記述言語
手法の改善
{ 変換、ブートストラップ
  構造
自己増殖

B1　コンパイラ作製自動化の諸方法

井 上 謙 蔵（富士通）

1. 緒

ソフトウェアの作製を自動化したいという欲求は，今にはじまったことではない．そもそも8進数や16進数などでプログラムを作ることはわずらわしくて，日常の仕事とはなり得ないから，EDSACが動きだした当初から，記号で表現した入力プログラムを機械語に翻訳する方法が考えられていた．しかしこの言語での1行は，機械語の1語に，大体対応している．

自動プログラミングという言語が，ひところ用いられたが，これはEDSAC的な記号言語やその後裔であるアセンブラなどに対応するものではなく，FORTRANクラスのものを指しているのであろう．しかし，それにしても，プログラムの内容を考えただけで，ひとりでにプログラムが出来てくるわけではないから，自動プログラミングなどという言葉よりは，自動コーディングのほうが適していよう．本小論の題目である自動化という単語も，その程度のことまでも含んでいるものと考えていただきたい．

数値計算という分野で，FORTRANやALGOLが輝やかしい業績をおさめたのであるから，OSの製作が，一つの応用分野として確立すると同時に，OSプログラム用言語が作られなければならなかった．しかし，その研究は非常におくれて，ここ2，3年の間に，主としてコンパイラ用に，その開発が試みられるようになった．

OSプログラム用言語の開発がおくれた理由としては，

1）OSの発達が急激で，関係者の間で，1つの巨大な応用分野としての意識が普遍化するひまがなかった，

2）OSに要求される特別な効率を，問題向き言語で実現する可能性が危ぶまれた，

3）OS製作集団の，特別な専門家意識による拒絶反応，

といったものが考えられる．

現在，このような言語の試作が盛んになってきたのは，いわゆるソフトウェア危機による圧迫と，特に日本的な特徴として1つのOSを各メーカーが共同して作製しなければならないプロジェクトの存在があげられる．しかし，それにしても主目標はコンパイラの記述に向けられ，OS全体をカバーする点に関しては，大部分慎重な態度がとられている．

そこで本小論もコンパイラ作製自動化ということで，コンパイラ・コンパイラの技術も含め，日本における開発を中心に，コンパイラおよびその周辺のソフトウェア作製自動化の技術の概観を試みたい．

Programming language
120　20 死んだ
　　35 あぶりつかわれぬ
　　50 特殊
　　15 つかわれる

2. コンパイラ作製自動化の2側面

アセンブラを用いてコンパイラを記述するにしても，できるだけ作り易く，かつ文書として役立つものにするために，徹底的にマクロを使用する立場がある．高級言語を使用するよりは，効率のよいコンパイラを作ることができるのはたしかであるし，保守用の文書としても，ある程度の改善が行なわれるが，コンパイラごとにアセンブラでプログラムを記述する労苦は変るものではない．しかし，この方向でのコンパイラ記述の可能性は，拡張可能言語の中に生かされているように思われるので，コンパイラ作り自動化の意図とは一寸ずれるけれども，のちに述べよう．

コンパイラ作り自動化の試みは記述言語の面での改善と，生産方法の改善との2方向の努力に分類して考えることができる．これらの試みは相補的な場合もあるが，ある場合には記述言語の改善だけで意味をもつことも，生産方法の改善だけで意味をもつこともある．前者は，たとえばアセンブラのかわりにPL／Iを用いた場合であるし，後者はのちに述べるブート・ストラッピング法である．コンパイラ・コンパイラは，記述面のある変更の上になり立つ生産方法である．

3. 記述言語

記述言語は，それによって記述されるべきコンパイラの論理からは，ある程度独立である．それゆえ，他のソフトウェア開発にも使用しうる可能性がある．

一般に，コンパイラでは，数値計算の場合とちがい，ビットや字の処理が多く，判断の種類が多く，加減剰除は少ない．これはコンパイラの特徴であるのみならず，OS全般に通じていわれることがらである．それゆえ，コンパイラ記述に適当した言語は，OS全般に使用して，良好な結果をうる見とおしが強い．

