

# Compiler Compiler

について

京都大学工学部 萩原 宏, 渡辺 勝正

## §1. Compiler と Compiler Compiler

すべての言語は、記号列の集合であり、Compiler (一般に translator) は、或る言語 (入力言語  $L_1$ ) の記号列を、他の言語 (出力言語  $L_2$ ) の記号列に変換する媒体である。

各言語の個々の記号および記号列は、ある定められた約束のもとに何らかの意味を持っており、入力言語から出力言語への変換は、系列としての記号の組み合わせだけでなく、その意味も考慮に入れておこなわれる。プログラミング言語の場合には、表現による言いまわしの違いはあっても、ニュアンス (nuance) の違いとか、あいまいさというものが無いため (無いのが望ましく、あつては困る)、入力記号列 (source program) は一意的に出力列 (object program) に変換される。

入力言語の一つの記号  $a$  が出力言語の一つの記号  $b$  または記号列  $b_1 b_2 \dots b_k$  に直接対応がつかう場合には、Compiler の機

構 (i.e. 変換の手順) は単純である。基本的な Assembler や、一つの計算機  $M_1$  の機械語で書かれたプログラムを他の計算機  $M_2$  の機械語のプログラムに変換する手順などはその例である。

一般には、対応はもっと複雑である。Compiler は、

(i) 構造分析過程: 入力記号列  $a_1 a_2 \dots a_m$  における各記号  $a_j$  の互いあひさの関係を解析する。

(ii) 意味解釈過程: 互いあひさの関係から明らかに  $n$  と  $k$  入力記号列の意味に基づいて出力列  $b_1 b_2 \dots b_m$  を構成する。

なる二つの働をする機構を持つものである。

(1) 一つの入力言語 (Problem Oriented Language, POL) と一つの出力言語 (Machine Oriented Language, MOL) に対して、Compiler は一つ。

(2) 一つの POL ( $k$  とせば PL/I) と  $m$  個の MOL に対して Compiler は  $m$  個。

(3)  $p$  個の POL と一つの MOL に対して Compiler は  $p$  個。

(4)  $p$  個の POL と  $m$  個の MOL に対して Compiler は  $p \times m$  個。

(5) Universal Programming Language の考えに従って共通の中間言語を採用すると Compiler は  $(p + m)$  個。

(6) POL と MOL の仕様をパラメータとして与えられる場合には、一つの Compiler と  $(p \times m)$  種のパラメータ。

最後の考えに基づいて (2), (3) は次のように変更される:

(2)' 一つの PDL に対して、一つの Compiler と  $m$  種のパラメータ。

(3)' 一つの MOL に対して、一つの Compiler と  $p$  種のパラメータ。

(6) の考えに準じて Compiler Compiler を狭い意味で解釈すると：

「入出力言語  $L_1, L_2$  が定義される時、 $L_1$  から  $L_2$  への Compiler を作り出す手順である」

と言う：とが出来ることが、ここでは、

「入出力言語を出来るだけ形式的 (formal) に定義し、Compiler の変換手順を単純明解なものにして、計算機を用いて、Compiler をより短期間に容易に作成する手順」

を広く Compiler Compiler と解釈する。

この時、Compiler では、入力言語  $L_1$ 、出力言語  $L_2$  の他に Compiler 自身が表現されている記述言語  $L_0$  が問題になる。すなわち、Compiler は三つの言語  $L_1, L_2, L_0$  を規定することによって、

$$\begin{array}{c} L_0 \\ \downarrow \\ L_1 \rightarrow L_2 \end{array}$$

で特徴づけられる。

更に、Compiler Compiler では生成される Compiler を特徴づけるために、

- コンパイル方法を表わす性質  $\alpha$
- Compiler の目的を表わす性質  $\beta$

がつけ加えられる。しかし、

- i. コンパイル方法は最良と思われるものを採用すればよい。  
ii. (入力言語  $L_1$  に依存する)。
- ii. 新しいコンパイル方法が開発された時には、*Compiler Compiler* にも変更を必要とする。
- iii. *Compiler* の目的によって、コンパイル方法も違ってくる。

などの理由のため、現在の時点では *Compiler Compiler* の問題を複雑に可子にすであるから、当面の問題として  $\alpha, \beta$  を一般的に扱うことにしよう。

## § 2. *Compiler Compiler* の方法

*Compiler* を特性づける  $\equiv$  の言語  $L_0, L_1, L_2$  の各々に観点をおいて、*Compiler Compiler* に対して次のような考察が加えられる。

$L_0$ : 機械語またはアセンブラ言語で *Compiler* を記述する場合には、機械の特長とあぐれなプログラミング技術を活用して効率の良き *Compiler* を作成することが出来るが、その作成とデバッグに長い期間と多くの人手を必要とする。また機械毎に *Compiler* を別個に記述しなければならぬ。これらの欠点を補う目的で *Compiler* 記述言語を採用することが *Compiler Compiler* の一方法となる。(記述型に当てる。)

L1: 入力言語を検討して出来るだけ formal の定義を与えて、単純なコンパイラ方法を究明するものである。現在までにこれだけ言語およびコンパイラ方法についての理論的研究の多くは、書きかえ規則から出発して構文論すなわち構造分析手順の解明に力が注がれて来ている。phrase structure grammar, BNF, LR(k)-文法, canonic system 更には automaton の理論はこの部類に含まれる。この書きかえ規則の参照の仕方によってコンパイラ方法は次のように分類出来る:

0. 入力言語の文法に従って Compiler 作成者がすべてを規定する。
1. 記号の順位 (precedence) に基づく方法
2. Transition Matrix から分析手順をえらぶ方法
3. 書きかえ規則を直接参照しながら分析を進める方法

L2: 入力言語の意味論的考察と関連して Compiler の出力言語のあり方を検討するもので、計算機言語あるいは計算機としての将来の姿に一つの示唆を与えんとするものである。Universal Language (5) の考へはその一例である。具体的には、multi-address symbolic code,  $\lambda$ -notation と ALGOL の対応づけ、B5500 の言語 などの研究がある。

L2 の条件としては、

{ 様な入力言語から比較的容易に変換出来る。

{機械語への変換あるいは計算機による直接実行が容易。  
 なることが要求される。

具体的には Compiler Compiler の手法は次のように分類される。

a. 変換型 : machine  $m_1$  で働く POL  $p$  の Compiler  $C_{p \rightarrow m_1}^{m_1}$  から machine  $m_2$  で働く Compiler  $C_{p \rightarrow m_2}^{m_2}$  を作成する。

- (i) 1.  $C_{p \rightarrow m_2}^p$  を記述する  
 2.  $C_{p \rightarrow m_1}^{m_1} (C_{p \rightarrow m_2}^p)$  により  $C_{p \rightarrow m_2}^{m_1}$  を得る。  
 3.  $C_{p \rightarrow m_2}^{m_1} (C_{p \rightarrow m_2}^p)$  により  $C_{p \rightarrow m_2}^{m_2}$  を得る。

POL  $p$  で Compiler  $C_{p \rightarrow m_2}^p$  が記述出来るか否かが問題である。

- (ii) 1.  $C_{m_1 \rightarrow m_2}^{m_1}$  と  $C_{m_1 \rightarrow m_2}^{m_2}$  を記述する。  
 2.  $C_{m_1 \rightarrow m_2}^{m_1} (C_{p \rightarrow m_1}^{m_1})$  により  $C_{p \rightarrow m_2}^{m_1}$  を得る。  
 3.  $C_{p \rightarrow m_1}^{m_1}$  と  $C_{m_1 \rightarrow m_2}^{m_2}$  が  $C_{p \rightarrow m_2}^{m_2}$  の分割を果す。

$m_1$  と  $m_2$  の機能・特性が似ている場合には有効である。

- (iii)  $p_1, p_2$  に対し  $C_{p_1 \rightarrow m_1}^{m_1}$  から  $C_{p_2 \rightarrow m_2}^{m_2}$  を作成する。  
 1.  $C_{p_2 \rightarrow m_2}^{p_1}$  と  $C_{p_2 \rightarrow m_2}^{p_2}$  を記述する。  
 2.  $C_{p_1 \rightarrow m_1}^{m_1} (C_{p_2 \rightarrow m_2}^{p_1})$  により  $C_{p_2 \rightarrow m_2}^{m_1}$  を得る。  
 3.  $C_{p_2 \rightarrow m_2}^{m_1} (C_{p_2 \rightarrow m_2}^{p_2})$  により  $C_{p_2 \rightarrow m_2}^{m_2}$  を得る。

b. Bootstrap 型 : POL  $p$  の Compiler の機能を増大させてゆく。

- (i) 1.  $p$  の必要十分なサブセット  $p_0$  に対し  $C_{p_0 \rightarrow m}^m$  を記述する。  $i := 0$ 。

- 2.  $C_{p_{i+1} \rightarrow m}^{p_i}$  ( $p_i < p_{i+1}$ ) を記述する。
- 3.  $C_{p_i \rightarrow m}^m$  ( $C_{p_{i+1} \rightarrow m}^{p_i}$ ) により  $C_{p_{i+1} \rightarrow m}^m$  を得る。
- 4.  $i := i + 1$  とし 2, 3 をくり返す。

k番目以降の対し  $i$  は  $p_k, p_{k+1}, \dots$  のうち Compiler の記述に必要の部分に同じに改ると考えられる。この杯は  $k$  をみつけたとき、Compiler 記述言語に関する一の問題である。

- (ii) 1.  $C_{p_0 \rightarrow m}^m$  を記述し、 $i := 0$ 。
- 2.  $C_{p_{i+1} \rightarrow p_i}^{p_i}$  を記述する
  - 3.  $C_{p_i \rightarrow m}^m$  ( $C_{p_{i+1} \rightarrow p_i}^{p_i}$ ) により  $C_{p_{i+1} \rightarrow m}^m$  を得る。
  - 4.  $\bigcup_{j=i}^0 C_{p_{j+1} \rightarrow p_j}^m$  と  $C_{p_0 \rightarrow m}^m$  が  $C_{p_{i+1} \rightarrow m}^m$  の実装をする。
  - 5.  $i := i + 1$  に対し 2, 3, 4 をくり返す。

作成された Compiler は multi phase になる。

C. 記述型 : Compiler 作成の手数を軽減し、計算機に独立に Compile 過程を記述するために、記述言語  $L_0$  を定義する。

- (i) macro assembler 言語を  $L_0$  に用いる。Compiler の基本となる動作を macro order とし定義しておく。
- (ii) ALGOL や PL/I などの POL を  $L_0$  とする。a か b の手法が適用出来るが、ALGOL では付加可能な機能が問題となり、PL/I はサブセットの選択が問題になる。また、構造化手順の記述が複雑になる可能性が大きい。
- (iii) 記号処理を行う List Processing 言語を  $L_0$  とする。

(iv) "Compiler を記述する" という問題向き言語を定義する。

Brooker R.A : The Compiler Compiler.

