Syntax Directed Analysis and The Compiler Compiler

By

Hiroshi HAGIWARA* and Katumasa WATANABE*

(Received March 29, 1969)

The Compiler Compiler is the procedure to define an input and an output language as formally as possible, to describe the translation algorithm of a compiler simple and clear, and to generate compilers in a short time easily by means of a computer.

As one method, we have constructed the Compiler Oriented Language COL suitable to describe compilers. The compiler described in COL parses an input string in the manner of the Syntax directed analysis, interprets the resulting syntactic structure and produces the output string.

In this paper, we discuss two methods of the Syntax directed analysis, Top-down analysis and Bottom-up analysis, and the Syntax Statement of COL in which the parsing procedure of a compiler is described.

As a result, it may be said that the resulting compiler described in COL can accept ALGOL programs and produce the object programs in symbolic language or in machine language, and that the parsing time is about 50 percent of the whole compiling time on the average.

1. Introduction

A language is considered as a set of strings of symbols. A compiler is a program which translates the string of the input language L1 into the string of the output language L2. Every string has information as the sequence of symbols and the information about meaning of symbols. Translation is performed depending not only on the sequence of symbols but also on the meaning of both languages.

When each symbol "a" of the input language corresponds uniquely to some symbol "b" or some sequence of symbols $b_1 \, b_2 \cdots b_k$ of the output language, the algorithm of the translation is simple. But, in general, the translation algorithm is more complicated and could be divided into the following two parts:

1. Parsing phase: analyzes an input string (source program) and constructs a resulting sequence of symbols which represents the syntactic structure of the string.

2. Translating phase: interprets the resulting sequence and generates the output string (object program).

* Department of Applied Mathematics and Physics

For a pair of an input language (generally, procedure oriented language POL) and an output language (machine oriented language MOL), one compiler is required. But it takes much time and human work to make a compiler conventionally. As the kinds of POL and MOL increase in number, it is necessary to make compilers easily in a short time.

So study has been begun on "the procedure to define the input and the output languages as formally as possible, to describe the translation algorithm of the compiler simple and clear, and to generate compilers easily and shortly by means of a computer."
This is extensively called the Compiler Compiler.

Then, besides an input language L1 and an output language L2, the third language L0 in which a compiler is described is mentioned on a compiler. A compiler is specified as $C^{L0}_{L1\to L2}$. Although we can discuss the Compiler Compiler from the view point of each language of these three, we classify the Compiler Compiler in practical way.

a. Converting-type: When we have a compiler $C^{m1}_{p\to m1}$ which works on a machine m1, we can generate a compiler $C^{m2}_{p\to m2}$ for POL p which works on the machine m2 as follows.

1. describe a compiler $C^{p}_{p\to m2}$ in POL p.
2. obtain $C^{m1}_{p\to m2}$ by $C^{m1}_{p\to m1}(C^{p}_{p\to m2})$.
3. obtain the objective compiler $C^{m2}_{p\to m2}$ by $C^{m1}_{p\to m2}(C^{p}_{p\to m2})$.

b. Bootstrapping-type: Extend the ability of a compiler for a POL p step by step.

1. make a compiler $C^{m}_{p_0\to m}$ for a subset p_0 of p. $i:=0$.
2. describe $C^{p_i}_{p_{i+1}\to m}(p_i \subset p_{i+1} \subset p)$.
3. obtain $C^{m}_{p_{i+1}\to m}$ by $C^{m}_{p_i\to m}(C^{p_i}_{p_{i+1}\to m})$.
4. repeat the step 2 and 3 with $i:=i+1$.

c. Describing-type: Define a compiler describing language L0 to write compilers easily (and, if possible, independently of a computer).

1. make a L0 processor $C^{m}_{L0\to m}$.
2. describe a compiler $C^{L0}_{p\to m}$.
3. obtain $C^{m}_{p\to m}$ by $C^{m}_{L0\to m}(C^{L0}_{p\to m})$.

d. Parametric-type: Given some formal definition of POL p (and MOL m), construct a compiler $C^{m}_{p\to m}$ with the prepared parsing procedure. The Syntax directed analysis is an instance of this type.

Based on the type c and d, we define the Compiler Oriented Language COL to describe a syntax directed compiler. So, we discuss the Syntax directed analysis and the part of COL to describe an Analyzer.

2. String and Rewriting rule

2.1 The properties of strings

Definition 1. When V is a finite set of symbols and V^* is the set of strings of symbols belonging to V, V^* is defined as follows:

1. If $X \in V$, then $X \in V^*$.
2. If $\alpha \in V^*$ and $\beta \in V^*$, then the product $\alpha\beta$ is uniquely determined and $\alpha\beta \in V^*$.
3. V^* contains an element ϕ such as $\alpha\phi = \phi\alpha = \alpha$ for any $\alpha \in V^*$. ϕ is called a null string.

The elements of V^* have the following properties:

pro. 1 if α, $\beta \in V^*$, then $\alpha\beta \neq \beta\alpha$.

pro. 2 if α, β, $\gamma \in V^*$, then $(\alpha\beta)\gamma = \alpha(\beta\gamma)$,

here, we don't consider any semantic properties depending on the strength of association.

pro. 3 if α, $\beta \in V^*$ and $\alpha = \beta \neq \phi$, then one of the followings is satisfied,

(1) there exist X and Y in V such that $\alpha = X \in V$, $\beta = Y \in V$ and $X = Y$.

(2) there exist α_1, α_2, β_1, β_2 in V^* such that $\alpha = \alpha_1\alpha_2$, $\beta = \beta_1\beta_2$ and $\alpha_1 = \beta_1 \neq \phi$, $\alpha_2 = \beta_2 \neq \phi$.

From these properties, it can be said that the algebraic system of V^* and the productive operator is semigroup and monoid.