記述言語として，現在2つのレベルが考えられている．その1つはハードウェアに依存するものであり，他はハードウェアから独立なものである．

3.1 ハードウェアに依存する記述言語

これは，問題向き言語の書き易さ，読み易さと，アセンブラの柔軟性を，何とか両立させようとする試みで，その最初のものはPL 360[1]であった．この言語の主要な特徴は，

1) 繰返し文（for，do）や条件文（if）を最も単純で，高能率の機械語の組合せに対してのみ設ける．
2) 加減剰除の順位は設けない．
3) データの割付けが，機械語との対応関係が明らかなようになされる．
4) アセンブリ語が挿入できる．

等である．この程度でもアセンブラよりは，はるかに見易く，書き易い．われわれの実験では，同一のプログラムを書くのに，アセンブラの場合の約1／2の行数ですますことができる．[2] このうち節約されるのは，主としてデータ構造の宣言部分であるから，プログラム部分にはアセ

今命数PL / 今命数Asc = 1/2

ンブラと殆んど同質の柔軟性が残される．これはまたデータの配置変えにより，長命令を短命令へ変更する最適化を含んでいるが，その結果は手書きの場合にくらべ，やや良好な結果を得ている．今回発生の

変形 PL 360 の文法とその解釈

浅井，藤村，稲見

コンパイラ記述言語の一つの試み

清水，中川，石田

の2論文が，同種のものである．

この種のものは，アセンブラと同程度の柔軟性をもつが，機種間での両立性は望めない．しかし，このような言語でソフトウェアの開発を大々的に行なっても，その品質が低下する心配はない上，プログラムの変更も比較的容易であるので，きわめて高い実用性をもつ．

3.2 ハードウェアから独立な記述言語

前項の言語の欠陥は，機種間での両立性が保ってないから，新機種の開発にあたって，あらかじめソフトウェアを開発する手段としては，必らずしも適当でないことであろう．もし，ハードウェアに依存しない記述言語があって，その処理プログラムが他の機種の上に存在していれば，新機種のソフトウェアの開発は，大変容易となる．実際に，大型プロジェクトの計算機やDIPSのソフトウェア開発に，ハードウェアから独立な言語を使用する主目的は，この点に依存している．

さて，このような記述言語の特徴として要求されるものは，何であろうか．1つの言語の仕様としては，それを使用する分野で最も普遍的なデータの形式や，その処理の方法を選んで，それらを明瞭に簡潔に記述できるようにする必要がある．数値解析であれば，長い伝統があるから，比較的容易にFORTRANやALGOLの形式が作られたが，コンパイラ書きに対してはそれに適切な言語の設計は遅れ，まず汎用言語であるPL/Iの流用ということで始まった．

実際にシステム書きのために，すでに

SYSL　（コンパイラ，通研）
PL/I *　（コンパイラ，日本ソフト）
BPL　（コンパイラ，日電）
PLD　（PL/I，　日本情発センタ）
PL/IW　（OS，　日立）
TWT　（PL/I，　東芝）

が作られ，それぞれ括弧内に記述されたシステム・プログラムの作製に使用されているが，これらはすべてPL/Iのサブセット，又はそれにハードウェアに依存する若干の機能を追加したものである．

これらの言語の開発については，一昨年の夏のシンポジウムで報告され，特に11回プログラム・シンポジウム（1970）では，PL/IWによるOS作製の総括的な報告がなされた．そ

の結果，OSの作製速度の点ではすぐれた威力を発揮しているが，OSの効率については良好な結果を得たとは云いがたい．そのさい述べられたように，最適化に充分な注意が払われなかったことも1つの原因かもしれない．今回発表の

「システム記述言語の使用経験と記述言語に対する要求」　伊藤

は，PLDの経験である．

汎用の言語を，システム書きに流用するのは，多くの場合，その言語の処理プログラムが何らかの形で使用できる点で有利である．しかし，汎用の言語であるゆえに，そのサブセットをとっても，システム書き専用の目的からは，表現が不適切であったり，記述上不便なところも生じてくる．それゆえ，専用言語の設計が次の課題となる．

