Compiler 記述言語：COL*

萩 原 宏** 渡 辺 勝 正**

It is required to construct compilers easily and effectively on some machine, and many efforts have been made for it.

As one method for it, we have constituted the compiler describing language COL which is suitable to write the compiling processes.

The compiler, which is described in the forms of the SYNTAX TABLE and M-routines, divides the source program into some Incremental Units (IU) which are able to be compiled independently of other parts of the program, constructs M-structure denoting the syntactic structure of IU and generates three-address pseudo codes depending on several informations in the M-structure.

In this paper, the COL is described informally and two simple examples are given, but any experimental results are not given.

1. 序 論

電子計算機のめざましい発展と，その応用面の拡大に対して，様々の問題向き言語（POL）の Compiler を短期間に能率良く作ることが要求される．そのため，機械語，アセンブラ言語で Compiler を記述する段階から，higher level language で記述することが種々試みられている．たとえば，ALGOL[1] のサブセットや FORTRAN[2] に必要な表現能力を加えた言語または PL/I のサブセットを用いて Compiler を記述する．あるいは「Compiler を記述・作成するという問題」に適した POL を設計し，それを利用する[3~7]．

前者は，既存の Compiler で新たな Compiler を作り出すという利点を持つ．他方，後者は，より能率良く，Compiler を記述し得る長所を持つ．本論文に述べる COL は，後者の立場に立って，Syntax-Oriented Compiler を簡明に表現し，その動作を的確に記述することによって，Compiler 作成過程におけるコーディングの労力を軽減し，デバッグを容易にして，各種の POL の Compiler や，目的の違った Compiler を短期間に作成するために構成された Compiler 記述言語である．

Compiler の syntax analysis の過程については，その自動化・定式化を行ないうる状態にいたっている．COL で書かれた Compiler では，さらに syntax analysis の結果，文脈中の3カ所からの情報によってコンパイル過程を効果的に進めることを試みるコンパイル方法が取られている．

まだ実験結果を報告するにはいたってないが Compiler Compiler へのアプローチの一方法（あるいは firmware[8] といわれるものへの移行）として，皆様の御批判をあおぐ次第である．

2. Compiler の構成

COL で記述された Compiler は，POL で書かれた Source Program を，Parsing phase および Translating phase を通して Object Program に変換する (**Fig. 1**)．

Parsing phase は

（i） Source Program を読み，input symbol あるいは string of symbols を識別・判定する．

（ii） 判定に基づいて Source Program の文脈の構造を知り，その構造を表わす M-structure を構成しながら，次の parsing 動作を決定する．

（iii） 文脈の解析が完全に行なわれた時点で，Translating phase にうつる．すなわち，Source Program の一部分において，その文脈の構造が完全に決定され，それだけで Object Program に変換できる

* Compiler Describing Language: COL, by Hiroshi Hagiwara and Katumasa Watanabe (Faculty of Engineering, Kyoto University)

** 京都大学数理工学教室

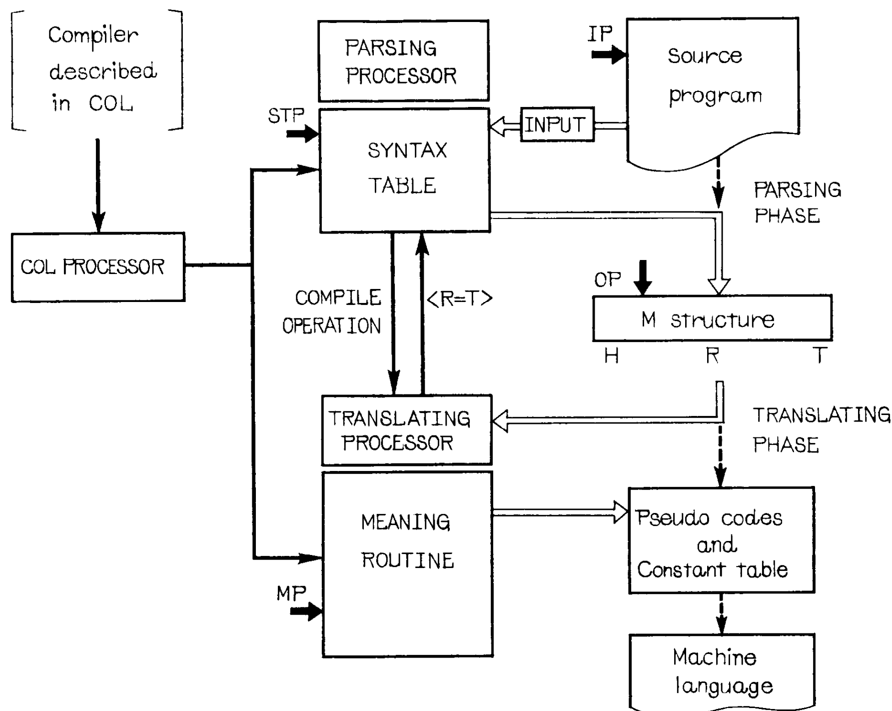

Fig. 1 COL System and Compiling process

部分（それを Incremental Unit と呼び IU と略す）があれば，その部分の parsing を終えたところで，syntax analysis を一時中断し，IU のシンタックス構造を表わす M-structure を持って，Translating phase にうつる．