Lemma 1. For n strings α_1, α_2, $\cdots$, α_n of V^*, the product $\alpha = \alpha_1\alpha_2 \cdots \alpha_n$ is uniquely determined without the dependence of the order of the product.

Definition 2. For $\alpha = \alpha_1 \cdots \alpha_n$ $(\alpha_i \in V^*,\ i = 1, \cdots, n)$, each α_i $(1 \leqq i \leqq n)$ is a substring of α and $\alpha_1 \cdots \alpha_j$ $(1 \leqq i \leqq n)$ is left substring of α. We denote it $\alpha_1 \cdots \alpha_j \in \mathrm{LS}(\alpha)$.

Lemma 2. A non-null finite string α $(\alpha \in V^*)$ can be represented as

$$\alpha = \prod_{i=1}^{n} X_i, \quad X_i \in V.$$

Definition 3. $l(\alpha)$ denotes the length of the string α $(\alpha \in V^*)$ and is defined as follows

$$l(\alpha) = \begin{cases} 0 & \text{for} \quad \alpha = \phi \\ n & \text{for} \quad \alpha = \prod_{i=1}^{n} X_i, \ X_i \in V. \end{cases}$$

The elements of V^* also have the following property.

pro. 4 (1) $l(\alpha\beta) = l(\alpha) + l(\beta)$ for α, $\beta \in V^*$.

(2) $l(\prod_{i=1}^{n} \alpha_i) = \sum_{i=1}^{n} l(\alpha_i)$ for $\alpha_i \in V^*$ $(1 \leqq i \leqq n)$.

Definition 4. For $\alpha=\prod_{i=1}^{l(\alpha)} X_i$ $(l(\alpha)\geqq 1,\ X_i\in V)$, it is defined that

$$LC_k(\alpha) = X_1X_2\cdots X_k\ (1\leqq k\leqq l(\alpha))\in LS(\alpha)\ .$$

Especially $LC_1(\alpha)=LC(\alpha)=X$ is called the most left character.

Example 1.

⟨assignment statement⟩: : =⟨variable⟩: =⟨expression⟩
LC(⟨assignment statement⟩) =⟨variable⟩.

2.2 Context free grammar

We divide the finite set of symbols V into two disjoint sets Σ and N. Σ is the set of terminal symbols and called alphabet. N is the set of non-terminal symbols.

Definition 5. A string $\alpha(\alpha\in V^*)$ is called a sentence if it consists of only terminal symbols. Namely, a sentence $\alpha\in\Sigma^*\in V^*$.

According to this definition, a language is regarded as a set of sentences selected by some rules. The rule, which is the basis of the selection, is the grammar. Here, by introducing the concept of the phrase, we consider the phrase structure grammar defined with the rewriting rules.

Definition 6. A context free phrase structure grammar G is specified by the 4-tuple (V, Σ, P, σ), where V is a finite set of symbols and called vocabulary, Σ is a finite set of terminal symbols and called alphabet, P is a finite set of rewriting rules $A\rightarrow\alpha$, $A\in N$, $\alpha\in V^*$, where $N=V-\Sigma(N\cap\Sigma=\phi)$ is the finite set of non-terminal symbols, and σ is an element of N and called initial symbol. σ does not appear at the right handside of any rewriting rule.

Definition 7. When $G=(V, \Sigma, P, \sigma)$ is a context free phrase structure grammar,

(1) we write $\varphi\Rightarrow\psi$ for φ, $\psi\in V^*$, if $\varphi=\gamma_1A\gamma_2$, $\psi=\gamma_1\alpha\gamma_2$ $(\gamma_1, \gamma_2\in V^*)$ and there exists $A\rightarrow\alpha$ in P.

(2) we write $\varphi\overset{*}{\Rightarrow}\psi$ for φ, $\psi\in V^*$, if (i) $\varphi=\psi$, or (ii) there exist finite number of strings φ_i $(\varphi_i\in V^*,\ i=1, 2, \cdots, n-1)$ such as $\varphi_i\Rightarrow\varphi_{i+1}$ $(i=0, 1, \cdots, n-1)$ where $\varphi_0=\varphi$, $\varphi_n=\psi$.

(3) the language $L(G)$ defined by G is represented as

$$L(G) = \{\alpha\,|\,\alpha\in\Sigma^*,\ \sigma\overset{*}{\Rightarrow}\alpha\}$$

and called a context free phrase structure language (CF-language). Namely, CF-language $L(G)$ is defined as a finite set of sentences obtained by applying some rewriting rules of P to σ. The syntactic structure of the programming language

ALGOL is defined by Backus Normal Form (BNF) recursively, then, without regarding the properties of identifiers, an ALGOL program is a sentence of a CF-language.

Definition 8. The string of terminal symbols $a_1a_2\cdots a_n$ $(a_i \in \Sigma)$ obtained by applying rewriting rules to a non-terminal symbol A is called the terminal form of $A(A \in N)$. For terminal form of A,

$$LT_k(A) = a_1 \cdots a_k \ (1 \leqq k \leqq n)\ .$$

Especially $LT_1(A) = LT(A) = a_1$ is called the most left terminal. $LT(A)$ has an important role in syntax analysis of a program as $LC(\alpha)$ has.

Now, a resulting sentence generated by a grammar G has the information of the sequence of symbols and the information of the associative relation among symbols and/or substrings according to the applied rewriting rules. Given the former information at the parsing phase the compiler deduces the latter information. It can be represented as a tree structure.

Definition 9. The correspondence between the generation of a sentence and its tree structure is as follows:

1. The symbol σ is the root of the tree and called node σ.
2. When a rule $A_p \rightarrow X_{p1}X_{p2}\cdots X_{pn_p}$ is applied, the node A_p has n branches and is connected to nodes X_{p1}, X_{p2}, $\cdots$, X_{pn_p}.
3. If X_{pi} $(1 \leqq i \leqq n_p)$ is a terminal symbol $(X_{pi} \in \Sigma)$, node X_{pi} is called a leaf and no new branch shoot out from it.

