ALGOL—翻訳の例*

三　浦　大　亮**

1.　はしがきと紹介

1962年から63年にかけて作成したALGOLコンパイラー（TORAY ALGOL）の翻訳の仕方の概要について報告する．まず使用した電子計算機とコンパイラーの構成などを紹介しよう．

第1表　使用電子計算機について

名　称	USSC 90（磁気テープ・システム）		
性　能	主記憶装置	磁気ドラム	5 kW（1 W＝10進10桁）
		回転速度	3.4 msec/r
	入出力装置	カード・リーダー	600 cd/min
		カード・パンチ	150 cd/min
		高速度プリンター	600 *l*/min
		磁気テープ装置	25 kc/sec
	その他：インストラクションはノン・シーケンシァルの実行		
		5 kW のうち 1 kW 分には読書ヘッドが4個ついている	
		処理速度（アクセスを除く）判定　68 μsec	
		加減　85 μsec	
		インデックス・レジスターは3個あるがいずれも mod. 200 で変化するので非常に使いにくい（object program を直接 machine code にしなかった理由はこれ）	

第2表　コンパイラーの構成とオペレーション

Phase	機　能	命令量	プリンター	処理時間	磁気テープ装置 2	3	8	9	10	カード
1	Source Prog. Deck & Mag. Tape に読み込む．Sequence Check をする Source Prog. をプリント・アウトする	600語	Source Prog. Sequence check	4%		○			Sys 1	Source prog
2	Identifier, Number, String, Delimiter, Declarator, Specificator などにわけ1Wに入れる	2600語	簡単な文法エラー	20%	○				Sys 2	
3	上記のものを記号化する Procedure, ブロック構造を考察して Identifier に仮アドレスをつける．文法エラーを見つける	3400語 表 800語	記号表 文法エラー	46%		○			Sys 3	
4	object prog. を作る	3800語 表 500語	文法エラー	18%	○					
5	完全な相対アドレスをつける アドレスは relocatable constant と working storage を前に詰められる	400語 表 200語	object prog.	12%			○		Sys 5	
	object prog. と Subroutine を load する．Library が必要であれば指定されたものを load する	400語 表 200語						○	Loader &	
	Executive Routine : Subroutine および Interpretive Routine	2000語							Sub Routine	

〔注〕 磁気テープ装置は最低3台必要である．
使用メモリーはテープ，コントロール・プログラムに共通に 200 W を，またインプット，アウトプット用に同じく 200 W ずつをとる．

第3表　Full ALGOL に対するおもな制限

(1) <identifier> はすべて英字を頭文字として英数字5文字までが有効である．
(2) assignment と power 以外で違った <type> どうしの演算は原則として禁止する．
(3) <array declaration> では <upper bound> だけ定数で示す．subscript は0からとする．
(4) <label> 以外の <identifier> は出現したときは必ず定義されていなければならない．
(5) <parameter> となる <array> は value で call できない．
(6) <procedure> の recursive call はできない．
(7) <parameter> となる <procedure> の <parameter> は value で call されるものでなければならない．
(8) name で call される <parameter> の <actual parameter> が <expression> であることはできない．
(9) specify されない <formal parameter> はすべて real simple variable とみなす．
(10) <switch list> は <label> だけゆるされる．
(11) <switch subscript> は integer だけゆるされる．
(12) <for statement> で control variable に最初に与えられる値が <for list> の条件を満たしていなくても実行される．また <for list> の値は <for statement> の実行前の値によって定まる．
(13) <comment> はすべて <comment quote> '～' によって示す．
(14) / は integer の演算はしない．ただし分母だけ integer のものはゆるす．
(15) <Boolean type>, <logical operator> ≧ ≦ while は使わない．
(16) その他若干の制限（システム・テスト中に現われたもの）

第4表　変更および付加したこと

(1) 記号の変更
ALGOL 60　:= ↑ > = ≠ < [ ] go to '⌣' 10±××
TORAY ALGOL : * * GTR EQL NEQ LSS ( )
GOTO ⋕⌣⋕ (±××)
(2) 小文字はない
(3) <label> は label part に書く <label> は <statement> 以外にもつけられる．
(4) real の zero は 0.0, integer の zero は0
(5) input/output process の付加
(6) string を <type declarator> に含め string type の variable を使う．
(7) <standard variable-array> BUFFER の付加
(8) <library specificator> と library procedure の導入