M-structure は，IU の構造解析の結果，Translating phase で取るべき処理ルーチンの名前と，identifier の内部表現とで構成されており，expression の部分は Reverse Polish form で表現される．

IU は POL に応じて適当に選ぶことができるが，ALGOL サブセットをモデルとした例（5．を参照）では，one statement[9], one declaration および one label をその単位としている．

次に Translating phase は，与えられた M-structure を左から右に走査して，指定された名前の処理ルーチン（これを Meaning Routine といい M-routine と略す）を実行する．各 M-routine は，番地の割付け，identifier の処理，必要な情報の伝達・記憶を行ないながら，Pseudo Code の Object Program を構成する．一つの M-routine の実行が終ると，M-structure の中に指定された次の M-routine にうつる．M-structure 中のすべての M-routine が実行されると，現在の IU に対する Object Program への変換が完了したことになる．Compiler のコントロールは Parsing phase に戻されて，Parsing を再開し，次の IU の構造解析を行なう．

一つの POL に対する Compiler は，Parsing phase をコントロールする parsing algorithm と，Translating phase に実行される M-routine とを COL で表現して与えられる．parsing algorithm は，POL のシンタックスに基づいて作られた SYNTAX TABLE に従う．SYNTAX TABLE には，Source Program を読む READ OPERATION や，IU の parsing を終えて Translating phase にうつることを示す COMPILE OPERATION などが含まれている．一方 M-routine は，一連の定められた BASIC STATEMENT によって記述される．

COL PROCESSOR は，各種の POL の Compiler に対して共通であり，一つの計算機にあらかじめ一つ用意されれば十分である．SYNTAX TABLE に指定される OPERATION および M-routine の BASIC STATEMENT は単純なものであるから，COL PRO-

CESSOR は，比較的簡単なものになるであろう．

3. Parsing Processor

3.1 SYNTAX TABLE

SYNTAX TABLE は，POL のシンタックスに基づいて，READ & TEST OPERATION とそれに対する ACTION OPERATION によって **Fig. 2** のように構成される．

0	1	2	3
LABEL	READ & TEST OPERATION	T. ACTION	F. ACTION

Fig. 2 format of SYNTAX TABLE

Parsing Processor は，SYNTAX TABLE に指定されたそれらの OPERATION に従って Source Program を読み，次の順序で parsing を進める．

（i） 1欄の READ & TEST OPERATION を行なう．

（ii） TEST の結果が「真」なら，2欄の T. ACTION に従い，「偽」なら，3欄の F. ACTION に従う．

(iii) READ & TEST は TABLE の上から順に行なう．

(iv) T. および F. ACTION で，次の READ & TEST を指定する場合，および TEST OPERATION がある syntactic unit の確認を要求する場合には，指定された READ & TEST にうつる．そのため，READ & TEST には LABEL をつけることができる．

Parsing Processor は，三つの pushdown storage を持つ．pushdown storage は

〔head〕↔〔tail〕

の構造を持ち，head をそれぞれ，IP, OP, STP と呼ぶ．

IP: Source Program の中で，現在調べられている symbol の位置を示す．

OP: M-structure の右端の位置を示す．

STP: SYNTAX TABLE の実行中の OPERATION の位置を示す (Fig. 1 参照).

Program の parsing に先立って，IP, OP にはそれぞれ必要な初期値が与えられ，STP には parsing 終了後の復帰番地が与えられる．

tail への情報の出し入れは，head を通じて last-in-first-out に行なわれる．すなわち，head の情報は，次の pushdown storage を操作する OPERATION によって格納され，回復される．

RESERVE: push down; (head) → (tail);
head の内容を tail に貯える．

RESTORE: (head) ← (tail); pop up;
head の以前の値を回復し，tail を一つ消去する．

REWRITE: (head) ← (tail);
head の以前の値を回復する．tail は変らない．

LOSE: pop up;
tail を一つ消去する．head は変らない．

ERASE: delete tail
tail を全部消去する．head は変らない．

3.2 READ & TEST OPERATION

Source Program を読み，その文脈を調べるために，次の READ & TEST OPERATION を定める．

*: Source Program から，one symbol を INPUT register に入れる．

I: Source Program の次の symbol が identifier か否かを調べる．

N: Source Program の次の symbol が number か否かを調べる．

"symbol": INPUT register に，指定された terminal symbol があるか否かを調べる．

CALL (NAME): Source Program の次の unit が指定された名前の syntactic unit であるか否かを調べる．
i. e., RESERVE (IP, OP, STP) を行なって，NAME の指す位置を STP にセットする．

（i） READ OPERATION

SYNTAX TABLE に READ OPERATION* が指定されると，Source Program から IP に従って one symbol を INPUT register に入れる．その時，すでに INPUT に symbol が入っているか，あるいは，新たに Source Program から取り出すかを，READ FLAG (RF) で判定する．

RF が OFF なら，IP に 1 加えて，Source Program から IP の指す位置の one symbol を INPUT に入れる．

RF が ON なら，新たに Source Program から取り出さず，単に RF を OFF にする．

（ii） IN TEST OPERATION

SYNTAX TABLE で，I または N が指定されると，Source Program の次の位置に，それぞれ iden-

tifier または number の出現を調べる. その出現が確認されると identifier または number を編集して, IN TABLE に格納する. その格納開始番地を内部表現として, a-register に入れる (**Fig. 3**).