Annual Review in Automatic Programming Vol 3. 1963

Feldman J.A : Translator Writing System.

C. ACM Vol 11 (Feb. 1968)

定義された  $L_0$  により、 $L_0$  を簡潔に表現出来るが、次の二つが短所となる。

1.  $L_0$  として採用する言語により、コンパイラ手法がある程度制限される。

2. 新たな言語  $L_0$  の Processor を必要とする。このため  $L_0$  の条件としては、理解しやすく書きやすいこと、Compiler の動作が完全に記述出来ること、さらに、 $L_0$  の Processor が出来ただけ簡単であること、が要求される。

d. Parameter型 : ①の(6)または(3)'に基づいて POL (および MOL) の定義を何らかの形で与えると、あらかじめ用意された構造分析手順に従ってコンパイルを行う Compiler が構成される。Syntax-directed Compiler はこの典型である。

d. と c(iv) とは本質的に区別を明確にしながら、

{	構造分析手順	は	d	Ex. Production Language
	意味解釈手順	は	c(iv)	Formal Semantic Language

の傾向が強い。



様々の Compiler Compiler の才力が考えられ、 $\llcorner \rightarrow \llcorner$  の試みが  
 行われてはいるが、その実用性も作成された Compiler の space  
 と compile-time に問題が残されている。

次に、記述型才力の一例として Compiler 記述言語 COL とその  
 根拠となる syntax directed analysis について検討する。

COL は、

- ・ 能率のよい Compiler を、より容易に作成する。
- ・ Compiler が何を為すものかを明らかにする。
- ・ Compiler にとって必要不可欠な機能を知る：とくに、 $\llcorner$  コ  
 ンパイルを行う計算機があるならば入力言語を直接理解して  
 実行する計算機に代わらばその機能を探索する。

ためのものである。

### § 3. Syntax-directed Analysis の問題点

phrase structure grammar  $G = (V, \Sigma, P, \sigma)$  において、

書き換え規則の組  $P$  は、

$$A \longrightarrow X_1 X_2 \cdots X_n, \quad A \in N, X_i \in V$$

で与えられるものとす。ここに、

$V$  : vocabulary

$\Sigma$  : alphabet i.e. a set of terminal symbols  $\Sigma \subset V$

$$N = V - \Sigma, \quad N \cap \Sigma = \emptyset$$

$P$  : a set of rewriting rules

$\sigma$  : initial symbol  $\sigma \in N$

phrase structure grammar  $G$  により、言語  $L(G)$  が定義される。

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

### 3.1 Top down 分析法

入力記号列を tree 構造に表現するとき、tree の root にある node  $\sigma$  から始めて、各 node を構成する要素を、与えられた書きかえ規則の組  $P$  を参照しながら、順次確認し、最終的に入力記号列  $\alpha = a_1 a_2 \dots a_n$  が  $\sigma$  に帰着されることを確かめるものである。これは sequential machine の状態図に基づいて状態を移行させてゆくことによく似ている。

規則の組  $P$  は次の諸点に注意して与えられる：

a. 同じ左辺の記号を持つ規則を  $\alpha$  とまとめ、node  $\sigma$  を持つ規則から順次ならべる。この時、規則の順番が参照順序になるため

$$A \longrightarrow \alpha_1$$

$$A \longrightarrow \alpha_2$$

なる二つの規則に対して、 $\alpha_2$  が  $\alpha_1$  の左の部分記号列となる可能性のある場合には、後者を後順にする。

例 1.  $\langle \text{for list element} \rangle \longrightarrow \langle \text{ae} \rangle \underline{\text{step}} \langle \text{ae} \rangle \underline{\text{until}} \langle \text{ae} \rangle$   
 $\langle \text{for list element} \rangle \longrightarrow \langle \text{ae} \rangle$

b. 左帰的規則 (left recursive rule),  $k$  とせば

$$A \longrightarrow Ab$$

は、帰的定義の出発とれた規則

$$A \longrightarrow a$$

と合わせてくり返しの形に表現する。すなわち、

$$A \longrightarrow Ab \implies Abb \implies \dots \implies abb \dots b = a b^*$$

$$\text{を } A \longrightarrow a^* \{b\}$$

と表わす。\*{}は {} の中の要素が零回または任意回くり返して規則の構成要素と見做せることを示す。

例 2.  $\left\{ \begin{array}{l} \langle \text{block head} \rangle \longrightarrow \underline{\text{begin}} \langle \text{declaration} \rangle \\ \langle \text{block head} \rangle \longrightarrow \langle \text{block head} \rangle ; \langle \text{declaration} \rangle \end{array} \right.$