One generating process has only one tree structure, but it is not necessarily that all the sentences have only one tree structure.

Definition 10. The grammar G is unambiguous if all of its sentences have unique tree structure.

The rules of ALGOL 60 are unambiguous, and generally it is desirable that the grammar of the programming language is unambiguous. Although a sentence is ambiguous, with some restriction we can determine only one tree structure of it.

Definition 11. In a tree the handle is the most left set of leaves $X_1, \cdots, X_n$ which neighbor with and are connected to the same node Y.

The syntax analysis of the given sentence $\alpha \in \Sigma^*$ is to seek handles of the corresponding tree in turn. Namely, if the grammar G has the rule $Y \rightarrow X_1X_2\cdots X_n$ and $X_1X_2\cdots X_n$ is the most left set of leaves of the string $\alpha = \alpha_1X_1\cdots X_n\alpha_2(\alpha_1, \alpha_2 \in V^*)$, then α is replaced by $\alpha_1Y\alpha_2$.

Next we discuss two methods of the Syntax directed analysis how a given sentence is parsed. They are Top-down analysis and Bottom-up analysis.

3. Top-down analysis

Beginning with the root node σ of the corresponding tree of a given input string $\alpha = a_1 a_2 \cdots a_n$, the Top-down analysis is the method to recognize the components of each node of the tree with referring to the set of rewriting rules P, and at last to recognize that α is reduced to σ. This is similar to the change of states according to the state diagram of a sequential machine. The set of rewriting rules P is given with attention to the following points.

a. The rules with the same symbol at left handside are grouped and every rules are numbered sequentially one group after another, beginning with the group which has σ at left handside. The number of the rule represents the order of reference. Then, between two rules

$$A \rightarrow \alpha_1$$
$$A \rightarrow \alpha_2$$

which have the relation of $\alpha_2 \in LS(\alpha_1)$, the former should precede the latter, in order that the longest string of symbols constructing A could be recognized.

Example 2.

⟨*for list element*⟩ → ⟨*ae*⟩ **step** ⟨*ae*⟩ **until** ⟨*ae*⟩
⟨*for list element*⟩ → ⟨*ae*⟩ **while** ⟨*be*⟩
⟨*for list element*⟩ → ⟨*ae*⟩

where ⟨*ae*⟩ is ⟨*arithmetic expression*⟩ and ⟨*be*⟩ is ⟨*boolean expression*⟩.

b. When a left recursive rule

$$A \rightarrow Ab$$

is referred, the parsing process will not go ahead and the syntactic unit A will not be recognized. Generally, besides the above rule, a rule such as

$$A \rightarrow a$$

is contained in the set of rules P, so both are rewritten as follows

$$A \Rightarrow Ab \Rightarrow Abb \Rightarrow \cdots \Rightarrow abb \cdots b = ab^*$$
$$A \Rightarrow a^*\{b\}$$

where *{ } means that the elements in { } can be repeated zero or any number of times.

Example 3.

⟨*block head*⟩ → **begin** ⟨*declaration*⟩
⟨*block head*⟩ → ⟨*block head*⟩ ; ⟨*declaration*⟩

are changed to

$\langle$*block head*$\rangle \rightarrow$ **begin** $\langle$*declaration*$\rangle$* {; $\langle$*declaration*$\rangle$}

c. The right recursive rules are also modified. With rules

$$B \rightarrow aB$$
$$B \rightarrow b$$

the parsing process goes ahead normally and a^*b is reduced to B. But, in order to decrease stack manipulation, they are changed to

$$B \rightarrow {}^*\{a\}b$$

Example 4.

$\langle$*compound tail*$\rangle \rightarrow \langle$*statement*$\rangle$ **end**

$\langle$*compound tail*$\rangle \rightarrow \langle$*statement*$\rangle$; $\langle$*compound tail*$\rangle$

are changed to

$\langle$*compound tail*$\rangle \rightarrow \langle$*statement*$\rangle$*{ ; $\langle$*statement*$\rangle$} **end**

d. Under the restriction of the condition a, rules in a group are ordered depending on the frequency of use. It only reduces the parsing time and has no effect in parsing procedure.

e. Among the rules in a group the rules with common substring at right handside are modified by one of the following manners.

e1. Introduce some new non-terminal symbols.

e2. Change them into the factoring form.

For example, the rules

$A \rightarrow aBC, \quad A \rightarrow aBd, \quad A \rightarrow ae$

are modified as follows.

*e*1. $A \rightarrow aS, \quad S \rightarrow BT, \quad S \rightarrow e, \quad T \rightarrow C, \quad T \rightarrow d$

*e*2. $A \rightarrow a\{B\{C|d\}|e\}$

It omits that the components which have been recognized are cancelled and recognized again.

Example 5. $\langle$*term*$\rangle$ is defined as follows.

*e*1. $\langle$*term*$\rangle \rightarrow \langle$*factor*$\rangle$ *{$\langle$*post factor*$\rangle$}

$\langle$*post factor*$\rangle \rightarrow \times \langle$*factor*$\rangle$

$\langle$*post factor*$\rangle \rightarrow$ / $\langle$*factor*$\rangle$

$\langle$*post factor*$\rangle \rightarrow \div \langle$*factor*$\rangle$

*e*2. $\langle$*term*$\rangle \rightarrow \langle$*factor*$\rangle$*{$\times \langle$*factor*$\rangle$ | / $\langle$*factor*$\rangle$ | $\div \langle$*factor*$\rangle$}

We suppose that the grammar G has the set of rules P modified according to the condition a,b,c and e1. Namely, each component of the numbered rule

$$p\colon A_p \rightarrow X_{p1}X_{p2} \cdots X_{pn_p}$$

is an element of V or {or}. To parse the input string $\alpha = a_1 \cdots a_n$ one stack E and six auxiliary variables are used.
G represents the goal which is the node being recognized.
REP has the logical value (1,0) and represents whether the component is repetitive or not.
r points the rule being referred.
k points the *k-th* component at the right handside of the rule r.
j points the *j-th* symbol a_j of the input string.
The result of parsing is given as the sequence of the numbers of the applied rules and m points the *m-th* symbol of the output string. The stack E memorizes the sets of values of these variables in first-in-last-out manner and is manipulated by the following push-down operations.