11回プログラム・シンポジウムでは

「ハードウェアに独立なシステム記述言語」　井上，藤崎

があるが，今回発表の論文では，

「システム・プログラム記述用言語D32，D34の試作」　高橋，島内，和田，筧，四条

がある．また

「機械独立なコンパイラの一構成法」　野下，雨宮

も，機械独立な記述言語を設定している．

「COBOLコンパイラ記述言語」　伊藤，増田

も，COBOLという特別な言語のコンパイラのみを対象としているが，やはり同じ目的を目指しているものであろう．また

「MINIMUM INSTRUCTION SETによるSYSTEM PROGRAM記述の試みについて」　長谷部，田場，国井，高橋

も，同様な意図をいだいているもののようである．

これらの言語を設定する効果は，作製自動化の観点から眺めるばかりでなく，ハードウェアの設計への反作用も考慮に入れなければならない．すなわち，もしある種の手順が，これらの言語の中で標準化され，それが計算機による情報処理の手順として，一般性が認められるならば，ハードウェアとしては，その手順を高速化するように設計すべきである．これは，ハードウェアへの設計へのソフトウェア側からの直接的な反作用である．

この意味で，OSを記述することを目的とした言語が，多くの人々によって，いろいろな観点から設計されることが望ましい．

4. 生産方法の改善

生産方法を改善する試みとして，以前より使用されていたのは，既製のコンパイラの変換によるものと，ブート・ストラッピングである．コンパイラ・コンパイラはIronsの「syntax directed compiling」の技術が開発されて以来のことであるから，[3] これも起原は古いけれども，盛んになったのは最近のことである．

4.1 変換法

これは，首勝（三菱）によって実施され[4]，多くの成果をあげたが，汎用性の点で問題があるので，その後あまり使われないようである．

つぎに簡単に，その方法を示そう．

まず process という処理を行なうプログラムPが，言語Lで書かれていることを，

$$P(\mathrm{process},\ L)$$

とあらわす．たとえば，ALGOLを機械語M_0へ翻訳する処理プログラムP_0が，機械語M_0で書かれていれば，

$$P_0(\mathrm{ALGOL} \rightarrow M_0,\ M_0)$$

である．これがあるデータ，すなわちALGOLで書かれた他のプログラムに働らいて，それを機械語のプログラムに翻訳する過程を，

$$P(\mathrm{process},\ \mathrm{ALGOL}) * P_0(\mathrm{ALGOL} \rightarrow M_0,\ M_0) \rightarrow P(\mathrm{process},\ M_0)$$

とあらわそう。別の計算機の機械語M_1で書かれたALGOLの処理プログラムP_1が必要であるときは，

$$P_1(\mathrm{ALGOL} \rightarrow M_1,\ \mathrm{ALGOL}) * P_0(\mathrm{ALGOL} \rightarrow M_0,\ M_0) \rightarrow P_1(\mathrm{ALGOL} \rightarrow M_1,\ M_0)$$

$$P_1(\mathrm{ALGOL} \rightarrow M_1,\ \mathrm{ALGOL}) * P_1(\mathrm{ALGOL} \rightarrow M_1,\ M_0) \rightarrow P_1(\mathrm{ALGOL} \rightarrow M_1,\ M_1)$$

とすればよい．プログラムP_1はALGOLで書かれるから，項3.2で述べたと同じように，比較的容易に作ることができる．

この方法は，ALGOLがシステム記述に不適切ならば，あまり質のよいコンパイラは得られない．また，すでにALGOLコンパイラが存在していることが必要である．

4.2 ブート・ストラッピング

種コンパイラを作る方法である．すなわち言語Lのコンパイラを作るときは，その言語の最小のサブセットL_0を定め，L_0の処理プログラムP_0を，機械語（アセンブラ）で記述する．ついでL_0によってLのコンパイラPを記述し，P_0で処理をする．

$$P(L \rightarrow M_0,\ L_0) * P_0(L_0 \rightarrow M_0,\ M_0) \rightarrow P(L \rightarrow M_0,\ M_0)$$

P_0の作製は，L_0が最小のサブセットであるから，比較的容易である．またPの記述は，L_0を用いるから容易である．この方法は，NELIAC[5]やWISP[6]の開発に使用され著名になった．

