Information Processing in Japan Vol. 10, 1970

On the Compilers described in COL

HIROSHI HAGIWARA* AND KATUMASA WATANABE*

In order to get compilers in short period easily, we have proposed to write compilers in Compiler Oriented Language (COL).

In this paper, it is discussed what class of programming languages can be accepted by the compiler described in COL, and some experimental results and several important points on describing compilers are reported. Through the experiments it may be said that compilers described in COL can accept ALGOL programs satisfactorily, but that it takes much compiling time than handcoded one.

1. *Three languages specifying a compiler*

In this paper, an attempt based on any method to make compilers as readily as possible with a computer is extensively called a Compiler Compiler. Then, from the view point of the Compiler Compiler, a compiler can be specified with three languages, input language $L1$ output language $L2$, and the third language $L0$ in which the compiler is described. We denote it $C^{L0}_{L1\to L2}$.

We can classify Compiler Compilers at the point of each one of these three languages.

a. To construct a compiler describing language $L0$, and to write compilers in it. ALGOL and PL/1 can be an example of $L0$. Generally $L0$ is refined suitably to describe the compiling process.

b. To define the input language $L1$ formally and to find a simple compiling method. The study of $LR(k)$ grammar and the automaton model of a compiler is of this type.

c. To consider the output language $L2$, and to research the better computer language and computer itself. Universal Language, and the correspondence between ALGOL and λ-notation is of this class.

The compiler describing language COL, which we have proposed previously [1], is constructed at the view point of a. With it, we try to produce compilers readily and attempt to find the necessary feature of the computers which accept the higher-level programming language directly.

2. *Consideration of the Input Language*

2. 1. *Class of Input Languages*

This paper first appeared in Japanese in Joho-Shori (the Journal of the Information Processing Society of Japan), Vol. 10, No. 6 (1969), pp. 375-383.
* Faculty of Engineering, Kyoto University.

In this paper, the input language is considered being defined with a context-free phrase structure grammar G

$$G=(V,\ \Sigma,\ P,\ \sigma)$$

V is a finite set of symbols and called vocaburary. Σ is a finite set of terminal symbols and called alphabet. P is a set of rewriting rules

$$A \rightarrow X_1X_2 \cdots\cdots\cdots\cdots X_m$$

here $X_i \in V$, and $A \in N(=V-\Sigma$ i.e., a finite set of nonterminal symbols). σ is an element of N and called initial symbol. P has to be redefined to write the parsing process effectively in COL, especially according to the following two points.

1) Left recursive rule is reformed into repetitive form. For example, $\{A \rightarrow Ab, A \rightarrow a\}$ are combined to $A \rightarrow a^*\{b\}$, where $*\{\ \}$ means that the elements in $\{\ \}$ can appear any times repeatedly.

2) Some rules which have same left substring are reformed into factoring form to avoid unfruitful parsing process. For example, $\{A \rightarrow aBC, A \rightarrow aBd, A \rightarrow ae\}$ are combined to $A \rightarrow a\{B\{C|d\}|e\}$.

With respect to Knuth's $LR(k)$ grammar [2], COL has the following feature:

Lemma 1. The procedure to parse the string generated by $LR(k)$ grammar is able to be described in COL.

Next, from the view point of the automaton model, the compiler described in COL C^{COL} is considered as a combined automaton of a pushdown automaton $A1$ to perform the syntactic analysis and a stack automaton $A2$ to perform the translation into the output string. So according to Ginsburg's automaton theorem [3], the following is said:

Lemma 2. The compiler described in COL C^{COL} can accept the context-free language (CF-language).

Practically, when $A1$ performs the syntactic analysis incompletely, $A2$ may make up for the incompleteness with describing the translating process in detail. So it may be said that C^{COL} accept the class of languages including CF-language. For example, ALGOL is not a CF-language, but its program is accepted by C^{COL}.

2.2. *Description of Semantics*

For the purpose of describing semantics (or meaning) of the input language $L1$ as formally as possible, we would consider the way of representing the semantic interpretation procedure in COL. We assume, "Semantics is the specification how some (abstract or concrete) machine acts on". So, the problem of describing the semantics of $L1$ is reduced to the problem of defining the semantics of a simpler language and of relating the semantics of them. That is, we have following two problems,

1. Define an object machine and its basic action.
2. Establish the procedure making correspondence between string of $L1$ and

the sequence of basic actions.

In COL, we choose one of the current computers as an object machine and its machine instruction (or its symbolic code) as a set of basic actions, and concentrate our effort on the procedure to make the correspondence between *L*1 and these basic actions. To specify the procedure formally and systematically as possible, input string is first transformed into M-structure as a result of syntactic analysis. M-structure gives the standared form of the input string. Its structure and elements are arbitrary defined by the compiler builder, but it is better that M-structure has community in form for various input languages. For example, the input string of ⟨expression⟩ is transformed into the M-structure equivalent to reverse polish form. The process making correspondence between M-structure and output strings is represented in Semantic Statement. The semantic feature of language is reflected on this process. For example, ⟨conditional statement⟩

a. **if** ⟨Boolean expression⟩ **then** *S*1 **else** *S*2;

b. **if** ⟨Boolean expression⟩ **then** *S*3;

is transformed into the following M-structure, respectively;

a′. [IF] *BE* [BL] [THEN]/*S*1 [ELSE]/*S*2 [CONE]/

b′. [IF] *BE* [BL] [THEN]/*S*3 [CONE]/.