は  $\langle \text{block head} \rangle \longrightarrow \underline{\text{begin}} \langle \text{declaration} \rangle^* \}; \langle \text{declaration} \rangle \}$

と表わされる。

c. 右帰的規則 (right recursive rule) もくり返しの形に表わされる。

例 3.  $\left\{ \begin{array}{l} \langle \text{compound tail} \rangle \longrightarrow \langle \text{statement} \rangle \underline{\text{end}} \\ \langle \text{compound tail} \rangle \longrightarrow \langle \text{statement} \rangle ; \langle \text{compound tail} \rangle \end{array} \right.$

$$\langle \text{compound tail} \rangle \longrightarrow \langle \text{statement} \rangle^* \}; \langle \text{statement} \rangle \} \underline{\text{end}}$$

d. 条件 a に従う範囲内で、同じ左辺の記号を持つ規則は参照頻度の高い順になる。これは分析速度をあげるためのもので手順の本質にかかわりない。

例4.  $\langle \text{basic statement} \rangle \longrightarrow \langle \text{assignment statement} \rangle$   
 $\langle \text{basic statement} \rangle \longrightarrow \langle \text{go to statement} \rangle$   
 $\langle \text{basic statement} \rangle \longrightarrow \langle \text{procedure statement} \rangle$   
 $\langle \text{basic statement} \rangle \longrightarrow \langle \text{dummy statement} \rangle$

e. 同じ左辺の記号を持つ規則が右辺が共通の部分記号列を持つ場合 $\kappa$ は、次の $\kappa$ のいずれかにより、 $\tau$ 規則を變形する。

e1. 新たな nonterminal symbol を導入する

e2. factoring とよぶ形に變形する。

例5.  $\langle \text{term} \rangle$  は次の $\tau$ により定義される。

e1:  $\langle \text{term} \rangle \longrightarrow \langle \text{factor} \rangle * \{ \langle \text{post factor} \rangle \}$

$\langle \text{post factor} \rangle \longrightarrow \times \langle \text{factor} \rangle$

$\langle \text{post factor} \rangle \longrightarrow / \langle \text{factor} \rangle$

$\langle \text{post factor} \rangle \longrightarrow \div \langle \text{factor} \rangle$

e2:  $\langle \text{term} \rangle \longrightarrow \langle \text{factor} \rangle * \{ \times \langle \text{factor} \rangle$

$| / \langle \text{factor} \rangle | \div \langle \text{factor} \rangle \}$

以上の變形の後 $\kappa$ 、 $\tau$   $p$ 番目の規則は、

$p: A_p \longrightarrow X_{p1} X_{p2} \dots X_{pmp} [w_p]$

で与えられる。ここで、 $w_p$ は規則 $p$ が成立した時に出力列 $\kappa$ に加えるべき記号列ある $\kappa$ は規則の番号 $p$ を表わす。くり返し要素 $\kappa$ に対し $\tau$ は、分析の結果、何回適用された $\kappa$ が明示される必要がある。(手順の詳細は省略する)

## 3.2 Bottom up 分析法

mode  $\sigma$  を最終目標とし、入力記号列  $\alpha = a_1 a_2 \dots a_n$  が  $\sigma$  に帰着されることを確認するわけであるが、Bottom up では、ともかく入力記号  $a_1$  を読んで、当面の目標 ( $\sigma$ ) に関係なく、 $a_1$  を一構成要素に持つ規則  $g$

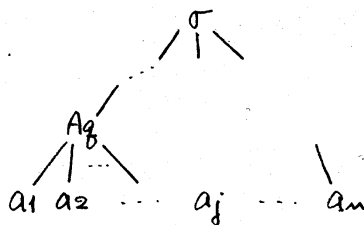
$$g : A_g \longrightarrow a_1 X_{g2} \dots X_{gn} g$$

を参照して、この規則の成立を確認する。規則  $g$  が成立したとき、 $A_g = \sigma$  なら分析は完了するが、 $A_g \neq \sigma$  の場合には  $A_g$  を一構成要素に持つ規則  $g'$

$$g' : A_{g'} \longrightarrow A_g X_{g'2} \dots X_{g'n} g'$$

を参照する。

参照する規則が、次の入力記号があることは確認された mode (これは目標とする mode の最も左の記号である) によって選択されたため、書きかえ規則の左辺と右辺を入れかえて、



$$g : X_{g1} X_{g2} \dots X_{gn} g \longrightarrow A_g [w_g]$$

と表現し、一構成要素に同じ  $a$  を持つ規則をまとめて順序づける。この時、条件  $a$  と  $e$  には注意する必要があるが、規則の帰帰括 (条件  $b, c$ ) には特に配慮する必要はない。帰帰規則は何回かくり返して参照される。

例 6.  $\langle \text{assignment statement} \rangle$  の構造分析を可子規則の組は左から右に次のように行うことができる。