RESERVE $E(\Pi)$: push down E and store the values of Π in E. Where Π means some or all auxiliary variables.

REWRITE $E(\Pi)$: rewrite Π with the values stored in E. Stack E is not changed.

ERASE E: pop up E to erase the set of values stored lastly.
With these background, given input string

$$\alpha = a_1 a_2 \cdots a_j \cdots a_n$$

is parsed according to the following procedure. As the result the sequence of numbers p_i of the applied rules

$$p_1 p_2 \cdots p_i \cdots p_m$$

is obtained.
Step A : set the initial value of each variable

$$(G, REP, r, k, j, m) := (\sigma, 0, 0, 0, 1, 1) .$$

Step B : in the order of the numbers, search the rule p such as $A_p = G$ and set $(r, k, REP) := (p, 0, 0)$.
Step C : $k := k+1$ and examine the component X_{rk}

1. if $X_{rk} \in \Sigma$ then examine the input symbol a_j
if $a_j = X_{rk}$, then $j := j+1$ and repeat Step C,
otherwise go to Step D.

2. if $X_{rk} \in N$ then set X_{rk} as a new goal

i.e., RESERVE $E(G, REP, r, k, j, m)$; $G:=X_{rk}$

and go to Step B.

3. if $X_{rk}=$"*" then $k:=k+1$ and set the repetitive state

i.e., RESERVE $E(G, REP, r, k, j, m)$; $REP:=1$

and repeat Step C.

4. if $X_{rk}=$"}" then add the indication to the output sequence that the repetitive component is recognized and $m:=m+1$. Repeat the recognition of the repetitive components

i.e., $k:=E(k)$; $E(j, m):=(j, m)$

and repeat Step C.

5. if $X_{rk}=\phi$(*i.e.*, $k>n_r$) then, because all the components of the rule r have been recognized, add the rule number r to the output sequence and $m:=m+1$. And if $G=\sigma$ then parsing is completed successfully, otherwise the goal A_r is accomplished and the previous goal is restored

i.e., REWRITE $E(G, REP, r, k)$; ERASE E

and repeat Step C.

Step D : when the component X_{rk} is not recognized

1. if $REP=1$ then skip over to the component "}"

i.e., $(REP, j, m):=E\ (REP, j, m)$; ERASE E

and $k:=k+1$ until $X_{rk}=$"}", then go to Step C.

2. if $REP=0$ then give up the current rule r and refer the next rule

if $A_{r+1}=A_r$ then $r:=r+1$; $k:=k-1$ and go to Step C,

otherwise it results that the node A_r could not be recognized, so if $G=\sigma$ then parsing stops unsuccessfully

otherwise restore the previous state

i.e., RESTORE $E(G, REP, r, k, j, m)$; ERASE E

and repeat Step D.

If $LT(X_{rk})$ (or $LT_i(X_{rk})$, $i>1$) have been examined beforehand for $X_{rk}\in N$, then the parsing time could be reduced slightly. Because, without setting the new goal X_{rk}, it can be known in Step C2 that the node X_{rk} has not the possibility to be recognized when $a_j\notin LT(X_{rk})$ (*or* $a_j\cdots a_{j+i-1}\notin LT_i(X_{rk})$).

4. Bottom-up analysis

The Bottom-up analysis is the method to set the node σ as the largest goal and to recognize that an input string $\alpha=a_1\cdots a_n$ is reduced to the node σ, as the

Top-down analysis is. But, contrary to the Top-down analysis, in the Bottom-up analysis the first input symbol a_1 is read, and the rule q

$$q\colon A_q \rightarrow a_1 X_{q2} \cdots X_{qn_q}$$

which has a_1 as the first component of the right handside is referred without regard to the current goal σ. Each component X_{qi} $(2 \leqq i \leqq n_q)$ is examined to find out whether the rule q is applied or not. In the case that the rule q is applied, if $A_q = \sigma$ then parsing is completed successfully, otherwise the rule q'

$$q'\colon A_{q'} \rightarrow A_q X_{q'2} \cdots X_{q'n_{q'}}$$

which has A_q as the first component is chosen and parsing is continued. For example, consider the case to parse the input string "cd" with the set of rules

$$\begin{array}{llll} P\colon & 1.\ \sigma \rightarrow AB & 3.\ A \rightarrow a & 5.\ C \rightarrow c \\ & 2.\ \sigma \rightarrow CD & 4.\ B \rightarrow b & 6.\ D \rightarrow d \end{array}$$

In the Top-down analysis the first rule $\sigma \rightarrow AB$ is applied, but cancelled with the unrecognition of A. Next, the second rule $\sigma \rightarrow CD$ is applied. "c" is reduced to C and "d" is reduced to D, and σ is recognized successfully. On the other hand, in the Bottom-up analysis, the first input symbol "c" is read and it is reduced to C according to the rule 5. Next, the rule 2, which has C as the first component, is applied, and the goal σ is attained with the recognition of the second component D.

In the Bottom-up analysis, because the rules are selected regarding input symbols, the parsing process could be gone ahead more effectively when each terminal symbol has the proper meaning.

To simplify the parsing procedure the rewriting rules are written in the form as

$$X_{q1} X_{q2} \cdots X_{qn_q} \rightarrow A_q\,.$$

The rules which have same first symbol are grouped and are numbered with consideration of the condition a.

Example 6. The set of rules P to generate ⟨assignment statement⟩ is represented as Ptop for the Top-down analysis and Pbot for the Bottom-up analysis.