Here, *BE*, *S*1, *S*2 and *S*3 mean the M-structure of Boolean expression and statements *S*1, *S*2, *S*3 respectively.

In each M-structure, the name in a bracket is the name of a M-routine and represents the context of the input string. Each M-routine is described in the following Semantic Statement.

(1) *left=right* means that the value of the right part is assigned to the variable in the left part.

(2) *⟨equal condition⟩ statement* is a conditional statement and the specified statement is executed when the condition in ⟨ ⟩ is satisfied.

(3) [M-routine name] treats the specified M-routine name as a subroutine entry.

(4) / denotes the end of a M-routine or a subroutine.

(5) $(-H)$ means that the contents of the specified register H is decreased by unit value. Here, H is one of the registers pointing to the cell of M-structure.

(6) The letter 'E' denotes the calling on the predetermined error processing routine, and EBL is the statement to call this routine and to print the error massage 'BL' in compiling time.

(7) LINK (α, β) denotes that the contents of the specified register β is inserted into the address part of the object code which is stored in the location α.

(8) RESERVE, RESTORE, REWRITE, and LOSE manipulate the pushdown storage m or j.

(9) ["symbolic code", a list of operands] represents the object instruction code to be generated in compiling time.

Following M-routines make a′ and b′ correspond with the output string a″ and b″.

```
IF:     RESERVE (m)/;
THEN:   RESERVE (j); ["JUMP ON ZERO" 0, (H)];
        (-H); REWRITE (m)/;
ELSE:   [TAKJ];
        RESERVE (j); ["JUMP" 0];
        LINK (w3, j); REWRITE (m)/;
CONE:   [TAKJ];
        LINK (w3, j); LOSE (m)/;
TAKJ:   Q=j; RESTORE (j); w3=j; j=Q/;
BL:     w2=(H); (-H); ⟨w2=3⟩/; EBL/;
a″.   |          m:=result of ⟨Boolean expression⟩;
      |          [JUMP ON ZERO    L1,    m    ];
      |          S1
      |          [JUMP            L2          ];
      |  L1:     S2
      |  L2:
b″.   |          m:=result of ⟨Boolean expression⟩;
      |          [JUMP ON ZERO    L3,    m    ];
      |          S3
      |  L3:
```

3. *Generation of Compilers*

3.1 *COL Processor*

Experiments of compiler generation were done on the computer system FACOM 230-10 (Core 8K byte, Drum 65K byte), referred as M10 in the rest of this paper.

It begins with producing COL Processor for M10. COL Processor $C^{\mathrm{M10}}_{\mathrm{COL}\rightarrow\mathrm{M10}}$ consists of Syntax Loader and Semantics Loader, and transforms C^{COL} into C^{M10}.

Syntax Loader transforms a Syntax Statement

SNAME: ⟨READ & TEST⟩ ? ⟨T. Action⟩ ε ⟨F. Action⟩;

into the appropriate M10 instruction sequence corresponding to

SNAME: ⟨READ & TEST⟩
if TEST **then** ⟨T. Action⟩ **else** ⟨F. Action⟩

here TEST is a Boolean variable in compiling time. Semantics Loader accepts a Semantic Statement and generates the corresponding sequence of M10 instructions. Usually some basic operations for compilers (for exmple, stack manipulation operations, identifier editing) are prepared as Compiling-time Subroutines for each compiling machine, and a subroutine call instruction with some arguments is generated instead of complete sequence of instructions. These Compiling-time Subroutines realize the function of compiler in software.

3.2 *Comparison of Compiling-time*

Four compilers, which accept ALGOL or subset of it, are described in COL and generated by its processor. The compiling speed of these four are compared with that of handcoded ones for M10, M10-ALGOL and M10-FORTRAN. Table 1 gives relative compiling time (ratio for M10-FORTRAN) of several programs and the real compiling time for the following program.

begin real $x, a, b, c, d, e, f, g, p, q, r$;

$L1$: $x := a+b+c+d+e+f+g+p+q+r$;

$L2$: $x := a\uparrow b\uparrow c\uparrow d\uparrow e\uparrow f\uparrow g\uparrow p\uparrow q\uparrow r$;

$L3$: $x := (a)+(b)+(c)+(d)+(e)+(f)+(g)+(p)+(q)+(r)$;

$L4$: $x := (a+(b+(c+\cdots\cdots+(q+r)\cdots\cdots)))$;

$L5$: $x := ((\cdots\cdots((a+b)+c)+\cdots\cdots+q)+r)$;

end

Table 1. Comparison of the Compiling Time.

