ソフトウェアの自動作成*

井　上　謙　蔵**

1. 自動化の意義

近年，計算機のソフトウェアは大型化して，その結果，利用者に対するソフトウェアの供給が，機械の引き渡し時点より，実質的にかなり遅延することが，しばしば起こる．

ソフトウェアの大型化は，利用者の大衆化，計算機の大型化，高速化に基づくものである．すなわち，前者は問題向き言語の存在を必然のものとしたし，後者はオペレーティング・システムを生みだした．計算機の用途の拡大につれて，言語の種類はさらにひろがり，オンラインや時分割的用法にオペレーティング・システムの構造は複雑化する一方である．

ソフトウェアの製作遅延は，プログラマの補充の困難さにもよるけれども，なによりもソフトウェアの理論と作成の方法論の展開が，大型化の速度に追いつかないことに最大の原因がある．

まず，方法についていえば，われわれは分業の手段を欠いているといってよい．職人が工場に集められ，一種の分業が行なわれているけれども，それは分業と呼ぶにはあまりに素朴である．アセンブラ言語で数十万文に及ぶシステムは，通常，数十人以上のプログラマに分けられて作製されるが，この分担の枠組は任意性の強い自然言語による仕様書だけである．この様式と，現実のプログラムの様式，アセンブラ言語との大きなへだたりから，プログラムの内部構造を明確に規定することが困難である．そこで，内部構造は各製作者の恣意にまかされざるを得ないし，そのことが各部分の不整合をひきおこす結果となるし，また，不整合の原因追求を困難にもしている．このことから，部分から全体への組立作業は，個々のプログラムの修正をひきをこす長い試行錯誤の過程となる．いわば，われわれは適切な設計図的手段をもたないため，部分の作製と組立てを，独立的に取り扱うことができないのである（それが完全な意味で可能であるかどうかは別として）．

本来分業とは，部分の作製を単純化するものでなければならない．アセンブラを用いるプログラミングは，コーディングの錯雑さのゆえに，プログラム本来の論理を背景へおしやり，事前の論理による誤りの予知を困難にする．それゆえ，プログラムの誤りは，その効果に基づいて発見されざるを得ない．その誤りは，個個のプログラムの試験によって発見される場合もあるが，組み立てられた後に，全体的な効果として発見される場合も多い．この場合，アセンブラ言語のゆえに，他人の領域に立ち入って点検することの困難さが，原因の発見の遅延につながる．

プログラムは，論理とコード以外に属性をもたないのであるから，上記の事情は，われわれが分業と協業の手段を，本質的に欠いていることを意味する．

次は理論である．たとえば，コンパイラをとってみよう．一つの言語のコンパイラは，論理的には，個々の計算機からは独立な，せいぜい数種の翻訳手段を用いるにもかかわらず，実際には，個々の機械に適応した技巧が，プログラムのすみずみに浸みとおっている．その結果，新しい機械には，全く新しい構想のもとに，設計のしなおしをしなければならない．しかし後に示すように，標準化された翻訳方法をとることによって，計算機ごとに異なる部分を分離して扱うことも可能なのである．これは，プログラム用言語の統一的な性格づけを行なう理論の結果である．このようなプログラム，あるいはプログラミングの理論は，コンパイラを除いてはなはだ未発達である．コンパイラにしても，まだ実用的な段階には達していない．それゆえわれわれは，ソフトウェアの構造やプログラミングについて，個々の機械の性格やプログラマの職人的な技術に，強く依存せざるを得ない段階にある．

ここ数年，PL/I のような高次言語による，システム・プログラム作りの報告[1]が，ぽつぽつ雑誌にみえている．このような言語で，システム・プログラムを書くことは，数値解析用のプログラムを ALGOL や FORTRAN を使って書くことの，直接の延長として，

---

* The Automation of Software writing, by Kenzo Inoue (Software Division, Fujitsu Ltd.) 昭和44年4月24日開催，第7回通常総会特別講演