Ptop	Pbot
1. $\langle as \rangle \rightarrow \langle left \rangle \langle e \rangle$	1. $\langle left \rangle \langle e \rangle \rightarrow \langle as \rangle$
2. $\langle e \rangle \rightarrow \langle t \rangle ^* \{\langle ap \rangle \langle t \rangle\}$	2. $\langle e \rangle \langle ap \rangle \langle t \rangle \rightarrow \langle e \rangle$
3. $\langle t \rangle \rightarrow \langle p \rangle ^* \{\langle mp \rangle \langle p \rangle\}$	3. $\langle t \rangle \langle mp \rangle \langle p \rangle \rightarrow \langle t \rangle$
4. $\langle p \rangle \rightarrow I$	4. $\langle t \rangle \rightarrow \langle e \rangle$
5. $\langle p \rangle \rightarrow (\langle e \rangle)$	5. $\langle p \rangle \rightarrow \langle t \rangle$
6. $\langle left \rangle \rightarrow I{:}=$	6. $I{:}= \rightarrow \langle left \rangle$

7. $\langle ap\rangle \rightarrow +$	7. $I \rightarrow \langle p\rangle$
8. $\langle ap\rangle \rightarrow -$	8. $(\langle e\rangle) \rightarrow \langle p\rangle$
9. $\langle mp\rangle \rightarrow \times$	9. $+ \rightarrow \langle ap\rangle$
10. $\langle mp\rangle \rightarrow /$	10. $- \rightarrow \langle ap\rangle$
	11. $\times \rightarrow \langle mp\rangle$
	12. $/ \rightarrow \langle mp\rangle$

Moreover, to make the parsing procedure more effective, the following character is derived from the set of rules beforehand.

Definition 12. The component X_{qi} $(1 \leqq i \leqq n_q)$ of the rule q has the first kind of attainability to the goal A_q.

Definition 13. The first component X_{q1} of the rule q has the second kind of attainability to the goal A_r, if A_q is the first component of the rule r or A_q has the second kind of attainability to X_{r1}.

It is obvious that if X_{q1} has the second kind of attainability to A_r then $X_{q1} \in LC(A_r)$.

Example 7. We can obtain the following Attainability Table from Pbot of Example 6.

Table 1. Attainability Table.

To / From	$\langle as\rangle$	$\langle left\rangle$	$\langle e\rangle$	$\langle t\rangle$	$\langle p\rangle$	$\langle ap\rangle$	$\langle mp\rangle$
$\langle left\rangle$	1						
$\langle e\rangle$	1		1		1		
$\langle t\rangle$			1	1			
$\langle p\rangle$			2	1			
$\langle ap\rangle$			1				
$\langle mp\rangle$				1			
I	2	1	2	2	1		
$:=$		1					
(			2	2	1		
$+, -$						1	
$\times, /$							1
)					1		

1 means the first kind of attainability.
2 means the second kind of attainability.

Definition 14. Symbol $X (X \in V)$ is attainable to the goal A $(A \in N)$ if X has the first or second kind of attainability to A, and it is represented as

$$T(X, A) = 1$$

otherwise $$T(X, A) = 0\,.$$

For example, in Example 7,

$$T(\langle p\rangle, \langle e\rangle) = 1, \quad T(\langle p\rangle, \langle t\rangle) = 1, \quad T(\langle p\rangle, \langle p\rangle) = 0\,.$$

In the Bottom-up analysis, the Attainability Table, auxiliary variables (G, REP, r, k, j, m) and one stack E are used. Variable REP is used to indicate whether a recursive rule is being referred and the goal G has been attained once or not. In order to correspond the longest substring to the goal, in place of the repetitive representation of the rewriting rules, the goal is repeatedly attained in the parsing procedure.

For example, the input string "$a+b+c$" once attains to the goal $\langle e\rangle$ with the substring "$a+b$" by the rule 2 of Pbot. And refering again the rule 2 which has $\langle e\rangle$ as the first component, "$a+b+c$" is reduced to $\langle e\rangle$. So, for the Bottom-up analysis, rewriting rules could be left recursive.

In the following parsing procedure it is supposed that no rule contains either the repetitive component or the factoring form.

Step A′: set the initial values of each variable

$$(G, REP, r, k, j, m) := (\sigma, 0, 0, 0, 1, 1)\,. \qquad (1)'$$

Step B′: read the next input symbol a_j, search the rule q such as $X_{q1} = a_j$ in the order of the number and

if $$A_q = G \quad \text{or} \quad T(A_q, G) = 1 \qquad (2)'$$

then $(REP, r, k, j) := (0, q, 1, j+1)$ and go to Step C′,
otherwise repeat the same test $(2)'$ with $q := q+1$.

When there exists no rule q which satisfies the condition $(2)'$, go to Step E′ with the possibility of some syntactic error in the input string.

Step C′ : $k := k+1$ and examine the component X_{rk},

1. if $X_{rk} \in \Sigma$ then examine the input symbol a_j

if $a_j = X_{rk}$ then $j := j+1$ and repeat Step C′,
otherwise go to Step F′.

2. if $X_{rk} \in N$ then examine the input symbol a_j

if $T(a_j, X_{rk}) = 1$ (*i.e.*, $a_j \in LT(X_{rk})$) then set X_{rk} as the new goal

$$\textit{i.e.}, \text{ RESERVE } E(G, REP, r, k, j, m);\ G := X_{rk}$$

and go to Step B′,

otherwise go to Step F′, because the left substring of $a_j \cdots a_n$ has no possibility to be reduced to X_{rk}.

3. if $k > n_r$ then, because all the components of the rule r have been recognized and the node A_r has been attained, add the rule number r to the output sequence and $m := m+1$. And

if $A_r = G$ then try to attain to the goal G repeatedly

$$i.e.,\ E(j, m) := (j, m);\ REP := 1$$

and go to Step D′,

otherwise (*i.e.*, $A_r \neq G$) go to Step D′.