$P_{\text{TOP DOWN}}$	$P_{\text{BOTTOM UP}}$
1 $\langle \text{as} \rangle \rightarrow \langle \text{left} \rangle \langle \text{e} \rangle$	1 $\langle \text{left} \rangle \langle \text{e} \rangle \rightarrow \langle \text{as} \rangle$
2 $\langle \text{e} \rangle \rightarrow \langle \text{t} \rangle * \{ \langle \text{ap} \rangle \langle \text{t} \rangle \}$	2 $\langle \text{e} \rangle \langle \text{ap} \rangle \langle \text{t} \rangle \rightarrow \langle \text{e} \rangle$
3 $\langle \text{t} \rangle \rightarrow \langle \text{p} \rangle * \{ \langle \text{mp} \rangle \langle \text{p} \rangle \}$	3 $\langle \text{t} \rangle \langle \text{mp} \rangle \langle \text{p} \rangle \rightarrow \langle \text{t} \rangle$
4 $\langle \text{p} \rangle \rightarrow I$	4 $\langle \text{t} \rangle \rightarrow \langle \text{e} \rangle$
5 $\langle \text{p} \rangle \rightarrow ( \langle \text{e} \rangle )$	5 $\langle \text{p} \rangle \rightarrow \langle \text{t} \rangle$
6 $\langle \text{left} \rangle \rightarrow I :=$	6 $I := \rightarrow \langle \text{left} \rangle$
7 $\langle \text{ap} \rangle \rightarrow +$	7 $I \rightarrow \langle \text{p} \rangle$
8 $\langle \text{ap} \rangle \rightarrow -$	8 $( \langle \text{e} \rangle ) \rightarrow \langle \text{p} \rangle$
9 $\langle \text{mp} \rangle \rightarrow \times$	9 $+ \rightarrow \langle \text{ap} \rangle$
10 $\langle \text{mp} \rangle \rightarrow /$	10 $- \rightarrow \langle \text{ap} \rangle$
	11 $\times \rightarrow \langle \text{mp} \rangle$
	12 $/ \rightarrow \langle \text{mp} \rangle$

分析過程を能率良く進めるために、規則の組  $P$  から次の性質を調べる必要がある。

I. 規則  $g$  の構成要素  $X_{gi}$  ( $1 \leq i \leq n_g$ ) は、目標  $A_g$  に才一種の到達可能性を持つと言う。

II. 規則  $g$  の才一要素  $X_{gi}$  に対し、 $A_g$  が規則  $r$  の才一要素であるか、 $A_g$  が  $X_{ri}$  に才一種の到達可能性を持つ場合、

$Xq_1$  は  $A_r$  に  $\sigma$  = 種 の到達可能性を持つと言ふ。

出発	目標	<as>	<left>	<e>	<t>	<p>	<ap>	<mp>
<left>		/						
<e>		/		/		/		
<t>				/	/			
<p>				2	/			
<ap>				/				
<mp>					/			
I		2	/	2	2	/		
:=			/					
(				2	2	/		
+ -							/	
x /								/
)						/		

定義. 記号  $X$  ( $X \in V$ ) から目標  $A$  ( $A \in N$ ) に  $\sigma$  = 種  $\sigma$  は  $\sigma$  = 種 の到達可能性を持つ時、 $X$  から  $A$  に到達可能と云ふ。

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

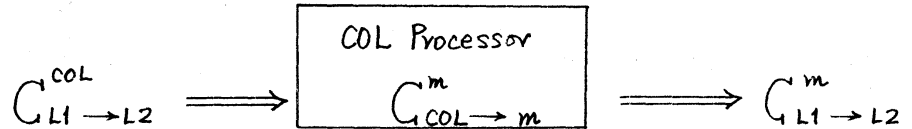
と表わす。そうでない時は

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

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

## § 4. Compiler 記述言語: COL

COL システムは、計算機によって Compiler を作り出すプログラムミニグオオのことで、COL 言語と各計算機に好する COL-Processor の二つの要素に分けられる。



COL 言語は Compiler の構造分析過程と意味解釈過程を記述する能力を持ち、前者を Syntax Statement, 後者を Semantic Statement によって表現する。生成される Compiler は、

$$\langle \text{COMPILER} \rangle ::= * \{ \text{Syntax Statement} \} \text{ENDMARK}$$

$$* \{ \langle M\text{-routine} \rangle \} \text{ENDMARK}$$

$$\langle M\text{-routine} \rangle ::= MNAME : * \{ \text{Semantic Statement} \}$$

で定義され、才一回の構成を持つ。

1. Syntax Statement の記述から表現される部分は、入力言語の記号列の構造を分析して、結果を表わす M-structure を構成する。 [ Analyzer ]。
2. Semantic Statement の記述から表現される部分は、Analyzer の解析結果に基づいて意味解釈を行われ、目的とする出力列を構成する。 [ Translator ]。