** 富士通株式会社ソフトウェア技術部

早くから問題とされていたけれども，実用的な観点から実施されなかったし，現在でも実験的作業の段階である．それは，この領域のプログラマは，まさに専門家の集団であって，アセンブラの使用を強制しうることと，システム・プログラムの効率に大きく作用し，計算センターの運営に決定的な影響を与えるからである．さらに，システム・プログラムの行動の論理を効果的に反映したプログラム用言語が存在しないためである．

しかし，システムのための適切なプログラム用言語の設定は，上に述べた第一の原因に対処する手段として重要である．すなわち，これを記述言語として用いるならば，各部分プログラムに対する条件を正確に，かつ簡明に記述することができ，これは厳密な設計図的役割を果たす．これは単なる記述用言語として設定されたものであってはならない．プログラマが，その恣意によって，部分的に表現方法を変えうるものであってはならない．その変更がコンパイラによって誤ったプログラムの生産に導びくところの，プログラマに対し内面的強制力をもつプログラム用言語でなければならない．

ソフトウェアについて論ずる際に，ぜひとも述べておかなければならない面がある．現在の計算機は，オペレーティング・システムを経由しないでは，一般使用者は全く利用できない．だから計算機の性能というものは，まさにオペレーティング・システムの性能である．金物の額面の性能で計算機を評価することは，全くのまやかしである．たとえば，命令系が不備である結果として，金物の速度が早いにかかわらず，プログラムの速度が意外に遅いということを，しばしば経験する．それゆえ，金物の設計にはオペレーティング・システムの立場が強く反映しなければならない．しかし，コーディングが個々のプログラマにまかされている段階では，コーディング技術の統一性がないために，この反映は長いサイクルをとおって，じぐざぐに進行せざるを得ない．ソフトウェアの自動作成が行なわれるようになれば，コーディングの統一化の結果として，この反映は，より直線的な形をとるであろう．あるいは，この統一化されたコーディング技術のもとに，最大の効率を発揮するように，計算機の命令体系を変革することが可能となる．

ソフトウェアの自動作成に関しては

システム・インテグレーション

システム・ジェネレーション

誤り処理

その他

モジュールの自動追加

とかいろいろ議論すべきことがある．しかし，ここでは先に述べた二つの問題に対処する方法の一つの典型としてコンパイラ・コンパイラに焦点を合わせよう．

2. コンパイラ作成自動化の古典的方法

コンパイラを自動的に作成する場合，すでにコンパイラが存在する計算機を用いて，他機種のコンパイラを作り出す場合[2]と，目的の計算機そのものを用いて作成する場合[3]とが考えられる．前者は，コンパイラを記述する言語を，ある機種から他の機種のものに変換する方法に基づいている．後者は，いわゆる，ブートストラッピングの方法に従う．

2.1 変換による方法

コンパイラは，ある言語 S を，他の言語 O に翻訳する．このコンパイラを記述している言語を D として，コンパイラを記号的に

$$C_{S,O,D} \tag{1}$$

と表わそう．O は通常，特定の機械語 M で，D も同じ機械語 M である．そうすると，あるコンパイラを他の機械上にうつすということは

$$C_{S,M:M} \tag{2}$$

を用いて，

$$C_{S,M':M'} \tag{3}$$

を作り出すことである．そのためには，S を M' に翻訳するコンパイラを S で書いて，コンパイラ (1) で翻訳する．その結果

$$C_{S,M':M} \tag{4}$$

ができる．すなわち，次の変換がおこる．

$$C_{S,M':S} \rightarrow C_{S,M':M}$$

$C_{S,M':M}$ は計算機 M 上で働くから，これを用いて，$C_{S,M':S}$ をもう1回翻訳すれば

$$C_{S,M':S} \rightarrow C_{S,M':M'}$$

がおこり，目的は達成される．変換のプロセスをもう1回書きなおすと

$$\left.\begin{aligned} C_{S,M':S} \ast C_{S,M:M} &\rightarrow C_{S,M':M} \\ C_{S,M':S} \ast C_{S,M':M} &\rightarrow C_{S,M':M'} \end{aligned}\right\} \tag{5}$$

