計算機を用いて実行するには，つぎのような方法による．

現在の状態はCレジスタが持っている．Cレジスタの内容と読み込まれた delimiter で（これは generatrix -handle pair を構成するが）その解釈ルーティンの最初のマイクロ order の記憶されているアドレスを構成する．結局，generatrix-handle pair が普通のマイクロプログラム制御方式の計算機の命令を構成する．つぎの状態（BまたはC）およびスタックへの入力は，そのマイクロ order の flow control area に書かれる．それによって計算機のつぎの状態が決定される．

この control formula は，identifier の declaration，その他の identifier メモリの制御は全く含まない．これらは，identifier の検索を含んでおり，それゆえ context sensitive な述語系を構成しているから，したがって，どのような表記法によるにしろ，なんらかのかたちで，処理デバイスを規定することなしには，これらの表現は簡単にはいかないと思われる．

(3) control transfer

このシステムでは，スタックで計算機の過去の情報を記憶しておくという，いわば，計算機全体が層をなした制御を行なっているため，control transfer が行なわれた場合，プログラムのどの地点に制御を移すかということだけでなく，状態のどの深さの層にもどるかという点が問題になる．これを以下で考察する．

conditional statement 中の implicit control transfer には，ほとんど問題はない．procedure についても control transfer に限っては簡単である．問題は goto statement による explicit control transfer にある．**Fig. 6** の statement のリストで goto statement によって移行可能なプログラムの range はどの部分であろうか．ただし，transformation phase における label の処理によって goto statement が出現した時点で，label はすでに定義されている．

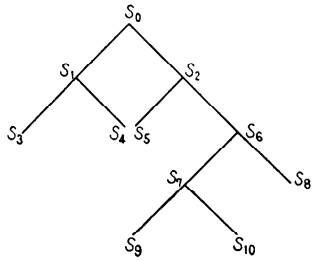


Fig. 6 Statement List

S_8 が goto statement であるとしよう．S_7 へは，nesting の level が同じであるから，スタック類の変更なしに到達可能である．S_9, S_{10} へは，スタックに新しい状態を push down することなしには，直接制御を変更できない．しかし，いかなる状態を記憶すればよいかを決定することはできないので，この計算機では到達不可能である．それゆえ，このような現象が起こることを防ぐ意味で，以下では，goto statement によって，for statement および if statement に飛び込むことは禁止する．

S_5 へは到達可能である．しかし，S_5 へいくときは，S_8, S_6 の状態を pop up し，S_2 の状態へはいらなければならない．やはり，ネットをつくる node の短絡が許されることはけっしてない．結局，label は（それゆえ，goto statement の transfer の range は）block に local であり，しかも，その block は，その label がつけ加えられた statement を囲む最小の仮想的 block と考えなくてはならないから，goto statement は statement のリスト構造を保存する．したがって，goto statement によってスタック類の順序系列が破壊されることはけっしてない．それは単にいくつかの

Table 1

Appendix A Syntax of a small language

<arithmetic expression>::=<term>|±<term>|
<arithmetic expression>±<term>
<term>::<factor>|<term>*er/<factor>
<factor>::<primary>|<factor>↑<primary>
<primary>::=<identifier>|(<arithmetic expression>)
<boolean expression>::=<arithmetic expression>
<relational operator><arithmetic expression>
<if clause>::=**if**<boolean expression>**then**
<if statement>::=<if clause><unconditional>
<assignment statement>::=<identifier>:=
*<arithmetic expression>|<identifier>:=
*<boolean expression>
<goto expression>::=**goto**<identifier>
<conditional>::=<if statement>|
<if statement> **else** <statement>
<unconditonal>::=<assignment statement>|<block>|
<goto statement>
<declaration>::=<type><type list>|
<declaration>; <type><type list>
<type list>::=<identifier>|
<type list>, <identifier>
<head>::=**begin**<declaration>|
<head>; <statement>
<block>::=<head> **end**
<statement>::=<conditional>|<unconditional>|
<identifier>:<statement>
<type>::**real**|**integer**
<relational operator>::==|>|<