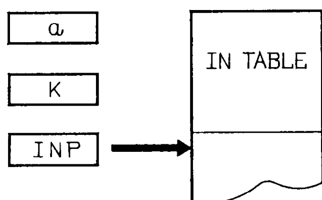

Fig. 3 IN TABLE and a-register

すなわち,

I: ∘a に INP の内容を記憶する.

∘identifier を <letter>*[<letter>|<digit>]*[1] の string として IN TABLE に格納する. INP は必要なだけ変更される.

∘End Mark の挿入;

∘IDENTIFY (a);

K=0 ならば, 手続を完了;

そうでなければ, INP を a に記憶された値に戻して, K の内容を a にうつす;

ここに, INP は IN TABLE の next available location を示し, IDENTIFY (p) は, IN TABLE の p 番目と同じものを探して, みつかればその番号を K にセットし, 同じものがなければ, K:=0 とする.

N: ∘number を <digit>, "·" および "$_{10}$" の string として IN TABLE に格納する.

∘type (real または integer) を確認する;

∘その type (real=0, integer=1) と INP の値を a に記憶する.

∘End Mark の挿入;

∘IDENTIFY (INP);

K=0 ならば, 手続を完了;

そうでなければ, type と K の内容を a にセットする;

IN TABLE を媒介として, source program 中で identifier および number を表す同じ記号列は, 同じ内部表現に変換される.

identifier は, のちに source program の文脈に従って, f-stack を用いて, 変数には記憶番地が割付けられ, label には program の番地が決定される.

一方, IN TABLE の number を表す部分は Compiler の output として, pseudo code と共に与えられて, pseudo code の machine code への変換時に, 機械語表現に変換される.

*1 *[ ] は零または任意回のくり返しを表わす.

3.3 T. & F. ACTION

SYNTAX TABLE の ACTION 欄では, 次の OPERATION によって parsing process のコントロールおよび M-structure の構成を行なう.

[M-routine NAME]:

OP に 1 加えて, 指定された M-routine の名前を M-structure に入れる.

[I] a: M-routine の名前 [I] と a-register の内容を M-structure に入れる.

OP に 2 加えられる.

[N] a: M-routine の名前 [N] と a-register の内容を M-structure に入れる.

OP に 2 加えられる.

/ : COMPILE OPERATION, すなわち, Parsing phase を一時中断して, Translating phase にうつり, 現時点までに構造解析された Source Program の M-structure に従って, Object Program への変換を行ない, 完了したら元に戻る.

→NAME: 指定された NAME を LABEL に持つ READ & TEST OPERATION にうつる.

→T.A: これは F. ACTION 欄にのみ指定がゆるされ, 対応の T. ACTION にうつる.

ON: READ FLAG を ON にセットする.

TR: CALL (NAME) で指定された名前の syntactic unit が確認されたので「真」なる答を持って calling point に戻る.

i.e. LOSE (IP, OP); RESTORE (STP); を行なって, STP に従って Parsing を続ける.

FR: CALL (NAME) で指定された名前の syntactic unit が確認できないので「偽」の答を持って calling point に戻る.

i.e. RESTORE (IP, OP, STP); を行ない, RF を OFF にして, STP に従って Parsing を続ける.

ER.: Error 処理ルーチンにコントロールをうつし, 指定された処理手続を行なって, 元の位置にもどる*[2].

*2 一般的には "ER. NAME" という表示を与えて, 対応の処理ルーチンを M-routine と同じ形式で与える. 本文では, その詳細は省略する.

ϕ: 何も指定がなければ，次の **READ & TEST** にうつる．

i. e. STP:=STP+1;

3.4 Example 1

〈asst〉=〈iden〉←〈exp〉

〈exp〉=〈prim〉*〔+〈prim〉〕

〈prim〉=〈iden〉|(〈exp〉)

なるシンタックスをもつ POL の SYNTAX TABLE は **Table 1** で与えられ，source program の parsing

Table 1 SYNTAX TABLE of Example 1

AS	I	[I] a	FR
	*"←"	[Left]	ER., → T.A
	CALL (EXP)	[Right],/TR	ER., [Z], → T.A
EXP	CALL (P)		FR
E 1	*"+"		ON, TR
	CALL (P)	[Plus], → E 1	ER., [Z], → T.A
P	I	[I] a, TR	
	*"("		FR
	CALL (EXP)		ER.
	*")"	TR	ER., [Z], → T.A

は第1行（label AS をもつ行）から開始される．たとえば，

xyz←a+(b+c)+d

に対しては，AS 行の I によって identifier の出現を調べ，xyz が確認されると，T. ACTION に従って M-routine の名前 [I] と，記号列 xyz が IN TABLE に格納された最初の番地（それを n_x とする）とを M-structure に挿入する．次に，one symbol 読込んで delimiter "←" の出現を調べる．"←" の確認に対して M-routine の名前 [Left] を M-structure に入れる．続いて syntactic unit 〈exp〉 の出現を確認するため，IP, OP, STP の現在値を記憶して，EXP の行にうつる．同様の手続で parsing を続け，〈exp〉が必要十分に確認されたら，TR の指示に従って，記憶した STP の値に基づいて第3行目に戻る．すなわち，a+(b+c)+d を parse して CALL (EXP) が完了し [Right] が M-structure に挿入される．M-structure は