である．＊はその右のコンパイラが，左のプログラムに働きかけて翻訳をする操作を表わす．

この方法は，次のように拡大して何種類ものコンパイラを生産するにも用いることができる．

$$\left.\begin{aligned} C_{S'M':S} \ast C_{S,M:M} &\rightarrow C_{S',M':M} \\ C_{S',M':S'} \ast C_{S',M':M} &\rightarrow C_{S',M',M'} \end{aligned}\right\} \tag{6}$$

または

$$\left.\begin{array}{l} C_{S,M':S} * C_{S,M:M} \rightarrow C_{S,M':M} \\ C_{S',M':S} * C_{S,M':M} \rightarrow C_{S',M':M'} \end{array}\right\} \quad (6')$$

ただし，(6) では S' を M' に翻訳するコンパイラを S と S' を用いて2回書かなければならない．(6′) では目的に達するためには，S を M' に翻訳するコンパイラが M 上に作られることが前提となっている．

2.2 ブートストラッピング

ほかの計算機上に働く，適当なコンパイラを利用できないときは，まず，目的とするコンパイラの言語のサブセット S_0 のコンパイラを，アセンブラで作らなければならない．これを

$$C_{S_0,M:M}$$

としよう．S_0 としては，S 全体を記述しうる最小限のものを選ぶ，これができれば

$$C_{S,M:S_0} * C_{S_0,M:M} \rightarrow C_{S,M:M} \quad (7)$$

によって，目的のコンパイラが得られる．この操作は

$$S_0 \subset S_1 \subset S_2 \cdots\cdots \subset S \quad (8)$$

なる言語列によって，何段階かに分けてもよい．

上記の方法は，いずれにしても，言語がコンパイラを記述する能力をもたなければならない．FORTRAN や ALGOL のような言語は，このような能力に不足しているから，これらのコンパイラを利用したり，これらの言語を作成したりするときには，特別な工夫をしなければならない．たとえば，ビットやバイトを扱う手段があれば，能率のよい書き方をすることができるし，結果のコンパイラも能率のよいものになることが予想されるが，それがなければ，整数とその演算によって，ビットやバイト処理にかかわる操作をしなければならない．この結果として，コンパイラが大きくなり，その翻訳速度がおちる．

3. 言語の一般的性質に基づく方法

前節の方法では，取り扱われる言語 S に制限が加わるか，コンパイラの効率が問題となる．しかし，(6′) に対して，S として，コンパイラ記述のための特殊な高次言語を定め，そのコンパイラ $C_{S,M':M}$ を作っておけば

$$C_{S',M':S} * C_{S,M':M} \rightarrow C_{S',M':M'} \quad (9)$$

の1回の操作で，任意の言語 S' の効率のよいコンパイラを作ることができる．これは 1. で述べたシステム・プログラムを PL/I などで書くことの，いわば，コンパイラ版である．しかし，$C_{S',M':S}$ は S' に依存するのみならず，M' にも依存するのであるから，その両者が異なるごとに書き直されなければならないし，依然として M' の変更に基づく設計変更の問題も含んでいる．これを解決するためには，言語の性質とコンパイラの機能について，立ち入った考察を必要とする．

コンパイラは，一つのプログラム用言語 S を，他のプログラム用言語（通常は機械語）に翻訳する．それゆえコンパイラは，原始プログラムとして与えられた記号の列（今後，原始記号列という）を分析して，それが S の文法にかなった正しいプログラムであるかどうかをたしかめる過程と，その分析の結果に従って，原始記号列と同じ内容を記述する目的プログラムの命令列を発生する過程とから成り立っている．前者を構文解析といい，後者を意味づけということにする．

コンパイラの機能が二つの部分から成り立つので，そのプログラムも二つの部分に分け，それぞれに構文解析と意味づけの働きをさせることができるはずである．それができれば言語 S にのみ依存する部分と，目的プログラム語 M にも依存する部分が分れるので，M の変わるときの書き換え部分が局在化することになる．

プログラム用言語は，Chomsky の理論に従って，大体のところ context free phrase structure language に属しているものと考えられる．それゆえ，この類の言語に対する一般的な構文解析の方法を適用することができるから，上に述べたコンパイラの構文解析部分を，言語 S によらない一般的な部分を，言語ごとに異なる部分とに分けることが可能である（次節参照）．

