普 及 課

47－S 002

コンパイラ・ジェネレータの開発

昭 和 48 年 3 月

JIPDEC

財団法人 日本情報処理開発センター

この報告書は，日本自転車振興会から競輪収益の一部である機械工業振興資金の補助を受けて昭和４７年度に実施した「コンパイラ・ジェネレータの作成」の成果をとりまとめたものであります。

序

当財団は情報処理に関する各種の研究開発をおこなっておりますが，本報告書はその一環としてとり上げた「コンパイラ・ジェネレータの開発」の成果をまとめたものであります。

現在，コンパイラ言語はコンピュータのプログラミング言語として最も広く使用されておりますが，新機種のための，あるいは新しい言語のためのコンパイラを，その都度開発する努力は多大なものであります。この労力を軽減し，かつ信頼性のさらに高いコンパイラを得るために，コンパイラ・ジェネレータが有効な手段であることはよく知られております。

しかし，ジェネレートされたコンパイラの効率の面，ジェネレータへの入力の技術などにまだ種々の問題があり，いま一歩，実用化への努力が必要とされております。

本研究開発は，これらの問題点の原因を解明し，実用に供し得るコンパイラ・ジェネレータの試作を目ざすものであります。

この報告書が各方面に利用され，わが国の情報処理技術発展の一助として寄与できるよう念願いたす次第であります。

昭和４８年３月

財団法人　日本情報処理開発センター

会長　難　波　捷　吾

ま　え　が　き

コンパイラ言語の構造と構文解析手法に関する研究は1960年頃から主として大学・研究機関において行なわれ，その成果として各種のコンパイラ・ジェネレータが開発されてきた。

コンパイラの作成における多大の労力を軽減し得るという理由でコンパイラ・ジェネレータには多くの期待がかけられているが，実用上ではいまだ多くの問題点が残されている。

例えば，セマンティックス記述においては，シンタックス記述のような形式的な記述方法がないこと，構文解析手法が一意的であるコンパイラ・ジェネレータによって作成されたコンパイラは，構文解析が各コンパイラ言語のレベルに最も適した手法で行なわれる hand-made のコンパイラよりコンパイル速度，メモリ占有量等において劣っていることなどである。

コンパイラ・ジェネレータＪPAC（ＪIPDEC Automatic Compiler generator）においては構文解析手法としては言語の複雑さによってSLR(k)文法とL(m)R(k)文法のいずれかが自動的に選択され，セマンティックス記述としてはシステム記述言語ＬＳＤ及び標準セマンティックス・ルーチンを用いる方法を採用した。

主眼としてはジェネレートされたコンパイラの効率が実用に耐え得るレベルを目標としている。

本報告書は２部にわけ，第１部は主として米国において開発された各種のコンパイラ・ジェネレータを紹介し，第２部にはＪPACの内部および外部仕様を記述してある。また付録として1968年までのコンパイラ・ジェネレータの著名な紹介論文である Translator Writing Systems の全訳を掲載した。

本研究開発は，２カ年計画でおこなっており，４７年度には仕様の決定，システムの設計および作成の一部を終了し，４８年度には標準セマンティックス・ルーチンの整備，ＪＰＡＣによるコンパイラと hand- made のコンパイラとの性能・労力などの比較をおこなう予定である。

総　目　次

第　1　部

コンパイラ・ジェネレータの調査

コンパイラ・ジェネレータあるいはコンパイラ・コンパイラの研究は，1960年頃から主として大学，研究機関に於ておこなわれてきたが，1968年 Stanford 大学の J, Feldman, D.Gries によって，米国におけるそれまでの研究成果がまとめられた。

" Translator Writing Systems "
Communication ACM. Vol. 11, No 2

がそれである。

この報告書では Transcator Writing Systems (TWS) 以後に発表されたシステムを中心に，それぞれのシステムのねらいおよび手法の概要を紹介するが，代表的なものはあえてTWSと重複させた。

なおTWS の全訳を付録として巻末に収録した。

目　　次

1. 概　　説

ランゲージ・プロセッサの開発には非常に多くの労力を要することが知られている。たとえば，超大型コンピュータのFORTRANコンパイラ作成のために要する労力の試算では８～１５人年を必要とするという報告がなされている。

ここ数年来この労力を軽減するために種々の方法が試みられている。現在コンピュータ・メーカの提供しているコンパイラの多くはまだアセンブラ言語で記述されているが，FORTRANまたはPL/Iのような汎用言語の使用，APL，LISPなどの特殊言語の使用，あるいはBPLのようにシステム記述言語を作成し，これによって記述することなどにより労力の軽減をはかることができる。

これ以外にメタ言語によりコンパイラを作成する方法もあり，汎用のコンパイラ言語等によるシステム作成よりも労力の削減が期待できるといわれている。

コンパイラ言語はContext Sensitive 文法の類に属するのでCanonシステムのように，この類の文法を記述し，解析する手段を提供するものがある。

しかしながら，その処理速度が実用化するにはほど遠いシステムになってしまうため，ほとんどのものは，コンパイラの記述をSyntax記述とSemantics記述に分離して考えSyntaxはContext Free 文法の範囲で記述し，Context Sensitive 文法になる部分は対応するSemantics という形で記述する方法をとっている。

Syntax記述に関しては，バッカス記法（BNF）というシンプルな方法があり，BNFあるいはextended BNF等を使用することでこと足りる。

Syntaxの解析には，TREE METAのようにrecursive機能を使用することにより下向きに解析するものと，順位文法等のContext Free 文法の一類を使用して上向きに解析する手法とがある。

TREE META のような方法は構文解析においてあともどりをおこなわなければならない。あともどりとは生成規則の適用の各段階を，１段ずつ逆戻り

して，それまでに試されなかった規則を適用することによって，他の可能性を調べることである。

この方法は，解析に必要なテーブル類は小さくなるが，解析速度が遅いといいう欠点を持っている。

順位文法等の方法では，あともどりは起こらないため解析速度を速くすることができるが，Syntex記述をその文法にありように書き変えなければならないこと，およびテーブルが大きくなる欠点を持っている。

構文解析に関しては，どちらの方法をとるにしろ決定的によいものはないので今後の研究が望まれるが，FORTRANクラスのコンパイラならば解析できるところまできている。

Semantics の記述に関してはSyntax記述におけるＢＮＦのようなよい記述方法が開発されてはいない。このためこの部分に関する技術の確立が待たれる。

Semantics の記述に必要な機能は，コンパイラ記述言語に必要な機能とほぼ同様で次のようなものがあげられる。

1. テーブル類のとりあつかいができる。
2. スタックのとりあつかいができる。
3. ビット，文字のとりあつかいができる。
4. 整数の簡単な演算ができる。
5. アドレスを作り出すことができる。
6. リスト，tree 等のあつかいが何らかの形でできる。
7. 入出力機能

現在までに開発されたコンパイラ・ジェネレータは，上記機能を有する言語を用意して，これによって自由にSemantics を記述するもの（ＸＰＬ，ＭＰＬ／Ｉ）と，これ等の機能を持つ標準ルーチンをいくつか用意して，これを呼び出

す方式（GULP，META）のいずれかによっている。

後者は，前者よりもユーザにとってはコンパイラを作る場合に楽ではあるがどのような標準ルーチンを用意すればよいかが問題となる。

2. 各種コンパイラ・ジェネレータ

2.1 ＸＰＬコンパイラ・ジェネレータ

ＸＰＬコンパイラ・ジェネレータは，スタンフォード大学においてW.M.. Mckeeman他によって１９６８年にＩＢＭ／３６０に対してインプリメントされた。

これは，ＢＮＦで書かれたSyntax規則を読み込んで，後のプロセスで必要となるテーブルを作り出すＡＮＡＬＹＺＥＲ，Semantics記述言語であるＸＰＬ言語のコンパイラＸＣＯＭ，およびジェネレートの対象となるコンパイラのソース・プログラムを読み込んでＡＮＡＬＹＺＥＲ で作り上げたシンタックス・テーブルにしたがい必要なセマンティックス・ルーチンにコントロールをわたす。ＸＰＬ言語で作られたＳＫＥＬＴＯＮより構成されており，その関係は（図２－１）ＸＰＬコンパイラ・ジェネレータの構成に示すとおりである。

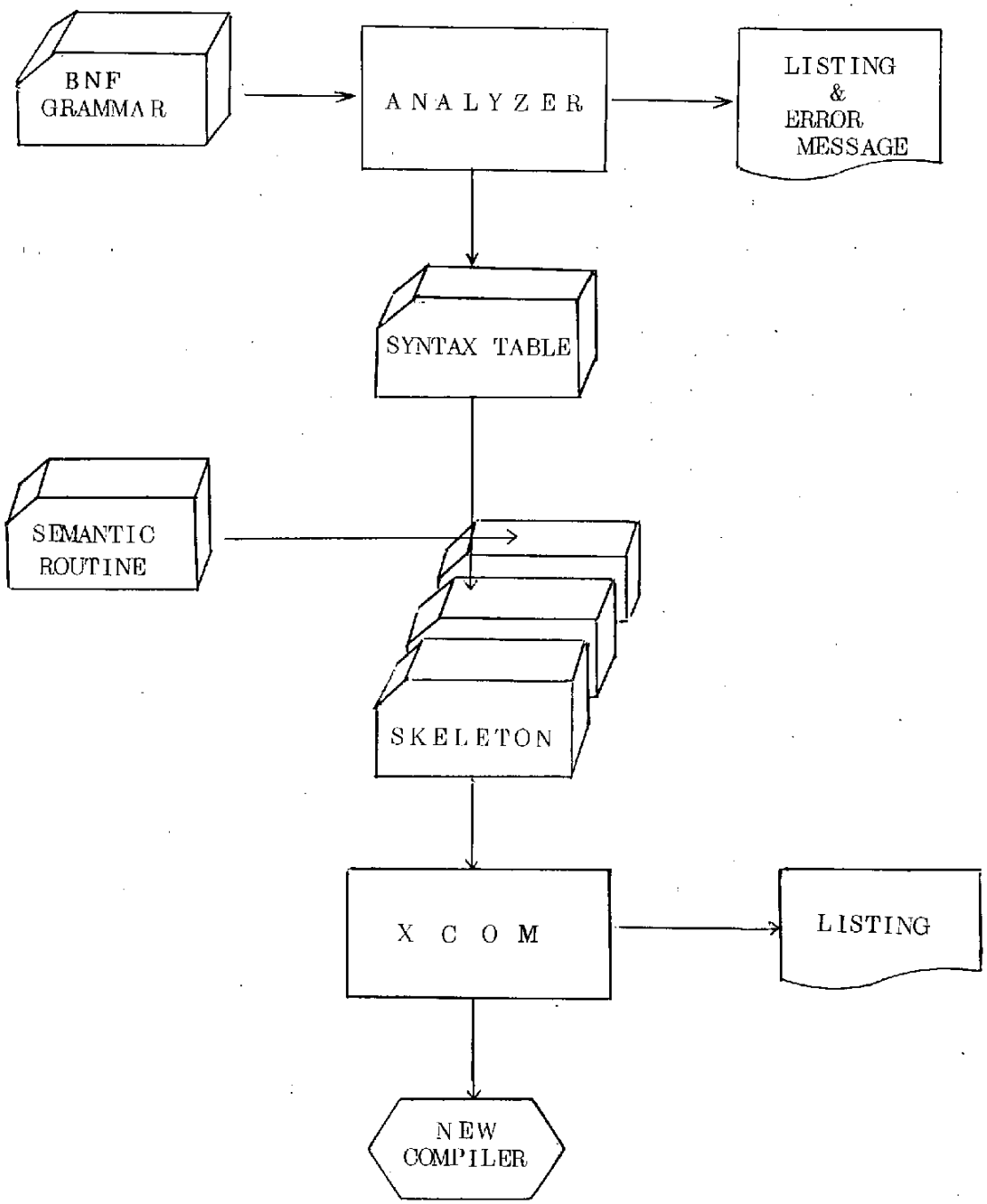


図２－１　ＸＰＬ　コンパイラ．ジェネレータの構成

ANALYZER

ANALYZERは構成図より明らかなように，単なるシンタックス・テーブル・ジェネレータであって，これが受け付ける文法は（２，１；１，１）位のMixed Strategy Precedence文法（以後ＭＳＰ（２，１；１，１）と略称する）である。ＭＳＰ（２，１；１，１）というのは，スタック中の記号と新しく入力したterminal記号を見てstackをするか，あるいは生成規則を適用してreduceするかという決定を（２，１）位のprecedence文法によっておこないreduceの決定がなされた場合において，生成規則の選択を（１，１）位のbounded contextによって確認する手法を意味している。

precedence文法の解析手法は，通常順位マトリックスをもちいる（順位関数を作る場合もあるが，常にこの関数ができ上がるとはかぎらない）。

順位マトリックスの大きさは，（１，１）位の場合

（記号の数）×（terminal記号の数）

となるが，（２，１）位の場合

$$(\text{記号の数})^2 \times (\text{terminal記号の数})$$

となり，マトリックスは非常に大きくなってしまう。

幸いにして，（２，１）位の文法において，stack あるいはreduceの決定のほとんどは（１，１）位でおこなうことができる。このために，順位マトリックスとしては（１，１）位に対応する大きさだけ用意する。（１，１）位の場合，マトリックス要素としてはstack，reduce，エラーの３種のみでよいが，（２，１）位の場合には，これに不明という情報を追加しておく。

不明の場合には，あらたに別のテーブルを調べて，stack，reduce の決定をすることになる。すなわち，スタックの頭の２個の記号と，新しく入力した記号の計３個の記号をテーブルにしておき，stackあるいはreduceを決定する。この時，３個の記号の連らなりがエラーであるかどうかのチェックをしないとすれば（エラーチェックを省略しても，生成規則を適用しようとして生成規則のテーブルをサーチした時，登録されていないのでエラーの発見はできる。）

stackあるいはreduceのいずれか一方だけをテーブルに登録しておけばよい。このため，あまり多くない情報の追加で（２，１）位のPrecedence文法の処理が可能となる。

ANALYZERの入力は，BNFで記述した文法であり，上記の思想により次のものがカード出力される。

1. V：terminal，nonterminal記号テーブル。
2. C1：（２，１）位Precedence文法に対応するstack，reduce，不明，エラーを示すマトリックス
3. C1TRIPLES：C1において不明の場合に調べるテーブルであってstackすべき場合のみがのっている。
4. ＰＲＴＢ：生成規則の選択のためのテーブル
5. PRDTB：セマンテイックス・ルーチンの番号テーブル
6. ＨＤＴＢ：生成規則が適用された時，どのようなnonterminal記号に還元されるかを示すテーブル
7. Context-Case：reduceの決定がなされた時，生成規則の選択にあたってContextを調べる必要の有無を示すテーブル
8. Ieft-Context，Context-Triple：生成規則の選択にあたってそれぞれ（１，０）Context，（１，１）Contextを調べる必要のある時のテーブル。
9. その他の補助情報

ANALYZERにおける特徴としてあげられるものに，入力ＢＮＦ記述の自動変更がある。bounded contextアルゴリズムにおいて"insulting comma"－区切り記号としての役割を果たすカンマ等－の問題がある。この問題に関しては，カンマに対応するnonterminal記号を定義し，それに

よって影響をうける生成規則を書き変えることによりしばしば解決できる。このためＸＰＬにおいてこのような場合には自動的に新しいnonterminal記号および生成規則の追加，変更をしてＡＮＡＬＹＺＥＲ が受付けられるように自動変更している。

ＳＫＥＬＴＯＮ

ＳＫＥＬＴＯＮはＡＮＡＬＹＺＥＲで作り上げたマトリックス・テーブル等にしたがって構文解析をし，生成規則の適用（reduce）が決定すると，生成規則番号をパラメータに持ってユーザが用意したセマンティックス・ルーチンにコントロールをわたす役割をする。すなわち，目的としたコンパイラの共通メインルーチンとみなすことができる。

（図２－２）によりＭＳＰ（２，；１，１）における構文解析アルゴリズムの概略を示し（図２－３）でＸＰＬが実際におこなっている方法を示す。なお，（図２－３）では説明を簡単にするために，実際に使われている記号をやゝ変えている。

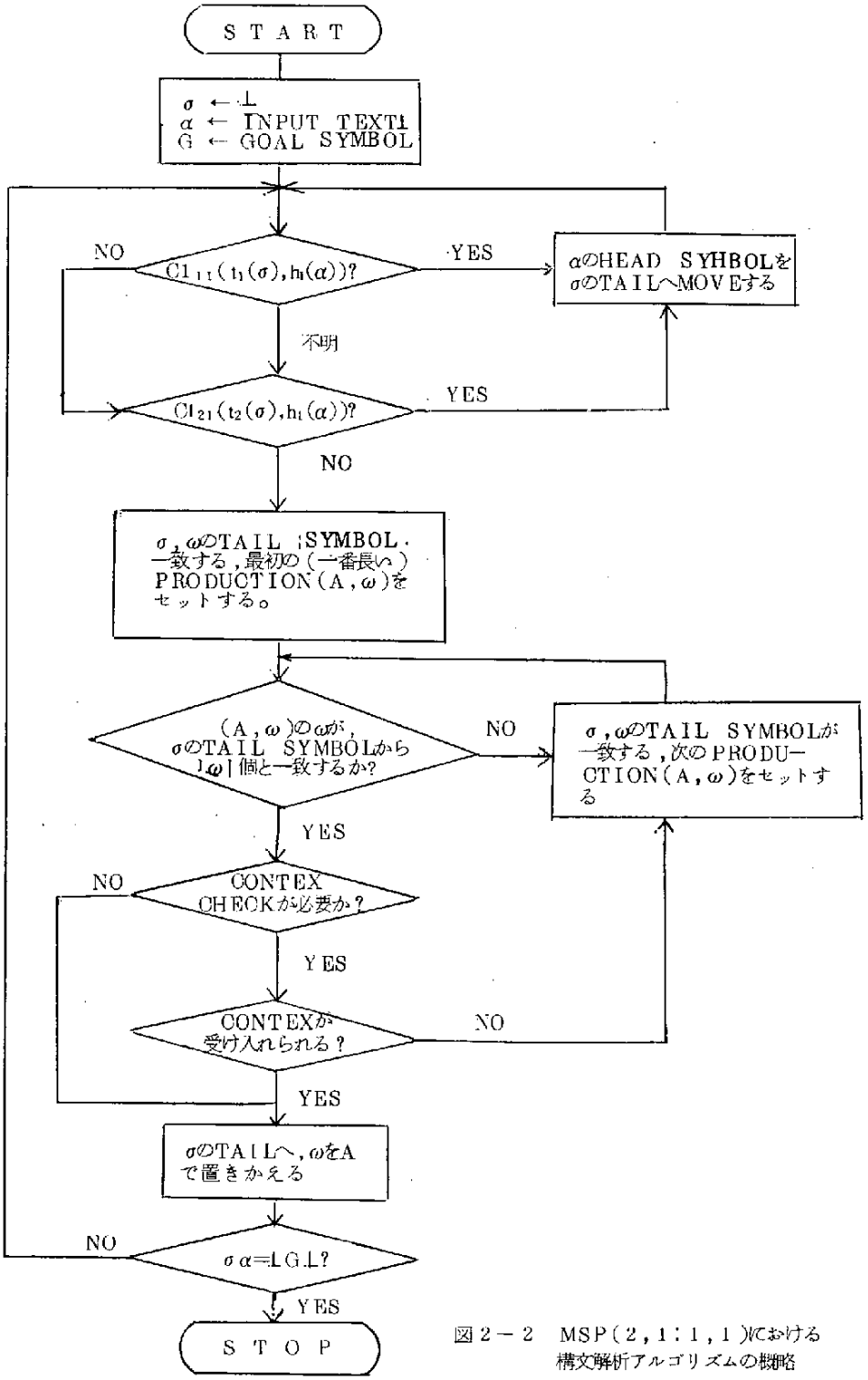


図２－２　MSP(２,１:１,１)における
構文解析アルゴリズムの概略

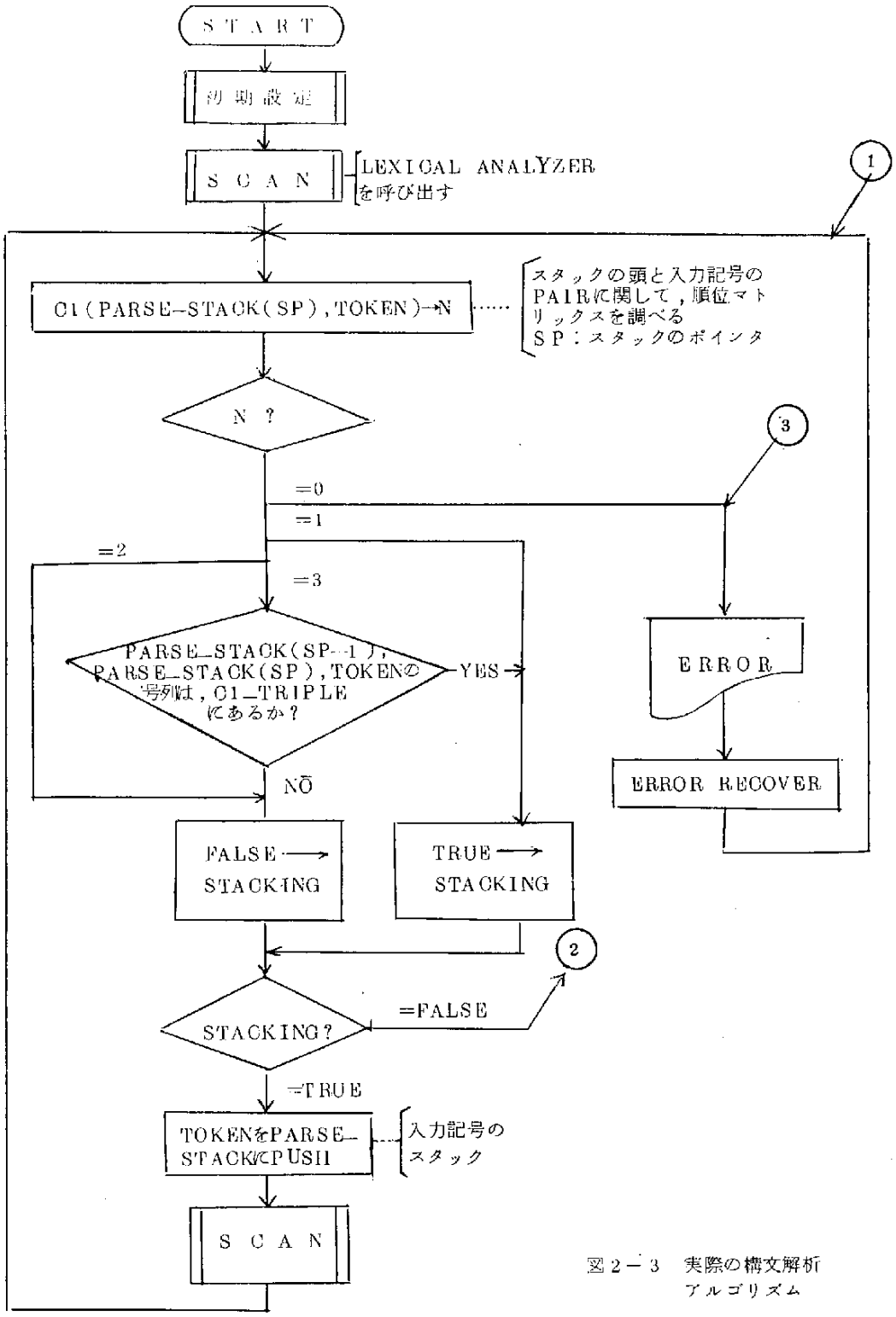
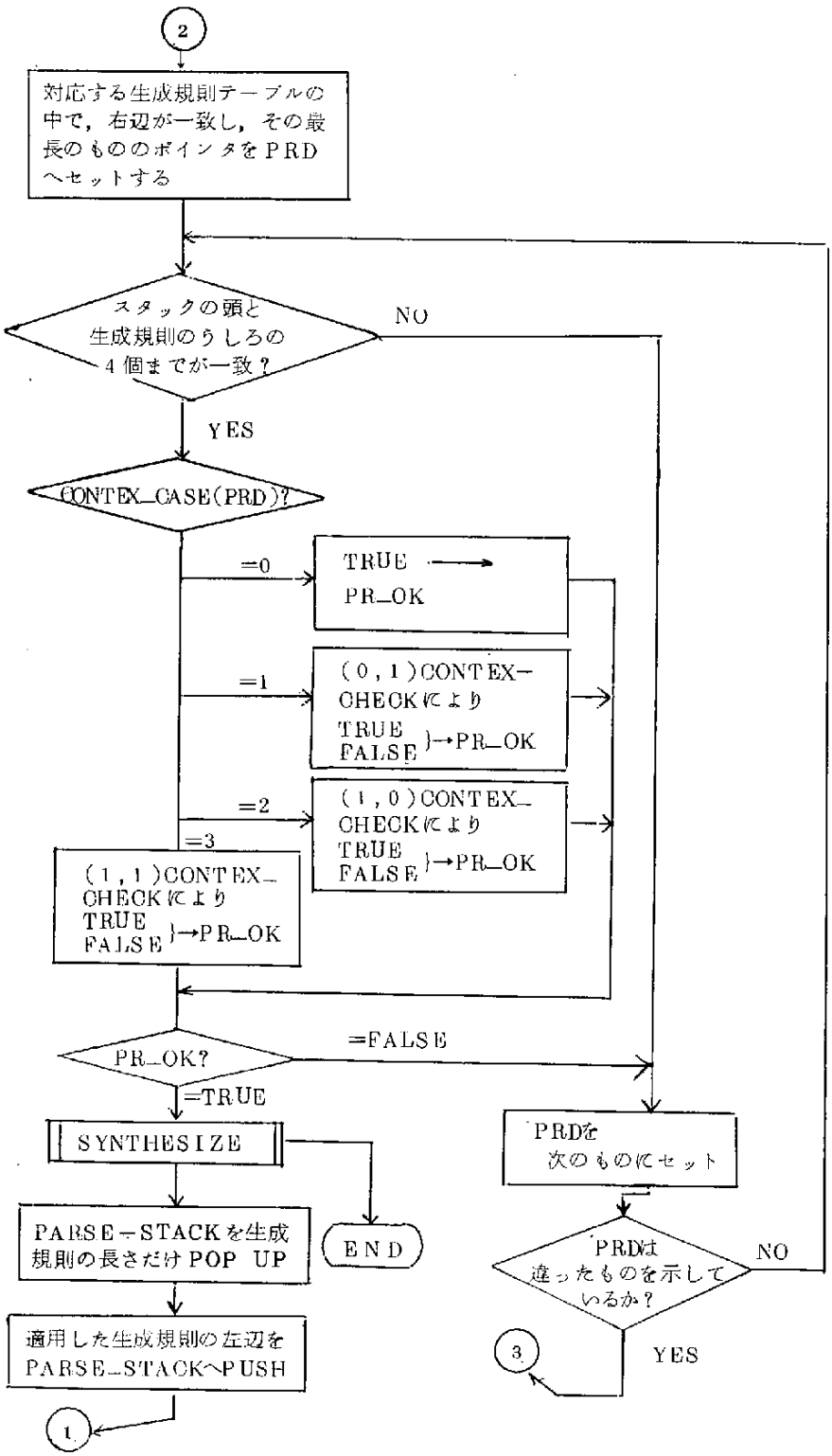


図2－3　実際の構文解析
アルゴリズム


2
対応する生成規則テーブルの中で，右辺が一致し，その最長のもののポインタをPRDへセットする
スタックの頭と生成規則のうしろの4個までが一致？
NO
YES
CONTEX_CASE(PRD)?
=0
TRUE → PR_OK
=1
(0,1)CONTEX_CHECKにより TRUE FALSE →PR_OK
=2
(1,0)CONTEX_CHECKにより TRUE FALSE →PR_OK
=3
(1,1)CONTEX_CHECKにより TRUE FALSE →PR_OK
PR_OK?
=FALSE
=TRUE
SYNTHESIZE
END
PARSE－STACKを生成規則の長さだけ POP UP
適用した生成規則の左辺をPARSE_STACKへPUSH
1
PRDを次のものにセット
PRDは違ったものを示しているか？
NO
YES
3

（図2－3）の中にあるSYNTHESIZEルーチンがセマンティックスの記述部分である。また，必要に応じてLexical Analyzerである SCANルーチンを組み込む必要がある。

XPL言語

XPL言語はPL／Iのサブセットに若干の変更を加えたものである。PL／Iに対する制限なり変更は，PL／Iの仕様の中でコンパイラ作成に必要な機能を残し，PL／Iの仕様にはないが，あった方が便利な機能を追加するという方向でなされた。

PL／Iから取り除いた仕様

① すべての変数は，それが使われる以前で宣言しなければならない。
② 配列の下限は，0である。
③ 属性等を指定する時などに，省略形はない。（CHARACTERをCHARと指定してはいけない）。
④ データのタイプはFIXED，CHARACTER，BITの3種類である。
⑤ 属性はカッコでくくって指定することができない。
⑥ ビットストリングのとりあつかいが違っている。
⑦ 文字ストリングはすべてVARY 属性を持ち0文字目から始まる。
⑧ DOループは正でなければならない。
⑨ PROCEDUREはnon—recursiveであってvalue parameterのみを持つ。
⑩ DO，IF等はreserved wordで名標として使えない。

PL／Iを更に拡張した仕様

① LITERALLY 属性を持つデータがある。……指定された変数をコンパイル時に定数におきかえる。

② ２，４，８，１６進数の表現ができる。

③ SHL(シフト・レフト)，SHR(シフト・ライト)という標準関数がある。。

④ FILE，INPUT，OUTPUTと書く(代入文等に)ことにより入出力ができる。

⑤ CASEステートメントにより一連のステートメントの一部を実行できる。

⑥ INLINEにより機械語の命令を書いて実行できる。

このＸＰＬ言語によって必要とするコンパイラのSemantics を記述するわけである。すなわちコンパイラのおこなうべき処理手順をプロシーデイアルに記述し必要なオブジェクトコードを出力することがＸＰＬ言語の役割である。

この言語の特徴はＤＯステートメントの拡張によりgo to ステートメントをほとんど使わないでコーディングできるために，他人がプログラムの解読する場合に非常にわかりやすい。

このシステムは，作成にあたって次のようなブート・ストラッピングによる方法がとられた。

まず次の記号の定義をする。

(1) (a→b，c)はａからｂへの変換プログラムがｃでコーディングされている。

(2) (a→b，c)×(c→e,f)→(a→b，e)は(a→b，c)のプログラムをトランスレータ(c→e，f)で実行させることにより(a→b，e)ができあがることを示す。

ＸＰＬのインプリメントにあたってまず

(ＸＰＬ→S／３６０ML，ALGOL)

(ＸＰＬ→S／３６０ML，XPL)

の２つが作られた。そして

① （XPL→S／360ML，ALGOL）×（ALGOL→B5500ML，B5500EX）→（XPL→S／360ML，B5500ML）

② （XPL→S／360ML，B550ML）×（B5500ML→B5500EX，B5500EX）→（XPL→S／360ML，B5500EX）

③ （XPL→S／360ML，XPL）×（XPL→S／360ML，B5500EX）→（XPL→S／360ML，S／360ML）

④ （XPL→S／360ML，S／360ML）×（S／360ML→S／360EX，S／360EX）→（XPL→S／360ML，S／360EX）

注　S／360　：IBM／360
　　B5500　：Burroughs 5500
　　ML　　：機械語
　　EX　　：実行形式

のステップにより，IBM／360 にインプリメントし，③，④のくり返しをおこなって機能拡張をおこなった。

したがってXPL自身がXPLで作られているわけでありその中のXCOMコンパイラに関していえば，SKELETON が作り出すシンタックス・テーブルの大きさは約3.8K＋α（αはXPL言語の変数領域アロケーション方法により増加する部分であるが0.5Kバイト以下である）バイトでおさまり実用上問題ない大きさである。コンパイルスピードに関しては，資料がないので不明であるが，構文解析手法から考えてそんなに遅いとは考えられない。

XPLを使ってコンパイラを作成するのにどのくらいの 工数が必要であるかということに関しては，スタンフォード大学のシステムプログラミングコースの学生が，このシステムを使うことにより3～6週間でコンパイラを作っているとの報告がされている。

XPLの欠点と思われる点は，XPL言語にはPL／1でいうEXTERNAL

属性がないので，新しいコンパイラを作成する場合に，すべてのソースプログラムを毎回コンパイルしなければならないこと，および<identifier><number>以外もterminal記号としてあつかえるようにした方がよい点であろう。

ＸＰＬに関しては，内部仕様書，リストが公開されており調査するのに非常に便利である。

参考文献

1 W.M.Mckeeman, J.J.Horning, D.B.Wortman
A Compiler Generator
PRENTICE-HALL INC.
（図２－２）は上記文献より引用させていただいた。

2. W.M.Mckeeman, J.J.Horning, E.C.Nelson,
D.B Wortman
The XPL compile Generator system
FJCC 1968 PP617～635
（図２－１）は上記文献より引用させていただいた。

3. W.M.Mckeeman
An approach to Computor language design
Stanford Univ. Technical Report NO.CS48, 1966

2.2 MPL／I

MPL／I は作成したいprocessor の仕様を記述する部分と，それを実行形式プログラムに変換する部分より構成されている。

MPL／I のSemantics 記述はPL／Iのサブセットによるプロシージャルな方法によっている。一方Syntaxの記述は，BNFに似てはいるが，次の点が違っている。

① 生成規則の右辺が全く同じものは，1つの式にまとめその選択はユーザの定義するprocedure による。

例

<array name>::=<identifier>;

<function name>)::=<identifier>;

なる構文があった時

FN(<array name>|<function name>)::=<identifier>;

```
FN: PROC;
    .....
    RETURN(N);
    END;
```

procedure FNでは，<identifier>を調べ<array name>であったならば1，<function name>ならば2を返す。

② reference variable の使用

<$>はすべての記号を代表するもの，<n>（nは自然数）は生成規則の右辺の記号で左から順に1より数えるものとした時<$>は右辺<n>を左辺に使用できる。

例

<2>::=<A><$>;

BNFによれば起こり得る場合すべてを記述するが，MPL／I においては，～以外のものという表現を許す。

例

＜C＞：：=＜A＞；

＜D＞：：=＜A＞＜B＞；

という時これを次のように記述することができる。

＜C＞＜2＞：：=＜A＞＜$—＜B＞＞；

前者の意味は，＜A＞のあとに＜B＞以外のものがきたときにこれを＜C＞＜B以外のもので，そのもの＞とおきかえることを指定する。

reference variable関係は，MPL／I の構文解析アルゴリズムにおいて，あともどりが起こることを避けるためにある。

つぎにMPL／Iによる例を示す。

```
/* CONTROL */
DECLARE KWORD(4) CHARACTER(15), TYPE(4) FIXED BINARY;
          KWORD(1) = 'TOM';       TYPE(1) = 1;
          KWORD(2) = 'Y';         TYPE(2) = 1;
          KWORD(3) = 'SMITH';     TYPE(3) = 2;
          KWORD(4) = 'LI';        TYPE(4) = 2;
READ: GET LIST(CH);
/* INITIAL PRODUCTIONS */
<L> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z;
''  ::= '';
/* PRODUCTIONS & SEMANTIC ACTIONS */
1 <WD> ::= <L>'';                 R(1) = V(1);
2 <PW> ::= <L><L>;                R(1) = V(1)||V(2);
3 <PW>::= <PW><L>;                R(1) = V(1)||V(2);
4 <WD> ::= <PW>'';                R(1) = V(1);
5 KW(<PN>|<LN>)  ::= <WD>;        R(1) = V(1);
6 <2> ::= ''<$>;                  R(1) = V(2);
7 <S> ::= <PN><LN>;               PUT EDIT (V(1),V(2))(A(15),A(15));
/* PROCEDURES */
KW: PROCEDURE;
          DO I = 1 TO N; IF KWORD(I) = V(1) THEN RETURN(TYPE(I));
          END;
          GO TO READ;
END KW;
```

図２－４　ＭＰＬ／Ｉの例

ＭＰＬ／Ｉ に提案されたものの中で考慮するに値するのは，同一右辺を持つた生成規則のとりあつかい方である。これは順位文法において制限されていることであるが，実際にＡＬＧＯＬのSyntax 定義等にもちいられており，この手法をもちいることにより，とりあつかいができることである。

