Compiler Describing Language: COL

HIROSHI HAGIWARA* AND KATUMASA WATANABE*

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.

1. *Constitution of a Compiler*

A compiler described in COL translates the source program represented in Problem Oriented Language (POL) into the object program through the parsing phase and the translating phase (Fig. 1). On parsing phase, the compiler,

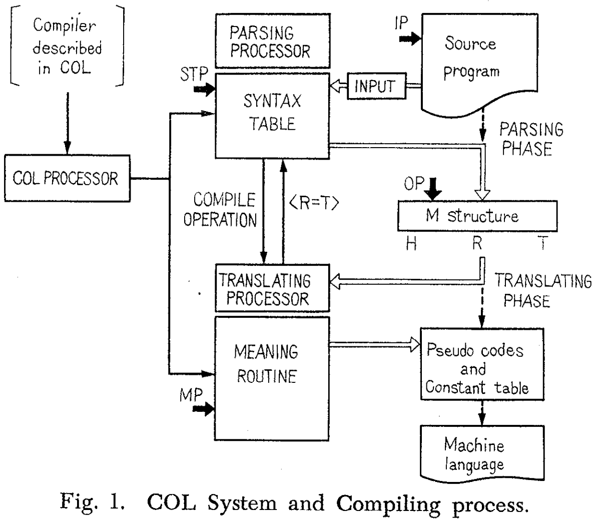

Fig. 1. COL System and Compiling process.

(1) reads the source program and recognizes an input symbol or a string of symbols,

(2) clarifies the context of the source program and constructs the M-structure

This paper first appeared in Joho Shori (the Journal of the Information Processing Society of Japan), Vol. 9, No. 4 (1968), pp. 187-196.
* Faculty of Engineering, Kyoto University.

denoting the syntactic structure,

(3) enters into the translating phase after the completion of the syntax analysis.

The part of the source program, the structure of whose context is determined completely and which is able to be translated into the object program by itself, is called an Incremental Unit (IU). For example, as to ALGOL 60, one statement and/or one declaration can be an IU. The parsing phase is paused at the end of the parsing of an IU, and the translating phase is begun with the M-structure of the IU.

M-structure denotes the result of the syntax analysis of an IU and consists of the inner representation of the basic items (identifier, number or string) and M-routine name. The M-structure of an (expression) is represented in the form of reverse-polish.

On translating phase the compiler reads M-structure (normally, from left to right) and executes the M-routines whose names are specified in the M-structure (these names are denoted MNAMEs). In M-routines it is specified to allocate the storage, to manipulate the identifiers, to store and restore the necessary items, and to construct a part of object program.

M-routines, whose MNAMEs are given in the M-structure, are executed one after another. Execution of all the M-routines in the M-structure of an IU means the completion of the transformation of the IU into the object program. Then, compiler returns to the parsing phase and restarts the syntax analysis of the next IU.

A compiler is made up of a SYNTAX TABLE and a set of M-routines. The former gives the parsing algorithm and the latter translating procedure. These are transformed into the executable form in a machine by a COL processor for the machine.

2. *SYNTAX TABLE*

Depending on the syntactic definition of POL, the SYNTAX TABLE is formed with READ & TEST OPERATION and ACTION OPERATION (Fig. 2). Each line of the SYNTAX TABLE is executed as follows :

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

Fig. 2. Format of SYNTAX TABLE.

If TEST **then** T. ACTION **else** F. ACTION ;

where TEST denotes the result of TEST OPERATION and has the value **true** or **false**. Compiler has three push-down storages to carry out the parsing. Each of them is the structure

(*head*)——(*tail*)

and information is reserved in tail, in first-in-last-out manner, through *head*. *Heads* are called IP, OP, STP.

IP : points the position of the input symbol which is examined.

OP : points the position of the right most item in the M-structure.

STP : points the position of the OPERATION being performed.

Pushodown storage E is manipulated by the following operation :

RESERVE (E) : pushdown *tail* E ; *head* $E \rightarrow$ *tail* E ;

RESTORE (E) : *head* $E \leftarrow$ *tail* E ; pop up *tail* E ;

LOSE (E) : pop up *tail* E ; *head* E is unchanged.

(1) READ OPERATION

* : takes one symbol in the source program which is designated by IP and puts it into INPUT register. But, if the inner switch RF is "on", only resetting of RF to "off" is done and any new symbol is not taken from the source program.

(2) TEST OPERATION

I : examines whether an identifier is found at the next position of the source program.

N : examines whether a number is found at the next position of the source grogram.

"delimiter" : examines whether the specified delimiter is found at the next position of the source program.

CALL (SNAME) : recognizes whether the next unit of the source program is the specified syntactic unit SNAME

i. e., RESERVE (IP, OP, STP) ; STP := SNAME ;

When these TEST OPERATION is satisfied, the inner variable TEST is given the value **true**. Otherwise TEST is given the value **false.**

(3) ACTION OPERATION

[MNAME] : specifies a MNAME to be inserted in the M-structure

OP := OP+1 ; (OP) := MNAME ;

a : requires that the inner representation n_x of a basic item x is inserted in the M-structure

OP := OP+1 ; (OP) := n_x ;

/ : instructs to stop the parsing phase and to begin on the translating phase with the M-structure constructed till then. After the M-structure has been translated into object program, the parsing phase is restarted at the position designated by STP.

$\rightarrow$SNAME : transfers the parsing procedure to the line labelled with the specified SNAME.

STP := SNAME ;

$\rightarrow$T. A : transfers the parsing procedure to the T. ACTION of the same line. This OPERATION is permitted only in F. ACTION.