以上の現在のプログラムは，USSC の特徴であるノン・シークェンシァル方式であるため，処理時間を短縮するために命令数を多くする反面，プログラムの修正段階で，その最適性が破壊されているので，結果としては非常にまずいものになっている．テープへの pass 回数の多いのは，テープの速度がドラムでの処

* Algol-An Example of Translation Process, by Daisuke Miura (Computing Sect. Controller's Dept Toyo Rayon Co., LTD)
** 東洋レーヨン株式会社管理部統計課

理時間に比べて速いのであまり問題ではないが，再編成することができれば，Phase 1, 2, 3 を first pass で Phase 4, 5 を second pass で処理し処理時間は数分の 1 にすることができよう．さらに，object program は machine code に assemble しないで inter を通して使っているので効率は悪い．

コンパイリング時間は Phase 1～5 までで object program 1 個 (one address) につき約 1 秒強である．実行時間は machine code で書かれたものに比べて 10 倍くらいかかると推測されるが実用されている．

なお，このプログラムはプログラミングの初めての女子が USSC の machine code で約 1 年かかって作ったものである．また当社では，USSC の使用を 64 年末でやめることが決定しているので維持改善は行なっていない．

2. Phase 1, 2 について

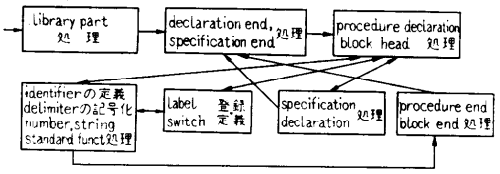
Phase 1 で card deck を読むときに card no. で sequence check で error が発見されると error indication がプリントされて，Phase 2 へ進む前にいったん停止する．Phase 2 では identifier, delimiters などの区切り（普通はスペース）を探して，一つ一つ後で処理しやすいために 1 語にする．また数字は real と integer の区別をして計算できる形にする．USSC では，左シフトで lower Acc. から upper Acc. へ 1 桁ずつ進めて比較することができないので，extract を全桁について行なうために，非常にめんどうで時間がかかっている．

3. Phase 3 について

ALGOL の性質から一般には，Phase 3 と Phase 4 をいっしょにした作業がコンパイリングの主作業になる．コンパイリング・テクニックとしてもいっしょにしたほうが手間が少なくてすむ．われわれの場合には，プログラム・テクニックが未熟であることを考慮して二つに分けた．また特にこの Phase に文法エラーの発見ルーチン的性格をもたせるつもりもあった．

作業はブロック構造の処理と identifier の定義，label の定義と記号化である．したがって，コンパイリング作業のうち object prog. をコーディングする部分以外はほとんど受け持つことになる．

プログラムの構成は次のとおり：



第 1 図 Phase 3 のプログラム構成

これによって記号化された言語がアウトプットされ Phase 4 にまわされる．次におもなカウンター，表を挙げる．

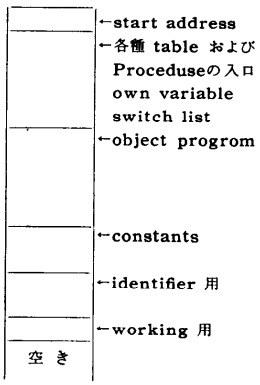
LAP: object program のステップを数える．array subscript table (object prog. を実行中に必要な subscript のキザミの表)，own variable, formal parameters table (object prog. 実行中に actual parameters の情報を入れるべき場所) や switch list table (switch の jump table) また procedure の入口はプログラムの中に含めるので，LAP で数える．

LAW: idenlifier の仮アドレスを決めるもので必要なアドレス数をカウントしている．(LAW) ≧5000

LS: label を記号化するためのもの

LC: constant table のカウント定数の記憶されている仮アドレスを与える．(LS)≦9000

したがって object program の構成は第 2 図のようになる．



第 2 図 Object Program の構成

LAW はあるブロックの処理が終ったときに，そのブロックだけで使われた identifier の分だけカウントを下げるのが通常である．この場合，そのブロックがある procedure の body であるときは，その procedure のほか identifier のアドレスとアドレスが重複しないように procedure の処理が終ったとき MAXW＝max(LAW) を LAW に入れる．この操作は，Phase 4 で，作業用アドレスやインデックスレジスターのようなものを数えるときにも行なう．