参 考 文 献

1. E.N.Ferentzy,J.R.Gabura

A syntax directed processor writing system

FJCC 1968 PP.637～647

（図２－４）は上記文献より引用させていたゞいた。

2.3 TREE META

TREE META は，ユタ大学において１９６９年に開発されたMETA 系列のコンパイラ・コンパイラシステムであって，ユーザが定義したSyntax 及びSemantics を読み込んで，その言語のコンパイラをFORTRANでコーディングしたものとして出力する。

このシステムを使用して作り出されるコンパイラは，原則として2 Pass となる。すなわち，第1 Pass で入力記号を見ながらStart記号より始めて下向きに構文を解析し，その構文に対応するtree を作り上げる。第2 Pass では作り上げられたtree を drivenしながらobject code を発生するという形をとっている。このためメタ言語の記述には，Syntax記述及びtree 作成記述部分とobjectコード発生部分の２つを指定しなければならない。

Syntax記述

このシステムのSyntax記述は，extendedＢＮＦを修正した形でおこなう。これがＢＮＦ記述と相違する部分をとりだせば，次のとおりである。

① nonterminal 記号は＜　＞でくくらない。

② terminal記号は，１文字ならば▼のあとにその文字を書き，１文字以の時は〝をその前後につける。

③ ：：＝の代りに＝を使う。

④ ｜の代りに／を使う。

⑤ 規則の最後に；をつける。

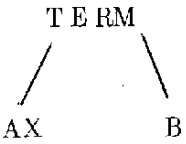
これ以外に，下向き構文解析であるため，左回帰の禁止をし，その代りとして，$（　）の形で（　）の中が0回以上くり返すことを指定できる。　またnonterminal記号的なあつかいとしてbasic recognizer及び，EMPTY（null にあたる）を指定することができる。basic recognizerとは，.ID，.SR，.NUM，.LET，.CHR，.DIGの６つであり，それぞれ，identifier（英字で始まる英数字列），string（文字列を〝でくくったもの），

数（数字の列），英文字，文字，数字を示す。

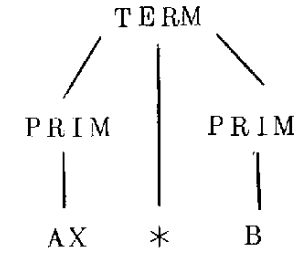
Syntax 部分には，上記のものに：のあとに tree 作成部分を書くことができる。たとえば

```
TERM=PRIM$(〝＊〃PRIM:MULT{2});
PRIM=.ID/.NUM;
```

ここでMULT{2}は，node MUITから２つのnode がでていることを示す。たとえばAX＊B というストリングを読み込んで tree を作れば次のようになる。



Syntax 部分に tree 作成部分を書くのは，Syntax 記述だけで作り上げられる tree



よりも冗長度が少く，tree としてとりあつかいやすいためである。

オブジェクトコード発生部分

それぞれの出力規則は，node のテストおよび出力記述より構成される。

node のテストは次のようなものがある。

① あるnode からでているnode の数のテスト。

ADD{－,－} はＡＤＤというnode から２個のnode がでているかどうかテストする。この時でているnode を左から順に＊１，＊２，…として参照することができる。

② node の一致

ADD{－,＊1}あるいはADD{＊2,－} は，第1 node と，第2node が等しいかどうかテストする。

③ basic recognizer とのテストは，その recognizer 名を指定する。

ADD{.ID,.NUM}は第1 node が identifier, 第2 node が number であるかどうかテストする。

④ リテラルとのテストは〝 リテラル 〃を書く。

EQ{－,〝AB〃} は第2 node が〝AB〃であるかどうかテストする。

⑤ generateされたラベルは，別の出力規則に持ちこされるが，このテストのために#number が使われる。

⑥ node のテストに特別の node 名を書くことができる。

STORE{－,ADD{＊1,〃1〃}}は例えばX=X+1のように,同一のiden－tifeir等と，リテラルの1がADDというnode にあるかどうかテストできる。

⑦ テストの省略は，node のあとに／をかく。

⑧ node 名を省略して同一node 名の選択子とすることができる。

ADD{－,＊1}=>

{－,－}=>

これはADDの第1 node と第2 node が等しければそのあとに書かれた出力記述を実行し，そうでなければ次の{－,－}のあとの出力記述を実行する。

出力の記述は次のものがある。

① ＼で改行を示す（カードの終了）

② ,でタブを示す。（カラムをあわせる）

③ リテラルストリングを書くことによりそのリテラルを出力できる。

④ node ：X （XはS，L，N，C）

＊1：S　＊1のリテラルを出力する。

＊1：L　＊1のストリングの長さを10進数で出力する。

＊1：N　もしnodeがストリング領域ならばそのインデックス・ポインタを，CHRならばnodeに対する10進コードを出力する。

＊1：C　もしnodeが.CHR，.LET，.DIG recognizerで作られたものならばその文字を出力する。

⑤ #numberはgeneratedラベルを出力する。

⑥ 出力記述の中にもテストを入れ，選択子をもうけることができる。

ADD{－,－}=>TNEG{＊2}MINUS{＊1,＊2：＊1}／＊1〝＋〃＊2；

これはnode ADDの第2 nodeがTNEG nodeテストでtrueならばMINUS{＊1,＊2：＊1}を実行し，そうでなければ＊1〝＋〃＊2を実行する。

このほか出力の記述部分には，treeの変換，エラーの処理，他のnodeテストの呼び出しを書くことができる。

コード発生部はnodeテストと出力の記述を=> でつなげて書いたものである。

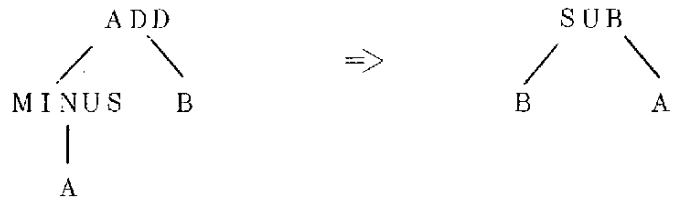
例

① DIVIDE{－,－}=>＊1〟／ ＊2；

はDIVIDEという名前のnodeから2つのnodeがでているかどうかテストして，trueならば，第1 node／第2 nodeというcodeを発生する。

② ADD{MINUS{－},－}=>SUB{＊2,＊1：＊1}

は下図の左のtreeに変換するものである。



ここで，＊n：＊mは現在処理中のnodeのn番目のnodeについているもののm番目のnodeをさす。この逆にtreeをさかのぼるものに↑n＊mがありこれは現在処理中のnodeのn代前のnodeのm番目のnodeをさす。

例　BNFで記述した時次のような形になるものをTREE METAで記述しSemantics記述を加える。

```
<EXP>::=<TERM>|<TERM>+<EXP>|<TERM>−<EXP>
<TERM>::=<FACTOR>|<TERM>＊<FACTOR>|<TERM>／
          <FACTOR>
<FACTOR>::=<PRIM>|−<FACTOR>
<PRIM>::=<identifier>|<number>|(<EXP>)
```

```
TREE META
 EXP=TERM(\+"EXP:ADD{2}/\−"EXP:SUB{2}/.EMPTY);
 TERM=FACTOR$(\＊"FACTOR:MULT{2}/"/"FACTOR:
       DIVD{2});
 FACTOR=\−"FACTOR:MINUS{1}/PRIM;
 PRIM=.ID/.NUM/\("EXP\)";
 ADD{−,−}=>TNEG{＊2}SUB{＊1,＊2:＊1}/＊1\+"＊2;
 SUB{−,MINUS{}}=>ADD{＊1,＊2:＊1}
     {−,−}=>＊1\−"＊2;
 TNEG{MINUS{}}=>.EMPTY;
 MULT{−,−}=>＊1\＊"＊2;
 DIVD{−,−}=>＊1\／"＊2;
```

TREE METAにおける特徴は，Semantics 記述の簡単さと，tree のおきかえという形をとるためにObject code の Optimize まで考えることができる点にある。しかしながらTREE META には他のMETA にみられるようなアセンブラ等の出力に対する考慮が欠けているのでこの点の充実が望まれる。

参考文献

1. C.S.Carr,D.A.Luther,S.Endmann
   The TREE META Compiler-Compiler System:A META Compiler System for the Univac 1108 and the General Electric 645
   TR 4-12（RADC-TR-69-83）

2.4 META PI

このMETA PI というシステムは，オン・ラインで，インタラクティヴなコンパイラ・コンパイラとして，１９６６年からプリンストンのＲＣＡ研究所で開発に着手したものである。このシステムは，基本的な言語構成としては，名前からも考えられるように，D.V.Schorre とそのUCLA computing facilityにおける協力者たちによって作られたＭＥＴＡシリーズを用いており，したがって厳密にはsyntax－directed Symbol Processor の範ちゅうに属するものである。さらに，ＭＥＴＡタイプのコンパイラと同じく，基本的な解析のアルゴリズムとしては，top － down，left to right，deterministicなものを採用している。

MITA PI が，他のMETA シリーズものと異なる最大の点は，インタラクティヴであるということである。すなわち，このシステムは同研究所の開発したＢＴＳＳ（Basic Time Sharing System Version Ⅱ）というオペレーティングシステムの下で働くように設計されており，開発当初の目的としては，FORTAN ＩＶ に基づくインタラクティヴな言語FORTRAN PI というコンパイラを作成するにあたって，先ずこのコンパイラ・コンパイラMETA PI を作り，できうるかぎりMETA PI を用いて，FORTRAN PIを書くことであった。これに使用した機種はＳＰＥＣＴＲＡ ７０／４５であって同シリーズの７０／４６でもオペレートできるようである。なお始めの目的であったFORTRAN PI は既に１９６８年 以前にでき上り，そのあとこのMETA PI を用いてDartmouth ＢＡＳＩＣをインプリメントしたことが報告されている。

META PI のステートメントは，他のＭＥＴＡ コンパイラと同じく，ＢＮＦに手を加えたextended BNF を用いている。このextended BNFはBNF とは記述方法の細かい点でも多少異なるが，大きな相違点は次のようなものである。

1. extended BNFによる Syntax の記述の中には Semantics （すなわちcode generation）を含めることができ，従ってソース・ステートメン

トのscanを実行中に，オブジェクト・コードをgenerateでき，scanの終了と同時にcode generationを終了できる。

2. くりかえしやくくり出しを，表現するために$および(　)を用いる。$の意味としては(　)の中にある任意の記号が0回以上続くということをあらわす。

たとえば　BNFステートメント

＜A＞：：＝　B|＜A＞C|＜A＞D

をMETA PIで表現すると

A：＝：B：　$(：C：／：D：)

のようになる。ここでBNFの：：＝は，META PIでは：＝，BNFの|はMETA PIでは／，BNFで＜　＞で囲んで示すnonterminalはMETA PIでは＜　＞をつけずに表わし，terminalを示すためにはMETA PIでは，：（コロン）で両側を囲んで表現する。

META PI	BNF
：＝	：：＝
／	|
ABC	＜ABC＞
：ABC：	ABC

3. META PIの創成するコンパイラがソース・ステートメントのscanを行なう場合，特定のコマンドを用いるとscanのある点からcode generationを逆戻りすることができる。但し，META PIそのものはdeterministicな言語であるので，META PIでコンパイラを作成する場合には，逆戻りは必要ないのである。

META PIにおけるextended BNFと通常のBNFとを比較するために

Dartmouth BASIC の2,3のステートメントをふたつの表現で書いてみると次のようになる。但し，この例では，META PIからSemantics に関するオペレーション部分は削除してある。

BNFを用いた例：

<READ statement>::=READ <read list>

<read list>::=<variable>|<read list>,
<variable>

<FOR statement>::=FOR<simple variable>
<expression><OPTEX>

<OPTEX>::=STEP<expression>|<EMPTY>

同じ例をMETA PI で書いた場合：

READST:=:READ: READLST

READLST:=VAR$(:,: VAR)

FORST:=:FOR: SIMVAR :=: EXP :TO:
EXP(:STEP:EXP/.EMPTY)

META PIのステートメントは次の3つのタイプの要素を含んでいる。

1. Syntactic な要素：これは，ユーザーのコンパイラ内のコードにコンパイルされ，ユーザーのコンパイラに対するソースの中のSyntactic な要素と比較されるものである。これはユーザーのコンパイラの syntax checker などに用いられる。
2. Sematicsの要素：ユーザーのコンパイラ内のコードにコンパイルされ object code を generate するのに用いられる。
3. META Syntactic 要素：これもユーザーのコンパイラ内のコードにコンパイルされ，逆戻りの機能を用いて，入力ソース・ステートメントをもう一度定義し直す時に生ずる矛盾を解決するのに有効なものである。

ユーザーはこれらの３つのエレメントを組合わせることによって独自のコンパイラを作成するようなMETA　PIのステートメントを書くことができる。META　PI ステートメントの一般型は次の如くである。

LABEL：＝expression

ここで，左辺は右辺の式を参照する時に用いる一意的な identifier である。たとえば，数字をMETA　PIのステートメントで定義すると

DIGIT：＝：0：／：1：／：2：／：3：／：4：／：5：／：6：／：7：／
：8：／：9：

となる。右辺にある式は，ユーザーのコンパイラの中のコードにコンパイルされるものでこの式は自分自身に対する間接的あるいは直接的な参照を含みうる回帰的なものであってもよい。

META　PIは，その式をユーザーのコンパイラの中のcode にコンパイルする際その式が後で呼び出された時に次の３つの結果のいずれかとなるようにgenerateするのである。

1. True ：インプットをscan した結果がその式を満足する場合で，呼び出されたルーチンは truth indicator が true を示すようセットしinput pointerを正しくscan できたデータの後に移してリターンする。
2. False ：インプットがその式を満足しないときでinput pointer は変らない。truth indicator は falseを示すようにセットされる。
3. Error：例えばGO　TO　20.3というステートメントのようにその式の最初のGO　TOの部分は正しく識別されるのに，途中から誤りであることがわかる場合，これが生じるとエラー・ルーチンが呼ばれ，input pointerは，一部更新され，正しくscan された最後の文字のあとに？を挿入する。

以上の3つのconditionは，META PIの式によってユーザーのコンパイラ内にgenerateされるコードの働きについて述べたものである。次にこれらのexpression を構成する要素をいくつか紹介することにする。

Syntacticな要素

：XXX…X：　Xは任意のストリングを表わす。このSyntactic な要素はそのストリングと現在のインプットを比較するためにユーザーのコンパイラ中にコードをgenerateするものである。
例えば，前述のDIGITステートメントの中では，0か，1か2か，…をテストするためのコードがgenerateされる。

ABC　これは，この名前（ここではABC）を持つルーチンを呼び出すようなコードをユーザーのコンパイラの中に作り出すものである。このルーチンはユーザーによってMETA PI で既に書かれているものと仮定する。INUM:=DIGIT$DIGITのような使い方をしてもよい。また，このルーチン名は，現在存在しているFORTRAN PIのルーチンの名前であってもよい。この使い方はMETA PI の中のサブルーチンをlink する方法にはもうひとつピリオドを先頭につけるやり方もある。このピリオドを先頭につける場合はそのルーチンが回帰的でなく，truth indicatorを返さねばならないことを意味している。ピリオドのない時は回帰的に呼び出すことも可能である。いずれの場合にも，truth indicatorをセットする。

ID　これはidentifierに対するテストであり，IDルーチンに対してlink されるコードがgenerateされる。先頭にピリオドがついているので自分自身をlink することはできない。

.EMPTY　これはtruth indicatorを強制的にtrue にセットする特殊な要素である。

.ＩＮＴ　　FORTRANの整数についてのテストである。

.ＮＵＭ　　これは，次のようなMETA　ＰＩステートメントによって規定される数値に対するテストである。

ＮＵＭ：＝＄ＤＩＧＩＴ（：.：／.EMPTY）＄ＤＩＧＩＴ（：Ｅ：（：＋：／：－.EMPTY）ＤＩＧＩＴ（DIGIT／.EMPTY）／.EMPTY）

.ＬＫＵＰ　これは，ＬＫＵＰルーチンにlinkするようなコードをgenerateするものである。このＬＫＵＰルーチンは，検出された最近のinputに対してlabelテーブルをscanする。labelテーブルの項目は文番号あるいは変数名である。各項目には，このほかに適当な制御情報を含んでいる。このルーチンからは次の３つの結果の１つを返す。

① そのインプットがlabelテーブル中にあり，代入する。memory locationを定義する。

② そのインプットが，labelテーブル内にない。

③ labelは見つかったがそのmemory locationはまだ定義されていない。これはまだ出現していない文番号を指すGOTOステートメントなどである。

.ＴＹＰＥ（：NNYY：）　このルーチンは，うけ渡されたインプットをlabelテーブル中で調べ，そのタイプを示すバイトが実引数ＮＮＹＹによって識別されるクラスにあるかをテストする。このルーチンはmixed mode errorのチエックなどに用いる。

Semanticsの要素

ＭＥＴＡ　ＰＩではSemanticsは，ＭＥＴＡ　ＰＩのステートメント中に埋めこまれ，このSemanticsは，code generationに用いられるものである。Semanticsは，

1. Semantic commands

2. Semantic operations

の２つの部分から成り，このうちSemantic operation は Semantic command の中に含まれ，一般型は，Semantic — command(Semantic — oprrations)である。

Semantic command

各 Semantic command はコンパイラによってgenerateされるcodeに直接効果を及ぼす。ＭＥＴＡ　ＰＩは入力ステートメント中のSemantic command に出会うとユーザーのコンパイラ中に，それらが，object program をgenerateするのに必要なobject code をgenerateする。Semantic commamd は基本的なものとしては５つある。

.OUT(…)　このコマンドはoutput area（そのユーザーのコンパイラによってコードが創成される一時的なエリア）の現在の内容を内部的な形に変換してユーザーのcode area に入れるものである。code area はcomputerが実行するobject code が入るエリアである。そのoutputそのものは括弧内で指定されるSemantic operation によって作られるものである。このコマンドはその中に含まれるSemantic operation の構造に従って３つのaction をとりうる。

① 第１文字目が英字，数字のいずれでもなければ，さいごの：（コロン）の対までの文字をそのままcode area にコピーする。

② 第４文字目がピリオドまたはスペースならば，そのoutputがインデックス・レジタスを用いる命令であり，第４文字目以降に記号番地をもつものと仮定する。そして，この記号番地を

labelテーブルから引いて，そこに含まれる情報からマシーン・アドレスをgenerateする。

③ 上記以外の場合，その文字ストリングは機械コードであると考えてそのまま変換してcode area へ入れる。

.LABEL(…) これはoutput area の現在の内容を取ってきて，labelテーブルに入れるものである。このlabelが定義済みであれば，エラーとし，このlabel は code areaのロケーション・カウンタの現在の値に関連している。

.ING(…) output area の内容を無視（削除）するもので，これは或るSemantic operation が code を generateする時，レジスタも解放しようとするような副作用を生み出すので有用である。

.NOP(…) このコマンドは他には何もせず，Semantic operation 効果を生ずるだけである。そのSemantic operationの結果はoutput area に残っている。

.DO(…) これは特殊なコマンドでMETA PIにすぐに( )内の命令を実行させるものである。

Semantic operation

Semantic operation は output area に code を generateするのに用いられるもので，これらのcode は記号的（シンボリック）な形をしているので，そのまま実行することはできない。また，これらのoperationは通常はinput pointer や truth indicator の変更は行 わない。output area内の次の有効な locationを指すためにはひとつのポインタを用いていて，このポインタは各Semantic operation のあとで更新する。以下 semantic operation を列挙する。

:CCC…C: コロンにはさまれているストリングをそのままoutput areaの続きに入れる。 syntactic な要素と同じ形であるが,seman-

tic command の内側か外側かで，いずれであるかは容易に判断できる。

＊　output area の内容に現在のインプットを続けて入れる。これは普通は有効な .ＩＤ のテストと共に用いられるものである。

S　push down list の中にout areaの中の現在の内容のコピーを退避して，リストを push する。

R　push-down list の一番上のものをoutput area に返し，リストを pop up する。

I　push-down list の一番上のエレメントを無視(pop up)する。

X　push-down list の上の二つのエレメントを入れかえる。

＊1　#で始まるグローバルに一意的な4バイトの文字ストリングをgenerateする。このストリングはローカルには一定でgenerate された code 内のロケーションにlabelをつけたり，これを参照したりするのに便利である。

output code 内でSemantic routine のあるものを用いればレジスタを使用することもできる。汎用レジスタと浮動小数点レジスタに対する push-down list のタイプは，実行時に保たれている。
このようなSemantic operationに対するレジスタは，汎用レジスタが6個浮動小数点用が4個である（ＳPECTRA 70 シリーズ）。これ以上のレジスタを要する場合には，コーディングによって自動的にレジスタの退避や restore をインプリメントするようにgenerateでき，それには次のようなSemantic routine を用いる。

ＯＦ　現在の汎用レジスタをoutput する。

Ｏ　現在の浮動小数点レジスタをoutput する。

＋　次の空いている汎用レジスタをoutput し，それを現在のものと

する。

＋2　次の空いている浮動小数点レジスタをoutputし，それを現在のものとする。

－　2個の汎用レジスタをoutputする。第1のものはひとつ前のレジスタで，第2のものは現在のレジスタである。このoperationが終了すると，ひとつ前のレジスタを現在のものとする。このoutputは，常に数値の対であり，そのフォーマットはSPECTR-A 70 におけるレジスタ同志のoperationに都合のよいようになっている。

－2　浮動小数点レジスタであること以外は－に同じである。

MEAT Syntactic command

以下に述べるコマンドは，ユーザーのコンパイラにおける逆もどりの機能をゆるすために設けるものである。

.LATCH(name)　これは(　)内の名前のルーチンを呼び出すようなコードをユーザーのコンパイラ内にgenerateするものである。さらに，もしlatch されたルーチンあるいはこれに続いて呼び出される任意のルーチンがエラールーチンに行きついたならば逆もどりを行なう。

C　このコマンドはSemantic operator の書けるところには，どこでも置くことができ，呼び出し側のルーチン内の .LATCHの発生をおさえるようなコードをユーザーのコンパイラ中にgenerate する。このコマンドはふつうは，subexpression内の初期的なあいまいさが解決したときに用いるものである。たとえばFORTRANのDOステートメントで，が見つかったときや論理IF文で論理演算子が見つかったときである。このときは，エラー

が生じても逆もどりは起らずエラーのlocationを指適する。

.CLAMP このコマンドはCを書けるところならばどこでも書くことができ先行 .LATCHで有効なものをすべてsuppressしてしまうことをコンパイラに指示するものである。

次に，META PIの例を少し挙げることにする。

〔例1〕 FORTRAN PIでは各ステートメントの区切りのセミコロンのあとにコメントをつけることができるが，この特徴をやめて，その代わり，1行にセミコロンで区切ることによって複数個のステートメントを書けるようにするには，次のようなMETA PIのコマンドを書けばよい。

USERCC:=LABST.NOP(.CLAMP)$LABST

ここでLABSTは FORTRAN PIのラベル付け可能なステートメントの定義を指すものである。

〔例2〕 次にBASICのReadステートメントをMETA PIで書くと次のようになる。

READ:= :READ: RID$(:,:RID):;:

このとき，META PI はこのステートメントを左から右へとscanし次のようなコードをgenerateする。

(1) 語READに対するテスト

(2) RIDの定義とのリンク。この定義はBASICに関するMETA PIの記述中に含まれるものである。

(3) readのidentifierのあとの，に対するテストがくりかえし行なわれるような命令

(4) 行の終了を示す；に対するテスト

このMETA PI ステートメントは SyntacticなものだけとなっているがこのREADステートメントに関するSemanticsはRIDの定義の中に行なわれているのである。

前にも述べたようにMETA PIは開発当初，FORTRAN PIの作成を目的としたものであったがこのMETA PI を他のコンパイラをインプリメントするのにも用いようという試みがなされた。その結果BTSS Ⅱ用のBASICがMETA PI で記述して作成された。この作成期間は1968年4月15日～同年6月6日で，

Total man hours　　96時間

Console hours　　33時間

Processor hours（CPU）1時間以下

の時間を要したとのことである。

でき上ったFORTRAN PIやBTSSⅡ用 BASICの処理効率がどの位のものであるかは明らかではないが，META PIのようなon-line のコンパイラのインプリメンテーションにも，また個々のユーザーの必要に応じてコンパイラを修正したりするのにも有力であると思われる。

参考文献

1. J.T.O'Neil Jr.

META－PI An on-line interactive compiler-compiler

FJCC 1968 PP201～218

2.5 VITAL

このシステムは，１９６７年初頭にＭＩＴの Lincoln Laboratoryにおいて開発されたコンパイラ・コンパイラで，ＴＸ－２というコンピュータに対してインプリメントされ，すでに実用に供されているものである。システムの設計思想については，Ｊ．Ａ．Feldman やＪ．Ｅ．Curry の Production Language （ＰＬ）およびFormal Semantic Language （ＦＳＬ）をとり入れ，このシステムの中でもSyntaxを記述する言語をＰＬ，Semanticsを記述するものをＦＳＬと呼んでいる。このようにＶＩＴＡＬシステムは，基本的な方向性としては，ＰＬやＦＳＬの思想を持っているがタイムシエアリングシステムのもとで動いていることがもともとのＦＳＬとの最大の相違点である。

我が国においても日立中央研究所でＰＬ／ＩＷ という言語を用いてＨＩＴＡＣ ５０２０ に対してＴＳＳ用ＴＩＴＡＬ システムに開発したことが１９７１年に発表されている。

ＶＩＴＡＬ という名前は（Variably Initialized Translator of Algorithmic Language）の略で，コンパイラ作成の際の細かな仕事を自動的に行なうことを目的としている。

このシステムの構成は次のようになっている。

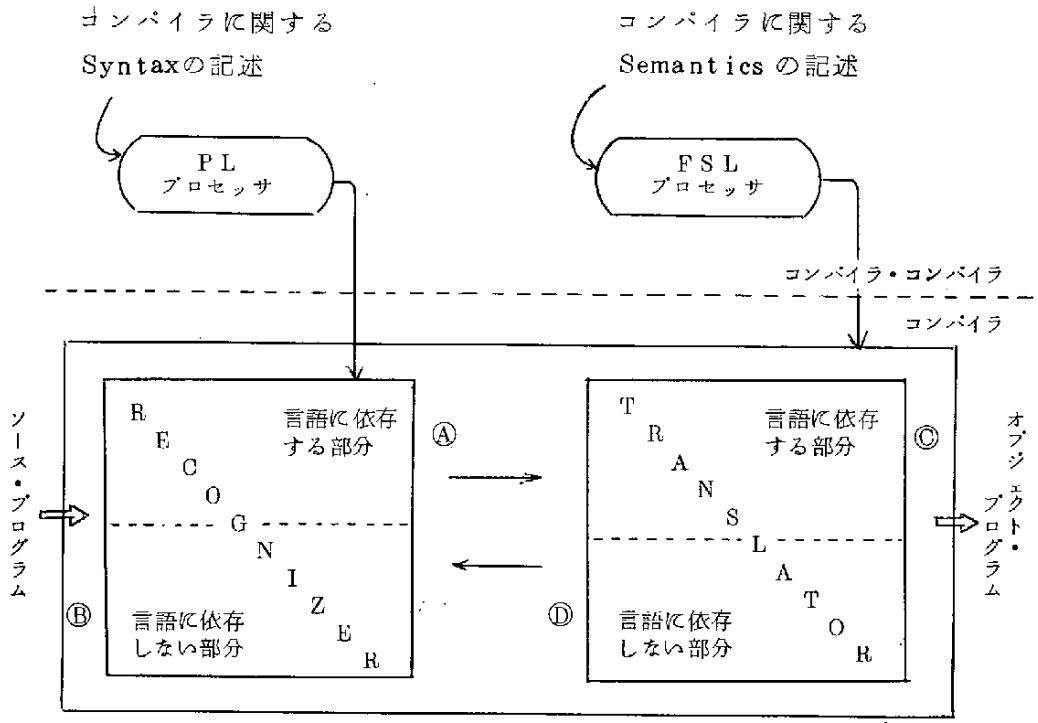


図２－５　コンパイラ構成図

すなわち，ＶＩＴＡＬ システムでは，コンパイラ・コンパイラはＰＬプロセッサとＦＳＬプロセッサの２つから成り，さらにその作り出すコンパイラはRECOGNIZERとTRANSLATOR の２つの部分に分けることができる。そして，その相互の関係は次の如くである。

ＰＬプロセッサは，現在作成しようとしているコンパイラに関するＰＬ（Production Language）で書かれたSyntax の記述を受け取ってprodnction tableを作り出す。これは，コンパイラ言語に依存するもので（図２－５）のⒶに当り，これにコンパイラ言語の体系には依存しないproduction interpreter Ⓑがつけ加えられてRECOGNIZERができ上る。このRECOGNIZER は，コンパイラにインプットされるソース・プログラムのSyntactic な性質を認識するものである。

これに対して，ＦＳＬプロセッサは，そのコンパイラに関するＦＳＬ（Formal Semantic Language ）で書かれたSemantics の記述を受け付けて，Semantic routineを作り出す。Semantic routine は，コード作成（code generator）用のサブルーチンの呼び出しの形であり，コンパイラ言語に依存するもので図の©である。これに実際にコードをgenerateさせるためのサブルーチン群Ⓓが加えられて，ＴＲＡＮＳＬＡＴＯＲが作られる。このサブルーチン群は，すべてのコンパイラ言語に共通なものであり，従って言語に依存しないものである。そしてＲＥＣＯＧＮＩＺＥＲとＴＲＡＮＳＬＡＴＯＲは相互に働きかけ，最終的にはＴＲＡＮＳＬＡＴＯＲによってアウトプットとしてのオブジェクト・プログラムのコードが作られるのである。

なお，ＰＬプロセッサとＦＳＬプロセッサはコンパイラとしては，前の図のように同じ構造を持っており，これらはハンド・コーディングとブート・ストラッピングによって創成された。その手順は，先ず言語に依存しない code-generator と production interpreter とがハンド・コーディングによって作られ，次にＰＬプロセッサのSemantics と productions がハンド・コーディングで作られ，最後にＦＳＬプロセッサのSemantics がハンド・コーディングされ，そのproduction をＰＬで書くことによって作成されたとのことである。

次に，コンパイラのオペレーションに関して少し説明し，そのあとVITALにおけるＰＬとＦＳＬについて簡単にまとめることにする。

コンパイラのオペレーション

Ｍａｉｎスタック

ＶＩＴＡＬのＲＥＣＯＧＮＩＺＥＲとＴＲＡＮＳＬＡＴＯＲは，Main スタックと呼ばれる１個の push-down スタックを用いて処理を行なう。このMainスタックは，インプット・テキストおよびそれを処理した結果としてできる解析列（parse ）を push down したり，pop up したりするものである。このスタ

ックのエントリは3つの連続したマシンワードから成っており，1ワードがsyntax word,残りの2ワードがsemantic word，として用いられる。

TRANSLATORとRECOGNIZERの間の情報のうけわたしは，このMain スタックを通じてのみ行なわれる。

インプット・テキストの処理

VITAL の作り出すコンパイラでは，インプット・テキストをその中に含まれるsymbolを認識しながら1度だけscanする。これらのsymbol はproduction によってreserved wordとして定められたもの（例えばFORTRANのDOや＋など）とその他のsymbol（すなわちidentifier）の2種類に大別できる。

reserved wordが見つかるとそのsymbolに対応する表現形式（representation）がmain スタックのsyntax wordにpush down され，identifier であることが認められた時には，そのidentifierの代わりに"identifier" であることを示す項目がpush down されるのである。この時点では，push down したsyntax word に対応するsemantic word は，関係ないものとして何も入れない。

production の一部としてexpected stack configurationのテーブル即ちスタックの中の予想される状態のテーブルが在り，上記のscan のあとmain スタックの上の方にある2,3のエントリのsyntax word をこのテーブルと比較する。一致するものが見つかった場合には，この一致したconfigurationに対応しているいくつかのルーチンが実行されるのである。
このルーチンは

(1) main スタックの一部の修正
(2) セマンティック・ルーチンへのjump
(3) エラーメッセージの記録

(4) さらに一つ以上のsymbolのscan

(5) mainスタックと比較すべき次のconfigurationの指定のうちのいずれかを行なうことができる。

このscanと照合という過程は，入力プログラムのすべてのsymbolをscanしてしまうか，あるいはproductionかSemantics内の終了コマンドが現われるかのどちらかが起るまで続けられるのである。

PL(Production Language)

Production Languageは，VITALコンパイラ・コンパイラ システムにおいて，コンパイラのSyntaxを指定するためのproduction を記述するのに用いる言語である。

そしてこのproduction は，(1)インプット・テキストを1度だけscanしてmainスタックの継続的な状態を作り出し，(2)スタックの中にあるsyntacticな要素に対応するSemantic routine にコントロールを移す働きがある。

PLプログラムの構造は，初めに宣言を書き，そのあとにproduction を続けて書いていけばよく，プログラムの終りにはENDという記号をつける。

PLにおける宣言文の中から二，三のものを挙げてみることにする。

RES <argument>

このargumentとしてはVITALのsymbolをスペースで区切って書くことができる。この宣言はargumentとして与えたsymbolが今作ろうとしているコンパイラのソース言語においてreserved wordとして用いられることを示すものである。

INT <argument>

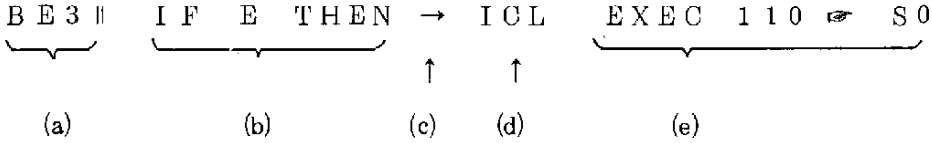
このargumentとしてはVITALのidentifierをスペースで区切って書くことができる。そしてこれはinternal symbolを定義するもので，ここで指定したinternal symbolは今作り出そうとしているコンパイラに対するソー

ス言語の一部でも，ＰＬの一部でもなく，そのコンパイラがスタックの Syntactic な部分に用いる名称である。

<argument 1> = <argument 2 >

argument 1としては，任意のＶＩＴＡＬ の identifier，argument 2としては，reserved word を並べたものを書くことができる。これは，記述するのに便利なようにクラス名を定義するもので，クラス名を含むproduc - tion を1つだけ書くことによって，対応するreserved word を含む production を沢山書き並べる代わりにすることができる。

次にproductionについては例を用いて説明することにする。



(1) ＢＥ３はラベルである。任意の productionに英数字から成るラベルをつけることができ，ラベルの終りは‖で示す。

(2) 次にＭＡＩＮ スタックの内容を調べる。上の例では，スタックの上から3つ分の内容が，上から順にＴＨＥＮ, Ｅ, ＩＦ であれば，スタックからこの3つのエントリをpop up し，その代りに，矢印→の右側に出現しているＩＣＬをスタック上に push down する。
矢印の左側にＳＧという symbol が出現した場合には，スタックの当該エントリの内容が何であってもこのＳＧと一致したとみなすことができる。また，Ｉも特別な意味を持っており，idenitfier が scan された場合にスタック上に push down される。

(3) スタックの内容がある production の矢印の左側と一致している場合は，変換を行ない，次の部分 ― ここでは，ＥＸＥＣ １１０と☞Ｓ０を実行する。この部分はactions と言われるもので，ＥＸＥＣ １１０

はSemantic routine 110を呼び出すことを示し，☞ＳＯはＳＯというルの付いたproduction にコントロールを移すことを示している。action にはこのほかに次のようなものがある。

ＵＮＳＴＫ　ｍ　：スタックの上からｍ個のエントリを取り除き，これをstorageに退避する。

ＳＴＫ　ｎ　：　ＵＮＳＴＫによって取り出された上からｎ番目のエントリをスタック上にpush downする。

ＳＴＡＫ<argument>：　argumentをスタック上にpush downする。

ＳＣＡＮ　：　インプット・シンボルを１個scanする。

ＳＣＡＮ　ｎ：　インプット・シンボルをｎ個scanする。

ＥＲＲＯＲ　ｎ：　出力バッファにエラーメッセージｎをストアする。

ＨＡＬＴ　ｎ：　現在実行中のproduction の実行を中止し，出力バッファにｎをストアし，システムにコントロールを返す。

ＮＥＸＴ<argument>：☞ <argument>に同じ

ＴＥＳＴ<argumeut>：　ＳＩＧＮＡＬが真の場合に限り，☞ <argument>と同じ働らきをする。

ＲＥＴＵＲＮ　：　最後にＣＡＬＬを行なったproductionへ，コントロールを移す。

(4)　スタックの内容が，矢印の左側の部分と一致しても，内容を変化させる必要のない場合には，矢印および変換後のスタックの内容に当る部分を省略すればよい。

(5)　スタックの内容があるproduction の左側と一致しない時には，物理的に次のproduction にコントロールが移され，再び新しいproductionの左側の部分との比較が行なわれる。このように構文解析は，直接的に行なわれるから逆もどりを避けるために，productionを書く際にユーザーが注意しなければならない。例えば