Appendix B intermediate form of a small language

<program>::=**begin**<block>
<block>::=**real/integer**<declaration>|
if<conditional>|**id**<assignment><state>|
goto<goto><state>|**begin**<block><state>
<state>::=; <tail>|**end**
<tail>::=**if**<conditional>|**id**<assigrment><state>
|**goto**<goto><state>|**end**
<conditional>::=⊢<exp><ifcl>
<ifcl>::=**then**<uncond><cond *A*>
<cond *A*>::=**else**<tail>|; <tail>
<uncod>::=**begin**<block>|**goto**<goto>|
id<assignment>
<goto>::=**id**
<declaration>::=**id**<dec *A*>
<dec *A*>::=, <declaration>|; <block>
<assignment>::=⊢<exp><asg *A*>
<asg *A*>::=:=
<term>::=**id**|<term><term><arithmetic operator>
<exp>::=<term>⊣|<term><term>
<relational operator>
<arithmetic operator>::=+|−|*|/
<relational operator>::= >|<|=

Appendix C flow control formula for a small language

(<program>**begin**)→<block>
(<block>*n*)→*α*(*n*)<labeldec>
(<labeldec>**label**)→<labeldec>
(<labeldec>**id**)→<labeldec *A*>
(<labeldec *A*>;)→<labeldec *B*>
(<labeldec *A*>,)→<labeldec>
(<labeldec *B*>*n*)→*α*(*n*)<tail>
(<block>**real/integer**)→<typedec>
(<typedec>**id**)→<dec *A*>
(<dec *A*>,)→<type dec>
(<dec *A*>;)→<block>
(<block>**if**)→<conditional>
(<block>**id**)→<assignment><state>
(<block>**goto**)→<goto><state>
(<block>**begin**)→<block><state>
(<state>**end**)→*ϕ*
(<state>;)→<tail>
(<tail>**if**)→<conditional>
(<tail>**id**)→<assignment><state>
(<tail>**goto**)→<goto><tail>
(<tail>;)→<tail>
(<tail>**end**)→*ϕ*
(<assignment>⊢)→<exp><asg *A*>
(<goto>**id**)→*α*(**id**)
(<conditional>⊢)→<exp><if cl>
(<if cl>*n*)→*β*(exp)$\begin{cases}\text{<if cl>}\\ \alpha(n)\text{<if cl>}\end{cases}$
(<if cl>**then**)→<uncond><cond *A*>
(<if cl>**else**)→<cond *B*>
(<cond *A*>;) →<tail>
(<cond *A*>**end**)→*ϕ*
(<cond *A*>*n*)→*α*(*n*)<tail>
(<uncond>**begin**)→<block>
(<uncond>**goto**)→<goto>
(<uncond>**id**)→<assignment>
(<cond *B*>**if**)→<conditional>
(<cond *B*>**begin**)→<block>
(<cond *B*>**goto**)→<goto>
(<cond *B*>**id**)→<assignment>
(<asg *A*>:=)→*ϕ*
n: the location of a program memory
α(*n*): advance the pointer to location *n*
α(**id**): advance the pointer to label **id**
β(exp) { }: select the upper case if the value of expression is 1 and select the lower case otherwise
Appendix *B* および *C* では，identifier を **id** と略記し，terminal symbol として取り扱っている．

pop up を含むだけである．したがって，もし delimiter を1つ1つ scan しながら指定された label へいく方式をとるならば全く問題はない．しかし，直接指定されたプログラムの位置に制御を移すなら，問題の lable が，statement の nesting の level のどの深さにあるか，それゆえ，どれだけの量の pop up が必要であるかをあらかじめ知っておかなければならない．

この仕事は transformation phase で行なわれる．すなわち，transformation phase で statement の nesting の level が計算され，lable の explicit declaration に際して，この値が書き加えられる．しかし，execution phase で実行が行なわれた場合の nesting structure は textually なもの（すなわち，静的な statement nesting structure）と異なるのではないかという心配