[I] n_x [Left] [I] n_a [I] n_b [I] n_c

[Plus] [Plus] [I] n_d [Plus] [Right]

となる．ここで "/" に従ってコントロールは一時 Translating phase にうつされ，M-structure で与えられた M-routines を実行する．M-structure 内の M-routines がすべて完了すると，TR に従って syntactic unit 〈asst〉 の確認が完了したことになる．

AS 行の FR は syntactic unit 〈asst〉 が確認されなかった場合である．

4. Translating Processor

4.1 M-routine の選択・実行

SYNTAX TABLE で COMPILE OPERATION "/" が出されると，Compiler のコントロールは Translating Processor にうつされる．Translating Processor は M-structure に指定された M-routine を順次実行させ，すべての M-routine が実行されると，STP によって Parsing phase にコントロールを返す．

Translating Processor は，三つの M-structure pointer H, R, T と，pushdown storage MP を持つ．H, R, T はそれぞれ M-structure の位置を示し，MP pushdown は M-routine の statement counter として使われる．すなわち Translating Processor は，

(i) Parsing Processor からコントロールを受け取ると，H:=R:=0; T:=OP; MP:=0; RESERVE (MP); の初期状態をセットする．

(ii) MP:=(R); によって指定された M-routine を実行する．

(iii) 一つの M-routine 実行後

R=T ならば →(iv);

R≠T ならば R:=R+1; →(ii);

(iv) ERASE (IP, OP); OP:=0;

STP によって Parsing Processor にコントロールを返す (Fig. 1 参照)．

M-routine の選択は，主として R の指す M-routine の名前に基づくが，その実行にあたっては H および T の指す位置の情報が使用される．つまり translation は，Source Program の IU に対応する部分の文脈全体をながめて行なわれる．

また pointer H, T を用いて M-structure に新たな情報を書き込んだり，M-structure を working stack として使用する．このように，IU の構造が完全に解析されて表現されている M-structure を自由に走査することによって，あるいは，いくつかの pointer を用いて IU の各所から情報を集めて，能率の良いコンパイルが行なえる．さらに複雑な POL のコンパイルも可能にし，プログラミング言語から自然

言語の翻訳に拡張，応用することもできるであろう．

4.2 f-stack*3

Translating Processor は，前節の pointer H, R, T と pushdown MP の他に，次のものを持つ．

stack: f=f 1(*a*), f 2(*t*), f 3(*a*);
pushdown: m(*a*); j(*a*);
working: W=W 1(*a*), W 2(*t*), W 3(*a*);
Q=Q 1(1), Q 2(*a*−1);
c(*p*); b(*p*);
CODE;

ここに，*a*, *p*, *t*, はそれぞれ format を表わし，

a: address part に必要な bit 数，
p: counter または pointer として必要な bit 数，
t: identifier の type 表示に必要な bit 数，

である．これによると，H, R, T は (*p*), MP は (*a*), M-structure の one cell は (*a*) である．CODE は character 4 文字分の bit 数を要する．これらの間の情報の伝達は最下位の bit をそろえて行なわれる．

pushdown m および j は，それぞれ，記憶の割付け，Object Program の発生の count に使われ 3.1 に定めた pushdown OPERATION によって操作される．

stack f は，identifier の登録および identification の媒介として用いられる．f-stack の tail F への情報の書き込みは，head f を通じて行なわれるが，tail F の内部の読み出しおよび消去ができる点が pushdown storage と違っている．すなわち，Compiler が "STACK AUTOMATON"[11] によってモデル化されるゆえんである (**Fig. 4**).

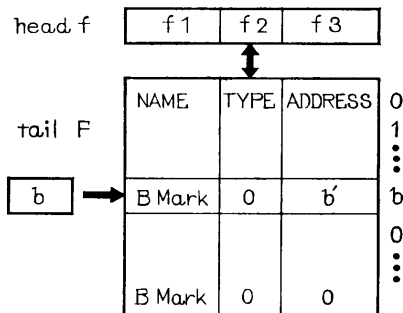

Fig. 4 format of f-stack

f および F は (*a*, *t*, *a*) の構造を持ち，identifier name (内部表現)，type およびその allocated location を示す情報を貯える．f は f 1, f 2, f 3 によって各々が別々に指定できる．tail F には，identifier が block (または subprogram) ごとに登録され[12]，block の区切りには Block Mark (B Mark と略す) が挿入される．tail F は最近に挿入されたものから 0, 1, 2…… と数えられ，現在の block に登録された identifier 数 (すなわち，現在の block の B Mark の位置) を pointer b で表わす．

f-STACK の操作には，pushdown OPERATION の他に identifier の identification を行なうために，次の OPERATION を定める．