## 4. 1 Syntax Statement

Syntax-directed analysis における Compiler の動作を分類する



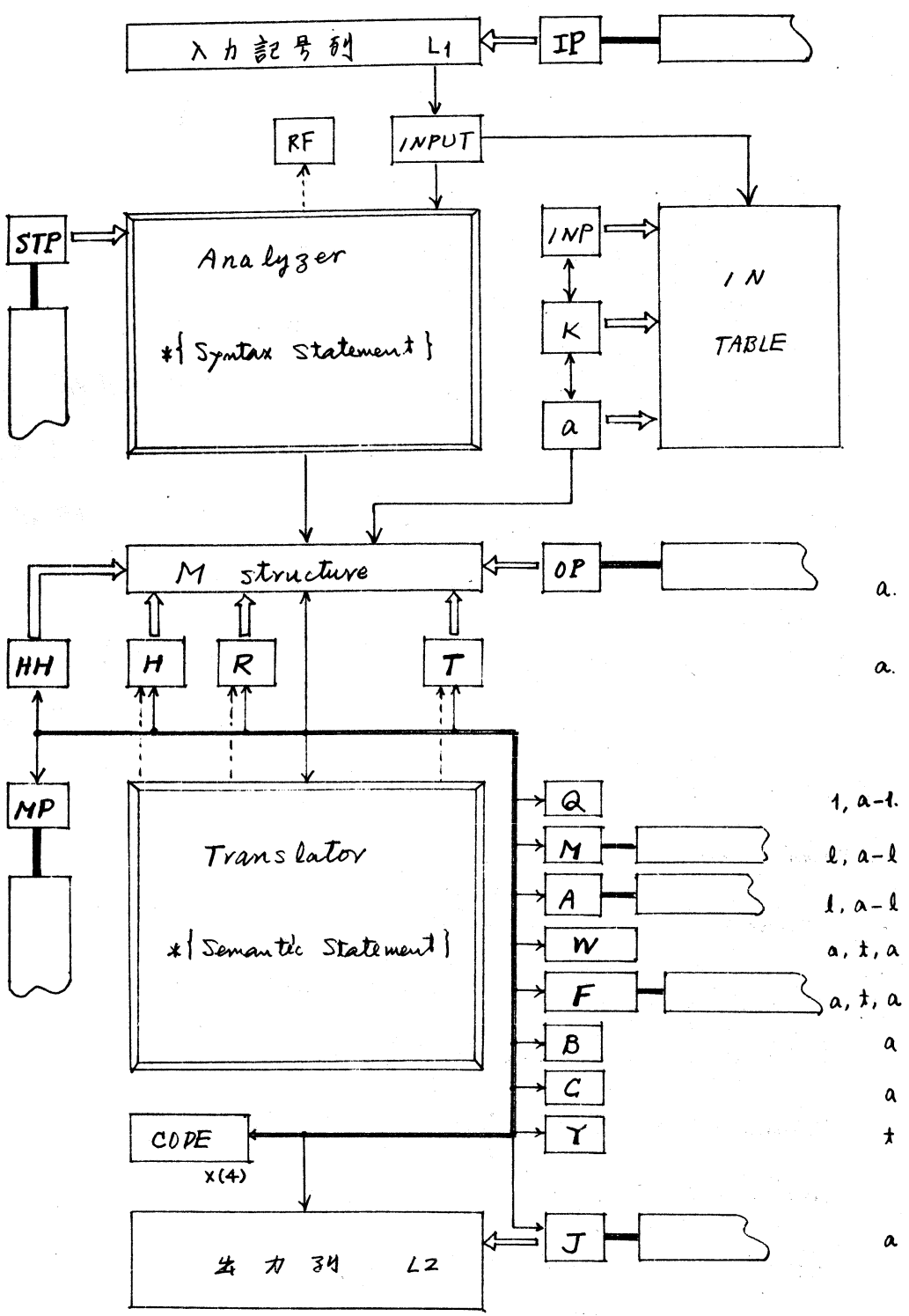


图 1 示

と次のようになる。

- i terminal symbol とする "delimiter" の確認.
- ii "identifier" "number" "string" などの basic item の編集と変換.
- iii nonterminal symbol とする syntactic unit の確認と之の応答.
- iv iii の手順に必要な情報の記憶と復元.
- v 分析手順の次の動作の指定 i.e. 参照規則の選択
- vi 分析結果の記録

COL による構造分析過程は規則の組  $P$  と分析手順が  $\rightarrow$  により定められて、一連の Syntax Statement で記述される。

$\langle \text{Syntax Statement} \rangle ::= \langle \text{Label} \rangle \langle \text{Read \& Test} \rangle ?$

$\langle \text{T. Action} \rangle \text{ \& } \langle \text{F. Action} \rangle$

$\langle \text{Label} \rangle ::= \text{SNAME} : | \phi$

$\langle \text{Read \& Test} \rangle ::= \langle \text{Read} \rangle \langle \text{Test} \rangle$

$\langle \text{Read} \rangle ::= ** | \langle \text{Read} \rangle ** | \phi$

$\langle \text{Test} \rangle ::= I | N | F | \text{"delimiter"} | (\text{SNAME})$

$\langle \text{T. Action} \rangle ::= \langle \text{Trans} \rangle | * \{ \langle \text{Action} \rangle, \} \langle \text{Trans} \rangle$

$\langle \text{Trans} \rangle ::= \text{TR} | \text{FR} | \# \text{SNAME} | \phi$

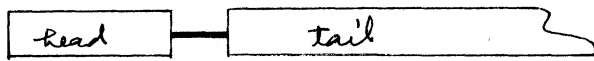
$\langle \text{Action} \rangle ::= \langle \text{Result} \rangle | \text{ON} | / | \langle \text{Error} \rangle$

$\langle \text{Result} \rangle ::= (\text{MNAME}) | @$

$\langle \text{Error} \rangle ::= E * \{ \text{letter} | \text{digit} \}$

$\langle \text{F. Action} \rangle ::= \langle \text{T. Action} \rangle | \langle \text{T. Action} \rangle, \text{AT}$

push down 記憶は



の構造を保持し、次の指令によって操作される。

RESERVE (E) : tail E push down ; head E → tail E ;