Table 2 Appendix D Description of the machine operation in the execution phase

routine NO.	current A		flow control area	identifier memory area	main memory area	arithmetic operation area
	A_1	A_2				
0	start		C←program, D←0 N←initial, p←0, A_1←1	I←0	O←0	
1			A←2		M←N, f_M: read	
2			f_A: read A_1←C, A_2←Q			
3	program	begin	C←block, p←p+1 N←N+1, A←4	H←I, J←I		
4			A←5	J←J−1		
5			A←6	K←(bh, P, 0, J; f_s: write		
6			A←7	I←I+1		
7			A←8	H←I, K←(0, 0, , 0) f_s: write		
8			A←1	I←I+1		
9	block	integer/real	C←type D N→N+1, A←1			
10	block	n*	C←Address A←2		M←Q, f_M: read	X→Q
11	block	if	C←conditional N←N+1, A←1			
12	block	goto	$H.S$←goto, C←state N←N+1, A←13			
13			D←D+1 A←2		M→N f_M: read	
14	block	begin	$H.S$←block, C←state N←N+1, A←15	H←I, J←I		
15			D←D+1 A←2			
16	block	id**	$H.S$←assign, C←state A←17	L←(Q, P, ,) J←I		
17			D←D+1, A←18			
18			$H.S$←C, N←N+1 A←19			
19			D←D+1, A←20			
20			$H.S$-C←21 A←79			
21			A←1			E←K^4, F←K^3 G←G+1
22	state	;	C←tail, N←N+1 A←1			
23	state	end	$H.S$-C←24 A←79	L←($b.h$, p, ,) J←I		
24			D←D−1 A←25	I←K^4 H←H+1, f_s: read		
25			C→$H.S$, N←N+1 A←1		$\bar{O}$←K^4	
26	tail	if	C←conditional N←N+1, A←1			
27	tail	id	$H.S$←state C←assign N←N+1, A←13			
28	tial	goto	$H.S$←tail, C←goto N←N+1, A←13			
29	tail	end	A←20			
30	type D	id	C←Dec$A$, N←N+1 A←31	H→I, K←(Q, p, type, O) f_s: write		
31			A←2	I←I+1	M←N, f_M: read $\bar{O}$←$\bar{O}$+1	
32	Dec A	,	C←type D, N←N+1 A←1			
33	Dec A	;	C←block A←1			
34	Address	label	C←label D, N←D+1 A←1			
35	Address	n	C←block, N←N+1 A←1	H←I, K^3←Q f_s: write		
36	label D	id	N→N+1 A←37	K←(Q, p, label.) H←I, f_s: write		
37			A←8	M←N, f_M: read		