意味づけ部分についても，のちに述べるように，言語 S に依存する部分と，目的プログラム言語 M に依存する部分とを分離することが可能である．

このような手段によって，コンパイラを言語 S から独立な部分，S に依存するが M からは独立な部分，および M にのみ依存する部分とに分けることができれば，機械の変更はもちろん，言語 S の変更に対しても，プログラムの書きかえが容易となる．そして，このような言語の性質に基づく，構造の正準化のうえに，context free phrase structure language（以下，句構造言語という）について一般的なコンパイラ・コンパイラを，それが処理するコンパイラと同一構造に作成することができる，次に句構造言語と，その構文解析について簡単に説明しよう．

3.1 句構造言語と構文解析

ALGOL 60 の文法を記述する際に，いわゆる，Backus-Normal Form と呼ばれる超言語式が用いられた．たとえば，その文法表記法で代入文の簡単な定義を作ってみよう．

⟨statement⟩::=⟨variable⟩=⟨expression⟩
⟨expression⟩::=⟨term⟩|⟨expression⟩+⟨term⟩
⟨term⟩::=⟨factor⟩|⟨term⟩×⟨factor⟩
⟨factor⟩::=⟨variable⟩|⟨constant⟩|(⟨expression⟩)
⟨variable⟩::=⟨identifier⟩
⟨constant⟩::=⟨unsigned integer⟩

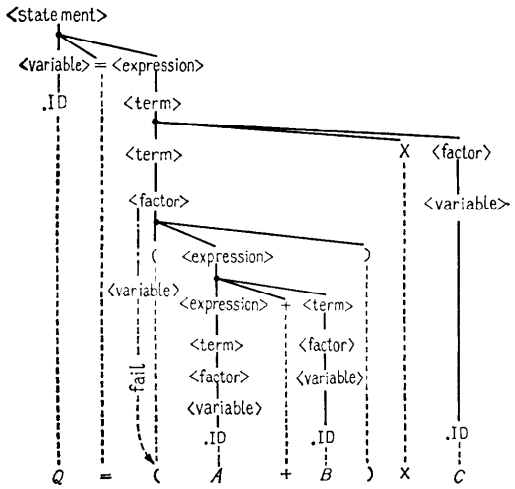
このような形に構文が表現できれば，その言語は句構造言語であるという．この場合，一つの記号，ここでは ⟨statement⟩ が左辺にしか現われていないことに注意しておく．

=，+，×，(，および) は原始プログラムに現われる記号で，超言語常数または端記号 (terminal symbol) といわれる．⟨statement⟩ などは超言語変数，または非端記号 (non-terminal symbol)，または被定義記号 (defined symbol) である．⟨identifier⟩ や ⟨unsigned integer⟩ も被定義記号であるが，そうすると，完全な定義のためには式が多くなり，話しが冗長になるから，それぞれ .*I* および .*N* とかいて，端記号のように扱うことにする．また，都合により，超言語記号として，' と ; を導入して，上式を次のように表わす。

⟨statement⟩=⟨variable⟩'=⟨expression⟩;
⟨expression⟩=⟨term⟩|⟨expression⟩'+⟨term⟩;
⟨term⟩=⟨factor⟩|⟨term⟩'×⟨factor⟩;
⟨factor⟩=⟨variable⟩|⟨constant⟩|'(⟨expression⟩');
⟨variable⟩=.*I*,
⟨constant⟩=.*N* (9)

' はその次の記号が端記号であることを示す．

さて，構文解析は原始記号列が，この言語の構文法にかなっているかどうかを調べることである．構文が (9) 式のようにあたえられた場合，この確認の一つの方法は，構文に従って端記号のみからなる記号列を作り出し，それを原始記号列と比較してみることである．すなわち，原始記号別は statement の一つであるはずであるから，われわれは (9) の第1式 ⟨statement⟩ を定義する超言語式をとりあげる．その定義の右辺の第1要素 ⟨variable⟩ は端記号でないから，その定義式，第5式によって，.*I* でとりかえる．もとへもどり，第1式の第2要素は端記号であるが，第3要素 ⟨expression⟩ はそうでない．そこで，これに第2式の定義を代入する．⟨expression⟩ は二つの定義 ⟨term⟩ と ⟨expression⟩'+⟨term⟩ をもつから，とりあえず ⟨term⟩ を入れる．⟨term⟩ は，また非端記号であるから，その定義によっておきかえる．このような手続きによって，端記号のみからなる記号列を生成することができる．そして，これと原始記号列を比較して，後者が前者に一致するかどうかをたしかめることができる．