SEARCH (*i*): 第 *i* 行から，番号の大きい方へ，B Mark に出会うまで，f 1 に与えられた name と同じ name を持つ行 *k* をさがす．
{ *k* がみつかると Q 1:=0; Q 2:=*k*;
{ B Mark に達すると Q 1:=1; Q 2:=0;

FIND: 第 0 行から，tail F 全体を f 1 の name と同じ name を持つ行 *k* をさがす．
{ *k* がみつかると Q 1:=0; Q 2:=*k*;
{ *k* がみつからないと Q 1:=1; Q 2:=0;

DELETE (*i*): tail の第 *i* 行を消去して，(*i*+1) 行以下を pop up する．

さて，identification の手順は，これらの OPERATION を用いて M-routine の一部として与えられる．tail F の内部への書き込みが許されないため，第 *i* 行を修正する場合には，その行を消去して，新たに f を通して書くという手続きを取る．その詳細については，ALGOL の block 構造と label をモデルとして，Table 5 の Head, End, Goal および Dlab の部分を参照されたい．

4.3 M-routine の記述

各 M-routine は，一連の次のような COL BASIC STATEMENT によって表現され，Object Program の発生およびそのために必要な各種の情報の伝送，記憶を行なう．

(i) ASSIGNMENT STATEMENT:
left:=right;

(ii) CONDITIONAL STATEMENT:
〈同等条件〉 STATEMENT;

〈 〉内の条件が充されたときのみ指定した STATEMENT を実行する．

---

*3 ここで stack は S. Ginsburg[11] の定義する "STACK AUTOMATON" と "PUSHDOWN AUTOMATON" に従って pushdown storage と区別している．すなわち pushdown storage では，tail への情報の出し入れは，head を通じて last-in-fist-out に行なわれるが，stack では，tail の内部の情報の読み出しもゆるされている．

(iii) TRANSFER STATEMENT：

a. go to NAME; NAME を label に持つ STATEMENT にうつる.

i. e. MP := NAME;

b. [NAME]; NAME をサブルーチンの入口として扱う.

i. e. RESERVE (MP); MP := NAME;

c. /; サブルーチンおよび M-routine の終りを示し，Calling point に戻る.

i. e. RESTORE (MP);

⟨MP=0⟩ go to Mend;

"MP により M-routine の実行を続ける".

Mend: ⟨R=T⟩ go to SYNTAX TABLE;

(+R); RESERVE (MP):

MP := (R);

"MP により M-routine を実行する".

(iv) POINTER STATEMENT: (+Π); (−Π);

指定された pointer Π の内容を，1だけ増減する.

(v) ERROR ROUTINE CALL: ER.;

3.3 と同様に，指定された Error 処理ルーチンを呼ぶ.

(vi) LINK STATEMENT: LINK (α, L);

アドレス部の未決定の Pseudo Code を完成する．address chain の始端となる α 番地の Code からはじめて，確定した値Lを，アドレス部の第1アドレスに挿入する．挿入前の α 番地のアドレス部の第1アドレスには，次の Code の番地 β が与えられており，終端となる Code のアドレス部には zero が与えられている.

(vii) PUSHDOWN OPERATION および STACK OPERATION

(viii) OBJECT CODE STATEMENT:

〔命令部 アドレス部〕;

発生すべき Pseudo Code を指定する.

このような，Compiler に特有な操作を指令する BASIC STATEMENT によって，各 M-routine は簡明に表現される.

4.4 M-routines of Example 1

Example 1 の言語に対する M-routine は，次の **Table 2** によって与えられる.

Source Program xyz ← a+(b+c)+d の各 identifier の declaration によって tail F が **Fig. 5** のように構成され，j=0, m=5 をもって 3.4 の M-structure に従って M-routines を実行すると，次の Pseudo Codes を得る.

Table 2 M-routines of Example 1

```
I:       (+R); f1:=(R); FIND;
         f:=F(Q2); (+H); (H):=f3/;
Left:    (T):=[Ag]; (+T);
         (T):=(H); (+T);
         (T):=[Right]/;
Right:   /:
Ag:      (+R); ['STO' (H), (R)]; (+j)/;
Plus:    [Tm];
         ['ADD' (H−1), (H), Q]; (+j);
         (−H); (H):=Q/;
Tm:      Q:=(H−1); <Q1=1>/;
         Q:=(H); <Q1=1>/;
         (+m); Q:=m; Q1:=1/;
Z:       (+H); (H):=C0**1/;
```

**1 C_0 は定数 0.0 の記憶番地を表わす.

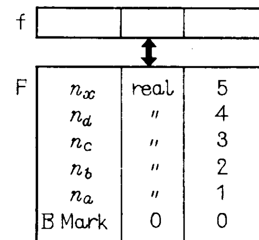

Fig. 5 f-stack of the Example 1

ADD	*m* 2,	*m* 3,	*t* 6
ADD	*m* 1,	*t* 6,	*t* 6
ADD	*t* 6,	*m* 4,	*t* 6
STO	*t* 6,	*m* 5	

Pseudo Code の命令部は symbolic code で表わされ，アドレス部は OPERAND の番地あるいは Object Program の番地を示す．OPERAND は，変数に割付けられた記憶番地または一時記憶の番地を表わし，後者には flag がつけられている (Example 1 では *m* と *t* で表示している).