000		CC	0000	0200	200 ……stack head sign, start address=(A_2)
1	LIB 01	H4	1000	0201	……library name, library の占領するアドレス
2	XX	00	AAAA	0202	……parameter list, (A_2) は parameter 用アドレス
3	II	01	AAAA	0203	
4	begin	HH	0000	0000	……block head, (A_1) は一つ上の block head の位置
5	X	00	0000	5000	……real variable (A_2) は仮アドレス
6	Y	03	1001	5001	……real array (A_1) は subscript table のアドレス, (A_2) は array のアドレス
7	FXT	HO	5001	1005	……procedure dec., (A_2) は入口のアドレス, (A_1) にはこの procedure を処理する直前の (LAW) がはいる
8	Z	00	AAAA	1006	……formal parameter list
9	K	01	AAAA	1007	
10	begin	HH	0004	0000	
11	W1	00	0000	5102	
12	W2	03	1008	5103	……subscript table は共有できる
13	W3	04	1008	5243	
14	begin	HH	00H0	0000	
15					←——STACK POINTER (15)
	⋮	⋮	⋮	⋮	(LAP)=1010 (W3 の subscript table を2として)
199				399	(LAW)=5244
	Name Table (10桁 5文字)	Sign Part (2桁)	A_1-part (4桁)	A_2-part (4桁)	

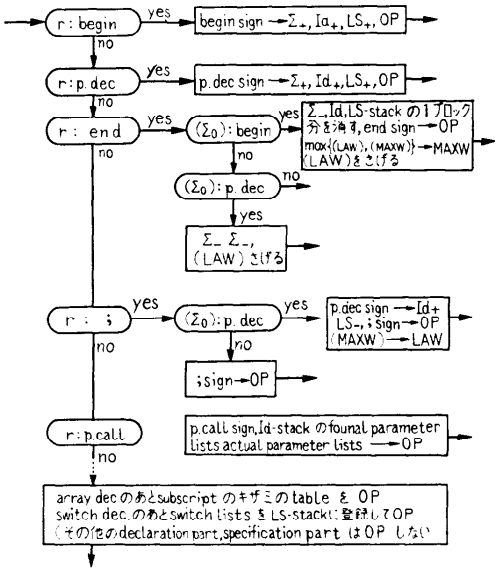
第3図　Identifier Stack (例)

Identifier Stack: stack 形式で, identifier の記号化に使う, 第3図のようになっていて, identifier と記号が対応されている. この table には, 処理中のブロック内の label と switch 以外の declare または specify された identifier がすべて載せられている. 処理済みのブロックは stack pointer の下にゆく.

Sign Part には, identifier の種類 (declaration, specification による) が入り, A_1-part には stack のアドレスおよびコントロール用の情報, A_2-part には, object program のアドレスとなるべきものが入れられている. Phase 4 のためにアウトプットされるものは, すべてこの形である.

Label-Switch Stack: label と switch を登録する. Id-Stack とほぼ同じ働きをする.

Σ-Stack: Id-Stack, LS-Stack その他をコントロールする. Stack の最後を Σ_0, stack につけ加えることを Σ_+, 除くことを Σ_-, 新しく処理されるものを r とすると, この働きの概要は次のようになる (他の stack についても同様; p. call = procedure call, p. dec= procedure declaration ' OP=out put する).



第4図　Phase 3 のコントロール

4. Phase 4 について

Phase 3 でアウトプットされたものを処理して sta-

tement のコーディングを行なう．主役である stack は Left-Stack と Formal Parameter-Stack である．

処理プログラムは多くのサブルーチンよりなっていて，どのサブルーチンを使うかは ALGOL の statement の場合には，(L_0) と r および stack の最後から2番目のもの (L_{-1}) と r の組み合わせによって決まる．これを明らかにするために，(L_0)-r Matrix と (L_{-1})-r Matrix を利用して処理プログラムを作成する．二つの Matrix の一部は第5図，第6図のとおりである．