Ｐ→Ｔ，Ｔ＊Ｐ→Ｔがある場合には，Ｔ＊Ｐ→Ｐを先に置くようにしなけれ

ばならない。

ここでproduction に関してもう一つ例をあげて考えてみる。ＰＬプログラムを書く場合，実際には，そのプログラミング言語の文法を直接ＰＬで記述すればよいわけであるが，ここでは，それを先ずBackus Normal Formで表わして，その次にＰＬで書いてみることにする。

ＢＮＦによる式の表現：

＜式＞::＝＜項＞｜－＜項＞｜＜式＞－＜項＞

＜項＞::＝＜１次子＞｜＜項＞＊＜１次子＞

＜１次子＞::＝＜identifier＞｜（＜式＞）

同じものをＰＬで記述したもの：

$$
\left.\begin{array}{lr}
E_1 \parallel & T \rightarrow E \\
 & -T \rightarrow E \\
 & E-T \rightarrow E \\
T_1 \parallel & P \rightarrow T \\
 & T*P \rightarrow T \\
P_1 \parallel & I \rightarrow P \\
 & (E) \rightarrow P
\end{array}\right\} \quad (イ)
$$

ここでＰ，Ｔ，Ｅは実際のプログラム上ではinternal symbolとして定義すべきものであり；＊，－，），（ はreserved wordとして宣言しているはずである。

上記(イ)のＰＬプログラムは，ＰＬ語の記述方法については正しいが，実際的なプログラムとして用いるためにはもう少し工夫を加えなければならない。第１に単純な構成のものの方から記述すること，および第２に例えばE_1 というラベルのところではＴ，－Ｔ，Ｅ－Ｔと並んでいて最後のsymbolは全部Ｔである。このような場合長いストリングのものからサーチしないと誤まった解析を行なうことになるので注意することである。

これを考慮して(イ)を書き換えると次のようになる。

```
P1 ‖   I→P
       (E)→P
T1 ‖ T＊P→T
       P→T
E1 ‖ E－T→E
       －T→E
       T→E
```

(ロ)

さて，productionの実行順序は，スタックの内容と一致するものが見つかった時には，無条件に，あるいはその次に在るsymbol を scanすることによって，それに従ってジャンプするわけで，シークェンシャルになってはいない。また，一致するものが見つかった場合には，適当なＦＳＬルーチンを呼び出して，main スタックのSemanticな要素を更新したり，ストレージを修正したり，code を作り出したりするはずである。さらにproduction は，インプットステートメント中のシンタックス・エラーも見つけねばならない。このことを(ロ)のＰＬプログラムにつけ加えると次のようになる。

```
B1 ‖  －                                  ＳＣＡＮ ☞ P1
P1 ‖  (                                   ＳＣＡＮ ☞ B1
      I→P                  ＥＸＥＣ  1            ☞ T1
P2 ‖  (E)→P                ＥＸＥＣ  2            ☞ T1
      ＳＧ                 ＥＲＲＯＲ  1
T1 ‖  T＊P→T               ＥＸＥＣ  3   ＳＣＡＮ ☞ ＴＢＲ
        P→T                              ＳＣＡＮ ☞ ＴＢＲ
ＴＢＲ‖ T＊                                ＳＣＡＮ ☞ P1
E1 ‖  E－T  ＳＧ→Ｅ  ＳＧ  ＥＸＥＣ４              ☞ ＥＢＲ
        －T  ＳＧ→Ｅ  ＳＧ  ＥＸＥＣ５              ☞ ＥＢＲ
        T    ＳＧ→Ｅ  ＳＧ                        ☞ ＥＢＲ
```

EBR‖	E)		☞ P_2
	E −	SCAN	☞ P_1
	SG	ERROR2	

（－A＊B－C）という式を考え，これが上のproductionでどのように実行されるかを追いかけてみることにする。

表2－6　VITALの例

ラベル	Production	スタック
		([initial state]
P_1		
	SCAN	(－
	☞B_1	
B_1	－	
	SCAN	(－I
	☞P_1	
P_1+1	I	
	P	(－P
	EXEC 1	
	☞T_1	
T_1+1	P	
	T	(－T
	SCAN	(－T＊
	☞TBR	
TBR	T＊	
	SCAN	(－T＊I
	☞P_1	
P_1+1	I	
	P	(－T＊P

```
                EXEC  1
              ☞ T1
T1              T*P
                T                   (-T
                EXEC  3
                SCAN                (-T-
              ☞ TBR
E1+1            -TSG
                ESG                 (E-
                EXEC  5
              ☞ EBR
EBR+1           E-
                SCAN                (E-I
              ☞ P1
P1+1            I
                P                   (E-P
                EXEC  1
              ☞ T1
T1+1            P
                T                   (E-T
                SCAN                (E-T)
              ☞ TBR
E1              E-TSG
                ESG                 (E)
                EXEC  4
              ☞ EBR
EBR             E)
```

	☞ P_2	
P_2	（E）	
	P	P
	EXEC 2	
	☞ T_1	
T_1+1	P	
	T	T
	SCAN	この例の終り

表２－６　ＶＩＴＡＬの例

この時のスタックの状態はこの式の解析列を形成していることになる。次にこの状態を tree であらわして，ＰＬについての説明を終ることにする。この tree では最初に scan された文字が一番下にあり，横の線はスタックの状態を示すものである。

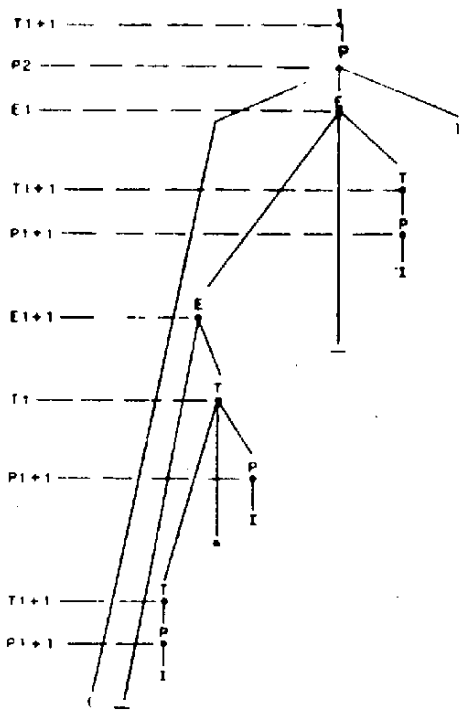


図２－７　Syntax tree

ＦＳＬ （Formal Semantic Language）

ＦＳＬは，これから作ろうとしているコンパイラに対するソース言語のSemantics を記述するのに用いる言語である。ＶＩＴＡＬ の特徴の一つは，Semantics を処理するＴＲＡＮＳＬＡＴＯＲが，言語に依存する部分と依存しない部分とに分かれていることであり，このため言語に依存しない code generator のサブルーチンさえ備わっていれば，ＦＳＬを用いて記述するSemantic routine は，特定のコンピュータを想定しなくてよい点である。したがって，ＦＳＬ自体はセマンティック・メタ言語であるということができる。

ＦＳＬプログラムの基本的な単位は，Semantic routine と呼ばれる番号を付けた１個のステートメント，あるいは一連のステートメントであって，production のaction の部分に，ＥＸＥＣ n （ｎはSemantic routine の番号）と書くことによって呼び出され，これによりコードをジェネレートさせたり，ＴＲＡＮＳＬＡＴＯＲの状態を変化させたりするものである。

ＦＳＬは，ＴＲＡＮＳＬＡＴＯＲの働きを記述するものであるので，コンパイルするプログラムの状態についての情報としてcell やテーブル，スタックなどを用いることができる。これらのcell やテーブルなどは，ＦＳＬの基本的な構成要素で，これらは１次子（primary）（論理的要素であれば論理１次子）と呼ばれるものであり，１次子の中には，ユーザーやシステムによって値の与えられる変数も含まれているのである。以下に１次子の中の代表的なものについて簡単に説明を述べることにする。

先ず，cell はユーザの管理できる最も簡単な変数で１個のマシーン・ワードに対応し，コンパイラ中の単一の storage location を指すものである。そしてcell は，ＴＲＡＮＳＬＡＴＯＲのワーク・エリアやカウンタとして用いることができ，通常はcell 宣言（cell declaration）によって定義しなければならない。

cell には，system cell と呼ばれるシステムで定義しているcell があ

る。その主なものは次のとおりである。

CODELOC……コンパイルされたコードの入り得る次の空き番地を指し分岐命令やサブルーチンの呼び出し等が行なわれる時に用いられる。

STORLOC… ソース・プログラムのデータの入り得る次の空き番地を指している。

PERSLOC …コンパイル・コードにおいて，システムの用い得る最初のワーク・エリアを指している。

ACCUM……実行時に用いるアキュムレータの使用状態を示す。

MAIN……Main スタックの一番上のエントリを指している。

前にも述べたように，system cell 以外のcell を用いる場合は，それを使用する以前に，次のようなcell 宣言で定義しなければならない。

CELL <argument>

ここでargumentとしては，cell identifierをカンマで区切って並べる。cell の名前には任意のVITALのidentifier を用いることができる。

FSLでは，コンパイル時に操作するテーブルを次のような形で宣言した上で用いることができる。

TABLE <argument 1> <argument 2> <argument 3>

ここでargument 1は，そのテーブル名を示すもので任意のVITAL のidentifier を使うことができ，argument 2はそのテーブルに含まれる行（row） の数を示す整数，argument 3はテーブルの1行に含まれる列(column)の名前としてVITAL のidentifierをスペースで区切って並べたものである。

例えば TABLE SYMBOL 400 ID SEMANT EXTRAは，ID，SEMANT，EXTRA という3つの列を持ち，サイズが400 行のSYMBOL という名のテーブルである。

テーブルの中に新しいエントリを設ける場合には，ENTER ステートメン

トを用いて行ごとに作成していく，たとえば前記のＳＹＭＢＯＬというテーブルの第１カラムにはＮＡＭＥ，第２カラムにＲＥＡＬ，第３カラムにＸＸを入れるのであれば，１＃　ＥＮＴＥＲ｛ＳＹＭＢＯＬ，ＮＡＭＥ，ＲＥＡＬ，ＸＸ｝と書く。

ＦＳＬで記述する時，テーブルの参照にはテーブル・オペランドという１次子を用いる。

これは<argument 1>｛<argument 2>,<argument 3>,<argument 4>｝という一般形を持つもので，

argument 1は　　テーブル名

argument 2は　　１次子もしくは0

argument 3は　　列の名（column name）

argument 4は　　0または1（モードとして使用）

である。テーブル・オペランドの値は，次のようにして見つけ出されるテーブルのエントリである。argument 2が１次子ならばargument 1のテーブルの第１列の中にその１次子と等しいものがあるかどうかサーチを行なって，見つかればその行の中でargument 3で指定された列の値をそのテーブル・オペランドの値とする。argument 2が0ならば最も最近テーブル・サーチが行なわれた行の指定列の値をセットする。テーブル・サーチしても見つからない場合には論理１次子ＳＩＧＮＡＬをＦＡＬＳＥにセットする。なお，argument 4はサーチの方向を示すもので0ならばforward search，1ならばbackward searchを行なう。

ＦＳＬで使用可能なスタックはpush - down スタックで，ＴＲＡＮＳＬ-ＡＴＯＲが用いるものはＳＴＡＫ；実行時に用いるものはＲＳＴＡＫ宣言を用いて定義する。これらのスタックを参照する際には，そのスタックの一番上のエントリを見ることになる。スタックの大きさは，特に指定しないかぎり（１００）$_8$である。

以上述べてきたcell，テーブル・オペランド，スタックはいずれもＦＳＬ

ステートメントで用いることのできる1次子（オペランド）であるが，1次子にはこのほかにproduction operandやＦＬＡＤ，その他のものがある。

production operandとは，あるproductionが実行される前，もしくは，実行された後のMainスタックのセマンテイック・ワードの内容を指すものでL1, LL1, L2, LL2, …, L7, LL7で実行前の上から7個のエントリ，R1, RR1, …, R4, RR4で実行後の上から4個のエントリを指すことができる。例えば，

Ｌ１ → Ｒ２；　ＬＬ１ → ＲＲ２；

Ｌ２ → Ｒ１；　ＬＬ２ → ＲＲ１；

は，Mainスタックの上の2個のエントリのセマンティック・ワードを互いに入れ替えるものである。

コンパイラを作成する際に，未だ出現していないステートメントへの分岐命令の処理は厄介なものであるが，ＦＳＬでは FLAD1, …, ＦＬＡＤ４を用いてこれを解決している。ＦＬＡＤはそれぞれスタックであって，この種の分岐命令の度にＰＵＳＨ FLADnを行ない，ＡＳＳＩＧＮ ステートメントが実行されるとその時のＣＯＤＥＬＯＣ の値がセットされる。

同じような働きを持つものにＣＨＡＩＮ があり，ＦＬＡＤ が前向きのchain を作るのに対して，ＣＨＡＩＮ は前後両方向へのchainを作ることが可能である。

さらに1次子としては定数や，他のオペランドの内容やアドレスを参照する時に用いるＣＯＮＴＥＮＴＳ やＬＯＣＡＴＩＯＮ，絶対値をとり扱かうＡＢＳ－ＯＬＵＴＥ ＶＡＬＵＥ，Main スタックの内のエントリの属性を調べるＴＹＰＥ，ビット単位の処理に用いるＴＡＧ，算術式をカッコで囲んだものなどがある。

1次子とならんでオペランドとなるものに論理演算用の論理1次子（Boolean primary）がある。この中に含まれるものにはＴＲＵＥ とＦＡＬＳＥ からなる論理定数（Boolean constant）をはじめ，指定された1次子の指定されたビットにタッグがつけられているかをテストするＴＥＳＴ，指定された1次子が定数であるか否かを調べるＣＯＮＳＴＡＮＴ ＴＥＳＴ，1次子どうしの間の大

小関係をみるRELATION，および特別なBOOLEAN CELLと呼ばれるOKとSIGNAL，それに論理式をカッコで囲んだものがある。テストを行なうような論理1次子は，与えられた条件が満たされればTRUEであり，それ以外の場合にはFALSEとなる。cell OKおよびSIGNALは，それぞれ，ユーザーあるいはシステムによってTRUEかFALSEの値を与えられるものである。もしcell OKがFALSEにセットされると，インプットはひきつづいてscanされるがこれ以降のコードはジェネレートされない。cell OKはこれ以上オブジェクト・コードを作り出したとしてもでき上ったプログラムが動かせないとか，致命的なエラーがあるものになってしまうような重症のエラーが生じた場合に用いるように設計されたもので，したがって一度FALSEになるとこれをresetすることはできない。

このシステムはあるルーチンが終了した時，それが正常に終ったかどうかに応じて，SIGNALにTRUEあるいはFALSEの値を与える。例えばtable searchを行なった時，該当するエントリがあればSIGNALはTRUE，なければFALSEとなる。

次にFSLで使用可能な式には，算術式と論理式の2種があり，算術式は1次子および単項演算子＋，－，二項演算子＋，－，＊，／およびベキ乗の演算子から成るものである。論理式は論理1次子および～，∧，∨（inclusive or），⊕（exclusive or）という演算子がら成る。演算の順序は一般の優先順位に従がい，括弧があれば，括弧内を優先する。

また，1次子および論理1次子はそれ自体，それぞれ算術式，論理式とみなすことになっている。

FSLにおける宣言には，前述のCELL，TABLE，STAK，RSTAKのほかにINDEXおよびDATAに関するものがある。INDEX宣言というのは，INDEX identifierで定義するもので，これは通常の方法で，添字あるいは算術式としてコンパイラが用いるものである。このシステムでは，8個のインデックスが使用可能である。

DATA 宣言は，コンパイル時にビット単位に処理を行なうオペランドにタッグをつけたり，テストを行なったりすることができるようにDATA identifierを定義するものである。VITAL では BOOL， INTGR FRACT， REALという4つのdata identifier をシステムが定義しているので，これらは宣言しないで用いることができる。

FSL中で用いることのできるステートメントとしては，先ず非条件ステートメント(unconditional statement)として，スタック類をとり扱かうPUSHステートメントとPOPステートメントがあり，これらはスタックにエントリを入れたり，とり出したりするのに使われる。テーブル類のとり扱かいには，前述のENTER 使用する。

左辺の内容を，右辺にストアする場合にはSTORE ステートメントを用いこれは

＜ argument 1 ＞→＜ argument 2 ＞

(argument 1は1次子，argument 2は1次子あるいはBoolean cell)

のように矢印で表わす。

ある1次子に算術式の値を加えるのにはTALLY ステートメントを用いてもよい。これは

TALLY ＜ argument 1 ＞

あるいは

TALLY｛＜ argument 1 ＞,＜ argument 2 ＞｝

の形でargument 1の1次子にargument 2の算術式の値を加えることを意味し，argument 2を指定しないと1を加えるものである。

このほか，前述のFLADやCHAINに値を与えるためのASSIGN ステートメント，ストレージをとりあつかうLOAD，SUBLOADステートメント，サブルーチンを定義するステートメントの☞，これを呼び出すためのCALL ステートメント，無条件に分岐することを示すJUMP，その他RETURN，STOP などのステートメントがある。

次に条件ステートメント（conditional statement）としては，ＩＦ＜argument 1＞ THEN ＜argument 2＞⊥，ＩFNOT＜argument1＞THEN＜argument 2＞⊥，ＩF＜argument 1＞THEN＜argument 2＞ＥＬＳＥ＜argument 3＞⊥の3つの形のＩＦステートメントがあり，いずれもargument 1の論理式の値に従ってargument 2のステートメント群（セミコロンで区切って複数個のステートメントを書くことができる）を実行するか否かを決定するステートメントである。
これらのＩＦステートメントは入れ子（nesting）を許している。

ＦＳＬプログラムの構造は，ＰＬの場合と同様，初めに宣言を書き，そのあとにSemantic routineをいくつも続けて書いていく形を採っている。そしてSemantic routineは次のような項目から成り立っているものである。

① ＃のあとに整数をつけたもの，（この整数はproduction のＥＸＥＣ nのnに当るものである。）
② 空白（スペース）
③ ステートメント（セミコロンで区切った条件あるいは非条件ステートメント）
④ ＃

このほか，プログラムの最初と最後はＢＥＧＩＮとＥＮＤでなければならない。またコメントは2個のアスタリスク＊＊で始まり，キャリッジ・リターンで終わるようにすれば，プログラムのどこに置いても構わない。

ＦＳＬにおいては，コード・ブラケットと呼ばれる特別な括弧⊠，⊠を用いて，ＦＳＬで書かれたオペレーションが，コンパイル時に行なわれるものであるか，実行時に行なわれるものであるかを区別している。コード・ブラケット内に書かれたオペレーションは実行時のもので，従ってコンパイル時にオブジェクト・コードが作り出される。一方，コード・ブラケットに囲まれていないオ

ペレーションは，コンパイル時にＴＲＡＮＳＬＡＴＯＲが実行するものである。例えば

Ｌ４＋Ｌ２　→　Ｒ２

の場合，コンパイル時にＬ４とＬ２の内容を加えてＲ２にストアするのに対して

⊴　Ｌ４＋Ｌ２　→　Ｒ２　⊵

の場合には，コンパイル時にはＬ４にＬ２を加えてＲ２にストアする演算のオブジェクト・コードをジェネレートし，実際にこの演算が行なわれるのはオブジェクト・プログラムの実行時である。

また，コンパイルの際に必要なデータの属性は，セマンティック・ワードにセットされるもので，このセマンティック・ワードの第１ワード目は，データの型やインデックスの番号などを含んでいて，システム側で定めたフォーマットを持っているが，第２ワード目は，このほかの必要な属性などを示すためにユーザーが自由に用いることができる。これらのとり扱かいについては，前述の１次子やステートメントの働きで比較的自由に行なうことができるようになっているのも，ＦＳＬの特徴となっている。

このＶＩＴＡＬシステムは，コンパイラ・コンパイラとしてかなり力のあるものである。このシステムは既に実用に供されていてこれによって作られたコンパイラもいくつもあるようである。

参照文献

1. L. F. Mondshein
VITAL Compiler-Compiler System reference manual
Lincoln Laboratory Technical Note 1967－12
（図２－５，６，７）は上記文献より引用させていただいた。

2. J．A．Feldman

A Formal Semantics for Computer Languages
and it's Application for Computer Languages
C．ACM Vol.9 No.1 1966 PP3～9

3. 二村良彦他

PL／IWによるTSS用コンパイラ・コンパイラの作成
第12回プログラミングーシンポジウム報告集 1971 PP43～53

2.6 CC（the Compiler－Compiler）

このシステムは，マンチェスター大学で研究開発されてきたもので，BrookerやMorris，Rohl などの手によるものである。この研究は，かなり古くから行なわれてきたものと思われ，1960年には，Brookerが，その基本的なアイディアを既に発表している。このシステムは，きわだって効率がよいとは思われないが，いくつものコンピュータ．システムでインプリメントされており，CCで書かれた ALGOL，FORTRAN Ⅱ，そのほか Atlas autocode，Mercury Autocode などが作り出されていることも報告されている。

ひとつのプロセッサというものを考える場合，そのプロセッサの使用目的によって，とり扱かうデータなどの種類が異なってくる。例えば，ALGOL やFORTRAN であれば，浮動小数点数の配例などを多く扱かっていると考えられるわけであり，これに対して，このCCは tree 構造のデータを操作できるよう設計されていると言える。 CCで記述するプログラムの構造はFORTRANや ALGOL のように，メインルーチンとサブルーチンから成り，そのサブルーチンのあるものは，作り出そうとしているコンパイラ言語のいろいろなソース．ステートメントを処理するものであり，またあるものは，コンパイラの処理を容易にするために導入する補助ステートメント（auxiliary statement）に対するものである。メインルーチンは，CCシステムによって作られていて，ユーザーの作り出すコンパイラ言語には関係なく，次のような定まった働きをするものである。すなわち，1度に1行ずつソース・ステートメント（コンパイラに対するソース．ステートメント）を読み，これをユーザーの書いた phrase と format の定義を用いて，その言語の構文規則に従って解析する。その解析方法は，top－down であり，ソース・プログラムの行がそれに対して対応しているような format を識別し，セマンティック・フェーズで用いられる完全な syntax tree を得る。そして，そのあと，メインルーチンは，この format に関連する routine を呼び出して，認識されるインストラクションを変換する。このあと，次のソース・ステートメントを読むように

コントロールは，メインルーチンに戻されるのである。

シンタックス・フェーズへのインプットの形は，拡張 ＢＮＦ （ extended Backus Normal Form ）を用いている。これは，普通の ＢＮＦ に，くりかえしを示す ＊ とオプションを表わす ？ を加えたものである。 例えば

〔Ａ１〕＝〔Ｂ＊〕〔Ｉ〕，　〔Ｂ〕＝ａ，　〔Ｉ〕＝ｉ

であれば， A1 はｂを１個以上続けて書いたあとにｉをつけたもの （ｂｉ，bbbi ，…… etc．）であり，

〔Ａ２〕＝〔Ｂ？〕〔Ｉ〕

であれば， Ａ２は bi またはｉである。

ＣＣで採用している ＢＮＦ の記述方法は，正統的な nonterminal をカッコで囲んで書き，terminal symbol はそのまま書くやり方をとっているものと， nonterminal をそのまま書き，terminal を引用符で囲むものなど，ＣＣをインプリメントしているコンピュータ・システムによって多少異なっているが，いずれの場合も ALGOL などで｜で示すメタ言語の orはカンマ(,)で置き換え，⋮＝ は ＝ で表わしている。

シンタックス・フェーズへ，このＢＮＦで書いた Syntax をインプットする場合，その並べ方の順序が問題になるのは ＦＳＬ などの場合と同様である。すなわち，

〔Ｘ〕＝〔Ｙ〕，　〔Ｙ〕〔±〕〔Ｚ〕

のような場合， 上の書き方では〔Ｙ〕〔±〕〔Ｚ〕に対応するソース・ステートメントも 〔Ｙ〕であると認識してしまって，正しく解析できないので

〔Ｘ〕＝〔Ｙ〕〔±〕〔Ｚ〕，〔Ｙ〕

のように書かねばならない。

さらに，〔ＦＤ〕＝〔ＢＤ〕〔ＯＤ〕〔ＯＤ〕〔ＯＤ〕，〔ＯＤ〕〔ＯＤ〕〔ＯＤ〕

（〔ＢＤ〕＝0,1 および 〔ＯＤ〕＝０，１，２，３，４，５，６，７）

のような場合，これは２進数１桁の後に８進数を３桁付けたものであるが，これを

〔ＦＤ〕＝〔ＢＤ？〕〔ＯＤ〕〔ＯＤ〕〔ＯＤ〕

と書くとまちがいである。その理由は，例えば（１５７）$_8$のような0または1で始まる3桁の8進数は後の定義によると，ＦＤに入らないことになってしまうからである。

ここで FORTRAN コンパイラからカッコを含まない算術式を例にとって，コンパイラ・コンパイラＣＣについて考えることにする。

FORMAT〔ＳＳ〕＝〔VARIABLE〕＝〔EXPR〕 ……(1)

ここでメタ変数は，次のように定義する。

```
PHRASE〔EXPR〕=〔PRECEDING±〕〔TERM〕〔TERMS〕
PHRASE〔PRECEDING ±〕=〔±〕, NIL
PHRASE〔±〕= ＋ , －
PHRASE〔TERMS〕=〔±〕〔TERM〕〔TERMS〕, NIL
PHRASE〔TERM〕=〔FACTOR〕〔FACTORS〕
PHRASE〔FACTORS〕=＊〔FACTOR〕〔FACTORS〕,
        ／〔FACTOR〕〔FACTORS〕, NIL
```

図２－８　ＣＣの例

(1)は format definition，（図２－８）は phrase definition と呼ばれ，ＳＳは，source statement を表わす format class を示している。(1)は簡単に言えば，〔VARIABLE〕＝〔EXPR〕という Syntax がこの言語の source statement として含まれることを表わしているのである。 phrase definition は format definition，あるいは，他の phrase definitionを補うためのものである。

さらに補助ステートメント（ auxiliary statement)を導入する。

```
FORMAT〔AS〕=LOAD〔PRECEDING ±〕〔TERM〕
```

```
FORMAT〔AS〕=ACC=ACC〔+〕〔TERM〕
FORMAT〔AS〕=DIV BY 〔FACTOR〕
FORMAT〔AS〕=MULT BY 〔FACTOR〕
FORMAT〔AS〕=LOAD〔PRECEDING ±〕〔FACTOR〕
FORMAT〔AS〕=DUMP ACC IN 〔VARIABLE〕
```

図２－９　CCの例

実際に，コンパイラに対するソース・ステートメントを読んで，それに対応する tree を format definition を用いて作り出すと，次にコンパイラは，その tree に対して，オブジェクト・コードをジェネレートするとか，宣言に対するリストを作るとかなどの処理を行なうことになる。その処理ルーチンを CC では format routine と呼んでいる。例えば (1)，および（図２－９）の第１番目の format に対する format routine は次のようになる。

```
ROUTINE〔SS〕=〔VARIABLE〕=〔EXPR〕
    LET〔EXPR〕=〔PRECEDING ±〕〔TERM〕〔TERMS〕
    LOAD〔PRECEDING ±〕〔TERM〕
 2) ⟶ 1  UNLESS〔TERMS〕=〔±〕〔TERM〕〔TERMS〕
        ACC=ACC〔±〕〔TERM〕
    ⟶ 2
 1) DUMP ACC IN 〔VARIABLE〕
    END

ROUTINE〔AS〕=LOAD〔PRECEDING ±〕〔TERM〕
        LET〔TERM〕=〔FACTOR〕〔FACTORS〕
        LOAD〔PRECEDING ±〕〔FACTOR〕
 2) ⟶ 1  UNLESS 〔FACTOR〕=/〔FACTOR〕〔FACTORS〕
```

```
    DIV BY 〔FACTOR〕
    ⟶ 2
1)⟶ 3 UNLESS 〔FACTOR〕≡ *〔FACTOR〕〔FACTORS〕
    MULT BY 〔FACTOR〕
    ⟶ 2
3) END
```

図2－10　FORMAT ルーチン

format routine の第1行目は heading と呼ばれ，"≡" の右側は format definition の符号の右側と同じ形になっている。これによって format definition に対応した format routine が呼び出されることになり，また heading の右側の部分は，format definition から format routine へのパラメタを渡すのに用いられている。ユーザーは後述する BSクラスのシステムに組み込まれている routine や，すでに定義済の format routine を用いながら新しい format routine を定義することができる。したがってこのシステムは 拡張可能であると言える。次に format や phrase definition などに関連して，もう少し説明を補うことにする。

CC に対するプログラムの入力システムは2つのフェーズから成っており，第1のフェーズは句構造文法の定義を受け取るところであり，第2のフェーズはその言語で書かれたソース・プログラムを変換するものである。

CC においては，基本的には4つの format class が用いられる。それらは次のとうりである。

〔SS〕　source statement の format の class

〔MP〕　master phrase の class（即ち，PHRASEやFORMATなど）

〔BS〕　format routine 内で用いられるシステムに組み込まれたステートメントの class

〔AS〕　ユーザーが，その format routine 内での使用のために導入

できる補助ステートメントの class

あとの3つは phrase，format および format routine すなわち，その言語の定義（definition），Syntax，および Semantics を読み込む CC の第1フェーズでのみ用いられる。その言語に固有の format や phrase definition はコンパイラが定義され，第2フェーズで operational になったら削除することができる。これを削除しない形で残すと，そのコンパイラはインクリメンタル・コンパイラとなり，ステートメントの定義を付け加えることができるようになるのである。

前にも述べたように，この CC は既にいろいろなコンピュータでインプリメントされており，特に Atlas に対して作られた CC に関しては，多くの文献が著わされている。その中には，実際に CC を用いて作成した Atlas Autocode コンパイラと同じものを手書きで作ったものとを比較したものもあるが，その結果は，CC を用いた方が，スペースについても，時間に関しても2倍程かかっている。これは，かなり CC を使い慣れた人が使った場合で，普通は4～5倍の時間を要するということも述べられている。コンパイル時間の遅いことの主な理由は，構文解析にも幾分原因があるが，それよりも算術式のとり扱かいによるところが大きい。これは X＝1 のような簡単な代入ステートメントでも，上記の算術式 〔VARIABLE〕＝〔EXPR〕として解析すると，多くの不要な枝を持った tree となってしまうためである。こういう場合には演算子順位などの方法による方がずっと効率がよい結果を生じるようになるはずである。

けれども CC の製作に当っている人々は，CC を用いてコンパイラを作る方が，手書きでコンパイラを作成するのよりは速いと書いている。そして，特に Syntax が複雑な場合には，有利であるとのことである。

CC については，巻末の TWS の翻訳の中にも説明があるので詳しくは，そちらを参照されたい。

参 照 文 献

1. R.A.Brooker, D.Morris

   A general translation program for phrase structure languag

   J.ACM Vol.9 No.1 1962 PP 1～10

   (図2－8, 9, 10)は上記文献より引用させていただいた。

2. R.A.Brooker, D. Morris

   Tree and Routines

   Comput.J. Vol 5 1962 PP 33～47

3. R.A.Brooker, D.Morris, et. al.

   The Compiler－Compiler

   Annual Review in automatic Programming Vol 3 1963

   PP 229～275

4. R.A.Brooker, D. Morris, et. al.

   Compiler－Compiler Facilities in Atlas Autocode

   Comput. J. Vol 9 1967 PP 350～352

5. R.A.Brooker, D. Morris, et, al.

   Experience with the Compiler－Compiler

   Comput. J. Vol 9 1967 PP 345～349

2.7 CABAL

CABALシステムは，FeldmannやCurryのProduction Language (PL) とFormal Semantics Language (FSL)の思想を取り入れたコンパイラ・コンパイラでCarnegie－Mellon大学で3年がかりで開発されたことが，1968年に発表されている。このため，同じくPL，FSLの流れをくむMITのVITALシステムと酷似している部分もあるが，CABALは，全体としてALGOL－likeな言語であることに付随して，言語の表現はかなり，異なっている。VITALと同じようにCABALもインタラクティヴなシステムである。

CABALプログラムの構成

前述のように，CABALはPL，FSLの思想を取り入れてはいるが，CABALプログラムの記述の際には，reduction関係のステートメントと，code generationおよび一般のCABALステートメントを混ぜて書くことができるようになっている。すなわち，CABALにおけるプログラム単位は，通常のALGOL procedureに類するものとCO－routineと呼ばれるものから成り，これらのプログラム単位内では，ALGOLと同じくBEGIN, ENDに囲まれたブロック構造やidentifierのスコープを持ち，宣言文およびステートメントから成っている。

CO－routineというのは，サブルーチンであるがタスクの概念を備えたもので，前にコントロールが中断された点から再び処理を続行できる機能を持っている。CO－routineでは，procedureの先頭に普通の宣言文に先立ってCOMMONステートメントを置いて，これによって処理の再開される場所を示すこと以外は，通常のALGOL procedureと同じ形を取っている。CO－routineが本来のCO－routineとして呼び出された時には，呼び出し側とそのCO－routineとには主従関係は存在しないが，これを普通のprocedureとして用いることも可能である。

CABALはVITALのようにPLプログラムとFSLプログラムを，記

述の際に区分する必要はないが，実行の上では，reductionステートメントを順に実行し，その中から他のステートメント，またはブロックを呼び出していくようになっている。次に先ず，ＣＡＢＡＬにおける宣言文，reductionステートメント，一般のステートメント，Code－generationに関するステートメントについて述べる。

CABAL の宣言文

ＣＡＢＡＬにおける宣言文は，どちらかというとＰＬ／Ｉ的なもので，この言語で使用可能な５種類のデータ構造 scalar，array，list，stack，queueを１つの形で宣言でき，その機能の違いは宣言につけ加えるパラメタの有無によって決められる。

宣言文のSyntax は次のとおりである。

＜declaration＞::＝DECLARE＜declaration list＞

＜declaration list＞::＝＜name list＞
(＜structure＞)

|＜declaration list＞,＜name list＞
(＜structure＞)

＜structure＞::＝＜nature＞＜element linkage＞

＜nature＞::＝NUMBER,＜bound＞
＜number option＞

|LOGIC,＜bound＞

|STRING,＜bound＞

|NAME,＜bound＞

＜bound＞::＝＜number expression＞|?|?
＜number expression＞

＜number option＞::＝,FIXED|＜empty＞

＜element linkage＞::＝,〔＜bound pair list＞〕
＜pop linkage＞

| <empty>

<pop linkage> ::= , LIFO | , FIFO | , RANDOM |
<empty>

<bound pair list> ::= <bound> : <bound>
| <bound pair list> , <bound> : <bound>

<name list> ::= <name> | <name list> ,
<name>

<name> ::= <letter> | <name> <letter> |
<name> <digit>

<empty> ::= (the null string of characters)

<number expression> ::= (expression of type
number)

図２－１１　ＣＡＢＡＬの例

例えば，次の例では，Ａ，ＢはNUMBER属性を持つ有効桁６のscalar，ＣはLOGICで有効桁４でエレメント数10×10の２次元のarray，ＤはSTRING属性を持ち，有効桁10でエレメント数が15コ位のlast－in・first－outのスタック，ＥはSTRINGで有効桁が不定でエレメント数も不定のlistを示している。

```
DECLARE A, B(NUMBER, 6),
  C(LOGIC, 4, [0:10, 0:10]),
 D(STRING, 10, [1:?15], LIFO),
 E(STRING, ?, [1:?], RANDOM);
```

ここでいう有効桁は，データ構造のとりうる４つの属性に対応し，その単位は

NUMBER (decimal) ..... digit

STRING (alphanumeric) ..... character

LOGIC ..... bit

NAME ..... その name 全体

のようになっている。

宣言の中で？が出て来た場合には，ダイナミック・アロケーションが行なわれる。

Reduction ステートメント

Syntaxの認識のためには，reductionステートメントが用いられる。これは例えば，

L:|'WHILE', EXP|→STA|WHCODE;RETURN|

のようなもので，スタックの内容を左半分の||に囲まれた部分と比較して一致していれば"WHILE"とEXPとをスタックから pop－up し，代りにSTAを入れる。