38			$A\leftarrow 39,\ N\leftarrow N+1$	$K^4\leftarrow Q,\ f_s$: write $I\leftarrow I+1$		
39			$A\leftarrow 40$	$H\leftarrow I$	$M\leftarrow N,\ f_M$: read	
40			$A\leftarrow 41,\ N\leftarrow N+1$	$K\leftarrow(0, P, 0, Q)$ f_s: write		
41			$A\leftarrow 2$	$I\leftarrow I+1$	$M\leftarrow N,\ f_M$: read	
42	label D	,	$N\leftarrow N+1$ $A\leftarrow 1$			
43	label D	;	$C\leftarrow$label DA $N\leftarrow N+1,\ A\leftarrow 1$			
44	label DA	n	$C\leftarrow S$-chain $A\leftarrow 2,\ N\leftarrow Q$		$M\leftarrow Q,\ f_M$: read	
45	assign	$\vdash$	$H.S\leftarrow$asg $A,\ C\leftarrow$exp $N\leftarrow N+1,\ A\leftarrow 13$			
46	goto	id	$H.S\text{-}C\leftarrow 49$ $A\leftarrow 79$	$J\leftarrow I$ $L\leftarrow(Q, P, ,)$		
47			$N\leftarrow K^4$ $A\leftarrow 48$	$H\leftarrow H+1$ f_s: read		
48			$D\leftarrow K^4+K^2+1,\ p\leftarrow K^2$ $A\leftarrow 2$	$J\leftarrow I$ $L\leftarrow(b.h, K^2+1, ,)$		
49			$H.S\text{-}C\leftarrow 50$ $A\leftarrow 79$			
50			$A\leftarrow 51$	$H\leftarrow H+1\ f_s$: read $I\leftarrow K^4$		
51			$C\leftarrow H.S,\ N\leftarrow N+1$ $A\leftarrow 1$		$\bar{O}\leftarrow K^4$	
52	cond	$\vdash$	$H.S\leftarrow$ifcl, $C\leftarrow$exp $N\leftarrow N+1,\ A\leftarrow 13$			
53	if cl	n	$\begin{cases} FF\ 1=1\rightarrow N\leftarrow N+1 \\ A\leftarrow 1 \\ FF\ 1=0\rightarrow N\leftarrow A_2 \\ A\leftarrow 2 \end{cases}$			
54	if cl	then	$H.S\leftarrow$cond$A$, $C\leftarrow$uncond $N\leftarrow N+1,\ A\leftarrow 13$			
55	uncon	begin	$C\leftarrow$block, $N\leftarrow N+1$ $A\leftarrow 1$			
56	uncon	goto	$C\leftarrow$goto, $N\leftarrow N+1$ $A\leftarrow 1$			
57	uncon	id	$C\leftarrow$assign $N\leftarrow N+1,\ A\leftarrow 1$			
58	cond A	n	$N\leftarrow Q,\ A\leftarrow 1$			
59	cond A	;	$C\leftarrow$tail, $N\leftarrow N+1$ $A\leftarrow 1$			
60	cond A	end	$A\leftarrow 23$			
61	if cl	else	$C\rightarrow$cond B $N\leftarrow N+1,\ A\leftarrow 1$			
62	cond B	if	$C\leftarrow$cond, $N\leftarrow N+1$ $A\leftarrow 1$			
63	cond B	begin	$C\leftarrow$block, $N\leftarrow N+1$ $p\leftarrow p+1,\ A\leftarrow 1$			
64	cond B	goto	$C\leftarrow$goto, $N\leftarrow N+1$ $A\leftarrow 1$			
65	cond B	id	$C\leftarrow$assign, $N\leftarrow N+1$ $A\leftarrow 1$			
66	asg A	:=	$N\leftarrow N+1,\ D\leftarrow D-1$ $A\leftarrow 67$			
67			$C\leftarrow H.S$ $A\leftarrow 68$		$V\leftarrow E,\ U\leftarrow F$	$G\leftarrow G-1$
68			$A\leftarrow 2$			check (type $V=F$)
69			$A\leftarrow 2$		$M\leftarrow N$ f_D: write	
70	exp	id	$H.S-C\leftarrow 71$ $A\leftarrow 79$	$L\leftarrow(Q, p, ,)$ $J\leftarrow I$		
71			$C\leftarrow$exp $N\leftarrow N+1$ $A\leftarrow 72$		$U\leftarrow K^4$ f_D: read	
72			$A\leftarrow 73$		$M\leftarrow N$	$E\leftarrow V,\ G\leftarrow G+1$
73			$A_1\leftarrow C,\ A_2\leftarrow Q$			$F\leftarrow E$
74	exp	$\overset{\times}{\mp}$	$N\leftarrow N+1$ $A\leftarrow 1$			$E\leftarrow F\overset{\times}{\pm}E,\ G\leftarrow G-1$
75	exp	/	$A\leftarrow 76$			$E=0\rightarrow$check
76			$N\leftarrow N+1$ $A\leftarrow 1$			$E\leftarrow F\div E$
77	exp	$\lessgtr$	$N\leftarrow N+1$ $A\leftarrow 78$			$F\lessgtr E\rightarrow FF.\leftarrow 1$ otherwise $FF.1\leftarrow 0$
78	exp	$\dashv$	$C\leftarrow H.S,\ N\leftarrow N+1$ $D\leftarrow D-1,\ A\leftarrow 1$			

79			$A\leftarrow 80$	$H\leftarrow J$ f_s: read		
80			$A\leftarrow 81$	$L=K\rightarrow FF.2\leftarrow 1$ otherwise$\rightarrow FF.2\leftarrow 0$		
81			$FF.2=1\rightarrow A\leftarrow 82$ $FF.2=0\rightarrow A\leftarrow 83$			
82			$A_1\leftarrow H.S,\ A_2\leftarrow C$			
83			$A\leftarrow 84$	$K^1=b.h\rightarrow FF.2\leftarrow 0$ othewise$\rightarrow FF.2\leftarrow 1$		
84			$FF.2=1\rightarrow A\leftarrow 85$ $FF.2=0\rightarrow A\leftarrow 90$			
85			$A\leftarrow 86$	$J\leftarrow J-1$		
86			$A\leftarrow 87$	$J=0\rightarrow FF.2\leftarrow 1$ $J\neq 0\rightarrow FF.2\leftarrow 0$		
87			$FF.2=0\rightarrow A\leftarrow 79$ $FF.2=0\rightarrow$check			
88			$A\leftarrow 89$	$H\leftarrow J+1,\ f_s$: read		
89			$A\leftarrow 79$	$J\leftarrow K^3$		
90	s-chain	n	$D\leftarrow D-1$ $N\leftarrow N+1$		$M\leftarrow Q$	
91			$A\leftarrow 1$		$Q\leftarrow I,\ f_D$: write	