しかしながら，生産されたコンパイラの質はその言語の仕様に依存するから，効率のよいコンパイラを求める立場からは，おのずと使用範囲が限定される．

われわれの目的からは，システム記述用言語に使われるならば，威力を発揮するであろう．実際にBPLでは，数十回のブート・ストラッピングで，BPLとそのプロセッサの改善を行なっている．

4.3 コンパイラ・コンパイラ

前々者と異なり，汎用の方法である．

この方法はIronsの直構文解析法（syntax directed analyzing method）に起原をもっている．以下に，直構文解析法と，それに基づくコンパイラ・コンパイラの構造について，簡単に説明したい．

ALGOL 60 の文法は，基本記号の並べ方の規則と，その記号列に対する解釈法からなりたっている．前者は構文で，後者は意味であり，構文はB.N.Fであらわされている．B.N.F.はchomskyのCFGと同類である．

コンパイラは，原プログラム（文）を読み，それが構文の規則にかなった構造をもつかどうかをしらべ，かなっていれば意味にしたがって翻訳を行なう．前者は構文解析で，後者は意味づけである．

直構文解析法というのは，B.N.F.(又は，その変形）と文を，直接比較することによって，文が正しい構造をもっているかどうかをしらべる．例えば，つぎの例を考える．

例1.

$$G=(V_N, V_T, P, S)$$

$$V_N=\{S, E, T, F\}$$

$$V_T=\{a, m, i, c_1, c_2, \vdash, \dashv\}$$

$$P=\{S\to\vdash E\dashv,\ E\to E\,a\,T,\ E\to T,\quad (4.1)$$

$$T\to T\,m\,F,\ T\to F,$$

$$F\to i,\ F\to c_1\,E\,c_2\}$$

ただし ::= のかわりに → を用い，| は使用しない．このとき1つの規則を生成規則（production）と呼ぶことにする．超変数として1字の大文字，端記号（terminal symbol）として1字の小文字及び ⊢ と ⊣ を使用する．

直構文解析法には下向き（top-down，top to bottom）法と，上向き（bottom-up，bottom to top）法といわれるものがある．ここでは，下向き法について説明する．

例1の記号Sは，この文法の目的であるところの概念である．これを出発記号というが，プログラム用言語であれば，これは＜program＞となるべきところである．例1では，式（に両端を示す記号 ⊢ と ⊣ をそえたもの）になっている．

さてSから出発して，超変数を，それを定義する生成規則の右辺で置換する．この置換をくりかえせば，やがて端記号のみからなる文を作りだすことができる．たとえば

$$\begin{aligned}S &\Rightarrow \vdash E\dashv \Rightarrow \vdash T\dashv \Rightarrow \vdash T\,m\,F\dashv \Rightarrow \vdash F\,m\,F\dashv\\ &\Rightarrow \vdash i\,m\,F\dashv \Rightarrow \vdash i\,m\,c_1\,E\,c_2\dashv \Rightarrow \vdash i\,m\,c_1\,E\,a\,T\,c_2\dashv\\ &\Rightarrow \vdash i\,m\,c_1\,T\,a\,T\,c_2\dashv \Rightarrow \vdash i\,m\,c_1\,F\,a\,T\,c_2\dashv\\ &\Rightarrow \vdash i\,m\,c_1\,i\,a\,T\,c_2\dashv \Rightarrow \vdash i\,m\,c_1\,i\,a\,F\,c_2\dashv\\ &\Rightarrow \vdash i\,m\,c_1\,i\,a\,i\,c_2\dashv \qquad (4.2)\end{aligned}$$

である．超変数は，ある記号の列がそれだけで一まとまりとなって，一つの文法上の概念に対応することを示すためにある．そこで，上記のようにSから出発して，端記号のみからなる文を作り出せば，その文はSという概念に従う正しい構造をもった文であることになる．

（4.2）を

$$S \overset{*}{\Rightarrow} \vdash i\,m\,c_1\,i\,a\,i\,c_2 \dashv = t \qquad (4.3)$$