スタックの内容を示すのに用いるエレメントは，端記号と超変数を区別して，端記号は" "で囲んで表現する。さらに

|A, B, A|→A, B|RETURN|

のような場合は，右辺のAが左辺のどちらのAに対応するか不明なので，これを避けるため，左辺のAに右から順に添字を付け，

|A, B, A|→A(1), B|RETURN|

のようにする。

またCABALでは，端記号をDEFINEステートメントを用いて前以ってまとめて超変数であらわすことができる。

DEFINE UNOP＝"－"|"＋"

BNOP＝UNOP|"／"|"＊"|"↑"|"0"|"ε";

スタックの内容が何であっても，置き換えを実行させる場合には，VITALなどの場合と同じような Sigma function を用いる。

さてスタックの内容が左辺と一致した時には，スタックの置き換えのあと，コントロールが右側の|以降のステートメントに移される。この部分には，

すべてのＣＡＢＡＬステートメントを書くことができる。但しreductionステートメントは BEGIN，ENDで囲まれたブロックの中でなければ使うことができない。

一致しない場合は，コントロールは次のreductionステートメントに移される。

ステートメント

ＣＡＢＡＬにおける reduction と code generation 以外の主なステートメントについて述べるがその前に，式について特異な点を示す。

ＣＡＢＡＬでは，データ・属性と同じく式も4つの属性をもつが，これはオペラントではなく，オペレーションによって決められる。但し式がオペランド1個のみから成る時はオペランドに依る。例えば，＋，－，＊，／，↑，などの演算は，オペランドが何であっても演算の結果は NUMBERタイプとなる。このほか，modを取る演算およびトランケーションは ＮＵＭＢＥＲタイプ，論理和，論理積，論理否定，関係演算は LOGICタイプ，concatenationは STRING タイプの演算結果をとる。このような方法を用いて，式を算定するのは変数中の情報を自由に部分的にアクセスできるようにするためである。

ＣＡＢＡＬの主なステートメントは，代入文，条件文，繰り返し，push － popの4種で前3者は，いくらかSyntax が異なるが，ALGOL とほぼ同じである。

push － pop ステートメントは，ＬＩＦＯ，ＦＩＦＯ，RANDOMの指定された順にリンクされた stack，queue や list を push したり，pop したりするものであって Syntaxは次のとおりである。

＜push--pop statement＞ ::＝＜stack operator＞

＜name＞

|＜stack operator＞＜group reference＞

＜stack operator＞ ::＝△ |＜stack operator＞△

| ▽ | ＜stack operator＞▽

＜group reference＞ ::＝＜name＞

〔＜group definition＞〕

＜group definition＞ ::＝＜index range list＞

＜number expression list＞

＜index range list＞ ::＝＜empty＞ | *, |

＜index range list＞*,

Code Generation

コード発生のためのステートメントは itemステートメントと codeステートメントを組にして用いる。

例えば

```
ITEM(SMT〔4〕, SMT〔1〕, #A, #B);
CODE(FOR#1←#4 TO #3 BY #2 DO);
```

では，＃1はSMT〔4〕に，＃4は＃Bに，のように itemステートメントの内容には左から順に番号をつける。item ステートメント内に＃が出てきたときは，それが定数であることを示している。code ステートメントはVITALのコード・ブラケットと同じように，このCODE( ) の中には，任意のCABALステートメントを書くことができて，それらのコードを発生させるものである。この例では，

FOR SMT〔4〕←#B TO #A BY SMT〔1〕 DO

に対応するコードが作られる。

未だ出現していないラベルへの ジャンプ命令に対しても，VITALと同じようにCHAINとASSIGN function を用いて，処理することができる。

CABALでは，GO TO ＜ラベル名＞ NOW を用いると，いまコンパイラ・コンパイラが作り出したコード内のラベル名で指定されるステートメントへコントロールを移すことができる。そしてgenerate されたコード内に

あるRETURN NOW ステートメントを実行すると，コンパイラ・コンパイラに戻ってくるのである。

ま と め

1968年に発表されたＣＡＢＡＬについての文献によれば，このシステムの設計思想は，

1. トランスレータの適応性。
2. 作り上げることのできうるトランスレータの範囲が多様であること。
3. 種々のエンバイロンメントに対して，柔軟性を持つこと。
4. モジュール化できること。
5. 専門のシステム・プログラマに対して，使いやすく簡潔であること。
6. 専門的でないユーザーに対して，平明であること。
7. 読みやすいこと。
8. 機種に依存しないこと。（ machine independence ）
9. 信頼性の高いこと。
10. エンバイロンメントとの適合性。

を基調とするものであって，実際にインプリメントした結果は，これらの要求事項をほぼ満足するものであると述べている。例えば，コンパイラ・コンパイラの適応性については，ＣＯ－routineの使用による並行処理，ストリング処理用の演算子，構文解析，master symbol tableがcode generatorシステム・ルーチンとユーザーの書いたセマンティック・ルーチンのいずれからのアクセスできること，高レベルの言語で code generatorが書けることによってある程度最適化したコードが作り出せることなどがある。

また，コンパイラ・コンパイラ CABAL の作り出すトランスレータは，インタープリータであっても，会話型のコンパイラ，あるいは，多重の言語システムであっても構まわない。さらにエンバイロンメントへの柔軟性にも富んでいてマクロの機能をつけ加えることも可能である。

そのほか，モジュール化や読みやすさ，信頼性も高いが，難点としては code generation に時間のかかることがある。これは，CODE ステートメントの書き方で，効率をよくすることができるであろうと書いてあるが，それには，このシステムでコンパイラを作成する人々の熟練度に依存する面が多いと解釈できる。

ＣＡＢＡＬは，コンパイラ・コンパイラとしてはかなり優秀なものであるらしいが，実際にこのシステムによって作り出されたコンパイラ，もしくはトランスレータがこの時点では数少ないことと，それ以降の文献が不足しているため，どの程度実用的なものであるのかという点は不明である。

参　照　文　献

1. P.K.Dove
   Design highlights of CABAL —— a compiler－compiler
   FJCC 1968　PP 1321～1328
   （図２－11）は上記文献より引用させていただいた。

2. R.A.Brooker, D.Morris, et.al
   Compilir－compiler facilities in Atlas autocode
   Comput.J., Vol 9　1967　PP 350～352

3. R.W.Floyd
   A description language for symbol manipulation
   J.ACM　Vol.8　№10　1961　PP 579～584

2.8 GULP

GULPは，ケンブリッジ大学において，大形コンピュータTITANにつながっているPDP－7（8kW／1W＝18 bit）及びライトペン等のついたグラフィック・ディスプレイ（CRT）のシステムに対して開発されたものである。

GULPの考え方は，Brooker & Morris の開発したCCから発想しているが，Syntax および Semantics の記述方法はかなりの違いがみられる。すなわち，CCにおいてはPHRASE，FORMAT，ROUTINEの3つで構成されていたが，GULP においては（図2－12）にみられるように同一ステートメントとして表現するようになっている。特にSemanticsの記述に関しては，ユーザがルーチンを別に作って指定することができるが，ほとんどの場合（表2－1）に示した21 個の標準ルーチンにより記述ができるとしている。

（図2－12）に示すのは，加算のみを許す代入文及び出力を指定できるユーザ言語の定義である。

ここで .PLUS. のように 前後にピリオドがあるのが nonterminal 記号を示し .ASTAT!. のようにエクスクラメーション記号がついているのが complete statement記号を示す。phrase の指定のあとに：があり それに続くのが Semantics 記述である。

```
1)  Language definition
CHARACTERS;
'CR';TTYPE;215;          (Define carriage return)
'RUB';TTYPE;377;         (Define a rub-out character)

DELIMITERS;
space;

ERASERS;
'RUB';

TERMINATORS;
'CR';

MNEMONICS;
LAC=200000;              (Load accumulator from store)
DAC=40000;               (Deposit accumulator in store)
ADD=300000;              (Add from store to accumulator)

CONVERSIONS;             (Octal conversion, numbered 1)
1;
0,1,2,3,4,5,6,7;

SYNTAX AND SEMANTICS;
0,1,2,3,4,5,6,7→.DIGIT.;
A,B,C,D,E,F,G,H,I,J,K→.LETTER.;

.LETTER..LETTER★?.,.DIGIT★?.→.NAME.;          (name of a variable)
=→.EQUALS.;
)→.KET.;
+→.PLUS.;
.PLUS..NAME.→.TERM.:INSTR(ADD,SYMADD(C2));    (positive term, to be added to sum)
.PLUS?..NAME.→.FTERM.:INSTR(LAC,SYMADD(C2));  (first term with optional plus sign. This term is
                                               the first loaded in the accumulator.)
.NAME..EQUALS..FTERM..TERM★?.→.ASTAT!.:INSTR(DAC,SYMADD(C1));
                                               (Complete arithmetic assignment statement, se-
                                               mantics to store result)
.NAME..KET..DIGIT★.→.CONDEF!.:INSTR(LAC,INPUT(C3,1)):INSTR(DAC,DEFSYM(C1,1));
                                               (Statement to define a symbol and its initial
                                               value)
OUTPUT→.OUT.;                                  (Defines a compound character)
.OUT..NAME.→.PRINT!.:OUTPUT(SYMADD(C2),1);     (Statement for printing value of a variable)
```

図2－12　GULPの例

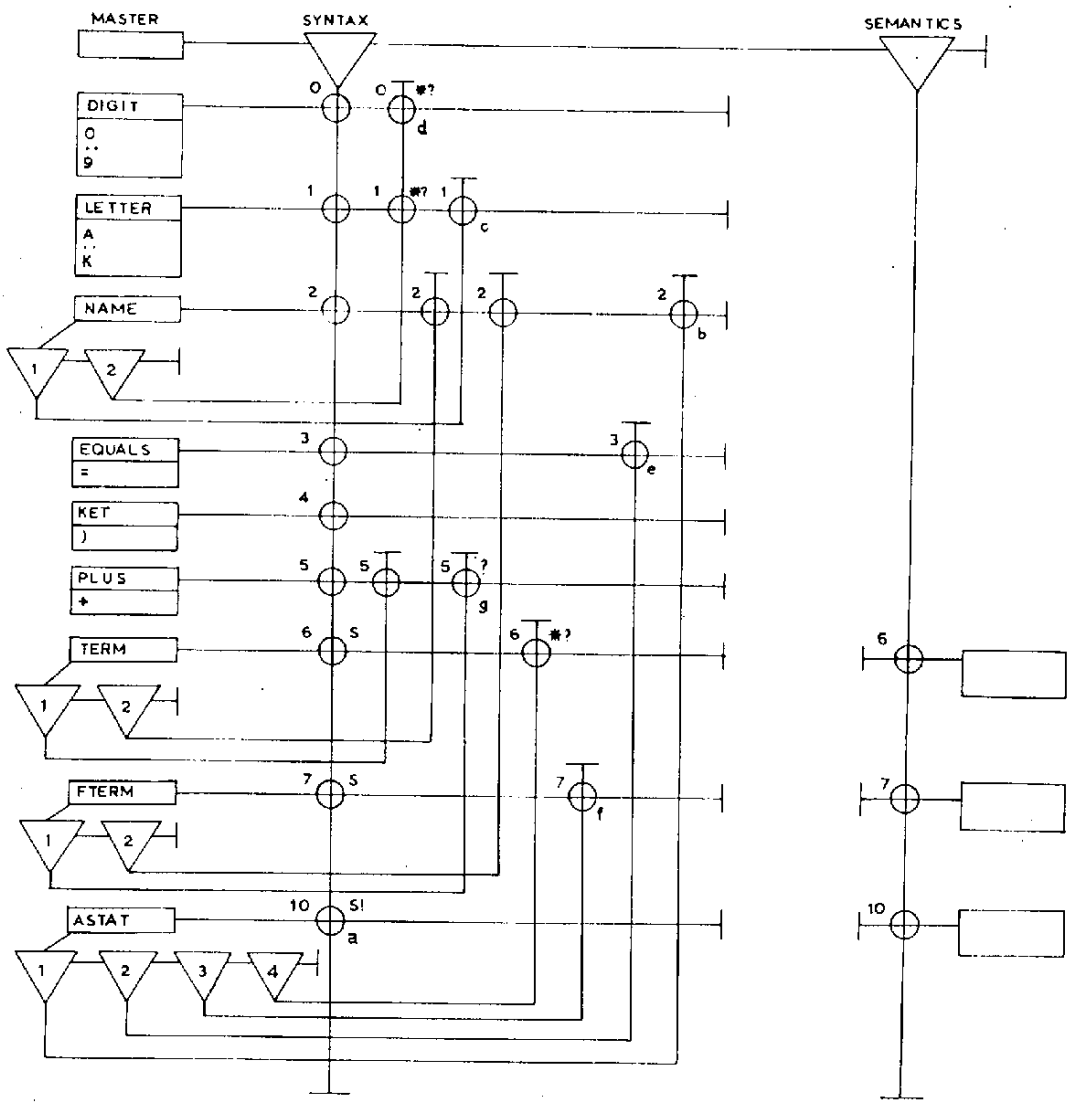


図２－１３　ＧＵＬＰによって使用されている ring structure

入力されたユーザの Syntax および Semantics の記述は，ＡＳＰ－７（ associative structure package ）によりまず（図２－13 ）に示すようなリング・ストラクチャーの形に変換される。

ユーザ言語のコンパイルは，入力ストリングにより complete statement 記号から始めて，構文解析のためにテストされた規則を入れるもの及び，構文解析の結果でき上った tree を入れる２つのスタックを利用して，リング・ストラクチャーをおっかけることによりおこなわれる。

このシステムは，ＣＲＴを端末として考えているために，ライトペンの割り込み，ファンクションキーの割り込みを，１つの文字に変換することにより通常の入力記号と同じようにあつかっている点に特徴がある。

同じようにグラフィックを対象としたシステムにＡＥＤがあると報告されているが，ＧＵＬＰが参照しているＡＥＤ関係文献ではその特徴が不明である。

表２－１　GULPのSemantics ルーチン

名　前	argument の　数	routine の結果	定　　義
ＩＮＳＴＲ	変　　数		すべての argument の値を加え，instruction か data word として，ＳＴＯＲＥ内にそれを置く。 dummy location counter に１を加える。
DEFSYM	2	reserve された location の address	１つのＳＹＭＢＯＬを定義する。 ARG. 1はＳＹＭＢＯＬ名を構成する文字列を示す。 ARG. 2は reserve される文字数を示す。 もし，ＳＹＭＢＯＬがすでに定義されていたら ERROR である。
SYMADD	1	ＳＹＭＢＯＬの address	ＳＹＭＢＯＬ ＴＡＢＬＥを look up する。 argument はＳＹＭＢＯＬを示す。

名　前	argument の　数	routine の結果	定　　義
INPUT	2	data word の address	input string を data word に変換する。 ARG.1 は input string を示す。 ARG.2 は変換数を示す。
OUTPUT	2		data word を string に変換し，それを output する。 ARG.1 は data を示す。 ARG.2 は 変換数を示す。（もしなければ変換はされない）。
SET	2		Semantic variable （ARG.1）を setする。 ARG.2 はなくてもよい。
INTENS	1		表示の intensity（強さ）を setする。
DSPLAY	1		ARG. が ON なら picture を出し， ARG. が OFF なら出さない。
PEN	1		ARG. が ON なら LIGHT-PEN使用可である。 ARG. が OFF なら不可である。
POINT	2		座標へPOINT(ARG.1,ARG.2)を画く。
LINE	2		X,Y increment を使って，line を引く（ARG.1,ARG.2 別々に）
ALINE	4		POINT(ARG.1,ARG.2)からPOINT(ARG.3,ARG.4)へ line を引く。
SETDIS	4		DISPLAY の set up ARG.1はintensity（強さ）を示す。 ARG.2はscaleを示す。 ARG.3はPENに対応する。 ARG.4はDSPLAYに対応する。

名　前	argument の　数	routine の 結　果	定　　義
CROSON	0		picture に tracking cross を付加する
SETX	2		座標（ARG.1，ARG.2）へ tracking cross を置く。
CROSS	2		tracking cross がある座標をARG.1 と ARG.2 へ読み込む。
DEFCHR	2		ENDCHR call があるまで， 表示character を定義する。 character については，ARG.1で名前が示され，ARG.2で示されるarray 長が reserve される。
ENDCHR	0		DEFCHR の end を示す。
STORE	変　　数		名前（ARG.1）で連想されるarrayに data を setする。 ARG.2は使用される最初の element を示す。 ARG.3はarrayに置かれるべき連続した data を先に与える。
LOCK	3		ARG.1によって指定される line にもっとも近い point に tracking cross を置き ARG.2と ARG.3にその座標を置く。
DELETE	1		（DEFCHR によって前もって定義されているとする）表示 character（ARG.1）をsystemから消去する。

参 照 文 献

1. P.J.Pankhurst

   GULP-A cumpiler－compiler for verbal and graphic languages

   Proc. ACM 1968 PP 405～421

   （図2－12，13），（表2－1）は上記文献より引用させていただいた。

2. R.A.Brooker, D.Morris

   A General Translation Program for Phrase Structurc Languages

   J.ACM 1961, 7. PP 1～10

3. R.A.Brooker, I.R. Maccallum, D.Morris, J.S.Rohl

   The Compiler Compiler

   Annual Review in Automatic Programmig Vol 3, 1963

   PP 229～275 Pergamon Press

4. D.T.Ross

   The AED approach to generalized computer－aided design

   Proc. ACM 1967. PP 367～385

2.9 INSCAN

INSCANはRADC(Rome Air Development Center)において開発されたDM−1(Generalized data management System) のためにインプリメントされたSyntax − directed language processor である。

このシステムの概略は，Action Graph の導入から始まる。Action Graph は入力ストリングに対しておこなう処理をコントロールするものと考え，各要素はそれぞれの命令とみなすことができる。

ここでその例を出してみよう。

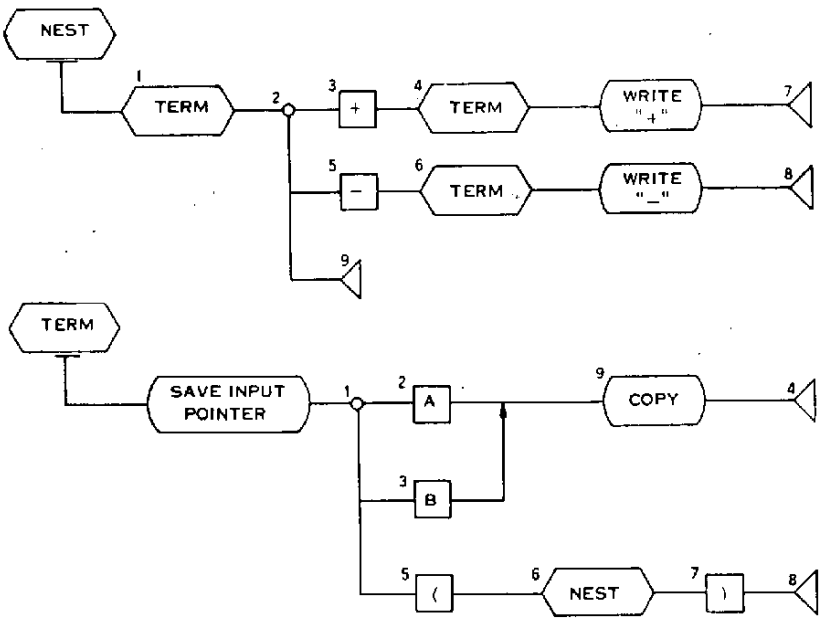


図２－１４　Action Graph の例

これは，加減算記号及びカッコより成る式を逆ポーランドの形になおす Action Graph である。

NEST，TERM はそれぞれのグラフ名をさす。

NEST から実行が始まり，１によりTERM が呼び出される。TERM に

おいては，SAVE INPUT POINTER により，入力記号のバッファポインタがSave される。グラフ 'TERM'の中の1は分枝を示している。すなわち，2において入力記号が"A"ならば，2，9の順に実行して呼んだグラフにもどる。そうでなければ，3において入力記号が"B"かどうか調べられ，等しければ3，9の順に実行，等しくなければ5の"("との比較がなされる。

グラフ'TERM'における9のCOPY は入力記号をSAVE INPUT POINTERにおいてSave した点から現在まで処理した点までを入力することを指定する。

すなわち，グラフ'TERM'においては，オペランドであればこれを出力し"(" であればNEST を呼び出し，そのあとに")"があるかどうか調べるものである。TERM からコントロールが帰ってくると，2の分析により"+"あるいは"−"と入力記号との比較がなされ，等しければTERM が呼び出されその後，記号"+"あるいは"−"が出力される。

Action Graph は計算機に対する入力としては不適当であるので，グラフをセンテンスの形にして入力する。

```
NEST:  EXECUTE TERM;CHOICE(3, 5, 9);
           3:"+";EXECUTE TERM; WRITE"+";GOOD;
           5:"−";EXECUTE TERM; WRITE"−";
           9:  GOOD.

TERM:  SAVE INPUT POINTER; CHOICE(2, 3, 5);
           2:"A";9: COPY; GOOD;
           3:"B"; GOTO 9;
           5:"("; EXECUTE NEST;")"; GOOD.
```

図2－15　INSCANの例

INSCANには，例に現われたセンテンス以外に次のものがある。

RECURSE ： 自分自身のグラフを呼び出す。

CALL ： 外部サブルーチンを呼び出す。

また，アクショングラフによる記述の中で次のものが禁止されている。

1. 左 回 帰

すなわち

```
A：CHOICE(1，2)；
  1：RECURSE；"A"；GOOD；
  2："A"；GOOD．
```

という表現は禁止されており，

```
A："A"；EXECUTE B；GOOD．
B：CHOICE(1，2)；
  1：EXECUTE A；
  2：GOOD．
```

というように書きなおさなければならない。

2. 入力記号を含めたあともどり

すなわち，

```
Q：CHOICE(1，2)；
  1："B"；EXECUTE R；GOOD；
  2："B"；EXECUTE S；GOOD．
```

という表現は禁示されており，

```
Q："B"；CHOICE(1，2)；
  1：EXECUTE R；GOOD；
  2：EXECUTE S；GOOD．
```

というように書かなければならない。これは前者の表現では1において入力記号と"B"と比較され等しければRを実行し，ここでfalse の状態の時は，すでに1グラフ要素を実行しているために分枝点にもどれないからである。

3. 分枝指定においてnullを最初にもってくること。

分枝は書かれた順にテストするためnullは最後にもっていかなければならない。

INSCANにおいてのaction Graphの使用は，flow chart的なものとしてわかりやすいと思われるが論文の中でSemanticsの記述に関しては，何も述べていないので断定はできないが，複雑な処理はCALLによって外部サブルーチンの呼び出しをおこなうのが普通の方法であると考えらえる。もしそうであるならば，Semantics記述の手段をもう少し考えておいてほしいと思う。また，DM－1システム中の何の役割をはたすかについても不明である。

参　照　文　献

1. M.Resnick, J.Sable

   INSCAN: a Syntax-directed language Processor

   Proc. ACM 1968 PP 423 ～ 432

   （図2－14，15）は上記文献より引用させていただいた。

2. P.J.Dixon, J.Sable

   DM-1-a generalized data management System

   SJCC 1967 PP 185 ～ 198

2.10 Canonic Translator

Canonic Translator はマサチュセッツ工科大学（ＭＩＴ）において Project MAC の一環として作成されたものである。

このシステムの最大特徴は，通常のコンパイラ・コンパイラが Context Free 文法をあつかうのに対し，Context Sensitive 文法をあつかえるように作られている点にある。

このシステムでは，規則は Canon と呼ばれたとえば次のように記述する。

① a_1 set A_1 ϕ a_2 set a_2 ϕ …… ϕ a_n set A_n ⊢ b set B

これは a_1 ， a_2 ，…… ，a_n がそれぞれ set A_1，set A_2，……，set A_n の要素の時 b は set B の要素であることを示す。

例

v letter ϕ x expression ⊢ v＝x assignment

これは，ＢＮＦで書けば

＜assignment＞ ::＝＜letter＞＝＜expression＞

の意味となる。

② a_1 a_2 …… a_n set B

これは， a_1, a_2, ……, a_n が setB の要素であることを示す。

例

0 1 2 3 ‥ ‥ ‥ ‥ 8 9 digit

これは，BNF で書けば

＜digit＞ ::＝0｜1｜2｜3｜……｜8｜9

の意味となる。

③ ＜a_1＜a_2 …… ＜a_n＞

これは， a_1 a_2 …… a_n の順にならんでいることを意味する。

例

x identifier ϕ y letter ϕ z digit ⊢ ＜x＜y，z＞

identifier

これは，BNF で書けば

<identifier>::=<identifier><letter>|<identifier>
<digit>

この記述方法をもちいて，Syntax および Semantics を記述するわけである。ここでPL/I－like なGO TO ステートメントを考えてみよう。

DCL 変数 LABEL；…… ラベル変数の定義

GO TO ラベル変数；…… L 1，ラベル変数
BR 1
の命令の発生。

GO TO ラベル定数； B ラベル定数
の命令の発生

このような場合

ℓ label－const ⊢ GO TO ℓ； goto stm with ref label ℓ
label－var Λ[*1] assembler stms
B * BRANCH TO ℓ

ℓ label－var ⊢ GO TO ℓ； goto stm with ref label Λ
label－var ℓ assembler stms
L 1，ℓ
BR 1

v label－var ⊢ DCL V LABEL；dcl stm with dcl
label－var

というように記述する。goto stm with ref label は，他の場所においてこのラベルが定義されているかどうかのチェックのために使用する。

*1 Λ は null（空）を意味する。

このシステムは，Context sensitive な文法を受け入れるために，理論的には，他のシステムにくらべると適用範囲ははるかに大きいが，次のような欠点を持っている。

① プログラムのスピードは普通のコンパイラよりうちわにみて1000倍遅い。

② ソースプログラムが多量の時に適応できない。

③ エラーが発生した時，ソースプログラムの中で最后に調べられた文字を示すことができるがコンパイルの続行ができない。

このため実用化には問題点が多いといえる。

参照文献

1. J.J. Donovan, H.F. Ledgard
   A formal system for the specification of the syntax and translation of computer languages
   FJCC 1967 PP 553～569

2. J.W. Alsop
   A canonic transfator
   MAC-TR-46 1967

3. 構文解析手法

構文解析手法の開発は，TWS に掲載されているものは省略した。また，Mckeeman によるMixed strategy precedence については，2.1 XPL で紹介したのでここでは省略した。

3.1 Weak precedenee 文法および右順位文法

weak precedence 文法は，J．D．Ichbiah 及びS．P．Morse により，右順位文法は井上によりほぼ同時期に定義されたものであり，文法の範囲は若干右順位文法の方が広いが，基本思想は全く同じものである。

Weak precedence 文法

次のような条件を満たすとき，文法をweak precedence 文法という。

① Sをstart 記号としたとき。

$S \overset{*}{\Rightarrow} \alpha AB\beta$

なる2つの記号A，Bの間に，一義的なweak precedence 関係が成立する。

② 右辺が全く同じである生成規則はない。

③ $U_1 \longrightarrow x\ S\ y$

$U_2 \longrightarrow y$

のとき，S と U_1 の間にweak precedence 関係がない。

weak precedence関係

① $U \longrightarrow \alpha\ U_1\ B\beta$ なる生成規則があるとき

$U_1 \overset{*}{\Rightarrow} z\ A$ なるA に対して

$A \doteq B$ である。

② $U \longrightarrow \alpha\ U_1\ U_2\ \beta$ なる生成規則があるとき

$U_1 \overset{*}{\Rightarrow} z\ A$，$U_2 \overset{*}{\Rightarrow} B\omega$ なるA，Bに対して

$A \lessdot B$ である。

③ $U \longrightarrow \alpha AB\beta$ なる生成規則があるとき

$A \lessdot\cdot B$ である。

④ $U \longrightarrow \alpha AU_1\beta$ なる生成規則があるとき

$U_1 \overset{*}{\Rightarrow} B\omega$ なるBに対して

$A \lessdot\cdot B$ である。

右順位文法

次のような条件を満たすとき，文法を右順位文法という。

① Sをstart記号としたとき

$S \overset{*}{\Rightarrow} \alpha AB\beta$

なる2つの記号A，Bの間に一義的な右順位関係が成立する。

② 右辺が全く同じ生成規則はない。

③ $U_1 \longrightarrow \alpha\beta$

$U_2 \longrightarrow \beta$

のとき

$C \longrightarrow \gamma Hi(\alpha)D\delta$，$D \overset{*}{\Rightarrow} Tn-i(\alpha)B\delta'$

は存在しない。

右順位関係

ここで次のものを定義する

$L(A)=\{F \mid A \overset{*}{\Rightarrow} F\alpha,\ A \in V_N,\ F \in V,\ \alpha \in V^*\}$

$R(A)=\{F \mid A \overset{*}{\Rightarrow} \alpha F,\ A \in V_N,\ F \in V,\ \alpha \in V^*\}$

$L_T(A)=\{F \mid A \overset{*}{\Rightarrow} F\alpha,\ F \in V_T,\ A \in V_N\} \cup \{A \mid A \in V_T\}$

① $U \longrightarrow \alpha U_1 U_2 \beta$ なる生成規則があるとき

$A \in R(U_1)$，$B \in L_T(U_2)$ なるA，Bに対して

$A \cdot\gtrdot B$

② $U \longrightarrow \alpha A U_1 \beta$ なる生成規則があるとき

$B \in L_T(U_1)$ なるBに対して

$A \leqq\cdot B$

構　文　解　析

weak precedence 文法，右順位文法のいずれにおいても構文解析手法は全く同じである。すなわち，スタックの頭の記号(a)と入力記号(b)の順位関係を調べる。a ⋖ b ならばスタックに push down し，a ⋗ b ならばスタックと生成規則の右辺が一致するものの中で最長の規則を適用し，新しい記号をスタックの頭に入れて解析を進めてゆく，a ⋖ b でも a ⋗ b でもなければエラーである。

参　照　文　献

1. J.D. Ichbiah, S.P. Morse
   A Technique for Generating Almost Optimal Floyd-Evans Production for Precedence Grammars
   C.ACM Vol 13 No.8 1970 PP 501～508

2. 井上謙蔵
   順位言語の右順位解析
   情報処理 Vol 11 No.4 1970 PP 231～233

3. 井上謙蔵
   右順位文法
   情報処理 Vol 11 No.8 1970 PP 449～456

4. 井上謙蔵 他
   右順位文法に対する構文解析プログラムの発生
   第12回 プログラミング・シンポジウム 1971 PP 64～76

3.2 Total Precedence 文法

この文法は，Alain Colmerauer によって定義されたものである。

Binary Relaions

Set E 上の binary relation を Cartesian 積 $E \times E$ の subset ρ とし定義する。

$$a \rho b \quad \cdots\cdots\cdots \quad (a, b) \in \rho$$

binary relationはギリシア小文字，又は特殊記号で示す。

$E \times E$ に関する ρ の complement $\bar{\rho}$ は

$$\bar{\rho} = E \times E - \rho$$

binary relation $\overset{\exists}{\rho}$，　on　set E

$\Rightarrow$　$\rho\sigma$ はE上の binary relation で

$$a \rho\sigma b \overset{def}{\equiv} \overset{\exists}{C} \in E \quad s.t. \quad G \rho C, \quad C \sigma b$$

この積が結合的であり，結合に関して左右に分配できることが容易にわかる。結合性によって，binary relation ρ についての（非負整数）累乗が定義できる。

$$\rho^i = \rho \rho^{i-1}, \quad a \rho^0 b \equiv [a = b]$$

transitive closure ρ^+ は次のように定義できる。

$$\rho^+ = \bigcup_{i=1}^{\infty} \rho^i$$

Eがn元からなる有限集合である時

対　$a, b \in E$　s.t. $a \rho^0 b$,　$k > n$

$\Rightarrow$　$\ell \leq n$　s.t. $a \rho^\ell b$

従って　$\rho^+ = \bigcup_{i=1}^{\infty} \rho^i = \bigcup_{i=1}^{n} \rho^i$

Relation α, λ, ρ の定義

文法Gの $V_T \cup V_N$ 上の3つの binary relation を定義することが必要で

ある。ここでαは"adjacent"，λは"left"，ρは"right"を指すことにする。

$^{\forall}A, B \in V_T \cup V_N$に対して

$A \alpha B \equiv [\ ^{\exists}U \quad V_N, ^{\exists}x, y \quad (V_T \cup V_N)^* \ s.t.\ U \rightarrow xABy\ ]$

$A \lambda B \equiv [\ ^{\exists}y \quad (V_T \cup V_N)^* \ s.t.\ A \rightarrow By\ ]$

$A \rho B \equiv [\ ^{\exists}x \quad (V_T \cup V_N)^* \ s.t.\ B \rightarrow xA\ ]$

あいまいでない total precedence relation の存在に関する必要十分条件は

$(\rho^+ \alpha \cap \alpha) \cup (\alpha \lambda^+ \cap \alpha) \cup (\rho^+ \alpha \lambda \cap \alpha) \cup (\rho^+ \alpha \cap \alpha \lambda^+) = \phi$ である。

3つの binary relation $\lessdot$，$\doteq$，$\gtrdot$が（文法G上の$V_T \cup V_N$の中の vocaburary についての）total precedence relation であり，これに関連する analyzer が$V_T \cup V_N$上のすべての Stringを解析することのできるのは，次の包含関係を満たすときであり，かつ，この時に限る。

$\alpha \subset \doteq$，$\alpha \lambda^+ \subset \lessdot$，$\rho^+ \alpha \subset \gtrdot$，

$\rho^+ \alpha \lambda^+ \subset \lessdot \cup \gtrdot$

Relation $\lessdot$，$\doteq$，$\gtrdot$ の計算

Boolean matrix によってα，λ，ρの各 relation を表現することができ $\lessdot$，$\doteq$，$\gtrdot$ を表現する Boolean matrix を得ることができる。これは，コンピュータ上での最も簡単な手続きである。

これを手で行なうためには，次のようにすればよい。

（x，y $\in (V_T \cup V_N)^*$）：

① $U \longrightarrow xXYy$ の形の規則が存在するようなX，Yの対の各々に対して十字を書き，X，Yを書きこむ。

X	Y

② $U \longrightarrow xX$ の形の規則が存在し，かつUが十字の左半分にあらわれており，かつXが同じ十字の左下にあらわれていないUとXの対の各々に対して左下の隅にXをつけ加える。これ以上の変更が行なわれないところ

まで step 2をくりかえす。

③ U⟶ Yy の形の規則が存在し，かつUが十字の左にあらわれていて，かつ，Yが同じ十字の右下にあらわれていないUとYの記号対の各々に対して，右下にYを加える。これ以上の変更が行なわれないところまで step 3 をくりかえす。

さて，記号X，Yの対の各々に対して，次のことが容易にわかるであろう。

① configuration  は $X\alpha Y$ に対応している。

② configuration  は $X\alpha\lambda+Y$ に対応している。

③ configuration  は $X\rho+\alpha Y$ に対応している。

④ configuration  は $X\rho+\alpha\lambda+Y$ に対応している。

そして，relation $\lessdot\cdot$，$\doteq$，$\cdot\gtrdot$ はただちに導かれる。

構 文 解 析

total precedence 文法による構文解析は，S_1，S_2 という2つのスタックをもちいる。

ここで $(\alpha, \beta) \longrightarrow (\alpha', \beta')$ とは，スタック S_1 に頭から順に α というものがはいっており，同様に S_2 に β があった時，スタック S_1 に α の代り α'，スタック S_2 に β の代りに β' を入れることを意味する。

構文解析は次の手順によっておこなわれる。

初期状態として，スタック S_1 には #，スタック S_2 には解析したい記号列 ω 及び # を入れておく。

① $A \lessdot\cdot B$ あるいは（$A=\#$ かつ $B\neq\#$）ならば

$(A, B) \longrightarrow (A\langle B, \Lambda)$

② $A \doteq B$ ならば

$(A, B) \longrightarrow (AB, \Lambda)$

③ $A \cdot\gtrdot B$ ならば

（A，B）⟶（A〕，B）

④ （〔u〕，Λ）⟶（Λ，U）

これは，U→uなる生成規則の適用を示す。

（注） Λ は空を示す。

この手順により，スタックS1 に＃，スタックS2 にS＃（SはStart記号を示す）になった時解析は終了する。

参照文献

1. A. Colmerauer
   Total Brecedence Relation
   J.ACM Vol. 17 No.1 1970 PP 14～30

3.3 SLR(k)

SLR(k)はF.L.DeRemerにより定義されたものである。

Context Free 文法がs＋1個の生成規則を持ち，$\#_0$，$\#_1$，…… $\#_s$ を文法の中で使用していない特殊記号とする。

P番目の生成規則がA→ωで，$\alpha' = \rho A \beta$ と $\alpha = \rho \omega \beta$ がStart記号Sから $S \xrightarrow{*} \alpha' \longrightarrow \alpha$ というcanonical derivation されるcanonical form であるとき，$\rho\omega\#_p$ はαのcharacteristic string であるという。