r / (L_0)	begin	end	:=	variable	+	−	*, /, % ↑	(	)	;	,	if	then	else	ω rel	Proc. Call
(empty)	r → L+ P_0									P_0						
begin		L- P_0		r → L+ P_0						P_0		r → L+ P_0				r → L+ P_{32}
ω arith				r → L+ P_0				r → L+ A_0 → L+ P_0								r → L+ P_{82}
variahle constant working		P_1	r → L+ A_0 → L+ P_0		P_1	P_1	P_1	r → L+ A_0 → L+ P_0	P_1	P_1	P_1		P_1		P_1	
Array		P_1			P_1	P_1	P_1	P_{100}	P_1	P_1	P_1		P_1		P_1	
A_0		P_1		r → L_0 P_0	P_0	r → L+ P_0		r → L_0 A_0 → L+ P_0		P_1		r → L_0 P_0				r → L+ P_{32}
if				r → L+ P_0	P_0	A_0 → L+ r → L+ P_0		r → L+ A_0 → L+ P_0				r → L+ P_0				r → L+ P_{32}
then	r → L+ P_0	A_0 → L+ P_1		r → L+ P_0	P_0	A_0 → L+ r → L+ P_0		r → L+ A_0 → L+ P_0	L- P_1	A_0 → L+ P_1		r → L+ P_0		A_0 → L+ P_1		r → L+ P_{32}
else	r → L+ P_0	A_0 → L+ P_1		r → L+ P_0	P_0	A_0 → L+ r → L+ P_0		r → L+ A_0 → L+ P_0	A_0 → L+ P_1	A_0 → L+ P_1		r → L+ P_0		A_0 → L+ P_1		r → L+ P_{32}
ω rel				r → L+ P_0		A_0 → L+ r → L+ P_0		r → L+ A_0 → L+ P_0				r → L+ P_0				r → L+ P_{32}
Proced. call		P_1			P_1	P_1	P_1		P_1	P_1	P_1		P_1	P_1		

第5図 (L_0)-r Matrix の一部

r / (L-)	end	+, −	*, /, %	↑	)	;	,	then	else	ω rel
begin	L- P_{01}					L- P_{01}				
:=	P_1	r → L+ P_0	r → L+ P_0	r → L+ P_0		P_1	P_{11}		P_{28}	
(		r → L+ P_0	r → L+ P_0	r → L+ P_0	P_{32}		(L_0) → L- L- P_1			
+−	P_2	P_2	r → L+ P_0	r → L+ P_0	P_2	P_2	P_2	P_2	P_2	P_2
*+/	P_3	P_3	P_3	r → L+ P_0	P_3	P_3	P_3	P_3	P_3	P_3
↑	P_4	P_4	P_4	P_4	P_4	P_4	P_4	P_4	P_4	P_4
Array		r → L+ P_0	r → L+ P_0	r → L+ P_0	P_{101}		P_6			
if	(L_0) → L- L- P_1	r → L+ P_0	r → L+ P_0	r → L+ P_0	P_{28}	(L_0) → L- L- P_1	P_{28}	r → L+ P_0	r → L+ P_0	P_{30}
then	P_{19}	r → L+ P_0	r → L+ P_0	r → L+ P_0	P_{19}	P_{19}	P_{19}	P_{24}	P_{20}	P_{30}
else	P_{18}	r → L+ P_0	r → L+ P_0	r → L+ P_0	P_{18}	P_{18}	P_{18}	P_{24}	P_{18}	P_{30}
ω rel		r → L+ P_0	r → L+ P_0	r → L+ P_0	P_{38}			P_{29}	P_{31}	
Proc. call		r → L+ P_0	r → L+ P_0	r → L+ P_0	P_{104}		P_{102}			

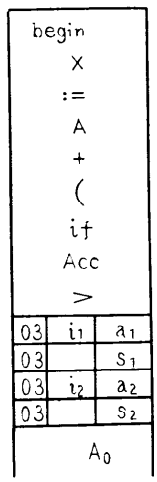
注：A_0 は dummy
P_0 は次の r をもってくる．
P_{01} は (L_0)-r Matrix を refer する．
P_1 は (L_{-1})-r Matrix を refer する．
他のサブルーチンも処理後 P_0, P_{01}, P_1 のいずれかへゆく．