とあらわす．通常1つの変数を定義する生成規則は複数個あるから，変数の置換にはその数だけの任意性がある．また規則の中には

$$E \rightarrow E\,a\,T$$

のような回帰的な定義があるから，置換の方法は無数に存在する．このような置換によって出来た文の総体を，文法Gから導びかれる言語といい

$$L(G) = \{\, t \mid S \overset{*}{\Rightarrow} t \,\} \qquad (4.4)$$

とあらわす．

与えられた文 t_0 が，

$$t_0 \in L(G)$$

であることをしらべるための一つの方法は，実際に（4.2）の導出を行なって，その結果導びかれた文 $t = t_0$ が一致しているかどうかを見ることである．その際，t にいたる途中の文形（sentential form）に，生成規則を適用するさいには，その中の最左端の変数を置換するようにすれば，文形の左端に端記号列が出来てくるから，文 t にまで到達しないうちに，t_0 とのくいちがいが発見される場合が多い．

t_0 と t がちがっていれば，別の導出方法について，試みなければならない．t_0 の長さの文を作り出すことごとくの導出に対して，

$$t \neq t_0$$

であれば，t_0 はL（G）に属していないことがわかる．

上記の方法は下向き解析法である．実際には解析の速度をあげるために，いろいろな工夫がされる．生成規則の表現法は変更することも，速度をあげるのに役立つ場合がある．

つぎに上向き法について説明しよう．まず，

$$S \overset{*}{\Rightarrow} \beta A \gamma \overset{*}{\Rightarrow} \beta \alpha \gamma$$

であるとき，α を句（phrase）といい，句の中で特に

$$S \overset{*}{\Rightarrow} u A \gamma \Rightarrow u \alpha \gamma$$

である α を把手（handle）という．ただし，α，β，γ は $V = V_N \cup V_T$ の要素からなる任意の列

$$\alpha, \beta, \gamma \in V^*$$

で，u は V_T の要素からなる任意の列

$$u \in V_T^*$$

である．上向き法は，与えられた文 t_0 から出発して，把手を発見し，それを超変数に置換する

操作をくり返すことによって，t_0 全体がSに置換されるかどうかをしらべる．生成規則が適用可能にみえる最左端の部分列は，一般に複数個存在するために，どれが把手であるかは，生成規則の適用の段階では不明である．t_0 がSに還元されてのち，はじめて確認される．結局，上向き法でも，一般には数多の試行を伴なう．

このようにして，下向き法でも上向き法でも，直構文解析では，与えられた文の長さを n とするとき，t_0 がL(G)に属するか，しかないかを確認するために

$$n^2 \log n$$

に比例する数の手順を必要とする．[7] これでは構文解析の時間が，手書きのコンパイラで通常とられている方法にくらべ長すぎるので，文法の表現法にいろいろな制限を加えて，解析時間を n に比例するところまで下げる方法が考えられる．

たとえば順位文法（precedence grammar）では，[8]

$$S \overset{*}{\Rightarrow} \beta A \gamma \overset{*}{\Rightarrow} \beta \alpha \gamma$$

であるとき

$$T_1(\beta) \lessdot H_1(\alpha), \quad T_1(\alpha) \gtrdot H_1(\gamma)$$

で，α の隣接する記号B，C間には

$$B \doteq C$$

であるように順位関係を，生成規則に従って定めておく．この関係を表にしておけば，容易に把手を発見することができる．順位規則のきめ方にはいろいろな方法があり，それによって順位文法，演算子順位文法，[9] 対称順位文法[10] 等々，いろいろな変種が考えられる．

もう1つは，ある記号の把手の右端であるかどうか，したがってその左に把手が存在するかどうかを決定するために，その記号の左 m 個，右 n 個の記号を眺める方法である．この方法は（m，n）限定文脈文法（bounded context grammar）といわれ，[11]（m，n）の文脈によって一義的に把手の存在を決定できる範囲の文法のみを取扱う．この方法にもいくつかの変種があるが，とくにその左側の制限を外して，左側の全記号列から与えられる情報と，右側の k 個の記号を眺めて手順を一義的に決定することの出来る文法をLR(k)といっている．[12] またこの変種でFloyd's productionと称するものもある．[13]