RESTORE (E) : head E ← tail E ; tail E pop up ;

REWRITE (E) : head E ← tail E ;

LOSE (E) : tail E pop up ;

ERASE (E) : tail E をおこなって消去する ;

#### 4.2 M structure

入力記号列の構造分析の結果、MNAME (M routine の名前) と basic item の内部表現から成る M structure が構成される。MNAME は入力記号列の文脈を表示すると共に、意味解釈過程で実行される M routine を指示してゐる。すなわち、M structure は入力記号列を標準型に表現したものであるため、意味解釈を容易にし、その手順を簡潔にするための媒介となる。

M structure は、入力記号列の文脈が確定する Incremental Unit (IU) 毎に構成され、意味解釈過程にかくられる。

#### 4.3 Semantic Statement

COL では Semantic Statement によって意味解釈過程すなわち入力言語 (たとえば ALGOL) の意味が表現される。これは才1回の register および記憶装置を持つ機械を起動させるも

1. 7. 3.

$\langle \text{Semantic Statement} \rangle ::= \langle \text{Uncond st} \rangle ; \mid \langle \text{Cond st} \rangle ;$   
 $\mid \text{MNAME} : \langle \text{Semantic Statement} \rangle$

$\langle \text{Uncond st} \rangle ::= \langle \text{Asst} \rangle \mid \langle \text{Object} \rangle \mid \langle \text{Mtrans} \rangle \mid \langle \text{Point} \rangle \mid$   
 $\langle \text{Pdstack} \rangle \mid \langle \text{Link} \rangle \mid \langle \text{Error} \rangle$

$\langle \text{Asst} \rangle ::= \langle \text{Left} \rangle = \langle \text{Right} \rangle$

$\langle \text{Cond st} \rangle ::= \& \langle \text{Egitem} \rangle = \langle \text{Egitem} \rangle ? \langle \text{Uncond st} \rangle$

$\langle \text{Object} \rangle ::= \$ \langle \text{Op} \rangle * \{ , \langle \text{Od} \rangle \}$

$\langle \text{Mtrans} \rangle ::= \# \text{MNAME} \mid \Phi \text{MNAME} \mid /$

$\langle \text{Point} \rangle ::= ( + \text{VP} ) \mid ( - \text{VP} )$

$\langle \text{Pdstack} \rangle ::= \{ \text{RESERVE} \mid \text{RESTORE} \mid \text{REWRITE} \mid \text{LOSE} \mid \text{ERASE} \} ( \text{Vpd} ) \mid$   
 $\text{SEARCH} ( \text{V} ) \mid \text{DELETE} ( \text{V} ) \mid \text{FIND}$

$\langle \text{Link} \rangle ::= \text{LINK} ( \text{V}, \text{V} )$

$\langle \text{Error} \rangle ::= E * \{ \text{letter} \mid \text{digit} \}$

#### 4. 4 Compile-Time

FACOM 230-10 FORTRAN	FACOM 230-10 ALGOL	Compiler I	Compiler II	Compiler III	Compiler N
1	0.8 ~ 0.9	1.5 ~ 2	2 ~ 3	4 ~ 6	3 ~ 4

Brooker :  $\left\{ \begin{array}{l} \text{space} \sim \text{handcoded version の } 1.7 \sim 2 \text{ 倍} \\ \text{compile time} \sim 4 \sim 5 \text{ 倍} \end{array} \right.$

Brooker R.A : Experience with the Compiler Compiler

Comp. Journal Vol 9 (1967)

時間のかかる理由

1. 構造分析と意味解釈を IU 毎に行っているため。

状態の表化

必要ページの取り出し

の回数が多くなる。

2. Syntax-directed analysis のため

push down の操作が多い (特に arithmetic expression)

入力記号の読み直しが多い

## § 5. 様々な問題

COL によって Compiler を記述・作成した結果、遭遇しない  
く→の問題点を、一般の Compiler Compiler の場合を含めて検討  
する。

5. 1 受け入れられる入力言語  $L_1$  とコンパイラ手法

Compiler 記述言語がすべてのアルゴリズムを記述表現出来る  
る場合には問題は無いが、一般にコンパイラ手順を簡潔にする  
るために Compiler 記述言語を定義した場合には、コンパイラ  
手法が何らかの形で規定され、それによつて受け入れられる  
(accepted) 入力言語  $L_1$  のクラスが制限されると考えられる。

a. COL で記述された Compiler  $C_{L_1 \rightarrow L_2}^{COL}$  ではない

i. 構造分析手順は LR(k) 文法に従う言語を受け入れる。

ii.  $C_{L_1 \rightarrow L_2}^{COL}$  の構造分析過程は pushdown automaton, 意味解釈

過程は stack automaton にモデル化して考えられるが、

"L を任意の CS-言語とした時、 $T(A) = L$  とする stack automaton A が存在する。"

"L を任意の CF-言語とした時、 $T(M) = L$  とする pushdown automaton M が存在する。"