Step D′ : If the attained node A_r is σ then the parsing procedure stops successfully. Otherwise search the rule q such that $A_r = X_{q1}$, and if

$$A_q = G \quad \text{or} \quad T(A_q, G) = 1 \tag{3$'$}$$

then $(r, k) := (q, 1)$ and go to Step C′,

else repeat the same test (3)′ with $q := q+1$.

When there exists no rule which satisfies the condition (3)′, go to Step E′.

Step E′ : In the case that no rule has the same first component as the next input symbol a_j or the attained node A_r,

1. if $REP = 1$ then, because the goal has been attained once,

$$\text{RESTORE } E(G, REP, r, k, j, m);\ \text{ERASE } E$$

and go to Step C′.

2. if $REP = 0$ then stop the parsing procedure because there exists some syntactic error in the input string.

Step F′ : In the case that the component $X_{rk} (k \geqq 2)$ could not be recognized, examine the following rules of the rule r. If there exists the rule $(r+i)$ for any i such that

$$X_{rl} = X_{(r+i)l}\ (l = 1, 2, \cdots, k-1),\ X_{rk} \neq X_{(r+i)k}\ ,$$
$$\text{and} \quad A_{(r+i)} = G \quad \text{or} \quad T(A_{(r+i)}, G) = 1 \tag{4$'$}$$

then $r := r+i$; $k := k-1$ and go to Step C′.

Otherwise give up the current goal

$$i.e.,\ \text{RESTORE } E(G, REP, r, k, j, m);\ \text{ERASE } E$$

and go to Step C′ when $REP = 1$ or repeat Step F′ when $REP = 0$.

The main difference between the Top-down analysis and the Bottom-up analysis is how to choose the next referring rule in Step B and Step B′ and D′. The former chooses it according to the node of the subtree (*i.e.*, the current goal) and the latter to the most left branch of the subtree (*i.e.*, LC(G)).

As seen above, the parsing procedure needs not be changed for various sets of rewriting rules (*i.e.*, for various input languages).

5. Parsing procedure of COL

As an example of the Compiler Compiler, we define the Compiler Oriented Language (COL) to describe compilers. The COL System is a programming system to generate compilers by means of a computer and consists of the language COL and COL Processor. COL is the language for a compiler builder to instruct various actions on the compiling process to a computer. The parsing phase of a compiler is described in Syntax Statement and the translating phase in Semantic Statement. In order that the compiler can work on a computer, these COL statements must be translated into the sequence of machine instructions. It is the COL Processor to perform this transformation.

The generated compiler is divided into two parts:

1. The Analyzer, which is described in the Syntax Statement, parses the string of input symbols and constructs its M-structure.
M-structure represents the syntactic structure of the corresponding string.

2. The Translator, which is described in the Semantic Statement, interprets the result of the parsing and produces the sequence of the object symbols.
That is, a compiler is defined as follows:

$$\langle COMPILER\rangle ::= *\{\textit{Syntax Statement}\}\ ENDMARK\ *\{\langle \textit{M-routine}\rangle\}\ ENDMARK$$
$$\langle \textit{M-routine}\rangle ::= MNAME\text{: } *\{\textit{Semantic Statement}\}\ .$$

As mentioned previously, the Syntax directed analysis is specified with a set of rewriting rules P and the parsing procedure Π. For various input languages the corresponding set of rules P is given, but the parsing procedure Π is unchanged. The compiler takes the following basic actions in the parsing procedure.

1. recognizing a "delimiter" as a terminal symbol,
2. editing a basic item such as "identifier", "number" and "string" and transforming it into an inner representation,
3. recognizing a syntactic unit as a non-terminal symbol,
4. storing and/or restoring the necessary information for the action 3,
5. specifying the next action in the parsing procedure i.e., choosing the rule referred to next,
6. recording the result of the parsing.

To write an Analyzer the set of rules P of the input language is modified on the conditions a, b, c, d and e. Then, according to the modified rules, the parsing procedure is described in the sequence of Syntax Statements defined as follows.

$\langle$*Syntax Statement*$\rangle$:: = $\langle$*Label*$\rangle\langle$*Read* & *Test*$\rangle$? $\langle$*T. Action*$\rangle$ & $\langle$*F. Action*$\rangle$
$\langle$*Label*$\rangle$ :: = *SNAME*: $|\, \phi$
$\langle$*Read* & *Test*$\rangle$ :: = $\langle$*Read*$\rangle\langle$*Test*$\rangle$
$\langle$*Read*$\rangle$ :: = $*\,|\,\langle$*Read*$\rangle * \,|\, \phi$
$\langle$*Test*$\rangle$ :: = $I\,|\,N\,|\,G\,|$ "*delimiter*" | (*SNAME*)
$\langle$*T. Action*$\rangle$ :: = $\langle$*Trans*$\rangle\,|\,*\{\langle$*Action*$\rangle,\}\,\langle$*Trans*$\rangle$
$\langle$*Trans*$\rangle$ :: = *TR* | *FR* | ♯*SNAME*
$\langle$*Action*$\rangle$ :: = $\langle$*Result*$\rangle$ | *ON* | / | $\langle$*Error*$\rangle$
$\langle$*Result*$\rangle$ :: = (*MNAME*) | @
$\langle$*Error*$\rangle$ :: = *E**{*letter* | *digit*}
$\langle$*F. Action*$\rangle$ :: = $\langle$*T. Action*$\rangle$ | $\langle$*T. Action*$\rangle$, *AT*

The Analyzer has an inner logical variable TEST. TEST is given the value (**true**, **false**) depending on the result of $\langle$*Test*$\rangle$ operation. In a Syntax Statement $\langle$*Read* & *Test*$\rangle$ is performed and $\langle$*T. Action*$\rangle$ or $\langle$*F. Action*$\rangle$ is taken in the manner such as

if TEST **then** $\langle$*T. Action*$\rangle$ **else** $\langle$*F. Action*$\rangle$.