またCFGの規則を，恰も正則文法のそれの如く見なして，解析を進めることによって，速度をあげようとする方法もある．この方法で解析できる範囲の文法を正則文脈独立（RCF）であるという．[14]

いずれの方法をとるにしても，構文解析プログラムを，ある標準的な形に統一することができる．たとえば順位文法では，順位表と生成規則を並べた構文表を作り，これらの表を使用して，構文解析を行なう．すなわち，文を左より右に走査しながら，順位表を利用して把手を発見する．発見された把手に右辺の一致する生成規則を構文表より取り出して，把手を，その左辺の超変数で置換する．このようにすれば，個々の言語に依存するのは順位表と構文表であって，手続き部分は言語には依存しない．そこで，1つの言語Xの生成規則の集合を読んで，そ

れより順位表と構文表を作り出し，言語から独立な手続き部分と結合して出力とする1つのコンパイラを作れば，順位文法で表現できる範囲の言語の構文解析プログラムを，構文の規則より自動的に生産することができる．これはコンパイラ・コンパイラの構文解析プログラム発生部分である．

コンパイラ・コンパイラの意味づけプログラム発生部分も，上記のように形式的な処理の方法ができるならば，コンパイラの生産は非常に容易となる．そのためには，言語の意味を形式的に単純に記述する方法が定められなければならない．この方面でも，ある程度の努力がなされている．たとえば Feldman の FSL[15] や TREE-META[16] の意味の規則はそれである．しかし意味づけの記述は，構文の場合にくらべ，実際に目的プログラムを発見することに関係しているので，単純な形式に統一することは中々困難である．まず決定的と思われる形式は発見されていない．

それ故コンパイラ・コンパイラの分野では，構文解析に関しては，

「出来るだけ簡潔で，自由度の多い文法の記述法と，出来るだけ小型で速度の速い構文解析法を見出すこと」，

意味づけに関しては，

「簡潔で自由度に富んだ文法の記述法を見出すこと」

が当面の課題である．

今回発生のものでは

「コンパイラ自動作成の1実験」　下村，藤野

がコンパイラ・コンパイラ構文と意味づけの両面にわたって新らしいコンパイラ発生の方法を示し，

「PL／1WによるTSS用コンパイラ・コンパイラの作成」　二村，西野，吉村

が，TSS用VITALの作製結果を報告する．また

「右順位文法に対する構文解析プログラムの発生」　井上

は，順位文法の1変数の問題を取扱う．

「Compiler Generatorへの入力データ」　渡辺

も，コンパイラ・コンパイラの問題に関係をもつ論文であろうが，内容を検討する機会をもたなかったので明白でない．

さて，プログラム用言語の文法は，構文と意味づけの規則に明確に分離できるものではない．たとえば，名前（identifier）は，1つのブロック又はプログラムの中で，同じ名前が同一の量をあらわすものと定められている．このような約束は，CFGの表現の範囲をこえるから，通常意味づけで述べられる．CFGで構文の規則を表現するのは，それで表現できる部分が多いから，そうするのであって，もし文脈依存的要素が強くなれば，それにふさわしい表現法と解析手段を考えなければならない．ALGOL 68やALGOL Nはその傾向を示している．コンパイラ・コンパイラの側からは，これは次の課題である．

4.4 拡張可能言語

コンパイラ・コンパイラは，もしその目標に到達できるならば，比較的に容易に自家用言語のコンパイラを生産できる方法となるであろう．

自家用言語のコンパイラを簡易に生産するためのもう1つの方法は，拡張可能言語である．これは，文法を記述するための超言語と，それによって記述される言語を同一レベルのものとして合体してしまおうとする考えに基づく．たとえば日本語で，日本語の新単語を定義して，日本語の記述能力を広げていくようなものである．

このような考えは，古くはアセンブラのマクロにその源をもっている．マクロは，プログラマが，自分のプログラムの中で，1つの名前が，任意の命令列を代表するように定義するための手段である．このような手段は，Burroughs の Extended ALGOL[17] や AEDO[18] に，プログラマが自分のプログラムの中で，文法的に合法的な記号列に，任意に名前をつけ，同じプログラムの中で任意に引用できる手段として採用されている．