ON : sets the inner switch RF to "on"

TR : gives the answer "true" for the OPERATION CALL (SNAME) and returns to the calling point

TEST :=**true** ; LOSE (IP, OP) ; RESTORE (STP) ;

FR : gives the answer "false" for the OPERATION CALL (SNAME) and returns to the calling point

TEST :=**false** ; RESTORE (IP, OP, STP) ;

ER : calls the error processing routine. After specified error processing, the parsing procedure is continued from the calling point.

ϕ : means empty OPERATION. Next line is executed.

STP : =STP+1 ;

Example 1.

The SYNTAX TABLE of the POL which has the following syntactic definition is given in Table 1.

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	CACC (P)		FR
E 1	* "+"		ON, TR
	CALL (P)	[Plus],→E1	ER., [Z],→T.A
P	I	[I] a, TR	
	* "("		FR
	CALL (EXP)		ER.
	* ")"	TR	ER., [Z],→T.A

⟨asst⟩=⟨iden⟩←⟨exp⟩

⟨exp⟩=⟨prim⟩ *[+⟨prim⟩]

⟨prim⟩=⟨iden⟩|(⟨exp⟩)

The parsing of a statement, for example,

$$x \leftarrow a+(b+c)+d \qquad (1)$$

is started from the first line, and the M-structure

[I] n_x [Left] [I] n_a [I] n_b [I] nc

[Plus] [Plus] [I] n_d [Plus] [Right] (2)

is constructed.

3. *M-routines*

On translating phase, compiler has following registers and auxiliary storages to read M-structure, to execute M-routines and to construct object program. There, a and t is the number of bits to represent the storage address and the type specification of an identifier, respectively.

(1) pointer H(a), R(a), T(a) : H, R, T point the cell of the M-structure. R points the cell containing the MNAME to be executed next. H and T point at first the left-most and the right-most cell of the M-structure respectively, these are used to know the detail of the M-structure and to write new information in the M-structure.

(2) pushdown MP(a) : MP corresponds to STP and points the Semantic Statement being executed.

(3) pushdown j(a), m(a) : j designates the next available position in which the object code is placed and m is used for the storage allocation.

(4) other working registers :

W=W1(a), W2(t), W3(a)

Q=Q1(1), Q2($a-1$)

b(a), c(a)

CODE .

where CODE is the register to manipulate the string of symbols for the operation part of an object code.

(5) stack f : stack f has the format f1(a), f2(t), f3(a) and is used for the identification of identifiers. The form of each element of stack f is

$$\begin{bmatrix} \text{inner representation} & \text{its} & \text{allocated} \\ \text{of an identifier } n_x, & \text{type,} & \text{location} \end{bmatrix}.$$

The identification is performed depending on the block structure of the source program with the following STACK OPERATIONs.

FIND : Beginning with the first element of the *tail* f, searches the same identifier as given in f1.

If such element k is found	Q1 :=0, Q2 :=k ;
otherwise	Q1 :=1, Q2 :=0.

SEARCH(i) : Beginning with the i-th element of the *tail* f, searches the same identifier as given in f1 in the part of *tail* f corresponding to the current block of the source program.

If such element k is found	Q1 :=0, Q2 :=k ;
otherwise	Q1 :=1, Q2 :=0.

DELETE(i) : deletes the i-th element of *tail* f and pops up its following elements.

Based on the semantic definition of the source language, M-routines are described procedurally in the sequence of the following Semantic Statement.

(1) ASSIGNMENT STATEMENT

left :=right ;

(2) CONDITIONAL STATEMENT

⟨EQUAL CONDITION⟩ STATEMENT ;

When the specified condition in ⟨ ⟩ is satisfied, the following statement is executed.

(3) TRANSFER STATEMENT

a. go to MNAME ;

MP := MNAME.

b. [MNAME] ; treats the specfied MNAME as a subroutine entry

i. e., RESERVE (MP) ; MP := MNAME.

c. / ; denotes the end of a M-routine or subroutine.

(4) POINTER STATEMENT

$(+\Pi)$; $(-\Pi)$;

The contents of the specified pointer Π is increased or decreased by unit value.

(5) ERROR ROUTINE CALL STATEMENT ER. ;

This calls the pre-determined error processing routine.

(6) LINK STATEMENT LINK(α, L) ;

α denotes the position of the object code whose address part is undefined. LINK(α, L) puts the contents of the register L in the address part of the code α. When the address part of α has the value β, this operation is performed for the code β until the value β is zero.

(7) PUSHDOWN OPERATION and STACK OPERATION

(8) OBJECT CODE STATEMENT

[OP part, address part] ;

This specifies the object code to be generated.

In these Semantic Statements suitable to represent the operation of the translating phase, compiler is described strictly and compactly.

Example 2.

M-routines of the POL given in Example 1 is described as Table 2. According to these routines, the M-structure (2) of the statement (1) is translated into the following sequence of object codes.

```
ADD  m2,  m3,  t6;
ADD  m1,  t6,  t6;
ADD  t6,  m4,  t6;
STO  t6,  m5;
```

where $m1$ to $m5$ mean the allocated location of a, b, c, d, x and $t6$ is the temporary storage location.

Table 2. M-routines of Example 1.

I:	(+R); f 1:=(R); FIND;
	f:=F(Q2); (+H); (H):=f 3/;
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):=C_0**/;

** C_0 represents the location of the constant 0.0.

4. *Conclusion*

We could describe compilers plainly in COL in the form of SYNTAX TABLE and Semantic Statements, but, as to the reflection of the characteristics of each computer on a compiler, we should have to leave it to each COL Processor. Especially, since operation to read source program and to edit identifier or number depends largely on each computer, they must be written in machine language or assembly language.

M-routines are executed with informations in several cells of the M-structure. So it is expected to describe M-routines easily and compactly even for the complicated POL. But, at the same time, we must take into consideration about the relation between M-structure and M-routines.