* a marker of address
** identifier

がある．これは ALGOL の場合正しい．事実 block 単位での textual な nesting structure は procedure にはいることによって完全に破壊される．しかし，block 内の，それゆえ block の head に相対な statement の nesting の level S_d は保存される．したがって，transformation phase では，block の head に相対的な statement の nesting level S_d はこれを知ることができるので，この値を計算し，explicit label declaration に付加する．execution phase で動的な block level を計算し，これに，さきの S_d を加えることによって，その時点のスタックの深さを知ることができる．

それ以外のスタックの場合は，ほとんど問題はないものと思われる．identifier メモリは label の所属する block までメモリの内容を消去すればよい．データメモリの消去は identifier メモリを介して行なわれる．しかし，function の abnormal exit は禁止する．これは，オペランドスタックを用いて算術演算を処理していることによるものである．

6. 計算機の動作の記述

この計算機では，transformation phase および execution phase におけるオペレーションは，すべてマイクロプログラムで制御される．したがって，プログラミング言語に対して，コンパイラを用意する代わりにマイクロプログラムで書かれた制御ルーティンを用意しなければならない．それゆえ，コンパイラ作製の場合と違って，記憶装置からレジスタ，あるいはレジスタからレジスタへの情報の転送や，フリップフロップのオン・オフ，演算回路の作動などのマイクロオペレーションによって，実行される演算を表現する必要がある．

ここでは，細かい点に至るまでは計算機の構成を与えていないので，Fig. 1 の概念図に従い，マイクロプログラムにきわめてよく似たテーブルを用いて計算機の動作を表現している．ALGOL machine $T=(\Xi, H, Z, \Theta, \Lambda, K)$ の構造に従い，テーブルの各行（1マイクロ order に対応）は，flow control area, identifier memory area, main memory area, arithmetic operation area の部分よりなる．

flow control area は，制御ユニット Ξ，固定記憶装置 Λ の操作を記述する．とりわけ，状態レジスタへの入力(つぎの計算機の状態)，状態スタックの push down, pop up（カウンタの制御），push down すべき状態などを指定する．execution phase において，Pカウンタは，動的な block nesting level を，Nカウンタはつぎの delimiter (transformation phase では，プログラムメモリのつぎに利用可能なアドレス）をそれぞれカウントすることから，PおよびNカウンタの制御もこの area で記述される．つぎに実行すべきマイクロ order の記憶されている固定記憶装置 Λ の番地が，直接 flow control area で指定されることもある．

identifier memory area には，identifier に関する操作がすべて記入される．identifier の検索はその interrogative part が L レジスタに記入され，Jカウンタによって順次 identifier メモリの内容がKレジスタに読み出され，それとLレジスタの論理積をとるこ

とによって行なわれる．Hレジスタには，読み出し，または読み込まれるべき番地を指定する．I レジスタは identifier メモリの next available address である．identifir メモリレジスタへの読み込み，読み出しなどのゲート信号もこの area に記入される．

main memory area には，プログラムメモリおよびデータメモリの2つの記憶装置に関する操作を含む．読み込み，読み出しなどのゲート信号もこの area に記入される．O レジスタは execution phase において，データメモリの2つぎに利用可能なアドレスを指定するために，この area でその制御が記述される．

arithmetic operation area は，算術演算ユニットへの呼び出し信号を指定する．オペランドスタックの push down, pop up (T カウンタの動作)，E, F レジスタの操作もこの area に属する．

以上のテーブルは，固定記憶装置 A へ記憶される，固定記憶装置内の各マイクロ order は，Aレジスタ内の delimiter と状態スタックの pair がAレジスタに入力されることによって呼び出されるか，または flow control area 内で，Aレジスタへの入力（番地）がそのまま指定されることによって呼び出されるかどちらかである．