第1図 構文解析の結果作られる樹構造

通常，各非端記号は | で分けられた若干の定義をもつので，生成される記号列は一意にきまらない．それゆえ，生成された記号列が，原始記号列と一致するまで，| で区分された各定義を一定の順に適用していく*．最後にいたるまで一致するものがなければ，与えられた記号列は，構文上，この言語には属さない，または間違って書かれている，ということになる．原始記号列が正しければ，結果として，**第1図**のような構文要素間のつながりが確認される．

ここに述べた解析方法は，言語の構造が (9) 式のような構文に表わされることに基づいている．しかし，その言語に特徴的な文法自体は，構文の中に表現されているのであって解析方法は全くそれからの独立である．それゆえ，プログラム上，各言語からは独立で，句構造言語に共通な解析プログラムと，各言語に依存する構文を表わすデータ (以下，構文表という) とを分けることができる．すなわち，構文解析プログラムは構文表をとりかえることにより，任意の句構造言語の

* 言語のあいまいさをさけるため，定義の適用には，右向き，最長一致の原則がとられなければならないが，その説明は省略する．

解析を行なうことができる．

3.2 意味の記述

構文解析に従って，正しいことが確認された原始記号列に対しては，それと同じ内容を記述する目的プログラムの命令列が生成されなければならない．この命令をアセンブラ的に表わせば，たとえば，一つの句 $A\times B$ に対しては

LOAD A
MULT B

なる命令列を発生する必要がある．たとえば，記号列の部分

$A+B\times C$

に対し，構文解析の結果

$B\times C$　　を　〈term〉
$A+B\times C$　を　〈expression〉

として，この順に句が確認されたとする．そうするとその順に

LOAD B; MULT C; STORE U

および

LOAD A; ADD U; STORE V

を発生すればよい．U や V は意味づけプログラムが内部的に発生する名前である．

この命令発生は，構文解析の完全に終了した後に行なってもよいけれども，ここでは簡単のために解析の過程で，一つの句の存在が確認されたとき，直ちにとられる処置として，行なうものとする．そのためには構文 (9) の各式に，それぞれに対応する意味づけプログラムの名前をそえてやり，それは解析プログラムが原始記号列の中に，句の存在を確認したとき，直ちに呼ばれるサブ・ルーチンとしよう．たとえば

〈term〉=〈term〉'×〈factor〉: MD

で MD は

```
generate ('LOAD' *1); generate ('MULT' *3);
replace (*1, *3, WORK); generate ('STORE'
  *1);
```

なるプログラムである．'LOAD', 'MULT' および 'STORE' は記号列 LOAD, MULT および STORE である．*1, *3 は句の第1および第3要素の内容，ここでは B と C であり，replace (*1, *3, WORK) は，WORK により名前を発生し，それによって，原始記号列中に確認された第1から第3要素 $B\times C$ をおきかえることである．おきかえた後，第1要素の位置には，たとえば U がある．

意味づけは，命令の発生に附随して，いろいろな手続きを必要とする．たとえば，前例でも，内部的な名前の発生を必要とした．このほか被演算子の型の分析や，名前が格納された表の索引，そのほか数多くの作業を必要とする．命令の発生は，むしろ，その中のごく一部分にすぎない．そこで，意味づけプログラムを，命令発生のような機械に依存する部分と，源プログラム言語にのみ依存する部分に分けることができれば，機械ごとの書き換え部分が，ここでも一部分に局在化することになる．

この一つの方法は，アセンブラ的な機械によらない中間言語を仮定し，その命令列を発生することである．この中間言語より機械語への変換は，コード・ジェネレーターと称するプログラムで行なうことにする．