5. COL Example

5.1 Syntax of model language

モデルとする言語 (ALGOL サブセット) のシンタックスを，次のような modified BNF で与える.

ただし，ここで，

├ および ┤ Source Program の始めと終りを示す.

*〔 〕 零または任意回のくり返しを表わす.

〔 | 〕 alternative を表わす.

φ Source Program の null string を表わす.

I または N identifier または number を示す.

Table 3 Syntax of COL Example

```
⟨PROG⟩   =⊢*[I:]⟨BLOCK⟩⊣
⟨BLOCK⟩  =begin*[⟨D⟩]⟨ST⟩*[;⟨ST⟩] end
⟨D⟩      =[real|integer]I*[,I];
⟨ST⟩     =*[I:][⟨CDST⟩|⟨UST⟩]
⟨CDST⟩   =⟨IFC⟩⟨UST⟩[else ⟨ST⟩|φ]
⟨IFC⟩    =if ⟨REL⟩ then
⟨UST⟩    =*[I:][⟨BLOCK⟩|⟨ASST⟩|go to I|φ]
⟨ASST⟩   =I ←*[I ←]⟨EXP⟩
⟨REL⟩    =⟨SA⟩[=|>|<]⟨SA⟩
⟨EXP⟩    =⟨IFC⟩⟨SA⟩else⟨EXP⟩|⟨SA⟩
⟨SA⟩     =[+|−|φ]⟨T⟩*[+⟨T⟩|−⟨T⟩]
⟨T⟩      =⟨F⟩*[×⟨F⟩|/⟨F⟩]
⟨F⟩      =⟨P⟩*[↑⟨P⟩]
⟨P⟩      =I|N|(⟨EXP⟩)
```

5.2 SYNTAX TABLE

parsing algorithm を与える SYNTAX TABLE は，次のようになる．

Table 4 SYNTAX TABLE of COL Example

LABEL	READ & TEST	T. ACTION	F. ACTION
PROG	*“⊢”	[Start]	→ PROG
PROS	I	[I] a	→ PROA
	*“:”	[Label]/ → PROS	ER.
PROA	CALL(BLOCK)		ER.
HALT	*“⊣”	[Fin]/TR	→ HALT
BLOCK	*“begin”	[Head]	FR
BL 1	CALL (D)	[Dh] → BL 1	
BL 2	CALL (ST)		→ BL 3
	*“;”	[Ste]/ → BL 2	ON
BL 3	*“end”	[End]/TR	ER., ON, → T.A
D	*“real”	[Real] → D 1	
	“integer”	[Int] → D 1	FR
D 1	I	[I] a	ER.
D 2	*“,”	→ D 1	
	“;”	[D]/TR	ER., ON, → T.A
ST	I	[As] [I] a	→ ST 1
	*“:”	[Label]/ → ST	→ AS
ST 1	*“if”	[If] → CDST	ON
UST	I	[As] [I] a	→ UN 1
	*“:”	[Label]/ → UST	→ AS
UN 1	*“go to”	[Goto] → GOTO	
	“begin”	ON, → BLOCK	
	“if”	ER., ON, TR	ON, TR
GOTO	I	[I] a [Jump]/TR	[Dummy]/TR
AS	“←”	[Left]	ER., → T.A
AS 1	CALL (LEFT)	→ AS 1	
AS 2	CALL (EXP)	[Right]/TR	ER., [0] → T.A
LEFT	I	[I] a	FR
	*“←”	[Left] TR	FR
CDST	CALL (REL)	[B]	ER., [0] → T.A
	*“then”	[Then]/	ER., → T.A
	CALL (UST)		
	*“else”	[Selse]/	[Conde]/TR
	CALL (ST)	[Conde]/TR	
REL	CALL (SA)	[A]	FR
REL 1	*“=”		→ REL 2
	CALL (SA)	[Eql] TR	ER. [0] → T.A
REL 2	“>”		→ REL 3
	CALL (SA)	[Grt] TR	ER. [0] → T.A
REL 3	“<”		FR
	CALL (SA)	[Lss] TR	ER. [0] → T.A
EXP	*“if”	[If]	ON, → SA
	CALL (REL)	[B]	ER., [0] → T.A
	*“then”	[Then]	ER., → T.A
	CALL (SA)		ER., [0]
	*“else”	[Else]	ER., → T.A
	CALL (EXP)	[Ed] TR	[0] → T.A
SA	*“−”		→ SA 1
	CALL (T)	[Umis] → SA 3	ER., [0] → T.A
SA 1	“+”		→ SA 2
	CALL (T)	[A] → SA 3	ER., [0] → T.A
SA 2	CALL (T)		FR
SA 3	*“+”		→ SA 4
	CALL (T)	[Plus] → SA 3	ER., [0], → T.A
SA 4	“−”		ON. TR
	CALL (T)	[Minus] → SA 3	ER., [0] → T.A
T	CALL (F)		FR
T 1	*“×”		→ T 2
	CALL (F)	[Mult] → T 1	ER., [1] → T.A
T 2	“/”		ON, TR
	CALL (F)	[Div] → T 1	ER., [1] → T.A
F	CALL (P)		FR
F 1	*“↑”		ON, TR
	CALL (P)	[Pow] → F 1	ER., [0], → T.A
P	I	[I] a, TR	
	N	[N] a, TR	
	*“(”		FR
	CALL (EXP)		ER.
	*“)”	TR	ER., [0], → T.A