このcharacteristic stringはregular言語なのでこれを受け付けるFinite State Machine を作ることができる。これをCharacteristic Finite State Machine（CFSM）と呼ぶ。

例として次のような生成規則を持つ文法のCFSM を作り上げると（図3－1）になる。

S ⟶ ⊢E⊣

E ⟶ E＋T

E ⟶ T

T ⟶ P↑T

T ⟶ P

P ⟶ （E）

P ⟶ i

CFSM において $\#_n$ の遷移と他の記号の遷移がある状態をinadequate 状態という。（（図3－1）においては7の状態）。

このinadequate 状態において，k個の記号のlook ahead により状態遷移を一意的に決定できる時，この文法をSinple LR(k)（SLR(k)）と言う。

例においては，$\#_4$ の1記号look ahead のsetは｛⊣，＋，）｝となり分離できるので，SLR(1)文法である。

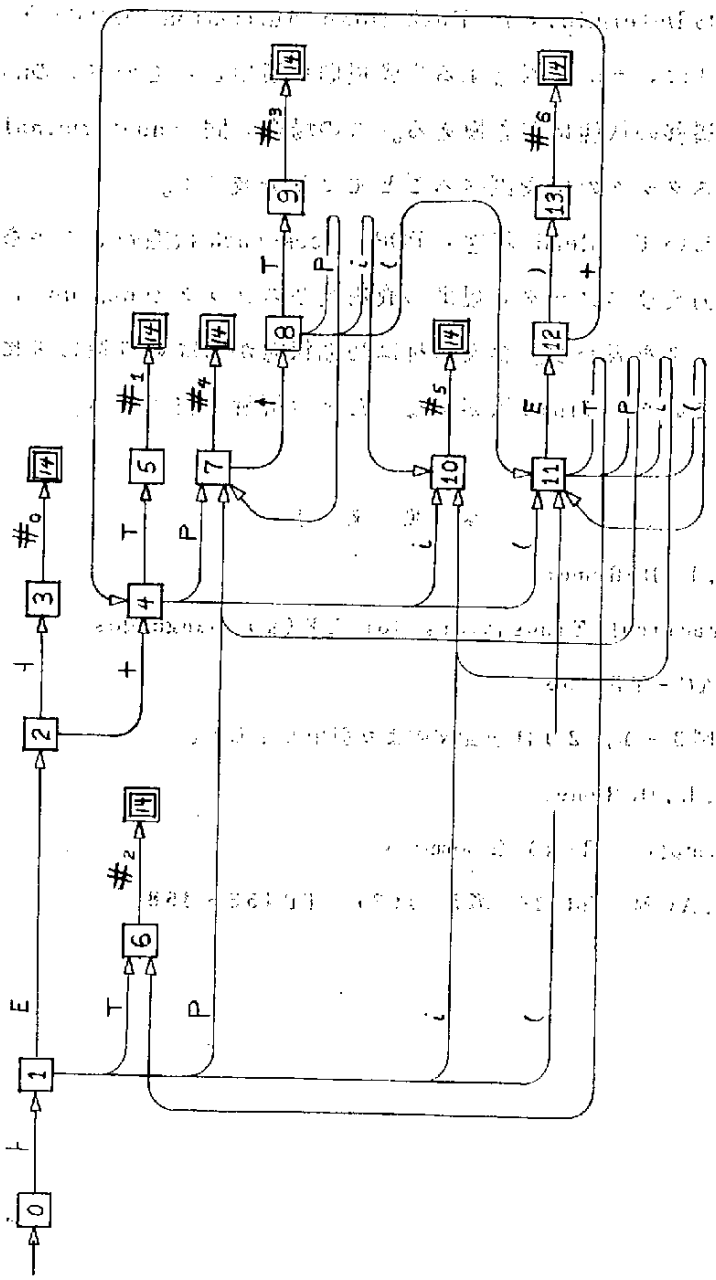


図3－1　CFSM

構　文　解　析

CFSM からDeterministic Push Down Automaton（DPDA） を作り上げる。この方法は，#p に対応する生成規則を適用し，その左辺のnonterminal記号に対する遷移の状態に置き換える。この場合，同一 nonterminal 記号が2個以上あればスタックの中を調べることにより分岐する。

ＤＰＤＡにおいて，Read 状態，POP，look-back 状態の３つがあり，それぞれ記号の入力及びスタック，規則の適用及びスタックの pop up ，スタックの内容のチェックを表わす。構文解析は初期状態からはいり遷移図にしたがって状態を移していき，final状態になったとき解析が終了する。

参　照　文　献

1. F.L.DeRemer
Practical Translators for LP(k) Languages
MAC-TR-65
（図３－１，２）は上記文献より引用しました。

2. F.L.DeRemer
Simple LR(k) Grammars
C.ACM VoL 14 №7 1971 PP453～458

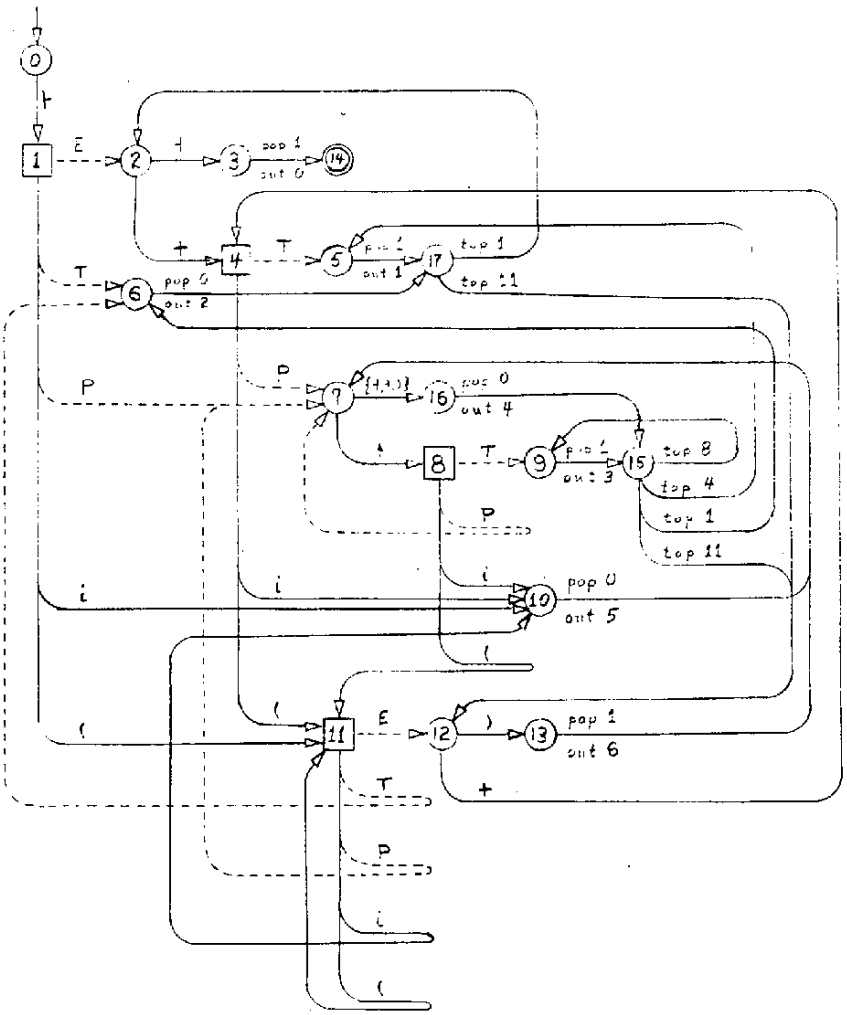

0
1
E
2
+
3
pop 1
out 0
14
+
4
T
5
pop 1
out 1
17
top 1
top 11
T
6
pop 0
out 2
P
P
7
{1,4,5}
16
pos 0
out 4
+
8
T
9
pop 1
out 3
15
top 8
top 4
top 1
top 11
P
i
i
i
10
pop 0
out 5
(
(
11
E
12
)
13
pop 1
out 6
+
T
P
i
(

図 3 − 2　D P D A

第　2　部

コンパイラ・ジェネレータの作成

第　1　編

コンパイラ・ジェネレータ ＪＰＡＣ

目　　次

1. JPACの概要

JPACはFACOM 230-60のMonitor Vの管理下で動作するコンパラ・ジェネレータであり，ジェネレートされるコンパイラもFACOM 230-60用のものである。

このシステムは，目的とするコンパイラ言語の文法のSyntaxおよびSemanticsを入力することにより所要のコンパイラを作り出すことができる。

現在までに開発された内外のコンパイラ・ジェネレータは研究的要素の特に強いものと，ある程度実用性を持ったものに分けられるが，JPACはどちらかと言えば後者に属し，特に次のような点を目標とした。

① 使いやすさ … 必ずしもコンパイラ作成の経験者でなくともコンパイラ言語の文法をBNFで記述すること及びプログラミングの力があれば使えることを目標とする。

② 実用性 … 今回作成するものはFORTRANクラスの言語のコンパイラが作成でき，コンパイラの効率（処理速度，メモリ占有量など）が実用上問題とならない程度のものであること。

③ 拡張性 … JPACの仕様は，この報告書に記載されている範囲で終わるものではない。すなわちこれを使用した結果，その仕様を変更する必要性がでてきた場合には修正，変更が簡単にできなければならない。

JPACで使用している構文解析手法，SyntaxおよびSemantics記述方法の特徴は次の点である。

① De Remerの手法の使用

De RemerはKnuthの作り出したLR(k)文法の解析が膨大なテーブルを必要とするため，これを小さくすることに努力した。その結果，SLR(k), L(m)R(k), LR(k)の順に，順次大きな文法を定義してそれらに対し統一的な解析手法を作り出した。この手法によれば小さな文法の範囲におさまるものはSLR(k)

を，大きな文法の範囲になればL(m)R(k) を使用するという方法がとり得る。

② LSD 言語の開発

LSD は第1部の概説で述べたコンパイラ記述言語に必要となる機能をすべて持っている言語である。この言語によりコンパイラの Semantics をプロシーデュアルに記述できる。

LSD は Semantics 記述をするに十分な機能を持っているが，今後 Semantics記述に必要となる標準ルーチンをまとめ，ライブラリ化することによりさらに強力な機能を持たせることができる。

LSD はコンパイラ記述言語として独立に使用することもでき，Syntax 解析プログラムも LSD を使用して作成した。LSD の外部仕様書はこれだけを独立して利用できるため第Ⅱ編にまとめた。

③ SYNTAXの記述

Syntaxの記述は，規則が適用された時に制御をわたすセマンティックス・ルーチンを BNF 記述のあとに指定するという形でおこなわれる。このセマンティックス・ルーチンは LSD で記述した Procedure あるいは標準ルーチンでなければならない。さらに標準ルーチンを整備すれば LSD によるプロシーデュアルな記述を使うことなく Semantics を簡単なルーチン・コールだけで済ますことができる。

2. JPACの動作概略

JPACでは，作成したいコンパイラの文法を入力してシンタックス・テーブル等を作り出すSTAGEとSemanticsを記述するためのシステム記述言語のコンパイラLSDの2つのステップより構成されている。

STAGEはバッカス記法（BNF）で書かれたコンパイラの文法を入力して，遷移テーブル等のSyntax 関係のテーブルを出力する。

STAGEが受けつける文法はSLR(k) (k＝0, 1, 2) あるいはL(m)R(k) (k＝0, 1, m＝1, 2)である。

ユーザの記述したSyntax は，まずSLR(0)文法が適用され得るか否かが調べられ適用できなかったならば，SLR(1)，SLR(2) の順で調べ適用できるレベルで遷移テーブル等が出力される。しかしSLR(k)自体が適用不　ならば今度はL(1)R(0)，L(1)R(1)，L(2)R(0)，L(2)R(1)の順に調べ，適用可能な文法における遷移テーブル等が出力される。この遷移テーブル等はアセンブラの形で出力される。SLR(k)，L(m)R(k) 文法のいずれも適用不可能な場合は，どの規則が原因となって不可能であるかを表示して処理を中止する。

LSD はコンパイラ作成の手法として一般に使用する機能すなわちテーブル，スタック，ビット，文字のとりあつかいが簡単にできるように考慮したシステム記述言語であって，Syntax 記述部分で定義した生成規則の適用によりどのようなアクションをとるかを指示する部分である。LSDも オブジェクト・プログラムとしてアセンブラを出力する。

STAGE，LSD の2つのステップで出力され，アセンブラ言語で書かれたオブジェクトは，FASP ステップによりリロケータブルになる。そして必要とあればLIBEによりユーザ・ライブラリの作成もできる。

コンパイラ作成のためにはリロケータブルおよびシステム・ライブラリ，必要ならばユーザ・ライブラリをつなげるLIEDにより実行形式を作ればよい。

おのおのステップは全く独立に実行することができる。すなわち文法の定義

だけを変える必要があればSTAGE-FASP-LIEDのステップを使えばよい。またコンパイラ作成途中においてSemantics記述部分に虫があった場合にはLSD-FASP-LIED ステップにより新コンパイラを作りあげればよい。この手順は(図2-1)JPACの概略に示しているとおりである。

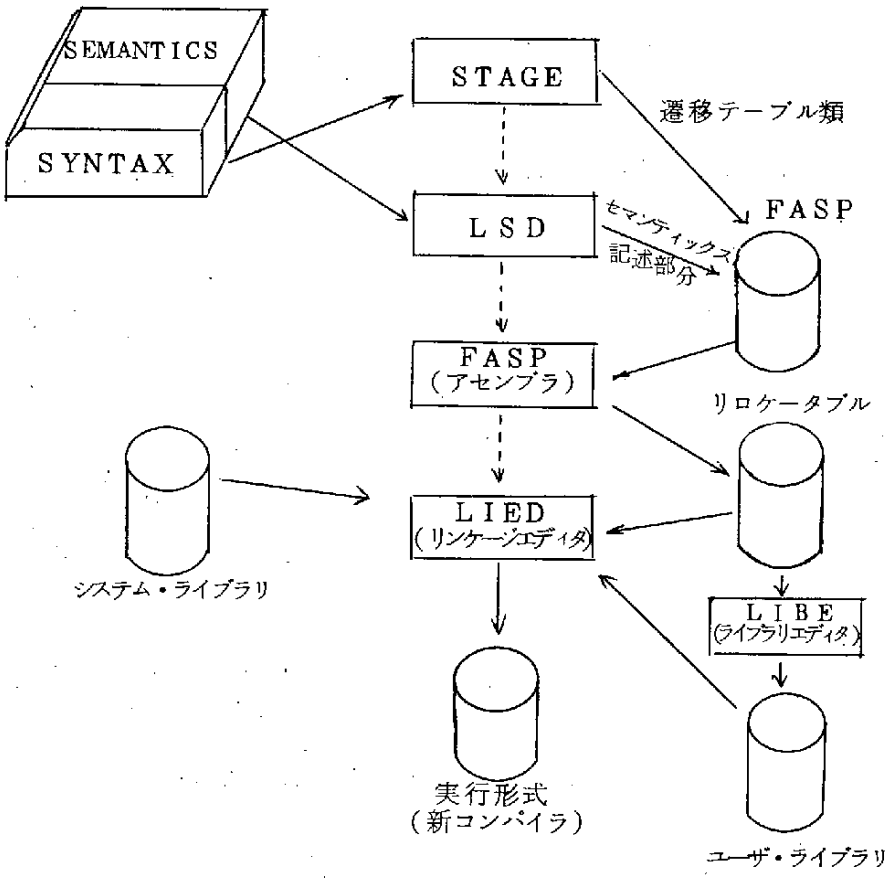


図2-1　JPACの概略

JPAC を使用することにより作り出されるコンパイラは(図2-2)の形をしている。この中で遷移テーブル類およびセマンティックス・ルーチンはそれぞれSTAGEおよびLSDステップによって作り出されるものである。

コントローラはまず最初にユーザがオウンコーディングしたInitiatorにコントロールをわたす。Initiator においてはセマンティックス・ルーチンの初

期設定等をおこなう。再びコントロールがInitiatorからもどってくるとLexical Analyzerを通してソース・プログラムを入力し，遷移テーブルにしたがって構文解析を進める。

遷移テーブルの中で，セマンテックス・ルーチンの呼び出し指定があると(図2-3)に示したCG_STACK 等に構文に関する情報をセットしてセマンテックス・ルーチスにコントロールをわたす。セマンティックス・ルーチンにおいては，構文に対応する処理をユーザの責任においておこなう。この場合，オブジェクト・プログラムをアセンブラ(FASP) で出力するとすれば，システム・ライブラリの中にあるコード発生ルーチンを使用できる。

コントローラあるいはセマンティックス・ルーチンにおいてエラーが検出された場合に，エラー・メッセージの出力，ソースプログラムの読みとばし（不必要な部分の)を行なうのがエラー処理ルーチンの役割である。エラーがあった場合には，コントローラはリカバリー可能な点までさかのぼって遷移テーブルのドライブを始める。

ソースプログラムを読み終わり，構文解析によってstart記号に還元されると，コントローラは制御をTerminatorにわたす。Terminatorはユーザオウンコードルーチンであって，最終処理をおこなう。

Terminatorからコントローラを呼び出して，再び新しいプログラムのコンパイルも可能であり，また任意のユーザ・ルーチンにコントロールをわたしてもよい。

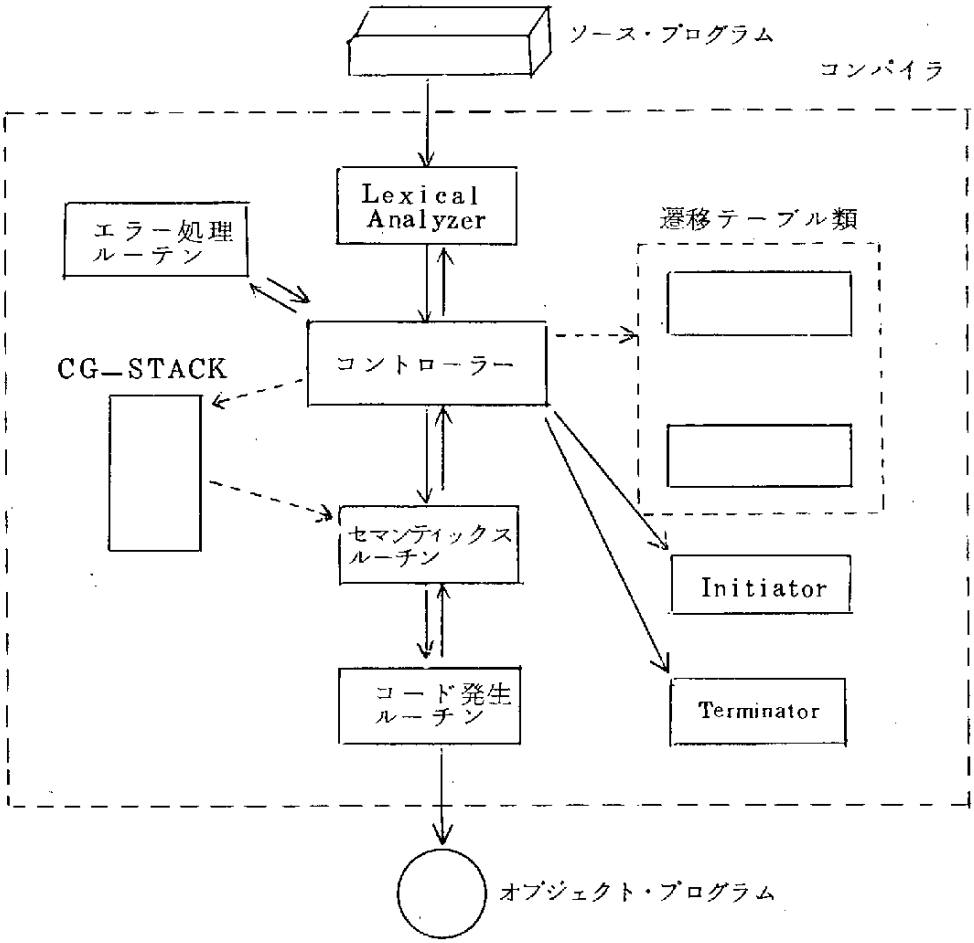
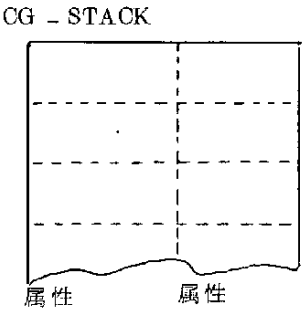


図 2-2　コンパイラ・ジェネレータによって作成されたコンパイラ概略



numeric　ユーザーオプション
ただし，ADDRESS，POINTER，数値，文字（4文字），BITのスカラー変数に限る。

図 2-3

3. JPACの言語仕様

JPAC は次の2つの定義部より構成される。

SYNTAX DEFINITION;

SEMANTICS DEFINITION;

以下に JPAC の文法を BNF で記述する。

< Alphabet > : : = A | B | C | D | …… | X | Y | Z

<Numeric> : : = 0 | 1 | 2 | 3 | …… | 7 | 8 | 9

< Alpha numeric > : : = < Alphabet > | <Numeric> |

<Name> : : = < Alphabet > | <Name> | < Alpha numeric > |

<Name> _ < Alpha numeric >

(注) 名前は 256 字以内で，かつ最初の 12 文字は同じものであってはいけない。

また，BNF 記述中次の2文字以上の記号の連なりは，右に書いた記号とみなす。

これ等は BNF 記述のための記号と terminal 記号を区別するためである。

▼▼ → ▼

▼< → <

▼> → >

▼| → |

▼::= → ::=

３．１　SYNTAX DEFINITION

この定義部においてはコンパイラ・ジェネレータで作成しようとしている言語の文法を BNF で記述し，その規則が適用された時にどのセマンティックス・ルーチンにコントロールをわたすかを指定するものである。

SYNTAX　DEFINTION は次の３つの節より構成される。

BASIC SYMBOL;
BASIC STATEMENT;
SYNTAX RULE;

① BASIC　SYMBOL

文法を BNF で記述する場合 terminal記号は文字そのものを，nonterminal記号は記号名を＜　＞でくくってそれを示している。

例

＜NUMERIC＞::＝0｜1｜2｜3｜4｜5｜6｜7｜8｜9

＜NUMBER＞::＝＜NUMERIC＞｜＜NUMBER＞＜NUMERIC＞

通常＜identifier＞，＜NUMERIC＞，＜NUMBER＞等の nonterminal 記号はプログラムにほぼ共通であり，SYNTAX定義部において定義するよりはLexical Analyzer を own－codingした方が処理も速く有意義なことが多い。しかしながら Lexical Analyzer を通してソースプログラムを入力する場合には，その最小単位は単語というレベルになるので単語を terminal記号とみなす必要がある。

BASIC SYMBOL節は Lexical Analyzerで解釈する単語はこれこれであるということを定義する部分である。

BASIC SYMBOL 節で定義された名前には定義された順に１から連番がふられる。これをBASIC SYMBOL 番号と呼び，Lexical Analyzerで Basic Symbol を解析した時に，この番号を呼んだルーチンに返さなければならない。

BASIC SYMBOL を定義しない場合にはコンパイラ・ジェネレータが

用意しているユーティリティ・ルーチンを使うことができるが，このルーチンはソースプログラムを1文字ずつとってくる役割のみを持っているため出来上がるコンパイラのスピードはかなり遅くなる。

一般形

BASIC SYMBOL
<Bacic Symbol definition>;

<Basic Symbol definition>::=<空>|<Basic Symbol list>
<Basic Symbol list>::=<Basic Symbol>|
<Basic Symbol list>,<Basic Symbol>
<Basic Symbol>::=▼<<Name>▼>

② BASIC STATEMENT

構文解析中にエラーが起った場合，すなわちソース●プログラムの中に文法に合わない文字列がでてきたときには，コンパイラはエラー・コンディションを発生する。

この場合に，コンパイラが処理を続行するためには，ソース・プログラムの一部を無視しなければならない。しかしながらソース・プログラムのどこからどこまでを無視するかということは非常に大きな問題である。

JPAC においては，この基準としてBasic Statementとして定義されたnonterminal 記号の直後の記号からユーザが指定する記号までを無視することにしている。

FORTRAN を例にとって考えてみよう。

<Program>::=⊢<Statement list> END ⊣
<Statement list>::=<Statement>|
<Statement list><Statement>
<Statement>::=<non executable>|<logical if>|<do>|
<sigma executable>
<logical if>::= IF (<logical exp>) <sigma executable>

＜sigma executable＞::＝＜Assign＞|＜GO TO＞|＜arith if＞| ……

というように定義されている時に＜statement＞をBASIC STATEMENTと宣言し，Lexical Analyzer においてステートメントのエンド（開始行のみの時はその行の最後，継続行がある時には最終継続行の最後）までを読みとばせるようにしておけば，エラーが起った場合にそのステートメントのみを無視することが可能である。

③ SYNTAX RULE

この節は作成したいコンパイラの文法をBNFで記述し，その規則が適用された場合に制御をわたすべきセマンティックスルーチンを指定するものである。

この節で定義されている生成規則からは（図2-2）で示されているものの中の遷移テーブル類が作成される。この遷移テーブル類は，コンパイラが動作する時にコントローラーが解読し生成規則が適用されると，CG_STACKの上からn個の要素を生成規則の右辺として処理するように指定されたセマンティックス・ルーチンにコントロールをわたす。セマンティックス・ルーチンから帰ると，CG_STRに還元されたnonterminal記号をセットする，そしてCG_STACKにスタックし再び遷移テーブル類の解読にはいる。

セマンティックス・ルーチンの指定がない場合には，コントローラーは〔生成規則の右辺の項の数－1項〕をpop upして，スタックの最上位に還元されたnonterminal記号をセットして遷移テーブル類の解読をつづける。

一般形

＜Production＞::＝＜Production rule＞|

＜nonterminal＞::＝＜Production element list＞

＜Production element list＞::＝＜Production element＞|

＜Production element list＞ |＜Production element＞

＜Production element＞::＝＜Right Part＞:＜Semantics exec Part＞

＜Semantics exec part＞::＝＜Semantics call＞

＜Semantics exec part＞,＜Semantics call＞

＜Semantics call＞::＝＜Name＞|＜Name＞(＜Parameter list＞)

＜Production rule＞::＝＜nonterminal＞▼::＝＜Right Part＞

＜Right Part＞::＝＜Symbol＞|＜Right Part＞＜Symbol＞

(注)＜Semantics call＞の中で使用する名前は8文字以内で，セマンティックス・ルーチン名でなければならない。＜Parameter list＞はスタック名あるいは定数あるいは＜Common Variable list＞中の変数でなければならない。

＜Production List＞::＝＜Production＞;|

＜Production List＞＜Production＞;

＜Syntax rule＞::＝SYNTAX RULE ;＜Production List＞

SYNTAX RULE END;

(注)＜Production list＞ の中で最初の＜Production＞はStart 記号の記述でなければならない。また，一つの nonterminal 記号に対しては一度しか ＜Production＞ を定義できない。

SYNTAX RULE は＜Production＞ ；を必要な個数だけ並びたものであるということができ，＜Production＞ はおおざっぱに考えれば次の2つの形のうちいずれかである。

＜E＞::＝＜E＞ ＋＜T＞：EXP；

＜P＞::＝＜id＞；

前者は，＜E＞＋＜T＞という形が現われた時，EXP という名前のセマンティックス・ルーチンにコントロールをわたすことを指示している。セマンティックス・ルーチンの処理が終ると，＜E＞＋＜T＞は新らしく＜E＞になる。後者は＜id＞が現われた時＜P＞に変えることのみを指定し，セマンティックス・ルーチンにはコントロールをわたさない。

＜Semantics exec part＞すなわちセマンティックス・ルーチンの実行指

定は，セマンティックス・ルーチン名及びコンパイラ・ジェネレータ標準ルーチンを(,)でくぎって任意の個数指定することができる。ここで指定されたものは，指定された順序にしたがって実行されてゆく。

一つの nonterminal 記号に対して2個以上の生成規則がある場合，たとえば，

<E>::=<E>+<T>：EXP；

<E>::=<T>；

という時には，これを(|)で区切って書かなければならない。すなわち

<E>::=<E>+<T>：EXP | <T>；

という形で指定しなければならない。

例3－1 生成規則が次の8個より成り立っているものとする。

表3－1

S → | A |

A → identifier＝E；

E → E＋T

E → T

T → P↑T

T → P

P → （E）

P → identifier

これをJPACの言語仕様にしたがって，SYNTAX定義部分のみを書けば，次のようになる。

表3－2

```
SYNTAX  DEFINITION;
BASIC  SYMBOL;
<IDENTIFIER>;
SYNTAX  RULE;
```

```
<S>::= ⊢<A>⊣ ;
<A>::= <IDENTIFIER>=<E>:ASS;
<E>::= <E> + <T>:EXP | <T>;
<T>::= <P> ↑ <T>:TERM | <P>;
<P>::= (<E>) | <IDENTIFIER>:PRIM;
'SYNTAX RULE END:
```

3．2　SEMANTICS DEFINITION

SYNTAX DEFINITION定義部においては，作成したいコンパイラ言語の文法の定義および，制御をわたすセマンティックス・ルーチンの指定をおこなった。すなわち，ある生成規則の適用がおこなわれた場合にセマンティックス・ルーチンの指定があれば，そのルーチンに一時的にコントロールがわたされる。

SEMANTICS DEFINITIONは，このセマンティックス・ルーチンのプログラムの集合ということができる。

セマンティックス・ルーチンの作成のためには（図3－1）に示した概略を正しく理解する必要がある。

＜E＞::＝＜E＞＋＜T＞：EXP；

という生成規則があって，EXPルーチンにコントロールがわたった場合に，コントローラからの連絡情報は，CG_STACKにあり，次のような形をしている。

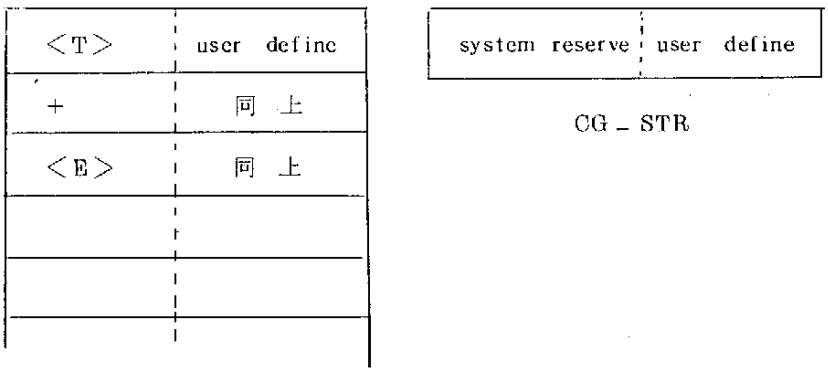


図 3－1

セマンティックス・ルーチンにおいては，必要な処理をおこない＜T＞，＋，＜E＞をpop upして，CG_STRの第2要素に還元された＜E＞に関する情報をセットしてコントローラに帰る。こうするとコントロラでは，CG_STACKの頭にpush downして構文解析を続行する。

セマンティックスの記述方法はLSDの外部仕様にしたがうが，メイン・ルーチンとして使用してはいけない。

SEMANTICS DEFINITIONの一般形

SEMANTICS DEFINITION;
<Common Variable list>
SEMANTICS BODY;
<Semantics Procedure list>
//END

<Common Variable list> ::= <Common Variable> |
<Common Variable list><Common Variable>
<Common Variable>:: =<DCL statement>
<DCL statement>はLSDの外部仕様書にあるDCLステートメントである。

<Common Variable list> として定義されたものについては<Semantics Procedure> の中で宣言してはいけない。すなわち<Common Variable list>は<Semantics Procedure list> 中にあるすべてのProcedureの中にDCLステートメントとしてさし込みがおこなわれる。

<Semantics Procedure list>::= <Semantics Procedure> |
<Semantics Procedure list> <Semantics Procedure>
<Semantics Procedure>::=// LSD <External Procedure>

<External Procedure>はLSD外部仕様書にあるexternal procedure であり，このprocedure名がセマンティックス・ルーチン名に対応している。

例3－2

(表3－2)で定義した文法に関するものをJPACで記述してみる。セマンティックス記述は，コンパイラの言語仕様により，あるいはLexical Analyzerの機能によりいちじるしく異なるが，ここではLexical Analyzerはidentifierを認識した時にname tableに登録し，そのテーブルのポインタがスタッ

クにのせられているものとする。

ここで示したのは，オブジェクト・コードをアセンブラ言語で直接出力する例であり，この他に種々の方法が考えられる。

```
*****  JPAC ( JIPDEC AUTOMATIC COMPILER GENERATOR )   LIST  *****

  1                SYNTAX DEFINITION;
  2                  BASIC SYMBOL;
  3                   <IDENTIFIER>,<FACT>;
  4                  SYNTAX RULE;
  5                   <S> ::="<A>";
  6                   <A> ::=<IDENTIFIER>=<E>:ASS;
  7                   <E> ::=<E>+<T>:EXP|<T>;
  8                   <T> ::=<P><FACT><I>:TERM|<P>;
  9                   <P> ::=(<E>)|<IDENTIFIER>:PRIM;
 10                SYNTAX RULE END;
```

```
*****  JPAC ( JIPDEC AUTOMATIC COMPILER GENERATOR )   LIST  *****

  1                SEMANTICS DEFINITION;
  2                    DCL OPTBL(50) TABLE CONTROLLED(OPT),
  3                        * OPNAME CHAR(6),
  4                        * OPDISP,
  5                        * OPADD  CHAR(4),
  6                        * OPATTR BIT;
  7                    DCL OPSTR STRUCTURE EXTERNAL,
  8                        * STRNAME CHAR(6),
  9                        * STRDISP,
 10                        * STRADD  CHAR(4),
 11                        * STRATTR BIT;
 12                    DCL NAMETBL(200) TABLE CONTROLLED(TBP),
 13                        * TBNAME CHAR(6),
 14                        * TBATTR BIT,
 15                        * TBADD;
 16                    DCL CG_STACK(100) STACK,
 17                        * CG_SYS,
 18                        * CG_SPNT POINTER;
 19                    DCL CG_STR STRUCTURE EXTERNAL,
 20                        * CG_SSYS,
 21                        * CG_SSPNT POINTER;
 22                    DCL WORKCNT EXTERNAL INIT(0);
```

```
23              SEMANTICS BODY;
24             //LSD
25                PRIM : PROC;
26                     POP CG_STACK INTO CG_STR;
27                    /* MOVE PRIMARY INFORMATION TO OPTBL */
28                     STRNAME=CG_SSPNT.TBNAME;
29                     STRATTR=CG_SSPNT.TBATTR;
30                     STRDISP=0;
31                     STRADD=' ';
32                     SET OPSTR INTO OPTBL;
33                     CG_SSPNT=OPT;
34                     RETURN;
35                   END;
36             //LSD
37                ASS : PROC;
38                    /*  GENERATE          LA     <T>    */
39                     CALL PUT(' ','LA',CG_SPNT,OPTBL);
40                     POP 2;
41                    /*  GENERATE          STA    <IDENTIFIER>  */
42                     CALL PRIM;
43                     CALL PUT(' ','STA',OPTBL);
44                     RETURN;
45                   END;
46             //LSD
47                EXP : PROC;
48                    /*  GENERATE          LA     <T>    */
49                     CALL PUT(' ','LA',CG_SPNT,OPTBL);
50                     POP 2;
51                    /*  GENERATE          A      <E>    */
52                     CALL PUT(' ','A',CG_SPNT,OPTBL);
53                    /*  GENERATE          STA    <E>    */
54                     CALL WORKGET;
55                     CALL PUT(' ','STA',OPTBL);
56                     POP 1;
57                     RETURN;
58                   END;
59             //LSD
60                TERM : PROC;
61                    /* GENERATE           SXJ,7  C.FACT    */
62                     STRNAME='C.FACT';
63                     STRDISP=0;
64                     STRADD='7';
65                     STRATTR=B'0';
66                     CALL PUT(' ','SXJ',OPSTR);
67                     POP  CG_STACK INTO CG_STR;
68                     POP 1;
69                     CALL PUT(' ','NOP',CG_SPNT,OPTBL);
70                     CALL PUT(' ','NOP',CG_SSPNT,OPTBL);
71                     CALL WORKGET;
72                     CALL PUT(' ','STA',OPTBL);
```

```
73                      POP 1;
74                      RETURN;
75                    END;
76              //LSD
77                 WORKGET : PROC;
78                      /* WORK AREA OPERAND SET  */
79                      STRNAME='L:WK';
80                      STRDISP=WORKCNT;
81                      STRADD=' ';
82                      STRATTR=B'0';
83                      SET OPSTR INTO OPTBL;
84                      WORKCNT=WORKCNT+1;
85                      CG_SSPNT=OPT;
86                      RETURN;
87                    END;
88              //END
```