とくに，execution phase においては，このテーブルは**2.**で述べたテーブルMの具体化であり，ALGOL の場合は Appendix D で与えられる．

7. おわりに

この論文では，文法構造の観点から，容易に，しかも高速にインタープリートできる中間言語を設定し，それを機械語とする計算機を考察した．中間言語には，さまざまのものが考えられる．しかし，従来の direct execution に関する試みでは，プログラミング言語の諸機能のうち，現在用いられている計算機の構成を多少拡張することにより，効率的に実行可能なものを選び出し，中間言語を設定して，プログラミング言語の他の諸機能をその中間言語に還元するという方式をとっているものが多い．当論文では，それとは反対に，インタープリテーションという処理の方法がしいる条件を満たす範囲で，できるだけプログラミング言語に近い中間言語に，対象とする言語を設定した結果，従来の方式の計算機とは，さまざまの点で異なる機構を考察する結果になった．

Von-Neumann 方式の計算機では，データと命令を区別せず，ともに演算の対象とする点にその特徴があるが，ここで構成した ALGOL machine では，データとプログラムは完全に区別される．すなわち，データに対しては，普通の記憶装置を，プログラムに対しては，比較的書き込み速度の遅い半固定記憶装置を用意している．さらに，普通の計算機では，命令は原則としてそのままで実行されるが，この計算機では，状態との pair によって，すなわち，スタックから常に修飾をうけることによって実行される．

従来のプログラミング言語の処理方式と，ここで述べた方式の優劣についての比較は，やはり，さまざまの場合を想定したシュミレーション実験に待つほかはないと思われる．コンパイラとの効率を比較するうえで重要な点は，identifier とアドレスの変換を transformation phase でなく，execution phase に実行していることである．これは ALGOL の動的な性質を尊重したことや，transformation の終わった段階で，プログラム，とくに identifier がほとんどそのまま残されており，それゆえ，デバックの作業が軽減されるためである．しかし，identifier とアドレスの変換は，必ず identifier メモリの検索を含むので，かなりの時間を必要とし，繰り返し計算の場合などは大きな損失である．しかし，それ以外の点に関しては，ここで述べた処理方式の方がはるかに高速であるので，計算機の規模や，使用目的によっては，十分コンパイラに対抗できるのではないかと考えられる．全体的な効率の点から見れば，コンパイラに常時占有されていた記憶装置を他の用途に用い，制御ルーティンは比較的安価な固定記憶装置を利用することによる有効性も考えられる．

謝 辞

このテーマに関して，ご指導いただいた京都大学・工学部，萩原 宏教授に感謝する．

参 考 文 献

1) P. Wegner: Programming Languages, Information Structures, and Machine Organization, McGRAW HILL, 1968.
2) H. W. Lawson: Programming Language Oriented Instruction stream, IEEE TRANS on Computer, Vol. C-17, 1968.
3) J. P. Anderson: A Computer for Direct Execution of Algorithmic Language, EJCC, Vol. 20, 1961.
4) J. Melbourne, et. al.: A small Computer for Direct processing of FORTRAN statement, Computer. J. Vol. 8, 1965.

5) T. R. Bashkow, et al: System Design for a FORTRAN Machine, IEEE TRANS on EC, Vol. EC-16, 1967.
6) H. Weber: A Microprogrammed Implementation of Euler on IBM System/360 Model 30, C. ACM. Vol. 12, 1965.
7) S. A. Greibach: A New Normal Form Theorem for Context Free Phrase Structure Grammar, J. ACM. Vol. 12.
8) D. E. Knuth: On the Translation of Language from Left to Right, Infrom & Control, Vol, 8, 1965.
9) S. Y. Levy, et. al.: System Utilization of Large-Scale Integration, IEEE TRANS on EC, Vol. EC-16, 1967.
10) Z. PAWLAK: New Class of Mathematical Languages and Organization of Addressless Computers, Colloquium on the Foundations of Mathematics, Mathematical Machines and Their Applications, 1962. AKADÉMIAI KIADÓ.
11) A. Opler: Fouth Generation Software, Datamation, Vol. 13, 1967.
12) S. A. Greibach: Formal Parsing System, C. ACM. Vol. 7.

(昭和44年12月17日受付)