3.3 コンパイラ・コンパイラ

前節までにコンパイラの構造を正準化する一つの手段について述べた．ここでは，この方向でのコンパイラ・コンパイラの形態について述べよう．

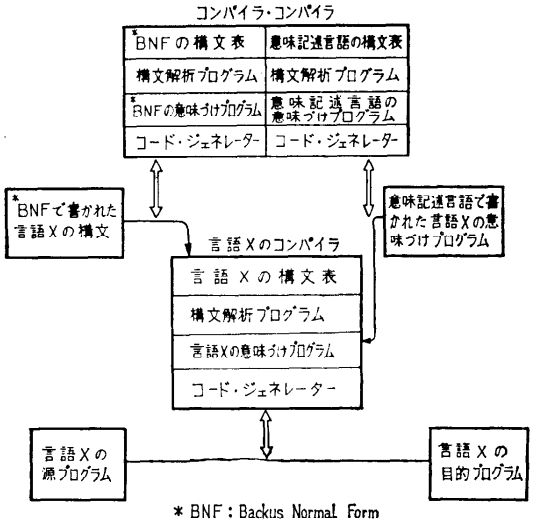
構文を表わす超言語式は =, |, ,, 〈, 〉, ', および・を超言語記号として用いているが，実際には，これだけでは，複雑な構文を簡明に表わすために不充分なことが多い．しかし，この問題には，ここではふれない．それよりもこれらの記号の使い方は，一定の規則に従うものであるから，それを超言語の文法とみなすことができる点に注目しよう．そこで，これを (9) のような式で表わせば

```
〈syntax〉=〈meta equation sequence〉;
〈meta equation sequence〉=〈meta equation〉
  |〈meta equation sequence〉'; 〈meta equation〉;
〈meta equation〉=〈defined symbol〉'=〈definiens〉;
〈definiens〉=〈phrase〉|〈definiens〉'|〈phrase〉;
〈phrase〉=〈terminal symbol〉|〈defined symbol〉
  |〈phrase〉〈terminal symbol〉|〈phrase〉〈defined
  symbol〉;
〈defined symbol〉='〈. I'〉;
〈terminal symbol〉='. . I|. ''. CHR          (10)
```

である．. CHR は任意の一記号を表わすものとする．すなわち，超言語を最も簡単な句構造言語とみなすことができる．それゆえ，われわれは句構造言語の構文解析プログラムを，その構文表を (9) に対応するものから，(10) に対応するものに変更するだけで，超言語式の解析プログラムとして使用することができる．この場合，(10) の構文につけられる意味づけプログラムの内容を構文表作成とするならば，構文解析プログラムは対応する意味づけプログラムと一緒になっ

て，任意の言語の構文を原始プログラムとして読み，対応する構文表を作成するコンパイラ・コンパイラとなる．

意味づけについては，適当な問題向き意味づけ言語を設定し，そのコンパイラを作成しておく．これはコード・ジェネレーターを変更することによって，意味づけ言語で問題向きに表現された任意のコンパイラの意味づけプログラムを，必要な機械のうえに変換することができる．



第2図　コンパイラ・コンパイラの構成と機能

これらの構造は，コンパイラとの関係を含めて，第2図に表わされている．コンパイラ・コンパイラの構文解析と意味づけ部分は同じ構造となっている．コード・ジェネレーターは再び意味記述言語で記述できるから，意味づけ部分の残りを，意味記述言語で記述しておけば2回の recompile でコンパイラ・コンパイラを他の機械上に移すこともできる（(6) 式参照）．

意味記述用言語としては，いままでのところ，あまりよいものはできていない．VITAL の FSL[4] や，SYMPL[5] や，TREE META の意味部分[6]は，専用に設計されたものである．PL/360[7] のように，ALGOL 的アセンブラを用いる試みもある．最近は PL/I を用いるのが一つの傾向となっている．