3.3 Lexical Analyzer

コンパイラ・ジェネレータを使ってコンパイラを作成する場合に，Basic Symbol の定義をすることができることは，すでに述べたとおりである。Basic Symbol を使用した時，ユーザは自分の責任でLexical Analyzer は（図2－2）に示されているように，出来上ったコンパイラのソースプログラムを読み込み，単語の列に変換し，コントローラに情報を渡す役割を持っているのでユーザは次のような情報を指定された変数にセットするように作らなければならない。

なおBasic Symbol を使用しない場合には，コンパイラ・ジェネレータのユーティリティ・ルーチンとして用意してあるLexical Analyzer があるので，ユーザはこれを使えばよい。しかしながら，出来上ったコンパイラの大きさ，コンパイルスピードに関してはいちじるしく悪くなることがある。

Lexical Analyzer のルーチン名

CG_LEX

情報の受け渡し

CG_BF： Terminal Symbol あるいはBasic Symbol をセットする場所である。

属　性　：　CHARACTER 256文字，EXTERNAL

CG_SORT: Terminal Symbol の場合には 0，Basic Symbol の場合には，Basic Symbol 番号を入れる。

属　性　：　NUMERIC，EXTERNAL

CG－CC：CG_BFにBasic Symbol がはいっている場合に，CG_STACKのuser define areaにセットしてほしいものを入れておく。

属　性　：　1 word，EXTERNAL

その他

ＬＳＤによってＣＧ_ＬＥＸを作成する場合は問題はないが，アセンブ

ラで作成する場合はＬＳＤとアセンブラのリンクの章をよく読んでつながるようにすること。

例2－3　FORTRAN　Lexical　Analyzer

前提

次の文字列のあとに空白あるいは特殊文字がくる場合は，これをリザーブ・ワードとして，文あるいは行の識別のためにもちいる。

END
COMMON
DIMENSION
EQUIVALENCE
SUBROUTINE
FUNCTION
READ
WRITE
FORMAT
GO
DO
IF
CALL
RETURN
STOP
CONTINUE

以上のものはBasic Symbolとして宣言されており，その番号はEND＝1から順でCONTINUE＝16となっている。

これ以外に＜identifier＞,＜Integer－number＞,＜real－number＞がそれぞれBasic Symbol番号17，18，19で宣言されているものとする。

```
CG_LEX:       PROC;
              DCL TAB(10) TABLE CONTROLLED(PP),
                   *NUM1 CHAR(1) INIT('0','1','2','3','4','5','6','7',
                        '8','9'),
                   *NUM2 INIT(0,1,2,3,4,5,6,7,8,9);
              DCL BASIC_TABLE(16) TABLE CONTROLLED(P),
                   *BASIC_SY     CHAR(12) INIT('END          ',
                                               'COMMON       ',
                                               'DIMENSION    ',
                                               'EQUIVALENCE  ',
                                               'SUBROUTINE   ',
                                               'FUNCTION     ',
                                               'READ         ',
                                               'WRITE        ',
                                               'FORMAT       ',
                                               'GO           ',
                                               'DO           ',
                                               'IF           ',
                                               'CALL         ',
                                               'RETURN       ',
                                               'STOP         ',
                                               'CONTINUE     '),
                   *BASIC_NO INIT(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
                        16);
              DCL CODE CHAR(1);
              DCL CG_BF CHAR(256) EXTERNAL,
                  CG_SORT EXTERNAL,
                  CG_CC EXTERNAL;
              DCL LABEL_SW BIT(1) EXTERNAL;
              DCL KEY_BF CHAR(12),OR_CHAR CHAR(1),NAME_SW BIT(1),
                  OR_SW BIT(1),TYPE_SW BIT(1),DIG_SW BIT(1),
                  EXP_SW BIT(1),ASTER_SW BIT(1);
              DCL BLANK_SW BIT(1);
      /*  START   LEXICAL-ANALIZER  */
              CG_BF = ' ';
              CG_SORT = 0;
              CG_CC = 0;
              KEY_BF = ' ';
              RESETB NAME_SW,1;
              CODE = OR_CHAR;
              IF OR_SW,1 IS OFF THEN GO TO LEX_7;
              RESETB OR_SW,1;
              GO TO LEX_9;
LEX_7:        CALL CODE_GET(CODE);
              IF LABEL_SW,1 IS ON THEN RETURN;
LEX_9:        IF CODE = ' ' THEN GO TO LEX_7;
              I = 1;
              IF CODE IS ALPHA THEN GO TO LEX_39;
              IF CODE IS NUMERIC THEN GO TO LEX_13;
              IF CODE ¬= '.' THEN GO TO LEX_54;
```

```
LEX_13:     RESETB TYPE_SW.1;
            IF CODE = '.' THEN SETB TYPE_SW.1;
LEX_15:     SUBSTR(CG_BF,I) = CODE;
            I = I + 1;
            CALL CODE_GET(CODE);
            IF CODE IS NUMERIC THEN GO TO LEX_15;
            IF CODE IS ALPHA THEN GO TO LEX_26;
            IF CODE = '.' THEN GO TO LEX_24;
            SETB OR_SW.1;
            OR_CHAR = CODE;
            IF TYPE_SW.1 IS ON THEN GO TO LEX_23;
            CG_SORT = 18;
            RETURN;
LEX_23:     CG_SORT = 19;
            RETURN;
LEX_24:     IF TYPE_SW.1 IS ON THEN CALL ERROR(10);
            SETB TYPE_SW.1;
            GO TO LEX_15;
LEX_26:     IF CODE ¬= 'E' THEN CALL ERROR(11);
            SETB TYPE_SW.1;
            RESETB DIG_SW.1;
            RESETB EXP_SW.1;
LEX_29:     SUBSTR(CG_BF,I) = CODE;
            I = I + 1;
            CALL CODE_GET(CODE);
            IF CODE IS ALPHA THEN CALL ERROR(12);
            IF CODE IS NUMERIC THEN GO TO LEX_38;
            IF CODE  = '+','-' THEN GO TO LEX_36;
LEX_35:     IF DIG_SW.1 IS OFF THEN CALL ERROR(13);
            SETB OR_SW.1;
            OR_CHAR = CODE;
            GO TO LEX_23;
LEX_36:     IF EXP_SW.1 IS OFF THEN GO TO LEX_37;
            GO TO LEX_35;
LEX_37:     SETB EXP_SW.1;
            GO TO LEX_29;
LEX_38:     SETB DIG_SW.1;
            SETB EXP_SW.1;
            GO TO LEX_29;
LEX_39:     SUBSTR(CG_BF,I) = CODE;
            IF NAME_SW.1 IS ON THEN GO TO LEX_50;
            KEY_BF = SUBSTR(CG_BF,1,I);
            SEARCH BASIC_SY     KEYED KEY_BF IF NOT GO TO LEX_50;
            I = I + 1;
            CALL CODE_GET(CODE);
            IF CODE IS ALPHA THEN GO TO LEX_49;
            IF CODE IS NUMERIC THEN GO TO LEX_49;
            CG_SORT = P.BASIC_NO;
            CG_CC = I-1;
LEX_48:     SETB OR_SW.1;
```

```
                OR_CHAR = CODE;
LEX_63:         RETURN;
LEX_49:         SETB NAME_SW.1;
                GO TO LEX_39;
LEX_50:         I = I + 1;
LEX_51:         CALL CODE_GET(CODE);
                IF CODE ¬= ' ' THEN GO TO LEX_53;
                SETB BLANK_SW.1;
                GO TO LEX_51;
LEX_53:         RESETB BLANK_SW.1;
                IF CODE IS ALPHANUMERIC THEN GO TO LEX_39;
                GO TO LEX_48;
LEX_54:         IF CODE IS ALPHANUMERIC THEN CALL ERROR(14);
                IF CODE = '*' THEN SETB ASTER_SW.1;
LEX_56:         SUBSTR(CG_BF,I) = CODE;
                I = I + 1;
                IF ASTER_SW.1 IS OFF THEN RETURN;
                CALL CODE_GET(CODE);
                IF CODE = '*' THEN GO TO LEX_62;
                SETB OR_SW.1;
                OR_CHAR = CODE;
                RETURN;
LEX_62:         RESETB ASTER_SW.1;
                GO TO LEX_56;
        /*  END   LEXICAL-ANALIZER */
        /*  START   CODE-GET  */
CODE_GET:       PROC(G_CODE);
                DCL INPBUF CHAR(80),COUNT INIT(72),STNO;
                DCL G_CODE CHAR(1);
                DCL JUMP_SW BIT(1);
                RESETB LABEL_SW.1;
                IF JUMP_SW.1 IS ON THEN GO TO GET_11;
                IF COUNT < 72 THEN GO TO GET_8;
GET_2:          READ SYSIN INTO INPBUF
                     AT END GO TO GET_20;
                WRITE SYSPRT FROM INPBUF;
                IF SUBSTR(INPBUF,1) = 'C' THEN GO TO GET_2;
                IF SUBSTR(INPBUF,6) = ' ','0' THEN GO TO GET_10;
GET_7:          COUNT = 7;
GET_8:          G_CODE = SUBSTR(INPBUF,COUNT);
                COUNT = COUNT +1;
                RETURN;
GET_10:         IF BLANK_SW.1 IS OFF THEN GO TO GET_11;
                SETB JUMP_SW.1;
                GO TO LEX_63;
GET_11:         RESETB JUMP_SW.1;
                STNO = 0;
                DO I = 1 TO 5;
                G_CODE = SUBSTR(INPBUF,I);
                SEARCH NUM1 KEYED G_CODE IF NOT GO TO GET_15;
```

```
                STNO = STNO * 10 + NUM2;
                GO TO GET_16;
GET_15:         IF G_CODE ¬= ' ' THEN CALL ERROR(1);
GET_16:         END;
                IF STNO = 0 THEN GO TO GET_7;
                CG_BF = STNO;
                CG_SORT = 18;
                CG_CC = 0;
                COUNT = 7;
                SETB LABEL_SW,1;
                RETURN;
GET_20:         G_CODE = '*';
                RETURN;
                END;
        /*  END   CODE-GET  */
ERROR:          PROC(NO);
                DCL MESSAGE CHAR(25) INIT(' CG_LEX ERROR    NO.    ');
                SUBSTR(MESSAGE,21,2) = NO;
                WRITE SYSPRT FROM MESSAGE;
                STOP;
                END;
                END;
```

３．４　エラー処理ルーチン

ＪＰＡＣ で作り出されたコンパイラにおいて，コントローラは構文解析中にエラーを見つけ出した場合に，ソース・プログラム中のどこでエラーが起ったかを表示することができるが，そのエラーがどういう種類のものであるかをユーザに分り易く出力することができない。また，ソース・プログラムのどこまでを無視してエラー・リカバリーするのが適当であるかを判断することができない。

このためエラーが起った場合に，適切なエラー・メッセージの出力とエラー・リカバリーのために，ユーザはエラー処理ルーチンをコンパイラの中に組み込むことが可能になっている。

エラー処理ルーチンは，ＢＡＳＩＣ　ＳＴＡＴＥＭＥＮＴを指定した場合には，ユーザがオウンコーディングしなければならないものである。

コントローラがソース・プログラムの構文解析中にエラーを検出した場合には，エラー個所をＳＹＳＰＲＴに出力したのちこのルーチンを呼ぶようになっている。セマンティックス・ルーチンの中でエラーが検出された場合には，ユーザは直接このルーチンを呼ぶようにしなければならない。

このルーチンではエラー・メッセージの出力とソース・プログラムの読みとばしをしてリターンすればよい。こうするとコントローラは ＢＡＳＩＣ ＳＴＡＴＥＭＥＮＴ に対応するところまで ＣＧ_ＳＴＡＣＫ を pop upして，ソース・プログラムの構文解析を続行する。

ＢＡＳＩＣ　ＳＴＡＴＥＭＥＮＴ の指定をしなければ，このルーチンは用意する必要はない。しかしながら そのような場合には，コントローラがエラーを検出するとエラー 個所を出力したのち Terminator にコントロールをわたすことになり，エラー個所以降は全くチェックをしないことになる。

4. STAGE内部処理

以下にSTAGEの内部処理について述べるが，説明に際してFranklin Lewis De Remer のPRACTICAL TRANSLATORS FOR LR(k) LANGUAGES を一部引用した。

4.1 処理概略

STAGEはSYNTAX DEFINITIONから遷移図を作成しFASPソースの形で遷移テーブル類を出力するものである。

STAGE の処理の流れは(図4-1)に示されるとおりである。

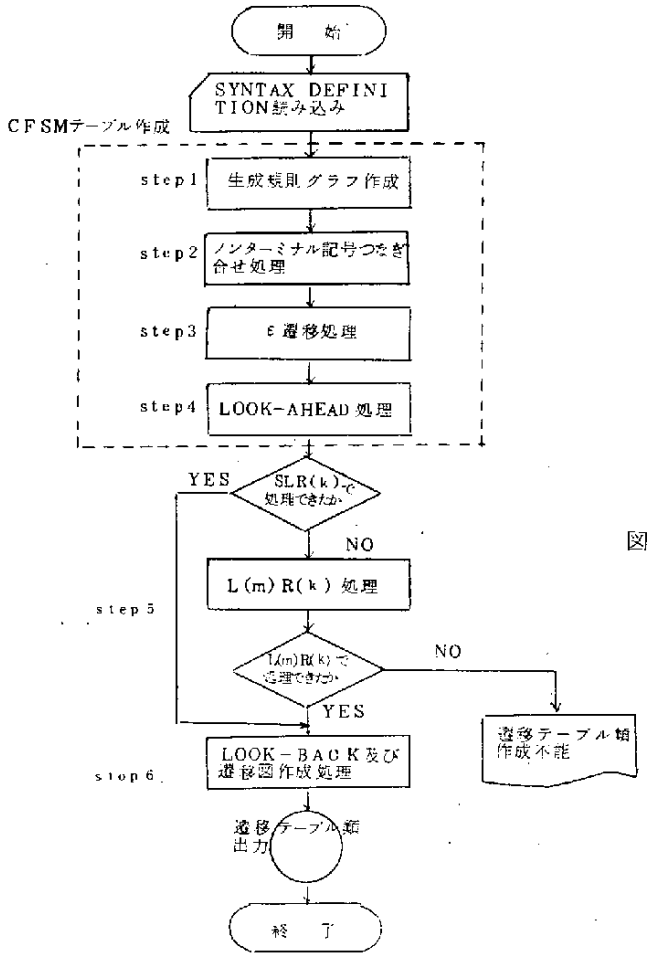


図 4-1 STAGEの処理の流れ

以下に（表4－1）で示すSYNTAX DEFINITIONが与えられた場合を例としてSTAGEの処理を説明する。

表4－1 SYNTAX DEFINITIONの例

```
SYNTAX DEFINITION ;
BASIC SYMBOL ;
<IDENTIFIER> ;
SYNTAX RULE ;
<S> ::= ⊢ <E> ⊣ : EXEC 0 ;
<E> ::= <E>+<T> : EXEC 1 | <T> : EXEC 2 ;
<T> ::= <P>↑<T> : EXEC 3 | <P> : EXEC 4 ;
<P> ::= <IDENTIFIER> | (<E>) : EXEC 5 ;
SYNTAX RULE END ;
```

４．２　生成規則グラフ作成

この処理は（図４－１）の step 1 に対応するものであり，与えられた生成規則（SYNTAX　RULE 節で与えられる）から node と arrow で表現した生成規則グラフを作成する。（arrow は遷移を表わし，node は状態を表わす）

生成規則グラフを作成する手順は以下の A，B の２つの部分に分かれる。

A. 生成規則から Characteristic String を作成する。

（表４－１）で示された例の場合の Characteristic String は（表４－２）で示される。

表 4-2　Characteristic String

0：	0：	$S' \rightarrow \vdash E \dashv$	$\#_0$ *-1)
	1：	$S' \rightarrow \vdash E'$	
1：	2：	$E' \rightarrow E + T$	$\#_1$
	3：	$E' \rightarrow E + T'$	
	4：	$E' \rightarrow E'$	
2：	5：	$E' \rightarrow T$	$\#_2$
	6：	$E' \rightarrow T'$	
3：	7：	$T' \rightarrow P \uparrow T$	$\#_3$
	8：	$T' \rightarrow P \uparrow T'$	
	9：	$T' \rightarrow P'$	
4：	10：	$T' \rightarrow P$	$\#_4$
	11：	$T' \rightarrow P'$	
5：	12：	$P' \rightarrow i$ *-2)	$\#_5$
6：	13：	$P' \rightarrow (E)$	$\#_6$
	14：	$P' \rightarrow (E')$	

↑ Characteristic番号

↑ 生成規則番号

*－1)　#n は n 番目の生成規則に対応する特殊記号であって，この記号にであうとその生成規則に対応するセマンティックス・ルーチンの実行をする。

*－2)　i：＜IDENTIFIER＞ を意味する。

B. Aで作成された Characteristic String から生成規則グラフを作成する。

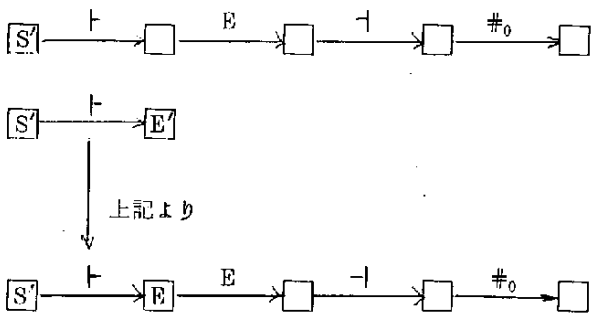


図 4-2　生成規則グラフの作成例

アルゴリズム

① Characteristic String 上ではじめてあらわれた記号，つまりダッシュのついている記号（例：S′, E′）はnode とする。

② 生成規則の各 terminal, nonterminal 記号はすべて arrow に対応する。

③ 各 arrow は必ず node から出て node に終るものとする。

④ 各 Characteristic String のうち，同一記号から始るものは1つにまとめる。

（表 4-2）では Characteristic 番号の（0,1),(2,3,4,5,6）等である。

⑤ node から node への arrow になんら該当する遷移記号が無いとき，その遷移記号として ε(無) 記号を割り当てる。

上記のアルゴリズムによって（表 4-2）の Characteristic String に対する生成規則グラフを作成すると（図 4-3）になる。

次に（図 4-3），(図 4-4) を例にして記号及び言葉の定義を行なう。

node 記号：（図 4-3）を見るといくつかの node はその中にダッシュのついている記号を持つ，この記号を node 記号という。

（図 4-3）では S′, E′, T′ etc …

SNODE　：（図４－３）を見てもわかるように，生成規則グラフはいくつかのグループに分かれている（各グループには生成規則グラフ番号が１つ対応している）そして各グループの最初のnodeを SNODE という。

（図４－３）では生成規則グラフ番号が０のSNODEはS′のnode 記号を持つnode である。

Ni　：ｉという状態にあるnode である。

（図４－４）ではNE′はnode 1 を示す。

arrow の START node 及び arrow の END node　：任意の arrow が Ni から出て Nj に終るとき，Ni，Nj をそれぞれ，その arrow の START node，END node という。

（図４－４）では arrow 3 の START node は node 1 であり END node は node 4である。

ARROW{S}：Ｓという遷移記号を持つ arrow である。

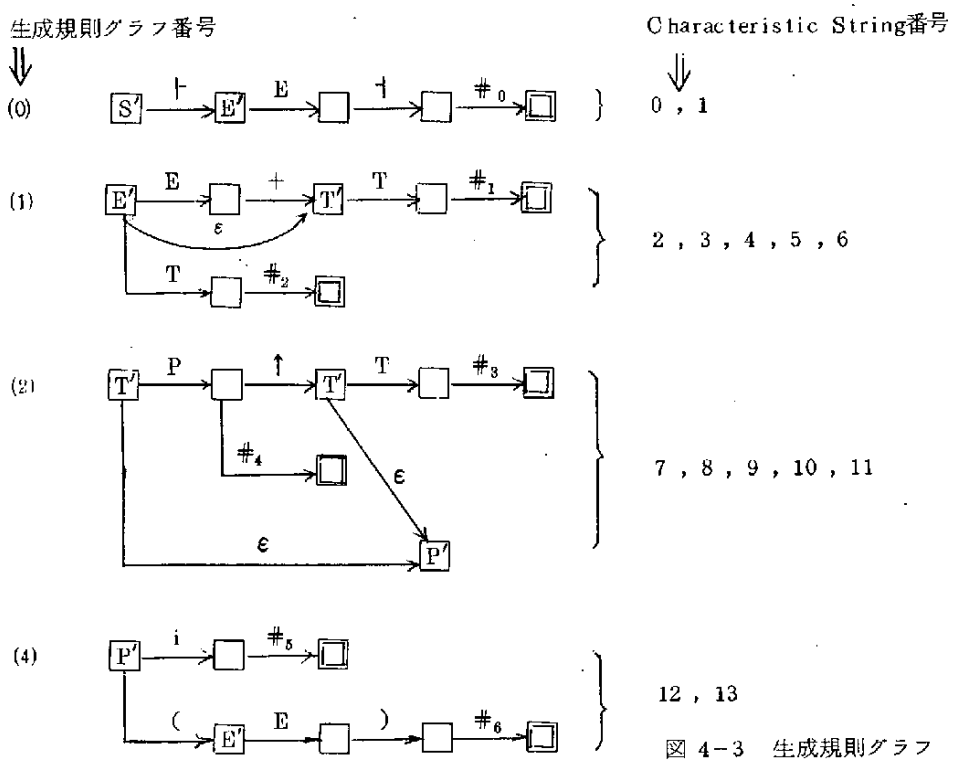


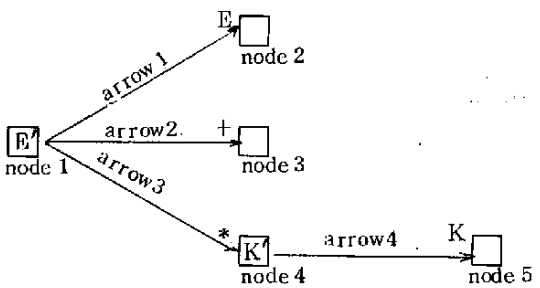
図 4－3　生成規則グラフ

Niの子 arrow：Ni をSTART node とする arrow の集合である。

Niの子 node：Ni の子 arrow の END node の集合である。

Niの子孫　：Ni から arrow 及び node を通って到達できる arrow 及び node の集合をいう。

（図4－4）では NE′（node 1）の子孫は arrow1～arrow4, 及び node 2～node 5 で，Nk′の子孫は arrow 4 と node 5 である。



E, +, *, Kはおのおの arrow1, arrow2, arrow3, arrow4 の遷移記号である。

図 4－4

Niの子 arrow{S}：Niの子 arrow のうち遷移記号としてSをもつものである。

Niの子 node{S}：Niの子 node のうちNiの子 arrow {S}を経由して到達する nodeo

#nの帰納記号　：#n 記号により reduce される nonterminal 記号。（表4－1）における #n の帰納記号はEである。

#n arrow　　：遷移記号として #n を持つ arrow で，arrow {#n}として参照されることもある。

４．３　Nonterminal 記号つなぎあわせ処理

この処理は（図４－１）のStep２に対応するものであり nonterminal 記号を結合してゆき，CFSM の母型となるグラフを作成する。（以下の説明ではこのグラフを母型グラフと呼ぶ）

結合処理アルゴリズム

① スタート・ステータスを表す node を母型グラフ上に作成する。
（（例４－１）の①参照）

② 母型グラフ上の node 記号を持つ node で未処理のものがあれば③へ，なければε遷移処理へ行く。

③ ②で見つけた未処理の node と同一の node 記号をSNODEとしてもつ生成規則グラフを見つける。そしてSNODE の子孫を未処理 node の子孫として追加した後②へ戻る。

結合処理の手続きは上記のように簡単なものだが，作成する母型グラフの冗長を無くすため結合処理アルゴリズム③は以下の手続きから構成されている。

3.1　未処理 node の子 arrow の遷移記号の集合をSUM１とする。SNODE の子 arrow の遷移記号の集合を SUM２とする。（但しε遷移のときは，その arrow のEND node の node 記号を仮の遷移記号とする）

3.2　SUM１，SUM２に含まれている記号を以下の３種に分類する。

3.2.1　SUM１にあってSUM２にないもの，これの集合をSUM３Aとする。

3.2.2　SUM２にあってSUM１にないもの，これの集合をSUM３Bとする。

3.2.3　SUM１，SUM２ともに含まれているが，その属性が異なるもの，これの集合をSUM３Cとする。

3.3　3.2 で分類した各集合について未処理 node への結合処理を行う。

3.3.1　SM∈SUM3Aなる記号SMを遷移記号とするarrow は，すでに母型グラフ上に作成済みなので考慮の必要はない。

3.3.2　SUM3 Bに含まれる記号についてはSNODEの子孫（この集合をSUM４とする）を母型グラフ上に複製し，その未処理 node の子孫とする。

但し該当生成規則グラフの結合が過去にされている場合は，作成するべき集合 SUM 4 はすでに母型グラフ上に作成されている。それゆえこの場合は，未処理 node から集合 SUM 4 を子孫とするようなarrowを母型グラフ上に作成するだけでよい。

また SNODE，未処理 node が一時的であるとき*-1) 集合 SUM 4 はすでに母型グラフ上に作成されている可能性がある。ゆえに作成済みか否かを調べ，前者の場合は未処理 node から集合 SUM 4 を子孫とするような arrow を母型グラフに作成するだけでよい。また後者の場合は集合 SUM 4 を未処理 node の子孫として母型グラフ上に作成する。

3.3.3 SUM 3 については未処理 node，SNODE ともに SM∈SUM3C となる SM に対して子 node{SM} を求め，それぞれ新しい未処理node SNODE（この新らしい未処理 node，SNODE は一時的なものなので一時的未処理 node，一時的 SNODE という）として 3.1 へ戻る。この作業は一時的未処理 node が無くなるまで繰り返される。

*-1) 3.3.3 参照

例 4-1

（表 4-1）で与えられた生成規則から，CFSMの母型となるグラフを作る手順を結合アルゴリズムの③に重点をおいて例示する。

(1) スタート・ステータスに該当する node を持ってくる。

図4-5 (1)終了時の母型グラフ

(2) （図 4-5）の NS' に対して生成規則グラフを結合する。この場合の生成規則グラフは（図 4-3）の生成規則グラフ番号(0)であらわされる。

3.1 より

SUM 1 = {φ}

SUM 2 = {| }

3.2 より

SUM 3 = {| }　　で SUM 3A = {ϕ}, SUM 3B = {| }

SUM3C= {ϕ}

3.3.2 によって(図4-6)ができる

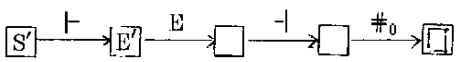


図4-6　(2)終了時の母型グラフ

(3)　(図4-6)を見るとE'という node シンボルを持つ node が新しく未処理 node として出現しているこれに対して生成規則グラフを結合するこの場合の生成規則グラフは生成規則グラフ番号(1)であらわされる。

3.1 より

SUM 1 = {E}

SUM 2 = {E,T',T}

3.2 より

SUM 3A = {ϕ}, SUM 3B = {T', T}, SUM 3C = {E} となる。

(4)　SUM 3B に対して 3.3.2 を適用すると(図4-7)となる。

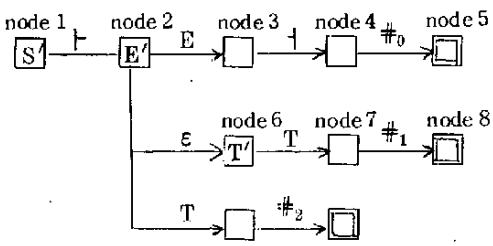


図4-7　(4)終了時の母型グラフ

(5)　次に SUM 3C に対して 3.3.3 を適用すると新しい一時的な未処理 node は node 3 で新しい一時的 SNODE は生成規則グラフ番号(1)であらわされるグラフの SNODE の子 node {E} である。

3.1 より

SUM 1 = {⊣}

SUM 2 = {+}

3.2 より

SUM 3A = {⊣}, SUM 3B = {+}, SUM 3C = {ϕ}となる。

SUM 3Bに対して 3.3.2 を適用する，この場合 SNODE，未処理 node は一時的ゆえ，node 3（未処理 node）の子 node {+} は母型グラフ上に作成済みの可能性がある。それゆえ，調べてみると node 2 の子 node{ε} すなわち node 6 として作成済みである（（図3-7）参照），それゆえnode 3 から node 6 へ遷移記号＋を持つ arrow を母型グラフへ追加する。結果（図3-8）となる。

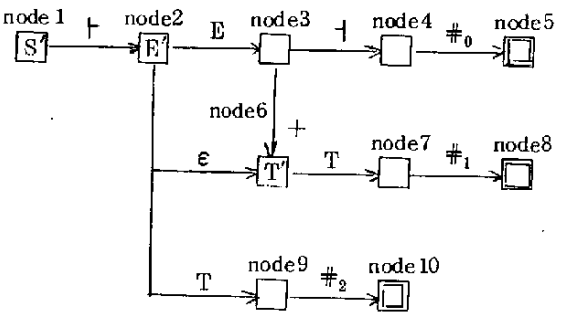


図4-8 (5)終了時の母型グラフ

(6) （図4-8）のT'に対して生成規則グラフ番号(2)のグラフを結合する。

3.1 より

SUM 1 = {T}

SUM 2 = {P, P'}

3.2 より

SUM 3A = {T}, SUM 3B = {P, P'}, SUM 3C = {ϕ}

となる。SUM 3B に対して 3.3.2 を適用すると（図4-9）になる。

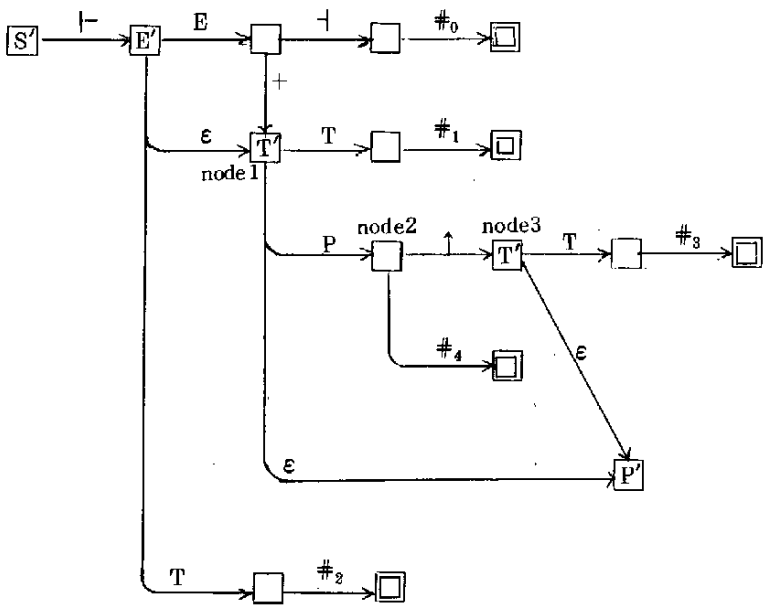


図4－9　(6)終了時の母型グラフ

(7)　(図4－9)をみるとnode シンボルT′とP′ を持つnode が新しく未処理nodeとして示されている。まずT′に生成規則グラフ番号(2)のグラフを適用

3.1 より

SUM 1 ＝{T, P′}

SUM 2 ＝{P, P′}

3.2 より

SUM 3A＝{T}, SUM 3B＝{P}, SUM 3C＝{P′} となる。

SUM 3Bに 3.3.2 を適用する，しかしこのとき生成規則グラフ番号(2)のグラフは(6)で結合済みなのでnode 3 の子 arrow {P}のEND nodeはnode 2になる。結果(図4－10) になる。

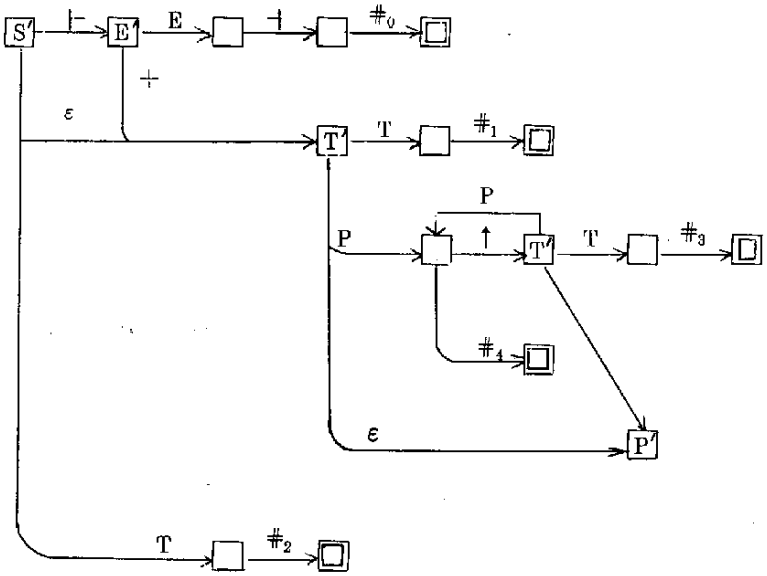


図 4－10　(7)終了時の母型グラフ

同様の手続きを繰り返すと最終的に（図 4－11）ができる。

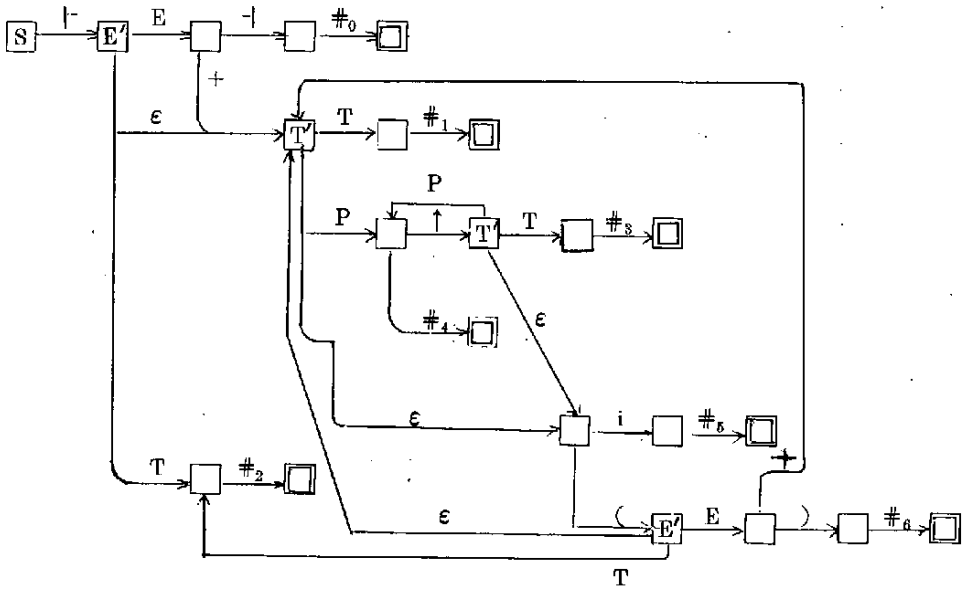


図 4－11　最終母型グラフ

４．４　ε遷移処理

この処理は（図４－１）のｓｔｅｐ３に対応する。

εによる遷移を，次のｎｏｄｅからの遷移におきかえる。

そのアルゴリズムを以下にのべる。

あるｎｏｄｅ Ｎｉ があってそのｎｏｄｅからε遷移があるとき，つまりＮｉの子ａｒｒｏｗ｛ε｝があるとき，子ａｒｒｏｗ｛ε｝のＥＮＤ ｎｏｄｅ Ｎｊ の子孫を子ａｒｒｏｗ｛ε｝のかわりにＮｉに結合する。