第6図 (L-)-r Matrix の一部

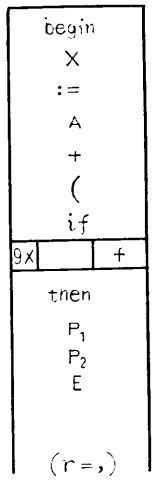
5. L-Stack 等の働き

いま，begin X:=A+(if B>C(I, K(J+M)) then P_1(D, P_2(E, F), L) else B*C/(−D↑G)); を処理しているとする．P_1, P_2 は function で P_2 はvalue で call されているとする．stack の内容とプログラムはおおよそ次のように進行する．太字の部分は objcet program となる．以下 $((L_0))$ は L_0 の A_2-part をaddress とする storage の内容を示す．

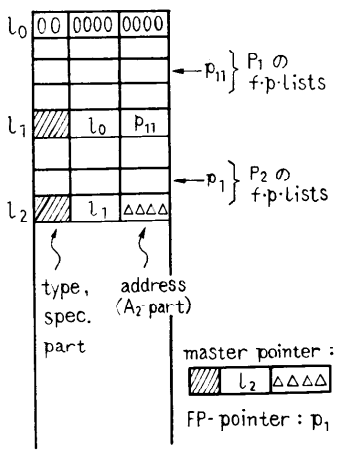
(P_0)r=begin, (P_{01}) begin→L_+, (P_0) r=X, (P_{01}) X→L_+, (P_0) r=:=, (P_{01}):=→L_+, A_0→L_+, (P_0) r=A, (P_{01}) A→L_0, (P_0) r=+, (P_{01}) (P_1)+→L_+, (P_0) r=(, (P_{01}) (→ L_+, A_0→L_+ (P_0) r=if, (P_{01}) if→L_0 (P_0) r=B, (P_{01}) B=L_+ (P_0) r=>, (P_{01}) (P_1) (P_{30}: **B→accumulator** Acc →L_-), (P_0) r=C, (P_{01}) C→L_+ (P_0) r=(,(P_{01}) (P_{100} : |03|i_1|a_1| → L_0, |03| |s_1| →L_+, A_0→L_+, s は subscript table の address, a は array C の address, i はこれから使う working storage または index register) (P_0) r=I, (P_{01}) I→L_0, (P_0) r=', (P_{01}) (P_1) (P_6 : $((L_0))$→i, $s_1+1=s_1$, A_0 →L_0), (P_0) r=K, (P_{01}) K→L_0, (P_0) r=(, (P_{01}) (P_{100}: |03|i_2|a_2| →L_0, |03| |s_2| →L_+, A_0→L_+)*1 (P_0) r=J, (P_{01}) J→L_0, (P_0) r=+, (P_{01}) (P_1) + → L_+, (P_0) r=M, (P_{01}) M→L_+, (P_0) r=), (P_1) (p_2: (A_{cc}) → Ws_1, $((L_{-2}))$+$((L_0))$ →A_{cc}, L_-, L_-, A_{cc}→L_0, stack 中の A_{cc} を Ws_1 に置き換え), (P_1) (P_{101} : (P_6 : $((L_0))$→i, $s_2+1=s_2$) L_-, L_-), (P_0) r=), (P_{01}) (P_1) (P_{101} : (P_6 : $((L_0))*(s_1)+(i_1)$→i_1, $s_1+1=s_1$ もし i_2 が index register でなければ，a_2 を modify するプログラムを作る) L_-, L_-), (P_0) r=then, (P_{01}) (P_1) (P_{29}: $((L_0))$ **Wrel** $((L_{-2}))$ **jump to** <f> , L_-, L_-, |9X| |f| →L_0, 9X は label sign, f は label の仮番号), (P_1) then→L_0, (P_0) r=P1, (P_{01}) P1→L_+, (P_{52}: 後続の formal parameter list→Formal Parameter Stack r=(になるまで, FP− pointer reset 1 A_0→L_+), (P_0) r=D, (P_{01}) D→L_0, (P_0) r=', (P_{01}) (P_1) (P_{102} : FP−pointer で指示された所の FP を refer して(L_0 **A_2-part**)→(**FP** の **A_2-part**), または ((L_0の**A_2-part**))→(**FP** の **A_2-part**), (pointer)+1→pointer, A_0→L_0), (P_0) r=P_2, (P_{01}) P_2→L_+, (P_{52} : formal parameter lists→FP-Stack r=(, FP-pointer reset 1, A_0→L_+), (P_0) r=E, (P_{01}) E→L_0, (P_0) r=, , (P_{01}) (P_1) (P_{102}: (L_0 の **A_2-part**)→(**FP** の **A_2-part**) または ((L_0の**A_2-part**))→(**FP** の **A_2-part**), (pointer)+1→pointer, A_0→L_0)*2, (P_0)r=F, (P_{01}) F→L_0, (P_0) r=), (P_{01}) (P_1) (P_{104}: P_{102}: 上に同じく; P_{103}: **linkage routine** L_-, FP-pointer reset 2 A_{cc}→L_0),(P_0) r=', , (P_{01}) (P_1) (P_{102}: 上に同じく), (P_0) r=L, (P_{01}) L→L_0, (P_0) r=), (P_{01}) (P_1) (P_{104}: P_{102}: 上に同じく; P_{103}: 上に同じく), (P_0) r=else, (P_{01})