上記のマクロをテキスト・マクロというが，これに対して，新らしい文法を追加できるマクロがシンタックス・マクロである．例えば ALGOL 68 では，本来の ALGOL 68 中には存在しないデータ構造と，それに対する演算子を，1つのプログラムの中で定義し，使用することができる．

拡張可能言語は，この方向で，プログラマが自由に言語機能を拡張できる種類のものであるから，コンパイラ・コンパイラよりも，プログラミングの柔軟性という点では優れているであろう．この方向の論文として，

「問題向き言語に対する意味論的メタ言語とそのコンパイラ」　渡辺

「増殖型言語 SELF について」　中田，浜田，霜田，野木

が出されている．

．あとがき

以上で，今回発表論文の位置づけをする心算で，コンパイラ自動作製の分野における種々の手段を一覧し分類してみたが，抜けた部分も多いであろうし，詳細な解説を行なうことはできなかった．この方面について興味ある方々は，参考文献 19) を参照して戴きたい．

参考文献

1) N. Wirth, "PL360, A Programming Language for the 360 Computers", J. ACM 15, No.1, p.37 (Jan. 1968).
2) 後藤真美，その他，"システム製造用言語 SL 45"，情報処理学会 第11大会予稿集
3) E. T. Irons, "A Syntax Directed Compiler for ALGOL 60", C. ACM 4, No.1, p.51 (Jan. 1963).
4) 首藤勝　その他，"計算機を用いたコンパイラ作成自動化の実験"，第7回プログラミン

グ・シンポジウム報告集 C−74, 1966.

5) M.H.Halstead, "Machine Independent Computer Programming", Spartan 1962.

6) M.V.Wilkes, "An Experiment with a Selfcompiling Compiler for a Simple List Processing Language", Annual Review in Automatic Programming 4, p.1, Pergamon 1964.

7) T.Kasami and T.Torii, "A Syntax-Analysis Procedure for Unambiguous Context Free Grammars", J.ACM 16, No.3, p.423 (July 1969).

8) N.Wirth and H.Weber, "EULER-a Generalization of ALGOL, and its Formal Refinition : Part I", C.ACM 9, No.1, p.13 (Jan. 1966).

9) R.W.Floyd, "Syntax Analysis and Operator Precedence", J.ACM 7, No.3, p.316 (July 1963).

10) A.Colmerauer, "Total Precedence Relations", J.ACM 17, No.1, p.14 (Oct. 1968).

11) R.W.Floyd, "Bounded Context Syntax-Analysis", C.ACM 7, No.2, p.62 (Feb. 1964).
E.T.Irons, "Structural Connections in Formal Languages", C.ACM 7, No.2, p.67 (Feb. 1964).

12) D.E.Knuth, "On the Translation of Languages from Left to Right", Inf.Contr. 8, p.607 (1965).
A.J.Korenjak, "A Procfical Method for Constructing LR (k) Processors", C.ACM 12, No.11, p.613 (Nov. 1969).

13) R.W.Floyd, "A Descriptive Language for Symbol Manipulation", J.ACM 8, No.4, p.579 (Oct. 1961).

14) V. Tixier, "Recwsive Functions of Regular Expressions in Language Analysis", Tech. Rpt. CS 58, Computer Science Dept., Stanford U., Stanford, Calif. (March 1967).

15) J.A.Feldman, "A Formal Semantics for Computer Languages and Its Application in a Compiler-Compiler", C.ACM 9, No.1, p.3 (Jan. 1966).

16) D.I.Andrews and J.F.Rulifron, "TREEMETA, A Meta Compiler System for the SDS 940 (working draft)", Stanford Research Institute, Mnlo Park, Calif. 1967.

17) "Burroughs B5500 Information Systems Extended ALGOL Language Mannual", Burroughs Corporation, 1966.

18) D.T.Ross, "AED bibliography", Mem. MAC-M-278-2, Project MAC, MIT Cambridge, Mass., Sept. 1966.

19) J.Feldman and D.Gries, "Translator Writing Systems", C.ACM 11, No. 2, p.77 (Feb. 1968).