5.3 M-routine

必要な M-routine は，次のようにまとめられる．

Table 5 M-routines of COL Example

```
Start:  ERASE (m, j, f);  m:=m0; j:=j0;
        f1:=BMark;  f2:=0;  f3:=0;
        RESERVE (m, i, f);  W:=0; b:=0/;
Fin:    RESTORE (f);
        <f1=BMark> go to Fine;
        <f2=0> ER.;
        go to Fin;
Fine:   /;
Head:   RESERVE (m); f1:=B Mark; f2:=1;
        <(T)=[D]> f2:=0;
        f3:=b;
        b:=0;
        RESERVE (f)/;
End:    c:=0;
Loop:   f:=F(c);
        <f1=BMark> go to Bend;
        <f2=0> go to Outer;
Delt:   DELETE (c); (-b); go to Loop;
Outer:  SEARCH (b+1);
        <Q1=1> go to Adc;
        W:=F(Q2);
        <W2=0> go to Adc:
        LINK (f3, W3); go to Delt;
Adc:    (+c): go to Loop;
Bend:   <f2=0> LOSE (m);
        DELETE (b); b:=b+f3; RESTORE (m)/;
Real:   f2:=1/;
Int:    f2=2/;
I:      (+R); f1:=(R);
        <(T)=[D]> go to Decla;
        <(T)=[Label]> go to Dlab;
        <(T)=[Jump]> go to Goal;
Prim:   FIND;
        <Q1=1> ER.;
        f:=F(Q2); (+H); (H):=f3;
        (+H); (H):=f2/;
Decla:  (+m); f3:=m; RESERVE (f); (+b)/;
Goal:   f2:=0; f3:=j; W3:=0:
        SEARCH (0);
        <Q1=1> go to Nof;
        W:=F(Q2);
        <W2=4> go to Obj;
        DELETE (Q2); (-b);
Nof:    RESERVE (f); (+b);
Obj:    ['JUMP' W3]; (+j)/;
Dlab:   f2:=4; f3:=j;
Lfind:  SEARCH (0);
        <Q1=1> go to Setl;
        W:=F(Q2);
        <W2=0> go to Chain;
          ER.; go to Dell;
Chain:  LINK (W3, f3);
Dell:   DELETE (Q2); (-b); go to Lfind;
Setl:   RESERVE (f); (+b)/;
Dh:     <(T)=[D]>/;
        RESERVE (m)/;
D:      /;
Goto:   /;
Jump:   /;
Right:  /;
Dummy:  /;
Label:  /;
0:      (+H); (H):=C0***1; (+H); (H):=2/;
1:      (+H); (H):=C1; (+H); (H):=2/;
N:      (+R); Q:=(R); (+H); (H):=Q2;
        (+H); (H):=Q1+1***2/;
B:      W2:=(H); (-H);
        <W2=3>/;
        <W2=2>/;
        (+H); CODE:='INTG'; [Conv]; (-H)/;
As:     <(T)=[Label]>/;
        H:=0/;
Left:   (T):=[Ag]; (+T)
        (T):=(H); (+T, -H);
        (T):=(H); (+T, -H); (T):=[Right]/;
Ag:     (+R);
        <(H)=(R)> go to Eqlr;
        CODE:='REAL';
        <(R)=2>  CODE:='INTG';
        (+R); [(CODE) (H-1), (R)]; (+j)/;
Eqlr:   (+R); ['STO' (H-1) (R)]; (+j)/;
A:      <(H)=1>/;
        <(H)=2>/;
          ER.; (-H); (-H); [0]/;
Ste:    REWRITE (m)/;
If:     RESERVE (m)/;
Then:   RESERVE (j); ['ZJMP' 0,(H)]; (+j);
        (-H); REWRITE (m)/;
Else:   [Atot];
Selse:  REWRITE (m);
        [Jtake];
        RESERVE (j); ['JUMP' 0]; (+j);
        LINK (W3, j)/;
Jtake:  Q:=j; RESTORE (j); W3:=j; j:=Q/;
Coude:  LOSE (m); [Jtake];
        LINK (W3, j)/;
Ed:     [Atot];
        <(H-2)=(H)> go to Same;
        <(H)=1> go to Ir;
Ri:     ['REAL' (H-1), (H-1)]; (+j);
Same:   [Conde]; (-H); (-H)/;
Ir:     W3:=j+2;
        ['JUMP' W3]; (+j);
        [Conde]; (-H); (-H);
        ['REAL' (H-1), (H-1)]; (+j); (H):=1/;
Atot:   Q:=(H-1); <Q1=1>/;
        W3:=(H-1);
        (+m); Q:=m; Q1:=1;
        (H-1):=Q;
        ['STO' W3, Q]; (+j)/;
Plus:   [Type]; CODE:=CODE·'ADD';***3 [Op]/;
Minus:  [Type]; CODE:=CODE·'SUB'; [Op]/;
Mult:   [Type]; CODE:=CODE·'MLY'; [Op]/;
Div:    [Type]; CODE:=CODE·'DIV'; [Op]/;
Pow:    [Type]; CODE:=CODE·'EXP'; [Op]/;
Eql:    [Type]; CODE:=CODE·'EQL'; [Opr]/;
Grt:    [Type]; CODE:=CODE·'GRT'; [Opr]/;
Lss:    [Type]; CODE:=CODE·'LSS'; [Opr]/;
Umis:   [A]; [Match]; CODE:=CODE·'NEG';
Conv:   (-H); [Tmu];
        [(CODE) (H), Q]; (+j)
        (H):=Q; (+H)/;
Type:   <(H)=(H-2)> go to Match;
        CODE:='REAL';
        <(H)=1> go to Tir;
Tri:    [Conv]; (H):=1; go to Match;
Tir:    (-H); (-H); [Conv]; (H):=1; (+H); (+H);
Match:  CODE:=0;
```

```
        <(H)=1> CODE:='F'**4/;
Opr:    (H-2):=3;
Op:     (-H); [Tmb];
        [(CODE) (H-2), (H), Q]; (+j);
        (-H); (H-1):=Q/;
Tmb:    Q:=(H-2);
        <Q1=1> go to Check;
Tmu:    Q:=(H);
        <Q1=1> go to Mset;
        (+m); Q:=m; Q1:=1/;
Check:  Q:=Q-1;
        <Q=(H)> go to Tmu;
        Q:=(H-2);
Mset:   m:=Q2/;
```