Each Syntax Statement is performed sequentially, but the sequence is broken by a $\langle$*Trans*$\rangle$ operation

Example 8. An input language is defined as follows.

$\langle$*BLOCK*$\rangle$ :: = **begin** $*\{\langle D\rangle\}\langle ST\rangle *\{;\langle ST\rangle\}$ **end**
$\langle D\rangle$ :: = **real** I $*\{, I\}$;
$\langle ST\rangle$ :: = $I = *\{I=\}\langle EXP\rangle$
$\langle EXP\rangle$ :: = $\{-\,|\,\phi\}\langle T\rangle *\{+\langle T\rangle\,|\,-\langle T\rangle\}$
$\langle T\rangle$ :: = $\langle F\rangle *\{\times\langle F\rangle\}$
$\langle F\rangle$ :: = $\langle P\rangle *\{\,!\,\langle P\rangle\}$
$\langle P\rangle$ :: = $I\,|\,N\,|\,(\langle EXP\rangle)$

where *I* and *N* represent an identifier and a number respectively. The Analyzer of this language is specified by the following Syntax Table (Table 2).

A push-down storage has the structure

(*head*)——(*tail*)

and the information is stored and/or restored first-in-last-out manner through the *head*. The Analyzer has three push-down storages whose *heads* are called IP, OP, STP.

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

OP indicates the number of symbols in the M-structure.

STP points the Syntax Statement which is performed.

Table 2. Syntax Table.

BL:	*“°BEGIN°”	? (HEAD)	& #BL
BD:	(D)	? (DH), #BD	&
BS:	(ST)	?	&
	*“ ; ”	? (STE)/#BS	&
	“°END°”	? (END)/	& E100, AT
EE:	*“ & ”	? TR	& #EE
D:	*“°REAL°”	? (REAL)	& FR
D1:	I	? (I) @	& E101
	*“ , ”	? #D1	&
	“ ; ”	? (TD)/TR	& E102, AT
ST:	(LEFT)	? #ST	&
	(EXP)	? (RITH)/TR	& (DUMY) TR
LEFT:	I	? (P)@	& FR
	“ = ”	? (LEFT)TR	& FR
EXP:	*“ – ”	?	& ON, #EP1
	(T)	? (NEG)#EP2	& (NUL) #EP2
EP1:	(T)	?	& FR
EP2:	*“ + ”	?	& #EP3
	(T)	? (ADD)#EP2	& E201, (NUL) AT
EP3:	“ – ”	?	& ON, TR
	(T)	? (SUB)#EP2	& E202 (NUL) AT
T:	(F)	?	& FR
T1:	*“*”	?	& ON, TR
	(F)	? (MUL)	& E203 (ONE), AT
F:	(P)	?	& FR
F1:	*“!”	?	& ON, TR
	(P)	? (EXP)#F1	& E204 (ONE) AT
P:	I	? (P)@, TR	&
	N	? (N)@, TR	&
	*“ (”	?	& FR
	(EXP)	?	&
	*“) ”	? TR	& E205, ON, TR
SYNTAX END			

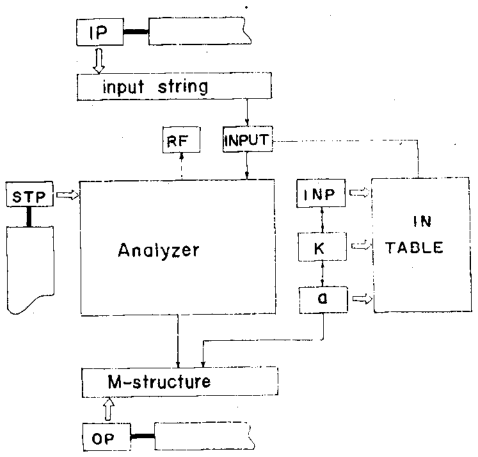

Fig. 1. The mechanism of the Parsing phase.

They are manipulated by the push-down operation.

RESERVE (E)	: push down *tail* E and *head* E→*tail* E.
RESTORE (E)	: *head* E←*tail* E and pop up *tail* E.
REWRITE (E)	: *head* E←*tail* E. *tail* E is unchanged.
LOSE (E)	: pop up *tail* E. *head* E is unchanged.
ERASE (E)	: erase the whole information in *tail* E.

Now, each operation of the Syntax Statement has the following meanings.

1. Read operation *

Analyzer has INPUT register and a logical variable RF.
if RF=on then RF:=off.
if RF=off then IP:=IP+1 and put the next one symbol of the input string into INPUT.

2. Basic item test operation I, N, G

Test operation I (or N, G) examines whether an identifier (or a number, a string) is found at the next position of the input string. If the specified basic item is not found then TEST is set to **false**, otherwise the basic item is edited in IN TABLE, its inner representation is set in the register *a* and TEST is set to **true**.

3. Terminal symbol test operation "delimiter"

This operation corresponds to Step C1 or Step C′1, and examines whether the specified "delimiter" is found at the next position of the input string. If it is found then TEST is set to **true**, otherwise TEST is set to **false**.

4. Non-terminal symbol test operation (SNAME)

This operation corresponds to Step C2 or Step C′2, and recognizes whether the left substring of the remaining input string is reduced to the specified syntactic unit or not. After the execution of

RESERVE (IP, OP, STP); STP: = SNAME,

the parsing procedure is continued from the specified Syntax Statement.

5. ⟨*Trans*⟩ operation

a. The operation ♯SNAME transfers the parsing procedure to the Syntax Statement with the specified SNAME

STP: = SNAME.

b. The operation TR is used to reply that all the components of the syntactic unit specified in 4 are recognized, and to return to the calling point with execution of

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

c. The operation FR is used to reply that the syntactic unit specified in 4 can not be recognized and to backtrack to the calling point with execution of

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

6. ⟨*Result*⟩ operation

This is the operation to construct the M-structure representing the results of parsing. M-structure is formed with name of M-routines and the inner representation of basic items.

a. (MNAME) specifies a M-routine name to be inserted in the M-structure

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

b. @ requires that the inner representation of a basic item, which is the result of the operation I, N or G, is inserted in the M-structure

OP: = OP+1; (OP): = *a*.