(sec)

	M10-FORTRAN	M10-ALGOL	Compiler I	Compiler II	Compiler III	Compiler IV
	1	0.8~0.9	1.5~2	2~3 (50%)	4~6 (50%)	3~4
L1	7.0	6.0	16.0	22.0 (8.0)	45.0 (23.5)	31.0
L2	9.0	10.5	12.5	18.5 (5.3)	30.0 (9.0)	52.0
L3	12.0	8.5	24.0	37.0 (23.5)	78.0 (58.0)	45.0
L4	11.5	8.5	23.5	32.0 (22.5)	72.0 (54.5)	39.5
L5	11.5	8.5	23.5	35.5 (22.5)	75.5 (54.5)	44.5
total	55.0	52.0	105.0	151.0 (83.5)	308.0 (201.0)	221.0

The characteristics of the generated compilers are as follows:

Compiler I: accepts a program which contains a declaration of simple variables and the sequence of simple assignment statements between **begin** and **end**, and generates the sequence of 3-address symbolic codes. For example, $x := a+b$ is converted to

ADD m_a, m_b, t;

STO t, m_x ;

where m_a, m_b, m_x, is the allocated address of a, b, x and t is the location of temporary storage.

Compiler II: accepts a subset of M10-ALGOL and generates 3-address symbolic code.

Compiler III: accept ALGOL program (with some restrictions and modifications) and generates 3-address symbolic code.

Compiler IV: accepts a subset of M10-ALGOL (same as Compiler II) and generates the sequence of machine instructions for another computer NEAC 2101.

In Table 1, the time in the parenthesis is the parsing time. It depends on the length of the path from the root to the leaf of the tree and the order of the rules referred, and it occupies about 50% of whole time in average. The ⟨arithmetic expression⟩ which has deeper tree structure takes much parsing time, and the block structure which requires processing of identifiers takes much time in translating.

The following are the main reasons why compilers described in COL take much compiling time than handcoded ones.

(i) Because of the parsing phase based on the syntax directed analysis, times of rereading input symbols and of manipulating pushdown storages increase in number as a result of trial and error.

(ii) To describe the compiling process simply, parsing phase is completely separated from translating phase and M-structure combines them. So redundant operations increase.

(iii) Because of 2-level storage (Core, Drum) of M10, generated compilers are segmented in pages (one page is 2K byte). As a compiler is bigger, much time are required to exchange pages of compiler. Moreover, exchanging of pages of automatically segmented compiler is more frequent than that of handcoded one.

3.3 *Compiler* III

To write Compiler III $C^{\mathrm{COL}}_{\mathrm{ALGOL}\rightarrow\mathrm{Symbolic,\ code}}$, we assume some restrictions and modifications on ALGOL.

1. Following concepts are deleted from ALGOL 60: program comment, own, unsigned integer label, formal parameter label called by value. Without label called by value, all ⟨designational expression⟩ can be dealt with as **go to** ⟨designational expression⟩.

2. The specifications of formal parameters are given in ⟨formal parameter list⟩. For example,

procedure *P* (*x*, *y*, *z*, *a*); **value** *x*; **real** *x*, *y*, *z*; **array** *a*;

is written as

procedure *P* (**value real** *x*; **real** *y*, *z*; **array** *a*);

3. To get the correct attribute of an identifier when it is referred, declarations are given in the following order.

type or array/switch or procedure

4. As an ⟨actual parameter⟩, ⟨statement⟩ is permitted. Corresponding formal parameter has the specification **procedure**. Hence, ⟨label⟩ can be deleted from ⟨formal parameter⟩. For example,

procedure *P* (**label** *M*); **begin** …… ; **go to** *M*; …… **end**;

    …… P (L) ; ……

is written as

  procedure P (**procedure** Q) ; **begin** …… ; Q ; …… **end** ;

    …… P (**go to** L) ; ……

4. *Conclusion*

As a result it may be said that the compiler described in COL, compared with handcoded one, is described more easily and produced in shorter period, but takes longer time for compiling. Then, we have the following problems in future.

(1) It takes much time to analyse the input string whose syntactic tree structure has deeper node such as ⟨expression⟩. So, COL is required to have the flexibility to write compilers which parse the input string in some deterministic method depending on operator precedence or $LR(k)$ parsing.

(2) To write translating phase more simply and systematically, it should be required to establish the formal definition of the semantics of languages.

Reference

[1] Hagiwara, H. and K. Watanabe, Compiler Describing Language: COL; *Information processing in Japan,* 9 (1969), 96–102.

[2] Knuth, D. E., on the translation of language from left to right; *Inf* & *Control.* 8 (Oct. 1965), 607–639.

[3] Ginsburg, S. and S. A. Greibach, Stack Automaton and Compiling, *J. ACM,* *14* (Jan. 1967), 172–201.