**1 C0 および C1 は定数 ZERO および ONE の番地を表わす.
**2 number は type (real or integer) を示す 1 bit とその格納番地とで表現されている.
**3 Symbol の concatenation を表わす.
**4 real operand に対して FLOATING OPERATION を表わす.

6. 結 言

COL は, Compiler に特有の操作を SYNTAX TABLE OPERATION および BASIC STATEMENT によって端的に表現するが, 計算機の特徴を Compiler に活かす点については, COL PROCESSOR にゆだねられる. 特に, Source Program の読み込み, identifier および number の編集・変換に対しては, 計算機に大きく依存するため機械語またはアセンブラ語によらねばならない.

また COL によって, SYNTAX TABLE に error syntax の事後処理をする部分を加えたり (本論文では省略したが), 各 Error Routine の記述の程度を変えたりして, syntax checker としての Compiler と, 正しいプログラムを速くコンパイルする Compiler を区別して作ることができる. しかし, 各 POL に関して次の点をさらに検討する必要がある.

(i) SYNTAX TABLE のみで, Source Program の syntax analysis が完全にできるか, あるいは完全にする必要があるか.

(ii) 文脈の二ヵ所または三ヵ所からの情報が, いかに有効に利用されるか. 本文の Example では, モデル言語が単純なため, その効果が十分に発揮されていないが, 多くの情報によってコンパイルは, より効果的に行なわれるのではないか.

(iii) IU の選び方と M-routine への影響. すなわち, 本文では, statement および declaration (label も含む) を IU として定めたが, ALGOL についていえば, ( ) でくくられた subexpression, subscript および list element (<for list>, <actual parameter list> など) を IU にすることができる. (ii) と関連して, もし, M-structure の数ヵ所からの情報によって変換を進める場合には, IU の選び方が M-routine にいかに影響を及ぼすかが問題となる. つまり, COL による記述の柔軟性の問題である.

参考文献

1) Peter Naur: Revised Report on the algorithmic language ALGOL 60; Comp. Jour. Vol. 5, No. 4 (Jan. 1963) pp. 349～367
2) W.P. Heising: History and Summary of FORTRAN Standardization Development for the ASA; C. ACM Vol. 7, No. 10 (Oct. 1964) pp. 590～625.
3) J.U. Garwick: Gargoyle: A language for Compiler Writing; C. ACM Vol. 7, No. 1 (Jan. 1964)
4) J.A. Feldman: A Formal Semantics for Computer Language and its Application in a Compiler Compiler; C. ACM Vol. 9, No. 1 (Jan. 1966) pp. 3～9
5) D.V. Shorre: A Syntax Oriented Compiler Writing Language; ACM Proc. 19 th 1964
6) F.W. Schneider & G.D. Yohanson: A Syntax Directed Compiler Writing Compiler to generate Efficient Code; ACM Proc. 19 th 1964
7) T.W. Pratt & R.K. Lindsay: A Processor Building System for Experimental Programming Language; FJCC 1966, pp. 613～621.
8) Ascher Opler: Fourth Generation Software; Datamation Vol. 13, No. 1 (Jan. 1967.) pp. 22～24.
9) J.L. Ryan: A Conversational System for Incremental Compilation; FJCC 1966, pp. 1～21.
10) P.Z. Ingerman: Syntax-Oriented Translator; Academic Press, 1966
11) S. Ginsburg: Stack Automaton and Compiling; J. ACM Vol. 14, No. 1 (Jan. 1967) pp. 172～201.
12) B. Randell & L.J. Russell: ALGOL 60 Implementation; Academic Press. 1964

(昭和 43 年 1 月 4 日受付)