7. RF set operation ON

This instructs the logical variable RF to be set "on".

8. Operation AT

This operation is permitted only in ⟨*F. Action*⟩ and indicates to perform ⟨*T. Action*⟩ in the same Syntax Statement. Operation AT is used to simplify the denotation of a Syntax Statement when ⟨*T. Action*⟩ and ⟨*F. Action*⟩ have the common operations or to continue parsing procedure after processing of a syntactic error of the input string.

9. Error indication operation E

The string of letters and digits following the letter E is printed to indicate the syntactic error of the input string or the state of parsing. When a syntactic error is found, parsing procedure should be continued after processing the error rather than be stopped.

10. Compile operation /

The compile operation means pausing of parsing phase and beginning of translating phase. This operation is instructed at the end of the context which has no anxiety of backtracking of the parsing. This is called an Incremental Unit (IU). As to ALGOL, one statement and one declaration becomes an IU, and also an ⟨actual parameter⟩ or a ⟨for list element⟩ could be an IU. By performing parsing and translating mutually for an IU, it is not necessary to obtain the large editing area for M-structure, and translation is done effectively with some information from various parts of the M-structure.

Example 9. The M-structure of the language of Example 8 is defined as follows:

$$\begin{aligned}
\langle p\rangle &= [P]a \mid [N]a \mid \langle ae\rangle\\
\langle f\rangle &= \langle p\rangle *\{\langle p\rangle[EXP]\}\\
\langle t\rangle &= \langle f\rangle *\{\langle f\rangle[MUL]\}\\
\langle ae\rangle &= \langle t\rangle\{[NEG] \mid \phi\} *\{\langle t\rangle[ADD] \mid \langle t\rangle[SUB]\}\\
\langle st\rangle &= [P]a[LEFT] *\{[P]a[LEFT]\}\langle ae\rangle[RITH] \;/\\
\langle d\rangle &= [REAL][I]a *\{[I]a\}[TD] \;/\\
\langle block\rangle &= [HEAD] *\{\langle d\rangle\ [DH]\}\langle st\rangle *\{[STE] \;/\; \langle st\rangle\}[END] \;/
\end{aligned}$$

where the symbol "/" represents the pause of IU and the name in the bracket is MNAME. The part of expression is represented in reverse polish form in M-structure.

In translating phase the M-routines, whose MNAMEs are specified in the M-structure, are performed and generate the sequence of output symbols (object program). After the interpretation of the M-structure of an IU, parsing phase is restarted and Analyzer parses the input string corresponding to next IU.

6. Results of experiments and conclusion

Four compilers were described in COL and generated by COL Processor. Their input languages are ALGOL or subset of ALGOL and output languages are three-address symbolic language or machine language.

The resulting compilers can accept ALGOL programs satisfactorily. The rate of the parsing time to whole compiling time depends on the context of the input string and it is about 50 percent on the average. Since the COL-parsing is based on the Syntax directed analysis, the parsing time of a string depends on the depth of the node of the corresponding tree structure and the order of referred rules. For example, Table 3 shows the compiling time and parsing time of some statements.

As a result, it may be said that the Syntax directed analysis is one basis of the

Table 3. Compiling time and parsing time.

	compiling time	parsing time
$x := z;$	9.0	5.0
$x := (z);$	12.5	8.5
$x := ((z));$	16.0	12.0
$x := y \uparrow z;$	11.0	5.3
$x := y \times z;$	12.0	5.9
$x := y/z;$	12.3	6.1
$x := y + z;$	12.5	6.5
$x := y - z;$	12.8	6.8
$x := A[i];$	18.0	11.8
go to $L1;$	6.0	2.0
if $x = y$ **then** $P;$	17.8	8.2
if $x = y$ **then go to** $L1;$	18.8	9.5
if $x = y$ **then** $x := y$ **else** $x := z;$	32.2	17.0
for $i := 1$ **step** 1 **until** n **do** $x := y;$	29.6	14.5

(sec.)

Compiler Compiler.

1. Once the parsing procedure Π has been made, the Analyzers of various input languages are easily composed with only the syntactic definition of the language. That definition is given in the form such as rewriting rules, BNF, canonic system, or the modified set of rules according to the conditions a~e. Meta II[2], Meta III[3] are its instances. This belongs to the Parametric-type.

But, the more complicated the structure of an input language is, the slower the parsing speed is, because rewriting rules increase in number and the possibility of backtracking of the parsing is increased.

2. The parsing procedure is simple and regular. So, in order to generate an Analyzer, compiler describing language is constructed. The Syntax Statement of COL and Feldman's Production Language[4] are its instances. They specify both the parsing procedure and rewriting rules of the input language together. This belongs to the Describing-type.

On the other hand, the translator, which takes different kind of actions from the Analyzer, is specified depending on the output language and the meanings of the input language. Any good way is not known to specify them formally. The describing language is only one well-known method. The Semantic Statement of COL and Feldman's Formal Semantic Language are its instances.

To develope the Compiler Compiler technology, it is required to find new methods to generate other parsing algorithms from rewriting rules and to specify the semantics formally.

References

1) H. Hagiwara and K. Watanabe; J. Inform. Proc. Soc. Japan, **9**, 187 (1968)
2) D.V. Schorre; Proc. ACM Nat. Conf. **19**, Dl. 3 (1964)
3) F.W. Schneider and G.D. Johnson; Proc. ACM Nat. Conf. **19**, Dl. 5(1964)
4) J.A. Feldman; Comm. ACM **9**, 3 (1966).