その場合Ｎｉ からの遷移とＮｊからの遷移に同一遷移があるとき，Ｎｉ の方の遷移を優先する。そしてこの操作を全てのε遷移が無くなるまで繰り返す。

例４－２

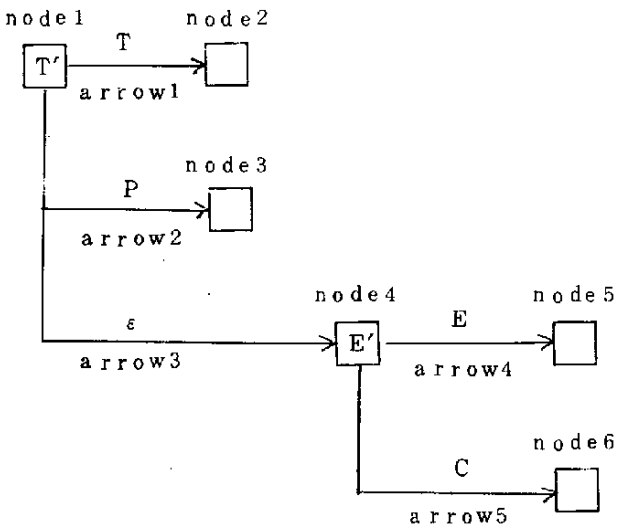


図４－１２　ε遷移処理前

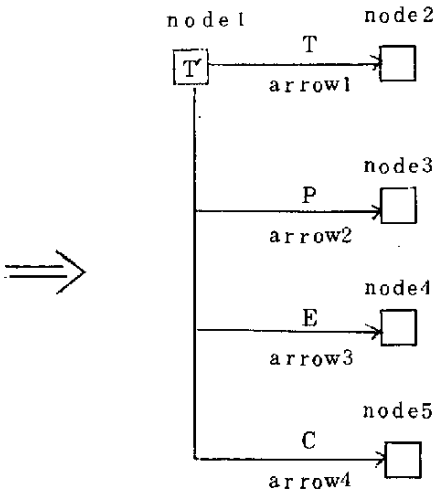


図４－１３　ε遷移処理後

（図４－１２）を例にとって上記手続きを説明する。node1からは｛T，P，ε｝の遷移がある。ゆえ上記手続きにより ε 遷移に該当する。arrow3を取り去りarrow3のEND nodeであるnode4の子孫つまりarrow4とarrow5をnode1に結合する。結果（図４－１３）となる。

例４－３

（例４－１）でできた母型グラフ（図４－１１）に対して上記ε処理を行ってできたものを（図４－１４）に示す。これがＣＦＳＭグラフである。

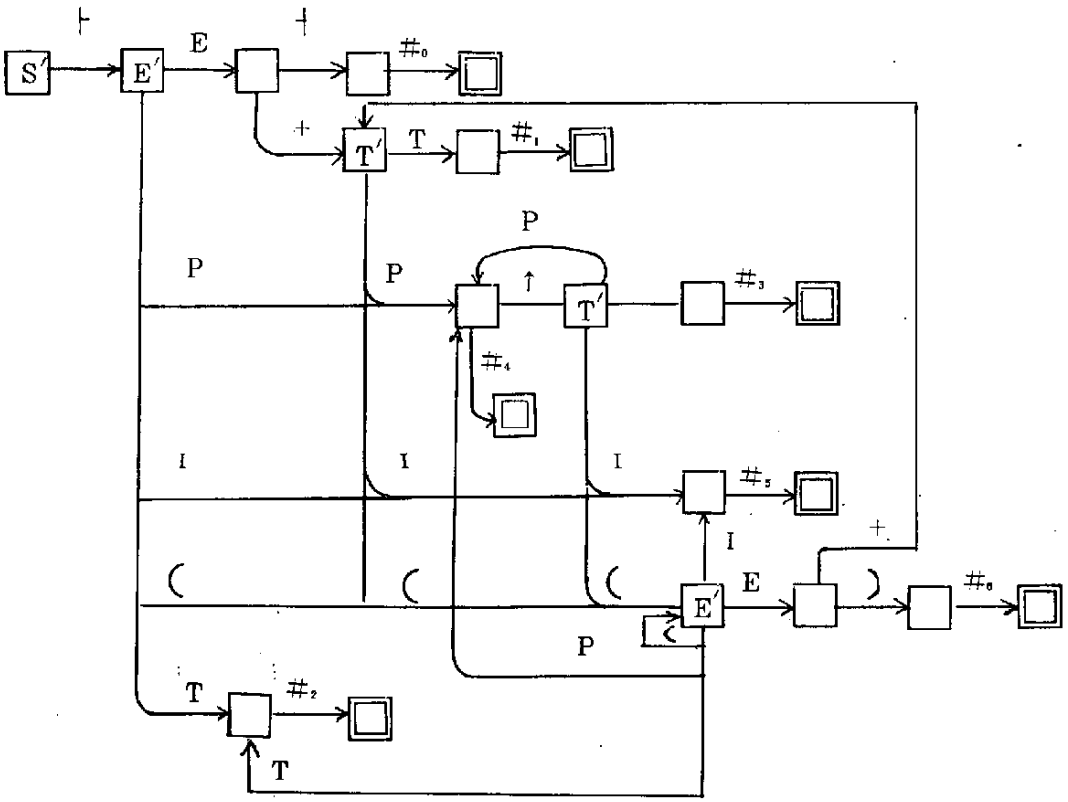


図４－１４　（表４－１）に示す文法に対するＣＦＳＭグラフ

４．５　LOOK－AHEAD　処理

この処理は（図４－１）のStep4に対応し，inadequate状態に対処するものである。

a・look－ahead　セットを求める。

（図４－１４）をみると同一nodeから#n arrowと他のarrowがでている場合があるが，この様な状態をinadequate状態という。

①　inadequate状態がなければ遷移図作成処理へ行く。

inadequate状態がある場合には該当する#nの帰納記号をAとするとF$^{K}_{T}$（A）と他のarrowの遷移記号と一致しないかどうか調べる。一致していれば②へ行く。一致していなければK＝K＋１として再び①へ行く。この場合K＝２までをおこなう。それでも駄目な場合L(m)R(k)処理へ行く。

②　#n arrowに関してF$^{K}_{T}$（A）をlook－ahead　セットと称し，その状態に達した時，K個の記号を先読みして，一致するものがあれば#nへ遷移するようにする。

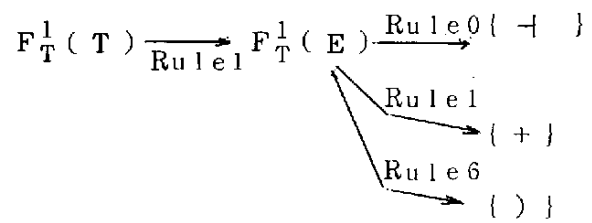
上記のF$^{K}_{T}$（A）で求めた記号列がlook－ahead　セットである。

例４－４

例として（図４－１４）の＃４の所のlook－ahead　セットを求めてみる。

＃４は（表４－１）によるとnonterminal記号TにreduceされるF$^{1}_{T}$（T）を求める

記号であるので



＃４のlook－aheadセットは｛⊣，＋，）｝となり（図４－１４）は

（図４－１５）となる。

b・ $F_T^K(T)$と$H_T^K(A)$

$F_T^K(T)$は，記号Tのあとにつづく K個の terminal 記号の列の集合を意味している。

$H_T^K(A)$はnonterminal 記号Aが terminal 記号になったとき，そのterminal記号とそのあとに続く terminal 記号列の始めのK個の記号列の集合を意味している。

T　　が$V_N \sqcup V_T$，

A，B，C，D，SがV_N　（Sはstart記号），

α，β，γ　　が$V_T^* \sqcup \phi$（ϕは空），

ρ　　がV^* のとき

$F_T^K(A) = \{ (K) : \beta \mid S \xrightarrow{*} \rho A \beta \}$

$H_T^K(A) = \{ (K) : \alpha\beta \mid S \to \rho A\beta,\ A \xrightarrow{*} \alpha \}$

とする。

$F_T^K(T)$の計算

$B \to \rho T\gamma$ のように右辺にAを含む生成規則をとりだす。

$\gamma = \phi$であれば　　$F_T^K(B) \in F_T^K(T)$

$\gamma \neq \phi$であるとき

$\gamma = \alpha$であれば

$|\alpha| \geq K$ならば　　$(k) : \alpha \in F_T^K(T)$

$|\alpha| < K$ならば　　$\alpha F_T^{K-|\alpha|}(B) \in F_T^K(T)$

$\gamma = \alpha C$であれば

$|\alpha| \geq K$ならば　　$(k) : \alpha \in F_T^K(T)$

$|\alpha| < K$ならば　　$\alpha H_T^{K-|\alpha|}(C) \in F_T^K(T)$

$H_T^K(A)$の計算

$A \to \gamma C\rho$のように左辺にAを含む生成規則をとりだす。

A→γ のとき

$|\gamma| \geq$ならば　　(k)：$\gamma \in H_T^K(A)$

$|\gamma| <$ならば　　$\gamma F_T^{K-|\gamma|}(A) \in H_T^K(A)$

A→$\gamma C \rho$ のとき

$|\gamma| \geq K$ならば　　(k)：$\gamma \in H_T^K(A)$

$|\gamma| <K$ならば　　$\gamma H_T^{K-|\gamma|}(C) \subset H_T^K(A)$

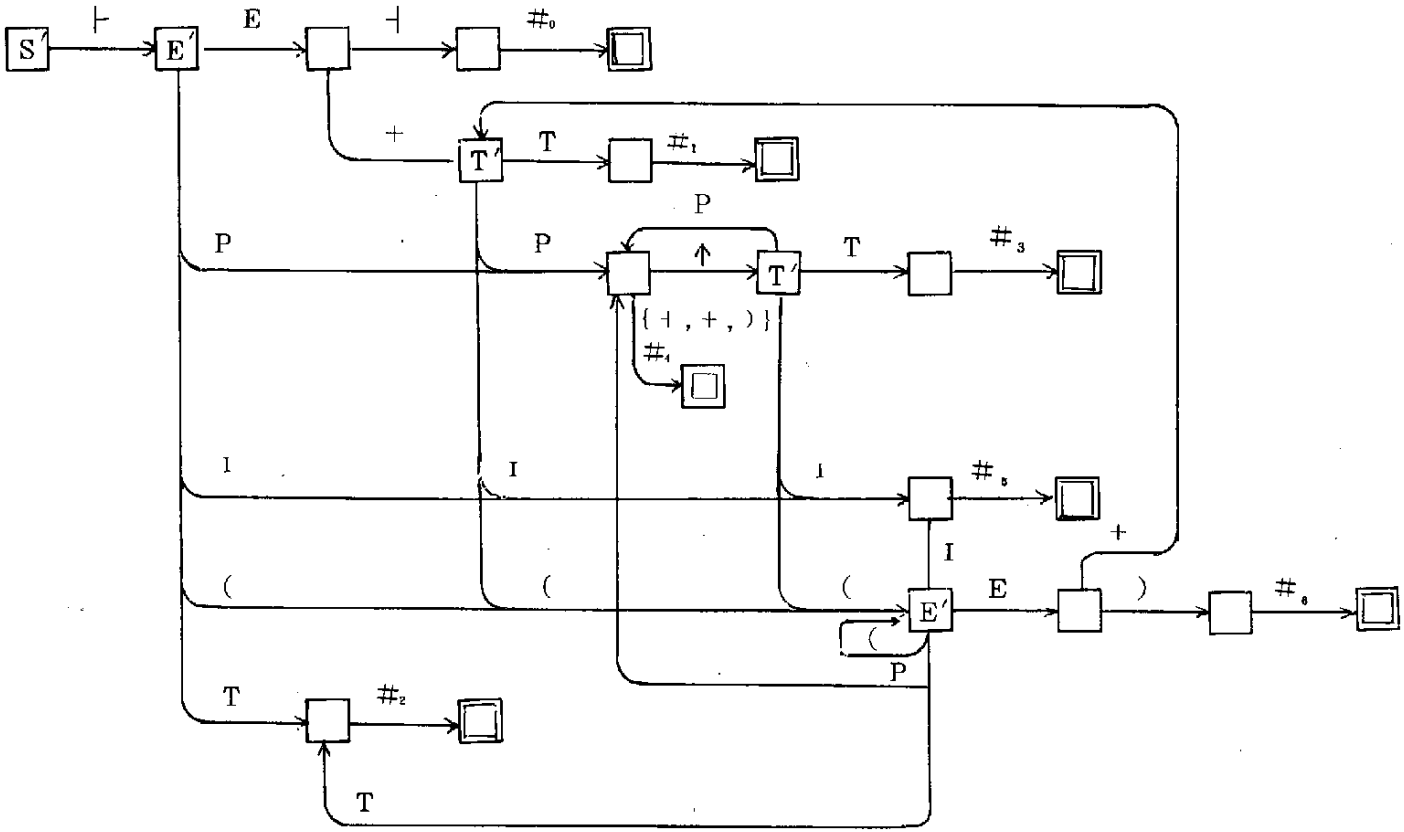


図4－15　LOOK AHEAD処理後のCFSMグラフ

４．６　Ｌ(ｍ)Ｒ(ｋ)　処理

この処理は（図４－１）のｓｔｅｐ５に対応するもので処理の概要は以下に述べるとおりである。

ＳＬＲ(２)でｉｎａｄｅｑｕａｔｅ　状態を分離できなかったときＬ(m)R(k)で分離できるか否かを試みる。その方法はＣＦＳＭグラフ上の該当ｉｎａｄｅｑｕａｔｅ状態のｎｏｄｅからの遷移が左のｍ個の記号と右のｋ個の記号によってユニークに決定できるか否かを調べることであり，その結果ユニークに決定できるならば該当ｉｎａｄｅｑｕａｔｅ　状態はＬ(m)R(k)で分離できるということになる。
そしてＬ(m)R(k)で分離できる場合，該当ｉｎａｄｅｑｕａｔｅ状態のｎｏｄｅをいくつかのｎｏｄｅに分割することによりＣＦＳＭグラフを作りかえる。この分割操作の結果できるグラフは ｌｏｏｋ－ａｈｅａｄセットのみでａｒｒｏｗ選択が可能になる。

ａ．記号の定義

ＳＴＡＧＥのＬ(m)R(k)処理の詳しいアルゴリズムを述べる。
前に（表４－３）で示すＳＹＮＴＡＸ　ＤＥＦＩＮＩＴＩＯＮ及び（図４－１６）で示す。ＣＦＳＭグラフを例として，アルゴリズムの説明で使用する記号を定義する。

(1)　φ ： m

φがストリングの時　φのうしろのｍ個のシンボル列を意味する。

(2)　$^{m}L(N)$

ＮがＣＦＳＭグラフ上の ｎｏｄｅ，φをＮの左側の文脈を構成するストリングとした時 $^{m}L(N)$はφ ： ｍである。

例　（図４－１６）の場合　$^{2}L(7)$は｛ａｅ｝である。

(3)　$FR^{m}(B)$

Ｃ→ rＢρという生成規則があるとき，$FR^{m}(B)$は次のようにあらわされる。

$|r| \geq m$　ならば　r ： ｍ

$|r| < m$　ならば　$FR^{m-|r|}(C)\,r$

例　（表４－３）で示すＳＹＮＴＡＸ　ＤＥＦＩＮＩＴＩＯＮでは

$FR^{1}(A)=\{a\}$

$FR^{2}(A)=FR^{1}(E)\,a$ となる。

(4)　${}_{m}C^{k}(T)$

①　Ｔが$\#_{N}$の場合

Ｎ番目の生成規則がＡ→ωであり，Ｂ→$\gamma A\rho$ なる生成規則があるとき ${}_{m}C^{k}$（$\#_{N}$）は次のようになる。

$|\gamma\omega|\geqq m$　ならば　$\{(\gamma\omega : m,\ F_{T}^{K}(A))\}$

$|\gamma\omega|<m$　ならば　$\{(FR^{m-|\gamma\omega|}(B)\gamma\omega,\ F_{T}^{K}(A))\}$

例　（表４－３）で示す構文の場合

の５番目の生成規則　Ａ→ｅの場合

${}^{2}C^{1}(\#_{5})=\{(be,\ c),\ (ae,\ d)\}$ となる。

②　Ｔがｔｅｒｍｉｎａｌ記号の場合

Ｂ→$\gamma t\rho$ なる生成規則があり，ｔがｔｅｒｍｉｎａｌ記号のとき ${}_{m}C^{k}$（ｔ）は次のようになる。

$|\gamma|\geqq m$　ならば　$\{(\gamma : m,\ {}_{t}F_{T}^{K-1}(t))\}$

$|\gamma|<m$　ならば　$\{(FR^{m-|\gamma|}(B)\gamma,\ {}_{t}F_{T}^{k-1}(T)\}$

例　（表４－３）で示す構文の場合

${}^{2}C^{1}(C)=\{(ae,\ c),\ (bA,\ c)\}$ となる。

(5)　${}_{m}BC^{k}(T')$ ……（ｍ，ｋ）－ｂｏｕｎｄｅｄ－ｃｏｎｔｅｘｔ－ｐａｉｒｓ。

Ｔ′がｎｏｄｅ Ｎ からの遷移記号Ｔを持つ遷移のとき

${}_{m}BC^{k}(T')=\{(\sigma,\ \mu)\in {}_{m}C^{k}(T)\mid \sigma\in {}_{m}L(N)\}$

となりσ，μをそれぞれＬｅｆｔ，Ｒｉｇｈｔ と呼ぶ。

例　（図４－１６）のＣＦＳＭグラフの場合

①　Ｎ＝ｎｏｄｅ　７，Ｔ＝$\#_{5}$，ｍ＝２，ｋ＝１とすると

${}^{2}L(7)=\{ae\}$

${}^{2}C^{1}(\#_{5})=\{(ac,\ d),\ (be,\ c)\}$

よって

$^{2}BC^{1}(\#_{5}')=\{(ae, d)\}$ となる。

② N＝node7，T＝C，m＝2，k＝1とすると

$^{2}L(7)=\{ae\}$

$^{2}C^{1}(C)=\{(ae, c), (bA, c)\}$

よって

$^{2}BC^{1}(C')=\{(ae, c)\}$ となる。

表4－3

```
SYNTAX DEFINITION;
SYNTAX RULE;
<S>  ::=├<E> ─┤  : EXEC0;
<E>  ::=a <A> d  : EXEC1 | acc:EXEC2 |
          b <A> c  : EXEC3 | bed:EXEC4;
<A>  ::=e  : EXEC5;
SYNTAX RULE END;
```

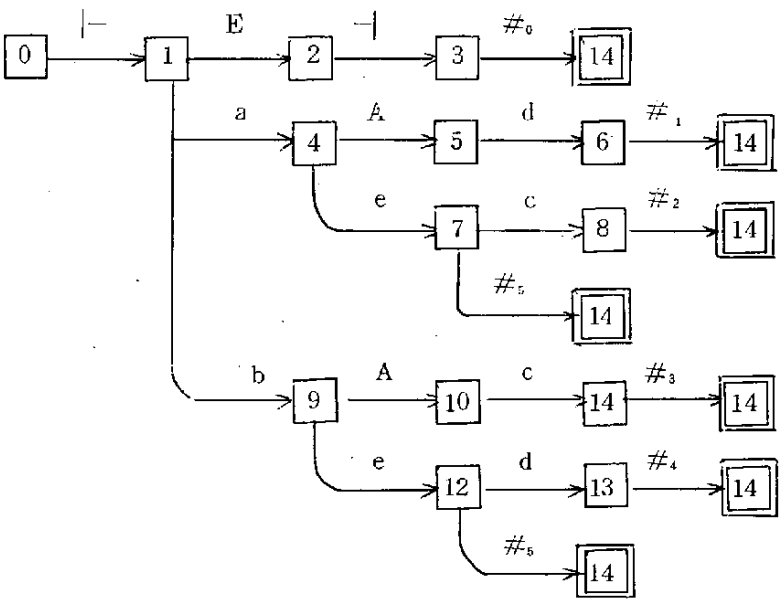


図4－16　（表4－3）に示す文法に対するCFSMグラフ

b ・ L(m)R(k)処理アルゴリズム

(1) ＳＬＲ(2)で分離できなかったｉｎａｄｅｑｕａｔｅ状態のｎｏｄｅNに対し，L(m)R(k)で分離できるか否かを下記の手順によりｍ＝２，ｋ＝１まで行なって調べる。

① ｎｏｄｅＮからの遷移の集合をＴ′その要素をＴｉ′とするときすべての遷移Ｔ′に対してｂｏｕｎｄｅｄ－ｃｏｎｔｅｘｔ－ｐａｉｒｓ

$${}^{m}BC^{k}(T'i)=\{(\sigma i_1, \mu i_1), (\sigma i_2, \mu i_2)\cdots\cdots\}$$

を求める。

② Ｔｉ′のｂｏｕｎｄｅｄ－ｃｏｎｔｅｘｔ－ｐａｉｒｓ がユニークであるならばL(m)R(k)で分離可能であるから(2)以降の処理を行なう。L(2)R(1)によっても分離できなかった場合は，どの生成規則が処理できなかったというメッセージを出力し処理を中止する。

(2) L(m)R(k)は遷移を行う場合Ｌｅｆｔの履歴を見る必要がある，これは効率が悪くなるため，ｉｎａｄｅｑｕａｔｅ状態のｎｏｄｅに関し，ＣＥＳＭグラフを作りかえＲｉｇｈｔすなわちｌｏｏｋ－ａｈｅａｄのみで遷移がユニークに決定できるようにする。

① (1)で求めたｂｏｕｎｄｅｄ－ｃｏｎｔｅｘｔ－ｐａｉｒｓがＲｉｇｈｔ μij だけでユニークになるとき，各遷移Ｔｉ′に対するｌｏｏｋ－ａｈｅａｄセットはそれぞれμi であり，この場合ＣＦＳＭグラフを作りかえる必要はない。(★－1) なぜならばこの場合はＬｅｆｔを見る必要がないからである。このあと③へ行く。

② (1)で求めたｂｏｕｎｄｅｄ－ｃｏｎｔｅｘｔ－ｐａｉｒｓがＲｉｇｈｔ μij のみではユニークに識別できないならば，ｎｏｄｅ分割を行なってＣＦＳＭグラフを作りかえねばならない，その手続きを以下に述べる。

Ｔｉ のｂｏｕｎｄｅｄ－ｃｏｎｔｅｘｔ－ｐａｉｒｓ$(\sigma ij, \mu ij)$に関してＣＦＳＭグラフ上をｎｏｄｅＮからσijを逆にたどっていく，そして到着したｎｏｄｅをＭとするとＭの子孫として$\{(\sigma ij, \mu ij)\}$の系列に対応するｎｏｄｅ，ａｒｒｏｗを付け加える。(★－2)

次に，NからMへ逆にたどってゆき複数の子arrowを持つnodeにはじめて行き当ったとき，その node の子孫からσ_{ij}以下のarrow node を取り除く。

③ 他にSLR(k)が適用できないinadequate 状態があれば，それを対象として(1)へ戻る。

(3) (2)において作りかえられた結果のグラフはnon−deterministicなグラフなので，これをdeterministic なグラフに変える。その方法は以下の通りである。

あるnode N の子arrowに同一遷移記号を持つ arrow が複数本あるとき，それらを1本の arrow にまとめる。そして，この操作をそのような状態のarrowが無くなるまで繰り返す。

＊−1）（例4−5）がこの場合に該当する。

＊−2）（例4−6）の(2)参照。

＊−3）（図4−19）参照。

c・STAGEのL(m)R(k)処理例を次に示す。

（例4−5）

前記のように（表4−3）で示す構文からCFSMグラフを作ると（図4−16）になる。

（図4−16）においてはinadequate状態はnode 7及びnode 12である。このときnode 7の#5 arrow，node12の#5 arrowに対して$F_T^2(A)$を求めると，それぞれ{c⊣, d⊣}，{c⊣, d⊣}となり，いずれもSLR(2)では処理できない。

(1) まずnode7からの各遷移(arrow{c}, arrow{#5 })に対する'BC'を求める。

arrow{c}に対して 'BC'(c')={(e, d)}

arrow{#5 }に対して 'BC'(#5')={(c, c)}となる。

よってnode 7からの各遷移に対するbounded−context−pair は各々ユニークになる。node 7のinadequate状態はL(1)R(1) で分離できる。

(2) (1)で求めたnode 7のbounded−context−pairsはそれぞれLeftは同じでRightがユニークである。それゆえnode 7 を分割しCFSMグラフを作りかえる必要はなく, node 7からの遷移はRightを見るだけで確定する(node 7からnode 14への遷移に対するlook−ahead セットは{d}となる)。同様にしてnode12からnode14への遷移に対するlook ahead セットは{c}となる。

上記の結果をまとめると(図4−17)ができ上がる。

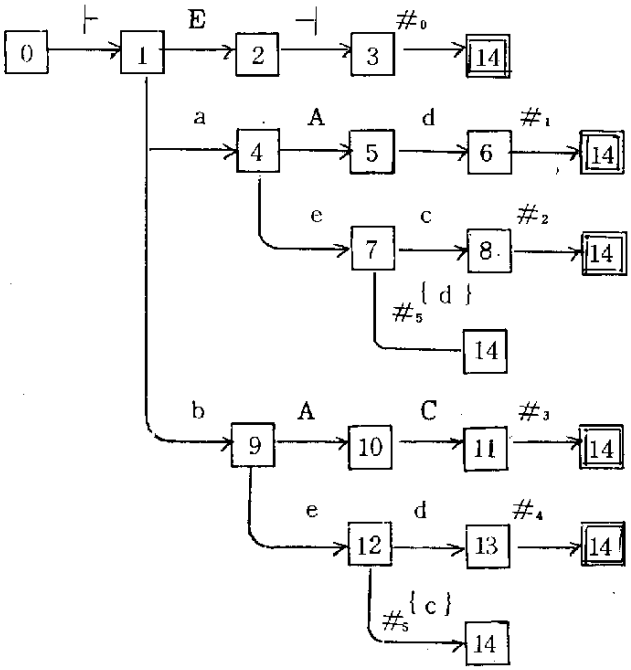


図４－１７　（表４－３）に示す構文に対するL(1)R(1)処理後のCFSMグラフである。

（例４－６）

（表４－４）に示すSYNTAX DEFINITIONからCFSMグラフを作ると（図４－１８）になる。

表４－４

```
SYNTAX DEFINITION;
SYNTAX RULE;
<S> ::=⊢<E>⊣ : EXEC0;
<E> ::=a<A>d : EXEC1 | a<B>c:EXEC2 |
        b<A>c : EXEC3 | b<B>d:EXEC4 |
        a e f : EXEC5;
<A> ::=e : EXEC6;
<B> ::=e : EXEC7;

SYNTAX RULE END;
```

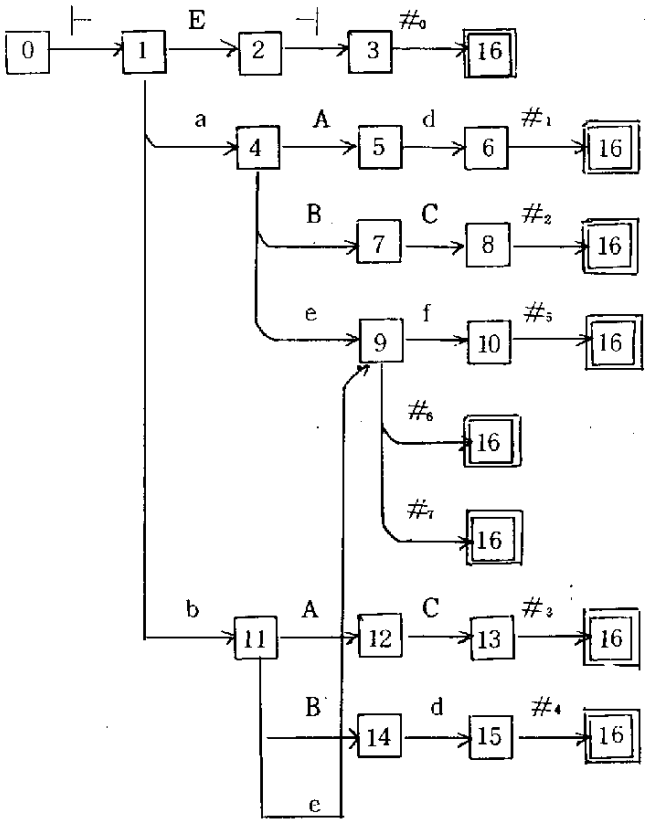


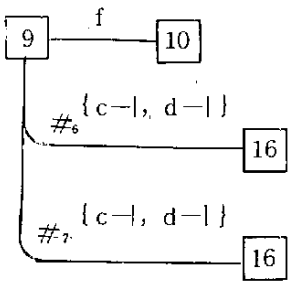
図４－１８　（表４－４）に示す構文に対するＣＦＳＭグラフ

（図４－１８）のＣＦＳＭグラフにおいてnode９にinadequate状態が生じている。

このときnode ９の#6　arrowに対してF_T^2（Ａ）を求めると

｛c⊣，d⊣｝であり，#7 arrowに対してF_T^2（B）を求めると

｛c⊣，d⊣｝　となり，ＳＬＲ(2) ではこの inadequate 状態は処理できない。



(1) node9からの3つの遷移（arrow{f}, arrow{#6}, arrow{#7}）をT′とし，それぞれに対して²BC¹（T′）を求める。

arrow{f} に対して ²BC¹（f′）={(ae, f)}

arrow{#6} に対して ²BC¹（#′6）={(ae, d), (be, c)}

arrow{#7} に対して ²BC¹（#′7）={(ae, c), (be, d)}

結果のbounded−context−pairsはユニークになる。それゆえnode9のinadequate状態はL(2)R(1)で分割できる。

(2) (1)で求めたbounded−context−pairsは各々のRightを見るだけではユニークにならない。それゆえnode9を分割してCFSMグラフを作りかえる。

① （図4−18）でnode9から各bounded−context−pairのLeftすなわち{ae, be}を逆にたどるとnode1に到達する。それゆえ，node1に²BC¹（f′），²BC¹（#′6），²BC¹（#′7）に対応したarrow, node 群を子孫として追加する。(*−4)

node4の子孫，及びnode10の子孫からarrow{e}をたどって到達できるnode, arrow を取り除く，その結果（図4−19）となる。

(3) (2)ででき上ったnon−deterministic なCFSMをdeterministic なCFSMに変更する。

（図4−19）のグラフはnon−deterministic なものである。これをdeterministic なグラフに変えると結果は（図4−20）となる。

（図４－１８）と（図４－２０）を見比べると（図４－２０）は（図４－１９）のnode9をnode9¹とnode9²の２つに分割した形となり，node9¹ 及びnode9² からの遷移は look-ahead のみに可能となる。

＊－４）（図４－１９）の点線で囲んだ部分。

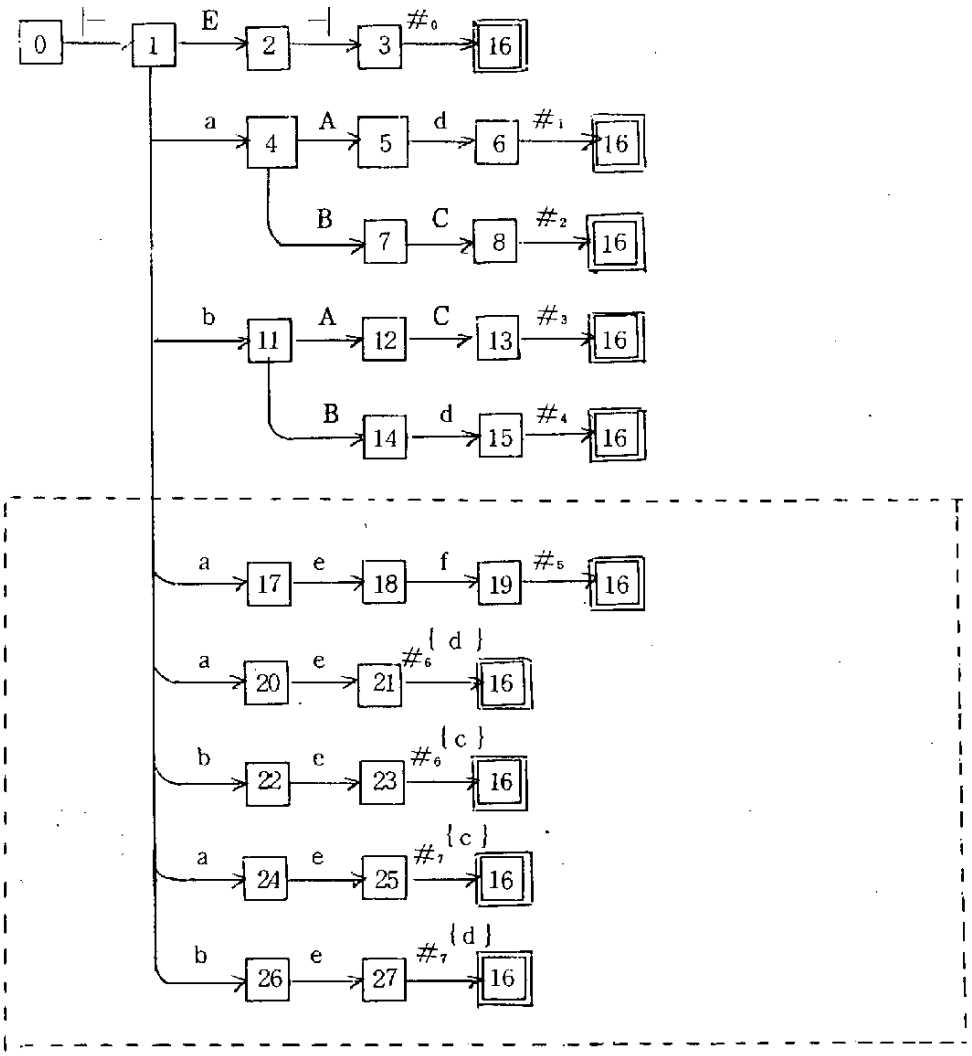


図４－１９　(2)処理後のnon-determintisticなＣＦＳＭグラフ

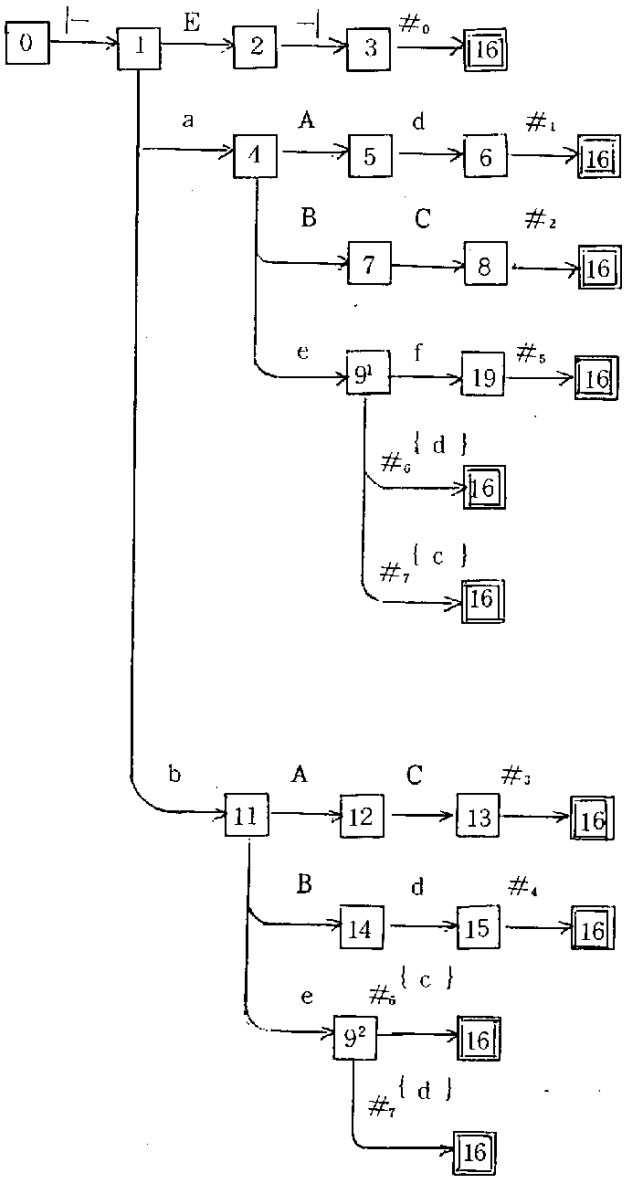


図４－２０　(3)処理後のＣＦＳＭグラフ

４．７　遷移図作成処理

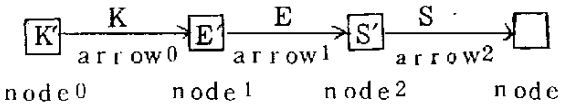
この処理は（図４－１）のstep６に対応するものでＣＥＳＭグラフより遷移図を作成する。