4.　その他の方法

3.2 で述べた構文解析の方法は下向き型 (top down) といわれる．これに対し上向き型 (bottom up) がある．

まず，(9) 式を左右両辺を逆にし，句の第1要素の同じものを一つにあつめ，かつ，超言語記号[と]を導入して，次のように書きなおす．

〈variable〉 ['=〈expression〉=〈statement〉
　|=〈factor〉]
〈term〉['×〈factor〉=〈term〉|−〈expression〉]
〈expression〉'+〈term〉=〈expression〉
〈factor〉=〈term〉
〈constant〉=〈factor〉
'(〈expression〉')=〈factor〉
. I=〈variable〉
. N=〈constant〉　　(11)

原始記号列の最左端の記号から出発して各記号に，(11) の構文を適用する．たとえば，原始記号列に名前が表われれば，(11) の第7式を適用し，それを〈variable〉におきかえる．ついで〈variable〉をより高位の定義におきかえる式は第1式だから，これを適用する．ところが，第1式は〈variable〉の右となりには = か〈factor〉におきかえられる記号列があるべきことを指示する．原始記号列上，次の位置に = があれば，= の次の記号からは，第1式に従えば〈expression〉に対応するものがなければならない．そこで，原始記号列上，=の次の記号を左端にもつ(11)の式から出発し，上記の手続きの繰返しによって〈expression〉の存在を確認する．= がなかったときは，最初の〈variable〉は〈factor〉におきかえられ，それは第5式により〈term〉におきかえられるというふうに，だんだん高位の定義にかえられていく．このようにして，原始記号列が〈statement〉になるかどうか調べられる．上向き型と下向き型はともに，即構文型 (syntax directed) であるといわれる．

上向き型の解析方法も，その手続き自体は，言語からは独立で，個々の言語に依存する部分は，すべて構文の中にもられている．それゆえ，前節に述べた方法によって，そのコンパイラを作成することができる．

上向き型と下向き型の相違は，前者が最終的な目標，たとえば，(9) や (11) では〈statement〉から出発して，構文に従って非端記号を，低位の非端記号または端記号の列でおきかえるのに対し，後者は原始記号列の記号から出発し，構文に従って，それを次第に高位の非端記号におきかえる．

これらのおきかえが正しいかどうかは，原則的に

は，前者の方法であればすべての非端記号が端記号でおきかえられたとき，後者であれば，原始記号列が最終的に目標の非端記号におきかえられたときに確認される．それにいたる道筋は，| で区切られる若干の定義を左から適用してみるだけで，最適な定義を予見する手段は厳密な意味では存在しない．ある定義が適用不可能であることは，適用の結果判明するし，その場合は適用直前の状態にもどって，次の定義を適用してみなければならない．すなわち，原始記号列を逆もどりして調べ直さなければならない．このため，即構文型解析は手づくりのコンパイラにくらべて，速度が数分の1である．[8]

速度をあげるためには，原始記号列の調べ直しをしなくてすむようにすればよい．そのためには，構文の書き方を制限する必要がある．たとえば，(9)や(11)の一つの式で | で区切られる定義がいくつかあるとき，各定義の先頭要素がすべて端記号で，お互いに異なるものであれば，原始記号列上の記号一つを見るだけで適用可能の構文は一義的に定まり，解析が進んでからあともどりすることはない．なぜなら，ある時点で適用可能の構文がなければ，原始記号列に誤りがあることになるからである．また，定義に共通部分があれば，新しい記号を導入し構文を細分することも，原始記号列上のあともどりをへらす手段である．たとえば

$$\langle Q\rangle = tr \mid ts$$

であれば

$$\langle Q\rangle = t\langle T\rangle\,;\ \langle T\rangle = r \mid s$$

と分割すると，r が適用不可能であったとき，t の位置までもどる必要はなくなる．ただし，r, s, および t は任意の端記号，または非端記号とする．

このような方法として優先順位（precedence）法[9]や遷移（transition）マトリックス法[10]がある．これらの方法は上向き型に属すけれど，構文解析は確定的で，ある時点で，いわば，局地的に原始記号列のあやまりを指適できるし，原始記号列のあともどりも必要がない．しかし，上記のような構文上の制限および書きかえきを伴う結果，この方法で解析可能な言語の型は，それぞれすこしずつ句構造言語よりせばめられる．通常，せばめられた部分の解析は，意味づけプログラムの中に，適当なパラメタを設けるか，あるいはそれに対応する手段で解決する．これらの方法は，手づくりコンパイラの伝統的手法の一般化であって，構文に関しては超言語による記述がなされ，したがって，コンパイラ・コンパイラの形に作ることができる．