よって、 $COL \rightarrow L1$  は CF-言語を受け入れたことが出来る。

b. 構造分析手順は Syntax-directed analysis に準じているため arithmetic expression のように対応の tree 構造の node が深くなるものは構造分析に時間がかかる。このためコンパイラ手法を operator precedence に基づく手法などに変更しなくては、COL のままでは少し困難である。

c. IU の選定により Semantic Routine (M-routine) の記述が変わってくる。たとえば、 $\langle \text{actual parameter} \rangle$  を IU に与えると  $\langle \text{expression} \rangle$  の途中で M structure が切れる可能性があるため、 $\langle \text{assignment statement} \rangle$  に関する M routine が少し変更される。

d. 記号の割り付け手法 (static or dynamic), side effect を考慮するかどうか、その他入力言語の意味論的性質により必要記述能力が変わってくる。

e. 添字付変数の着地計算と FOR ループ (DO ループ) の関係のように、コンパイラ結果の optimization を行う場合、之



## Compiler I

## ◦ 入力言語

$$\langle \text{BLOCK} \rangle ::= \underline{\text{begin}} \{ \langle \text{D} \rangle \} \langle \text{ST} \rangle^* \{ ; \langle \text{ST} \rangle \} \underline{\text{end}} \ \epsilon$$

$$\langle \text{D} \rangle ::= \underline{\text{real}} \ I^* \{ , I \} ;$$

$$\langle \text{ST} \rangle ::= I = \{ I = \} \langle \text{EXP} \rangle$$

$$\langle \text{EXP} \rangle ::= \{ - | \phi \} \langle \text{T} \rangle \{ + \langle \text{T} \rangle | - \langle \text{T} \rangle \}$$

$$\langle \text{T} \rangle ::= \langle \text{F} \rangle \{ \times \langle \text{F} \rangle \}$$

$$\langle \text{F} \rangle ::= \langle \text{P} \rangle \{ ! \langle \text{P} \rangle \}$$

$$\langle \text{P} \rangle ::= I \mid N \mid ( \langle \text{EXP} \rangle )$$

ただし  $I, N$  はそれぞれ identifier, number を表わす。

## ◦ M-structure

$$\langle p \rangle = [P] a \mid [N] a \mid \langle ae \rangle$$

$$\langle f \rangle = \langle p \rangle \{ \langle p \rangle [EXP] \}$$

$$\langle t \rangle = \langle f \rangle \{ \langle f \rangle [MUL] \}$$

$$\langle ae \rangle = \langle t \rangle \{ [NEG] | \phi \}^* \{ \langle t \rangle [ADD] | \langle t \rangle [SUB] \}$$

$$\langle st \rangle = [P] a [LEFT]^* \{ [P] a [LEFT] \} \langle ae \rangle [RITH] /$$

$$\langle d \rangle = [REAL] [I] a \{ [I] a \} [TD] /$$

$$\langle block \rangle = [HEAD]^* \{ \langle d \rangle [DH] \}$$

$$\langle st \rangle^* \{ [STE] / \langle st \rangle \} [END] /$$

ただし、/ は IU の切れ目を表わす

[ ] 中の名前は MNAME である。



## ◦ 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:	*"X"	?	& ON, TR
	(F)	? (MUL) #T1	& 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

## ◦ M-ROUTINES

```

HEAD:      ERASE(M, F, A, J); M1=1; M2=0;
           J=0800; A1=1; A2=1;
           F1=8100; F2=0; F3=0; B=0;
           RESERVE(M,A,F,J);
           $"STAT" M; /;
END:       LOSE( M ); RESTORE( M );
           RESTORE(J);      $"HALT" J; /;
STE:      REWRITE( M ) /;
REAL:     Y=1/;
I:        (+R); F1=(R); F2=Y;
           (+M2); F3=M; RESERVE(F); (+B)/;
TD:       Y=0/;
DH:       &(T)=(TD)? /;
           RESERVE(M);/;
P:        (+R); F1=(R);
           FIND; F=F(O2); (+H); (H)=F3; (+H); (H)=F2;/;
LEFT:     (T)=(AG); (+T);
           (T)=(H); (+T,-H); (T)=(H); (+T,-H);
           (T)=(RITH)/
AG:       (+R);(+R);(-H);  $"STOR" (H),(R);
           (+H)/;
RITH:     /;
DUMY:     /;
NEG:      (-H); &TMU;      $"FNEG" (H),Q;
           (H)=0; (+H)/;
ADD:     CODE=" ADD"; #OP;
SUB:     CODE=" SUB"; #OP;
MUL:     CODE=" MUL"; #OP;
EXP:     CODE=" EXP";
OP:      CODE=CODE+"F ";
           (-H); &TMB;      $CODE, (H-4),(H), 0;
           (-H); (H-2)=0/;
TMB:     Q=(H-4); &O1=1? #CHEK;
TMU:     Q=(H); &O1=1? #MSET; #TEMP;
CHEK:    Q2=Q2-1;
           & Q=(H) ? #TMU;
           Q2=(H-4);
MSET:    M=O2;/;
TEMP:    (+M2); O2=M; Q1=1 /;
N:       (+R); Q=(R);
           (+H); (H)= O2; (+H); (H)=Q1+1;/;
NUL:    (+H); (H)=1X00; (+H); (H)=1/;
ONE:    (+H); (H)=1X01; (+H); Y.7=1; (H)=Y/;

```

M ROUTIN END