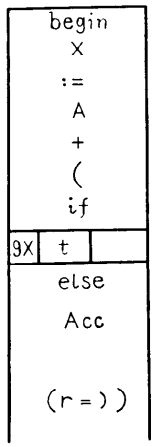
第7図　*1 のときの L-Stack



第8図　*2 のときの L-Stack



第9図　*2 のときの FP-Stack



第10図　*3 のときの L-Stack

(P_1) (P_{20} : if "(L_0)=label or switch" then "label/switch program" else if "(L_0) ≠<t, f>" then [((L_0)) → A_{cc} & jump to < t > & L_-, L_-, (L_0)=<f> label → op, |9X| t || → L_0, else → L_+] else (L_{-2}) → OP, (L_0) → L_{-2}, L_-, else → L_+], (P_0) r=B, (P_{01}) B → L_+, (P_0) r=*, (P_{01}) (P_1)* → L_+, (P_0) r=C, (P_{01}) C → L_+, (P_0) r=/, (P_{01}) (P_1) (p_3: (L_{-2}) **ω arith** (L_0) → A_{cc}, L_-, L_-, A_{cc} → L_0), (P_1) / → L_+, (P_0) r=(, (P_{01}) (→ L_+, A_0 → L_+),(P_0) r=−, (P_{01})− → L_+, (P_0) r=D, (P_{01}) D → L_+, (P_0) r=↑, (P_{01}) (P_1) ↑ → L_+, (P_0) r=G, (P_{01}) G → L_+, (P_0) r=), (P_{01}) (P_1) (P_4:(A_{cc}) → Ws_1, A_{cc} を Ws_1 に置き換え, (L_{-2}) **ω arith** (L_0) → A_{cc}, L_-, L_-, A_{cc} → L_0), (P_1) (p_2: −(A_{cc}) → A_{cc}), (P_1) (P_{32}: (L_0) → L_-, L_-), (P_0) r=), (P_{01}) (P_1) (p_3: (L_{-2}) **ω arith** (L_0) → A_{cc}, L_-, L_-, A_{cc} → L_0)*3, (P_1) (P_{18}: L_-, L_-, (L_0)=<t>→ OP), (P_1) (P_{28}: (L_0) → L_-, L_- …… (L_0)= |9X | t | | は演算に対しては A_{cc} となる. X は <type> を示す). (P_1) (P_{32}: L_0 → L_-, L_-) (p_2: (L_{-2}) **ω arith** (L_0) → A_{cc}, L_-, L_-, A_{cc} → L_0), (P_1) (p_2: (L_{-2}) **ω arith** (L_0) → A_{cc}, L_-, L_-, A_{cc} → L_0), (P_1) (p_1: (L_0) → (L_-), L_-, L_-, Acc → L_0), (P_1) L_-, (P_{01}) (P_0)

上記のプログラムの中には actual parameter が各種 formal parameter の場合の処置も含まれている.

6. Phase 5 について

これは, おもに label に実際のアドレスを与えることである. Phase 5 からアウトプットされるものには begin と end のサインが付けられていて, ブロック構造に対処することができるように, Label-Stack が用意されている. アドレスのつけ方はイモヅル方式である.

また object program のスタートアドレスから最後の作業用アドレスまですき間なくつなげることができる.

おわりに

コンパイラーを作る際には良い assembler を用意することが望ましい. それによって処理プログラムばかりでなく, object program も作りやすくなる.

このシステムの Phase 3 は error analyzer としては実は不成功に終ったが, 実用上のコンパイラーとしては, コーディングルーチンに入る前に error message を出すようにすべきである.

(昭和39年9月24日受付)