原始記号列を解析していく過程そのものを production language[11] と称する超言語によって表現する方法がある．これも上向き型で確定的な解析方法である．しかし，句の置換のさいに文脈をみることができるので，つまり，隣接の記号を参照できるので，扱いうる言語の範囲は，前二者よりは広い．

これらの方法に対する構文の表現方法は，句が細分され，余計な非端記号が導入される結果，Backus-Normal Form にくらべて，式の数がふえ，構文が著しく見にくくなるのが，もう一つの欠点である．

構文解析には，そのほか多くの方法がある．それらはすべて，それに属する言語の類を定める．問題は，その類が広いか狭いかということ，解析，および表現の明快さである．

5. あとがき

比較的実用的な観点からコンパイラ・コンパイラの方法について述べた．述べられた方法よりは，もっと幅の広い文法に対応するが，実際には適用できない若干の方法がある．それらは低位の方法の文法的なレベルの関係を論ずるものとして意味をもつであろう．[12]

現在の多くの計算機は，ここに述べられた型のコンパイラに対して，それを効率よく実現できるようにはできていない．たとえば，簡単なスタックや，テーブルやリスト処理に都合のよい命令があれば，それだけでも効率は著しく改善されるであろう．このような機能は，モニタの技術とも強い関連をもっているのであるから，それがオペレーティング・システムを，大きさと速度の点で改善することが予想される．大切なことは，このようなコンパイラ・コンパイラの方法は，必然的にコーディングの標準化に導びくのであるから，機械の論理体系の方向づけに，役割を果たすであろう．

参考文献

1) 戸田　巌，その他：PL/I コンパイラ構成法の研究，研究実用化報告，**18**, 1, p. 115 (1969)；高橋延匡，その他：HITAC 5020 TSS スーパイザの PL/I による記述上の問題点，昭和44年連合大会講演論文集39，情報処理 (5)，3805；その他多数．

2) 首藤　勝，魚田勝臣，居原田邦男：計算機を用いたコンパイラ作成自動化の実験，第7回プログラミング一シンポジウム報告集 C-74 (1966年1月)；藤野喜一：Compiler Generating System，情報処理，**7**, 5, p. 245 (1966)．

3) M. H. Halstead: Machine-Independent Computer Programming, Spartan (1962).
4) L. F. Mondshein: VITAL Compiler-Compiler System Reference Manual, Technical Note (1967-12), Lincoln Lab. MIT.
5) SYMPL, CSC (Sep. 1 1968).
6) D. I. Andrews and J. F. Rulifson: Tree Meta, A Meta Compiler System for the SDS 940, Working Draft (Dec. 29, 1967) SRI.
7) N. Wirth: PL 360, A Programming Language for the 360 Computers, J. ACM, **15**, 1, p. 37 (1968).
8) 荻原 宏, 渡辺勝正: Compiler-Compiler について, 算法言語の記述および処理方法の研究, 3, 24 (1969)
9) N. Wirth and H. Weber: Euler-A Generalization of ALGOL, and its Formal Definition, Part I and II. C. ACM. **9**, 1, and 2, p. 13 and 89 (1966).
10) D. Gries: The Use of Transition Matrics in Compiling. C. ACM **11**, 1, p. 26 (1968).
11) R. W. Floyd: A Descriptive Language for Symbol Manipulation, J ACM. **8**, 4, p. 579 (1961).
12) S. Ginsburg, S. Greibach, and M. A Harrison: Stack Automata and Compiling, J. ACM. **14** 1, p. 172 (1967); D. E. Knuth: On the Translation of Languages from Left to Right, C. ACM. **10**, 10, p. 611 (1967).
13) J. Feldman and D. Gries: Translator Writing Systems, C. ACM, **11**, 2, p. 77 (1968).

(昭和44年5月16日受付)