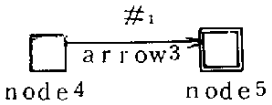
その場合必要があれば look−back セットを求める作業を行なう。遷移図作成の手続きを述べる前に記号の定義をしておく。

^{k}ＬＢＮ（Ｔ）　：arrow{Ｔ}のＥＮＤ　nodeに到達するまでにあらわれる長さｋの記号列の集合。

例　（図４－２１）で^{1}ＬＢＮ（Ｅ）：{Ｋ}

^{1}ＬＢＮ（Ｓ）：{ＫＥ}である。





注）　$\#_1$ はnonterminal記号Ｅを生成する生成規則に対する＃ｎ記号

図４－２１

(1)　ＣＦＳＭグラフから遷移図を作成するアルゴリズム

Ｔをnonterminal記号とすると，記号Ｔを帰納記号とする＃ｎ　arrowのＥＮＤnodeからのarrowでarrow｛Ｔ｝を置きかえるという操作を全てのnonterminal記号に対して行なう。

例　（図４－２１）の場合arrow 1をnode 5からnode 2へのarrowで置きかえる。

上記の操作を行なうにあたって考慮すべきことが２つある。

①　同一のnonterminal記号を帰納記号とする＃ｎ　arrowが複数本あるとき，ＥＮＤnodeを１つにまとめる。

②　同一のnonterminal記号をその遷移記号として持つarrowが複数本あるとき，各セマンティックス処理arrowのＥＮＤnodeからarrow

が複数本出るが，ドライブの際その経路選択を行なわねばならない。その為に該当するnonterminal記号に到着するまでの履歴が必要となり，これを見つけるのがLook-BACK 処理である。その場合必要なのがlook-back セットである。

(2) look-back セットを求める方法

Tをnonterminal記号とすると，arrow{T}が複数本あるとき各arrow{T}に対して^{k}LBN{T}を見つける，この操作はk＝1より始めてkを1ずつ増加しながら，各^{k}LBN(T)がユニークになるまで行なう。そして各^{k}LBN(T)がユニークになったとき，#n arrowのENDnodeからの各arrow（各arrow{T}にかわるもの）のlook-backセットは各^{k}LBN(T)となる。

例4－7

（図4－14）のCFSMグラフ上に上記の手続きを適用した結果できた遷移図は（図4－22）で示される。

（図4－22）の説明

① node6，node7における#n arrowはともにnonterminal記号{E}に対するものなので，そのENDnode は node17の1つだけにしてある。

② （図4－14）を見るとnonterminal記号Eを，その遷移記号として持つarrowは2本ある，そして各arrowに対して^{1}LBN(E)を求めると，各々{⊣}，{(}となる，よって（図4－22）のnode17からのarrowは2本あって，各々{⊣}，{(}のlook-backセットを持つ。

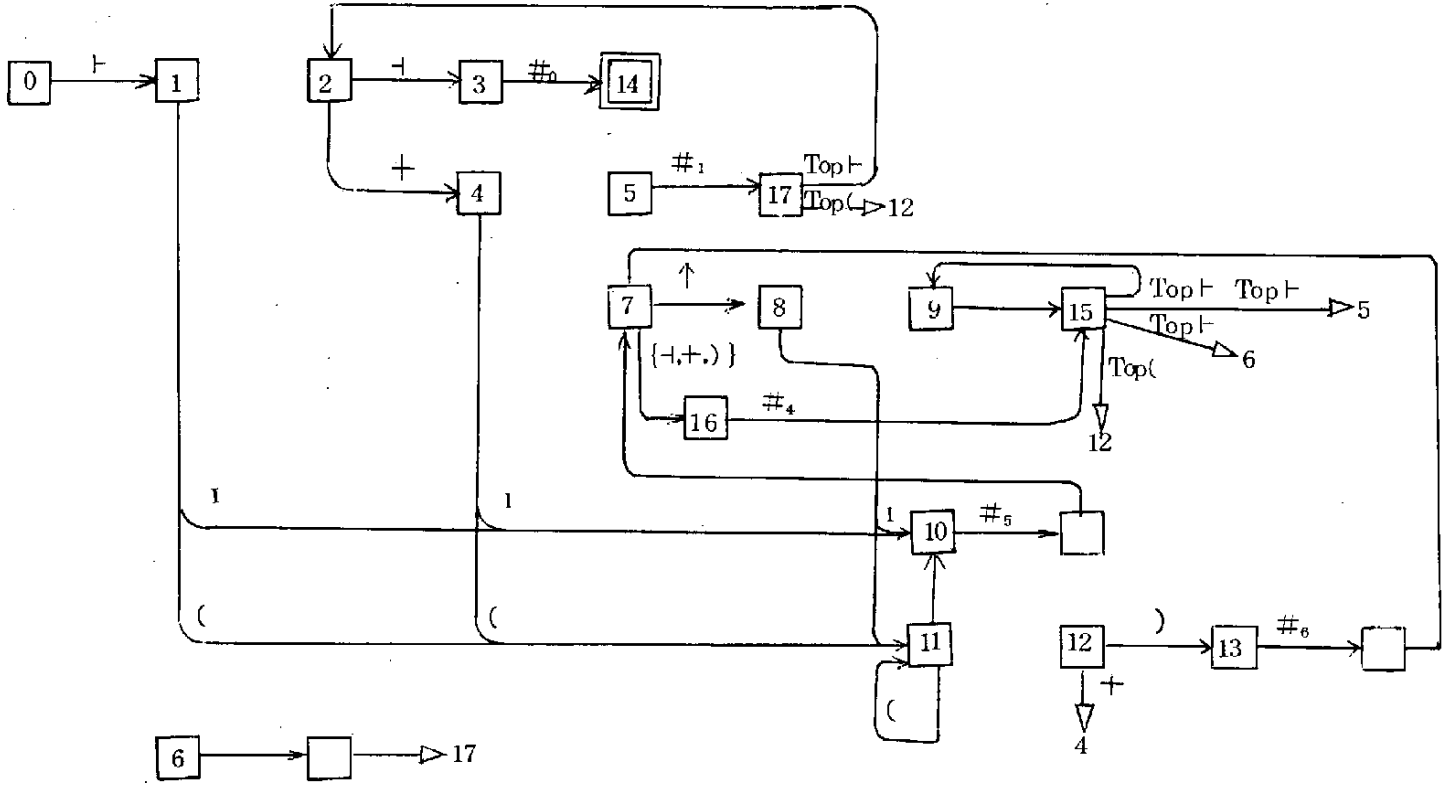


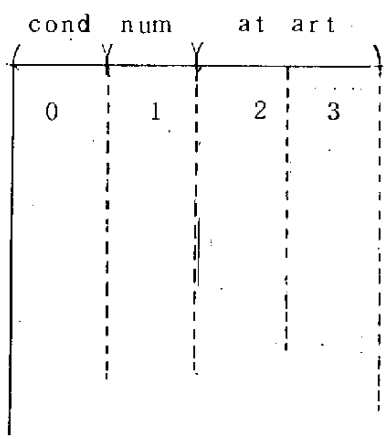
図４－２２　（表４－１）に示す文法に対する遷移図

5. 遷移テーブル類

ＳＴＡＧＥで作成される遷移テーブル類は（図４－２２）のような遷移図を計算機で取り扱える形式（ＦＡＳＰソース）で表現したもので（図４－２２）のｎｏｄｅに対応するものとしてＣＧ．ＮＯＤＥ，ａｒｒｏｗに対応するものとして，ＣＧ．ＡＲＴ，ＣＧ．ＳＹＭ，ＣＧ．ＬＡＢＴとしてＦＡＳＰソースの形で作成される。

(1) ＣＧ．ＮＯＤＥテーブル

ＣＧ．ＮＯＤＥテーブルは（図５－１）で示す様に３要素で１ＷＯＲＤを占め１つのｎｏｄｅ情報を示す。



① ｃｏｎｄは第０バイト目を占め，このｎｏｄｅの状態を示す。

０ｂｉｔ目ＯＮの時：Ｓｔａｒｔ ｎｏｄｅ

１　〃　：ＥＮＤ　ｎｏｄｅ

２　〃　：テーブルドリブンの際ドライブの対象をこのｎｏｄｅまで引き戻す。

図５－１　ＣＧ．ＮＯＤＥテーブル

② ａｔ ａｒｔは第２，第３バイトを占め，ＣＧ．ＡＲＴへのポインタがＢｉｎａｙではいる。つまりこのｎｏｄｅから出ているａｒｒｏｗの情報が入っているエリアの先頭番地を示す。

③ ｎｕｍは第１バイトを占め，ＣＧ．ＡＲＴのａｔ ａｒｔで示す所から何項目にわたって情報がはいっているかがＢｉｎａｙではいる。つまりこのｎｏｄｅから出ているａｒｒｏｗの個数を示す。

(2) ＣＧ．ＡＲＴ テーブル

(3) ＣＧ．ＳＹＭ テーブル

ＣＧ．ＡＲＴ及びＣＧ・ＳＹＭは１対１の対応をしており，各１WORDづつ合計２WORDで１つのａｒｒｏｗ情報を示す。

ＣＧ．ＡＲＴテーブル　　ＣＧ．ＳＹＭテーブル

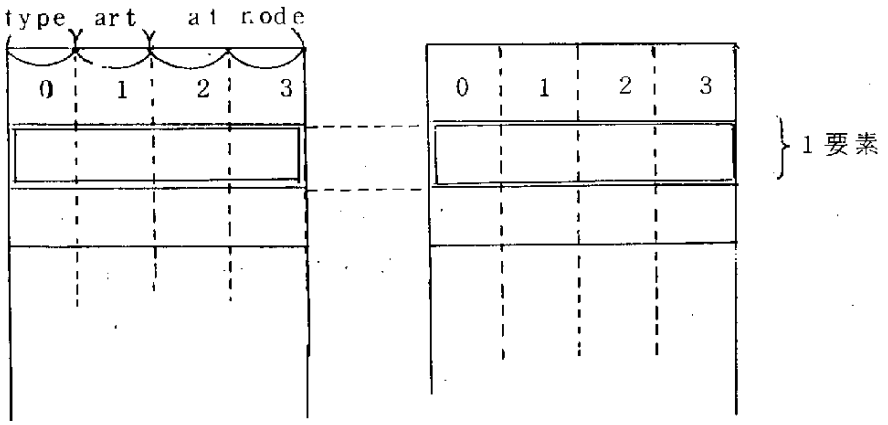


図５－２　ＣＧ．ＡＲＴ及びＣＧ．ＳＹＭ
テーブル

表5－1　CG．ART，CG．SYMの内容説明表

CG．ART			CG．SYM	
0バイト	1バイト	2，3バイト	0，1バイト	2，3バイト
0bit目ON READ	0bit目ON	CG．NODEへのポインタがBinaryではいる。	0バイト目に文字コードでterminal記号がはいる。	
	1bit目ON		0，1バイト目にBinaryでBASIC　SYMBOL番号がはいる	
	2bit目ON		0，1バイト目にBinaryでnonterminal記号番号がはいる	
1bit目ON LOOK－BACK	LOOK－BACKの個数がBinaryではいる		CG．LABTのLOOK－BACK情報組数がBinaryではいる。	このarrowに該当するCG．LABTへのポインターがBinaryではいる。
2bit目ON LOOK－AHEAD	LOOK－AHEADの個数がBinaryではいる		CG．LABTのLOOK－AHEAD情報組数Binaryではいる。	このarrowに該当するCG．LABTへのポインターがBinaryではいる。
3bit目ON EXEC			0，1バイトにプッシュダウンすべきnonterminal 記号番号がBinaryではいる。	2，3バイトに実行すべきセマンティックスルーチングループ通番がBinaryではいる。
4bit目ON POP				2，3バイトにpop up すべき数がBinaryではいる

(4) ＣＧ．ＬＡＢＴテーブル

ＣＧ．ＬＡＢＴテーブルは，ＬＯＯＫ－ＡＨＥＡＤ，ＬＯＯＫ－ＢＡＣＫの場合の情報をセットしておくもので（図５－３）に示すように１項目２要素で１項目で１ＷＯＲＤを構成する。

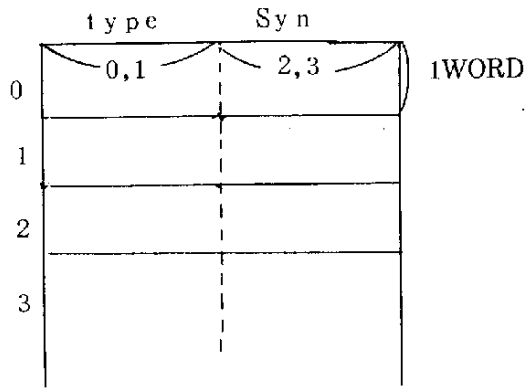


図５－３　ＣＧ．ＬＡＢＴテーブル

①　ｔｙｐｅはＣＧ．ＬＡＢＴを構成するＷＯＲＤの０，１バイトにあたり，Ｓｙｎにはいっている記号のタイプを示す。Ｓｙｎは２，３バイト目に当り，ｔｙｐｅで示された型に従って記号がはいる。

表５－２　ＣＧ．ＬＡＢＴの内容に対する説明表

ｔｙｐｅ	ｓｙｎ
０ｂｉｔ目ＯＮ	ｔｅｒｍｉｎａｌ記号が文字コードではいる
１ｂｉｔ目ＯＮ	ＢＡＳＩＣ　ＳＹＭＢＯＬ番号がＢｉｎａｒｙではいる
２ｂｉｔ目ＯＮ	ｎｏｎｔｅｒｍｉｎａｌ記号番号がＢｉｎａｒｙではいる

例５－１

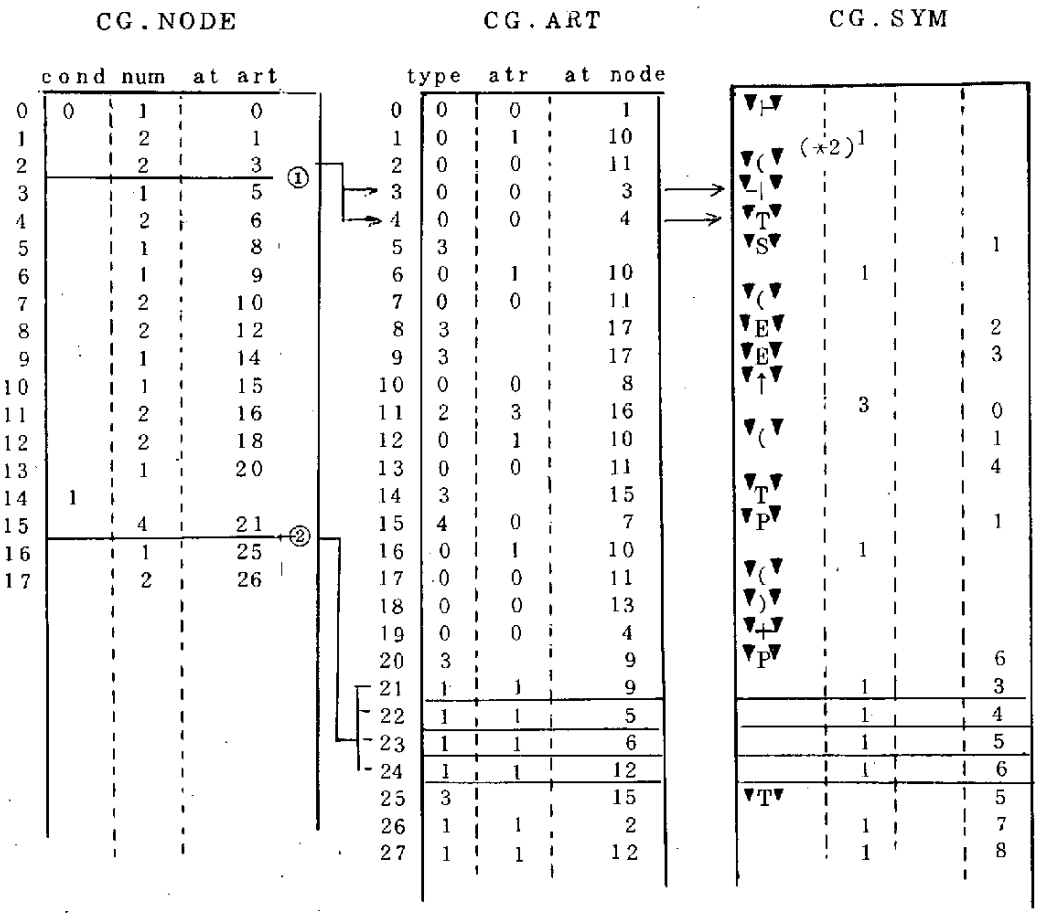
（表４－１）のＳＹＮＴＡＸ　ＤＥＦＩＮＩＴＩＯＮが与えられた場合，遷

移テーブル類はどのようになるかを示す。

（表４－１）から遷移図を作成すると（図４－２２）になり，それを基としてCG・NODE，CG・ART，CG・SYM，CG・LATを作ると（図５－４）になる。以下に（図４－２２）と（図５－４）の関係を少し述べる。（図－２２）のnode２を見ると，子arrowはnode３へのものと，node４へのものの２本存在し，node3へ行くarrowはREADしたものがterminal記号 ▼─┤▼ であり，node4へ行くarrowはREADしたものが ▼┼▼ のときである。（図５－４）のCG・NODE の第２項を見るとnumは２，at artは３である，つまりCG・ART の第３項から２項目にわたって２本分のarrow情報がはいっている事を示す。

CG・ART第３項のtype（０バイト目）を見ると0bit目ON なのでこのarrow はREADであり，atr（１バイト目）は０bit 目ONなのでCG・SYMの第３項にはterminal記号が入っていることを示す。実際CG・SYMの第３項０バイト目には ▼─┤▼ が入っている。

CG・ARTの第３項のai node（２，３バイト目）にはBinaryの３がはいっている，つまりこのarrowのEND nodeはnode 番号３すなわちCG－NODEの第３項という事になる。CG・ARTの第４項はCG・NODEの第２項のもう１つの子arrow情報がはいっている。どうように（図４－２２）のnode番号１５の子arrowは全てLOOK－BACK である。（図５－４）のCG・NODEの第１５項はnum：４，at art：２１なのでこのnodeの子arrowは４本あってその情報はCG・ARTの第２１項以降４項目に入っていることを示す。例としてCG・ART の第２１項を見るとtype：０bit ON，atr：１，at node：９であり，CG・SYMのupper：１，lower：３，つまりこのarrow はLOOK－BACK でその数及び組数はそれぞれ１で，その情報はCG・LABTの第３項に入っていることを示す。実際CG・LABTの第３項にはnon－terminal ▼↑▼ が入っている。



CG.LABT

0	0	▼-|▼
1	0	▼-+▼
2	0	▼)▼
3	0	▼↑▼
4	0	▼+▼
5	0	▼|-▼
6	0	▼(▼
7	0	▼|-▼
8	0	▼(▼
9		

注）

（＊1）CG.NODEの第0バイト，CG.ARTの第0バイト CG.LABTの第0バイト及びCG.ARTの第0バイトが0のときのCG.ARTの第1バイトは対応するbitがONになっていることを示す。

（＊2）▼ ▼でかこまれているのは，文字コード，但し nonterminal記号の場合，実際には nonterminal記号番号がBinaryではいる。

（＊3）その他はすべてBinaryである。

図5－4　（表4－1）に示す文法に対する出力遷移テーブル類

6. 中間テーブル類

コンパイラ・ジェネレータは遷移テーブル類を作成するまでの中間テーブルとして次のような2グループのテーブルを持つ。

a・第1グループ

(1) CG_NODEテーブル（TABLE）

これは1項目4要素で構成されnodeの情報をセットするのに用いられる。その内容は（表6－1）に示される。

表6－1 CG—NODTの内容説明表

名称	型	機能	初期値
CG_NDY1	P(pointer)	node記号の入っているCG_NDNMへのポインター	null
CG_NDY2	B(Bit)	nodeのstateを示す ＊－1）	B 000000000
CG_NDY3	P	このnodeをEND nodeとするarrow情報が入っているCG_PARTへのポインター	null
CG_NDY4	P	子arrow情報の入っているCG_ARTへのポインター	null

＊－1） CG NDY2

第1bit目ONの時 …… このnodeはStart node

2 〃 〃 End node

3 〃 〃 Basic Statemont 対応node

(2) CG_ARTテーブル（TABLE）

これは1項目5要素からなりarrow情報セットの為に用いられる。

表6－2　CG-ARTテーブル内容説明表

CG_ARTテーブル CG_ARY1(B)	CG-ARTテーブル CG-ARY2 (P)	CG-ARTテーブル CG-ARY3 (P)	CG-ARTテーブル CG-ARY4 (P)	CG-ARTテーブル CG_ARY5 (P)
1bit目ON (READ)	シンボルテーブル CG-SYMへのポインター	このarrowと同位のarrowが他にある時それへのポインター	このarrowのEND-nodeへのポインター(CG-NODTのポインター)	このarrowのstart-nodeへのポインター(CG-NODTへのポインター)
2bit目ON (LOOK-BACK)	CG-LABT1へのポインター			
3bit目ON (LOOK-AHEAD)	CG-LABT1へのポインター			
4bit目ON (EXEC)	CG-SHATへのポインター			
5bit目ON (POP)				

(3)　CG_SYMテーブル（TABLE）

1項目4要素からなりBASIC　SYMBOL，terminal記号，nonterminal記号の情報をセットする。

表6－3

CG_SYMY1 (B)		CG_SYMY2 (C)	CG_SYMY3 (P,N)	CG-SYMY4 (P)
1bit目ON		terminal記号が入る	———	———
2bit目ON		BASIC SYMBOL名が入る	Numeric型でBASIC SIMBOL番号が入る	———
3bit目ON	19bit目ON (Start記号)	nonterminal記号名が入る	nonterminal記号を生成する生成規則グラフのSNODEへのポインター，CFSMグラフ作成後はnonterminal記号番号	nonterminal記号を生成規則グラフが結合された母型グラフ上のnodeへのポインター
	20bit目ON (Basic Statement)			

(4) CG_SHATテーブル（TABLE）

1項目2要素からなり#n 記号に関する情報がセットされる。

表6－4

名　称	型	機　　能
CG-SHAY1	P(pointer)	#nの帰納記号へのポインター（CG_SYMへのポインター）
CG-SHAY2	N(numeric)	CG_ARY1の4bit目がON(EXEC)のとき，セマンティックスルーチングループ通番
		CG_ARY1の4bit目がON(POP)のときPOPすべき数

(5) CG_LABT1 テーブル（TABLE）

(6) CG_LABT2 テーブル（ARRAY）

CG_LABT1，CG_LABT2テーブルはLOOK－BACK，LOOK－AHEAD の時の情報をセットするもので，CG_LABT1はインデックス，CG_LABT2はボディ部に当る。

CG_LABT1（1項目3要素）

表6－5 CG-LABT1の内容説明表

CG_LABY11 (N)	CG_LABY12 (N)	CG_LABY13 (N)
LOOK-BACK，LOOK-AHEADの個数がはいる	LOOK-BACK，LOOK-AHEADの組数がはいる	CG-LABT2へのポインターがはいる

CG_LABT2（1項目1要素）

表6－6 CG-LABT2の内容説明表

CG-LAB21(P)
シンボルテーブルCG_SYMへのポインターがはいる

例６－１

（図６－１）で示すようなグラフがあったとき，ＣＧ－ＬＡＢＴ１及びＣＧ－ＬＡＢＴ２はそれぞれ（図６－２）で示すような形となる。

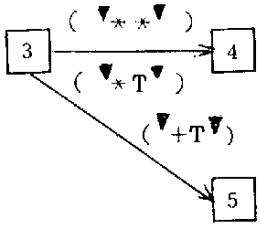


図６－１

注）① ｎｏｄｅ3からｎｏｄｅ4へのＬＯＯＫ－ＢＡＣＫは（▼**▼）とする。

② ｎｏｄｅ3からｎｏｄｅ5へのＬＯＯＫ－ＢＡＣＫは（▼*T▼），（▼+T▼）とする。

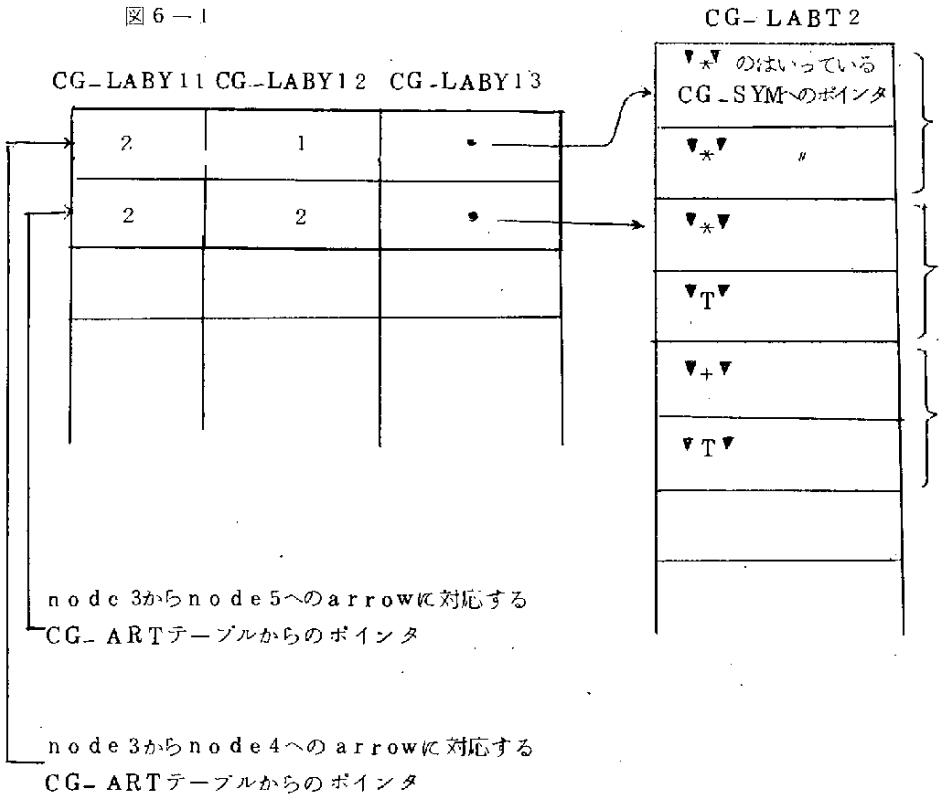


図６－２

(7) CG_PARTテーブル（TABLE）

1項目2要素からなりnodeに入るarrow情報がはいっているCG_ART テーブルへのポインターがはいっている。

表6－7 CG-PARTテーブルの内容説明表

名 称	型	機 能	初 期 値
CG-PART1	POINTER	流入arrow情報が入っている CG_ARTへのポインター	null
CG_PART2	POINTER	流入arrowが他にもある時，そのPointerが入っているとCG-PARTへのポインター	null （他にない時も null）

(8) CG_NDNMテーブル（TABLE）

1項目2要素からなりnode記号に対する情報がセットされる。

表6－8

名 称	型	
CG_NDNY1	P	node記号に対応するnonterminal記号の情報が入っているCG_SYMへのポインター
CG-NDNY2	P	nodeが以外にnode記号を持つとき，その情報がセットされているCG_NDNMへのポインター

b．第2グループ

第1グループの各種テーブル上の情報から第2グループのテーブルをセットする。これらは第1グループのテーブルをガーベージしたもので，遷移テーブル類の為のものである。

(1) OCG_NODTテーブル（TABLE）

OCG_NODTはnode情報セットの為に用い，1項目3要素からなる。

表 6－9　OCG_NODTテーブルの内容説明表

OCG_NDY1 (B)	OCG_NDY2 (P)	OCG_NDY3 (N)
Bit型でこのnodeの stateを示す。 1bit目ON ……Start node 2bit目ON ……End node 3bit目ON ……Basic Statement 対応node	Pointer型で OCG_ARTへのポインタ ーがはいる，つまりこの nodeから出ている arrow情報のはいってい るエリアへのポインター。	Numeric 型でOCG_ NDY2 でしめされたエリ アから何項目にわたって arrow情報がはいってい るかを示す，つまり arrowの数がはいる。

(2) OCG_ARTテーブル（TABLE）

OCG_ARTテーブルは1項目5要素からなる。

表6－10　OCG_ARTテーブル内容説明表

OCG_ARY1 (B)	OCG_ARY2 (B,N)	OCG_ARY3 (P)	OCG_ARY4 (P,C,N)	OCG_ARY4 (P,N)
1bit目ON (READ)	1bit目ON (B)	OCG−NODTへのポインター	terminal記号(C)	
	2bit目ON (B)		BASIC SYMBOL番号 (N)	
	3bit目ON (B)		nonterminal記号番号 (N)	
2bit目ON (LOOK-BACK)	LOOK-BACKの個数 (N)		LOOK-BACK情報組数 (N)	CG_LABT 2へのポインター (N)
3bit目ON (LOOK-AHEAD)	LOOK-AHEADの個数 (N)		LOOK−AHEAD情報組数 (N)	CG_LABT 2へのポインター (N)
4bit目ON (EXEC)			プッシュダウンすべきnonterminal記号番号が入っているエリアへのポインター(P)	セマンティックスルーチングループ通番 (N)
5bit目ON (POP)				POP UPすべき数 (N)

第　2　編

L　S　D 外 部 仕 様 書

目　　次

1. プログラム要素

1.1 言語の基本的構造

ＬＳＤのステートメントは，カードあるいはカード・イメージのファイルから入力されるものであるが，カードのカラムによる制限はなく，セミコロンで区切られるまでを１ステートメントとする。また，カード上の有効なカラムをユーザがコントロール・カードのパラメタで指定することができる。このパラメタを省略すると２～７２カラムが有効，すなわち前のカードの７２カラムが，次のカードの２カラムに続くものと考える。

1.2 文　　字

LSDでは,以下の５５個の文字を用いることができる。

英文字 … A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
Q | R | S | T | U | V | W | X | Y | Z　（２６文字）

数　字 … 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9　（１０文字）

特殊文字 … ＝ | ＋ | － | ＊ | ／ | （ | ） | ▼ | , | . | : | ; | ¬ | & | ⊥ | ＜ |
＞ | （空白）　（１８文字）

下　線 … _（under - bar）　（１文字）

これらの文字のほか，カタカナやこれ以外の特殊文字も文字ストリングやコメントの中に書くことができる。

1.3 区切り記号

特殊記号や特殊記号の組合せは区切り記号として用いられ，これらは演算子，カッコおよびその他の区切り記号に大別できる。

演算子

＜算術演算子＞ :: = ＋ | － | ＊ | ／

＜比較演算子＞ :: = ＞ | ¬＝ | ＝ | ＞＝ | ＜ | ¬＜ | ¬＞ | ＜＝

＜論理演算子＞::＝ ¬ | & | ⊥

＜連結演算子＞::＝ ||

カッコ

カッコはリストを囲むためや，いろいろなキーワードと結びついた情報を指定するのに用いる。

(　　左カッコ

)　　右カッコ

/*　　コメントの始まりを示す

*/　　コメントの終了を示す

▼　　ストリング定数を囲むために用いる

その他の区切り記号

,　(コンマ)　　リストの区切りとして用いる

;　(セミコロン)ステートメントの終わりを示す

:　(コロン)　　ラベルのあとに続く

=　　代　入

空白　　separator として用いる

.　　名前の修飾(qualify)のために用いる

*　(アスタリスク)テーブル etc. の要素の宣言のために用いる

1.4 名　　前

LSD において，値の参照，処理手続きの参照は名前によって行なう。

名前は英字で始まる8文字以内の英数字の列でなければならない。もし8文字を越える場合にはワーニング・エラーを表示し，はじめの8文字を有効とする(ただし30文字を越える名前があらわれた場合にはそのステートメントを無効とする)。但し，2～7文字目までは下線(under - bar)があってもよい。

名前は変数名，ラベル名，手続き名，ファイル名などに用いる。LSD では

組み込み関数の名前はreserved wordとするので名前として用いることができない。

組み込み関数としてはADDR, SUBSTRがある。

1.5 コ メ ン ト

コメントは空白の出現し得るところならば，どこでも置くことができる。コメントは／*，*／で囲むことによって示し，コメントの中には使用可能なすべての文字を書くことができる。ただし，*／の組み合わせだけは書いてはならない。

1.6 プログラムの構造

LSDのプログラムはPROCEDUREステートメントで始まり，ENDステートメントで終わる1群のステートメントをprocedureと呼び，ステートメントには1個のラベルをつけることができる。宣言文につけたラベルは無視される。

LSDではprocedureは1レベルだけ入れ子にすることができる。すなわち1個のprocedureの中にPROCEDUREステートメントを書くことはできるが，その内側のprocedureの中にさらにPROCEDUREステートメントを書くことはできない。この外側のprocedureのことをexternal procedure，内側のprocedureのことをinternal procedureと呼ぶ。

external procedureの中には，複数個のinternal procedureを書くことができるが，その位置はexternal procedureの後の部分におかなければならない。すなわち，external procedureの中ではinternal procedureに含まれない部分が先に来て，そのあとにinternal procedureが来るようにしなければならない。

名前の宣言でDECLAREステートメントによるものは，その名前の参照以前に宣言されていなければならない。このDECLAREステートメントは

internal procedureの中では書くことができないので注意すること。

さらに名前は，特にEXTERNAL と明示しないかぎり，そのexternal procedure 内だけで有効である。

ただしexternal procedure の入口名，およびexternal procedure内でinternal procedure に含まれない部分にあるENTRY ステートメントによる2次的な入口名はすべてのexternal procedure 間で有効である。

2. データ要素

LSD の目的プログラムで実行時に操作できる情報をデータと呼ぶ。各データ項目は一定の型と表現を持っている。

2.1 データの構成

データはスカラ・データすなわち単一のデータとデータ項目の集合として構成される。

2.1.1 スカラ・データ

スカラ・データは定数あるいはスカラ変数のいずれかである。

2.1.2 集合データ

変数データ項目を配列，構造体，テーブル，スタックなどのようにまとめることができる。これらを総称して集合データと呼び，それを構成しているスカラ変数のことを集合データの要素と呼ぶことにする。

次に集合データの個々のものについて簡単に述べることにする。

(1) 配　　列

同じ属性を持つスカラ変数の集合で1次元の順序づけられた要素の集合である。配列の大きさ（上限）は宣言文によって指定する。下限は1である。

配列を形成するスカラ変数のひとつひとつを配列要素と呼び，これを参照するときは配列名のあとに添字式を括弧( )で囲んで書けばよい。配列要素の参照方法については後述する。

(2) 構　造　体

配列とは異なり，異なる属性のデータ項目をまとめるのには構造体を用いる。構造体は宣言文でその構造体の名前とそれに含まれる要素名とを指定することによって作られる。構造体の要素は単純変数あるいは配列である。

(3) テーブル・スタック

異なる属性のデータをひとまとめにし，さらにそのまとまりを複数個集めたものにテーブルとスタックがある。テーブルもスタックも同じ構成の構造体を複数個集めたものと考えてよいわけであるが，この構造体のことはテーブルあるいはスタックの項目と呼ぶことにする。

テーブルやスタックを宣言するときにはその項目の個数，すなわちテーブルまたはスタックのサイズとその１項目を構成する単純変数あるいは配列を構成順に列記する。テーブルの場合には，さらにこのテーブルの項目を参照するために用いるテーブル固有のポインタ変数を共に指定しなければならない。

テーブルとスタックの違いは，テーブル内の項目あるいはその項目の要素を参照したり，これに値を入れたりする時に，ポインタ変数によってその項目の位置を識別する（特にポインタ変数を指定しなければ固有のポインタ変数とみなされる）のに対してスタックの場合には，その項目あるいは項目の要素を参照したり，これに値を入れる時には常に現在使われているスタックの一番上の項目と考えられることにある。

例えば数値変数Ａ１と文字変数Ｂ１から成るテーブルＴ（項目数１０）と数値変数Ａ２と文字変数Ｂ２から成るスタックＳ（項目数１０）があるとする。

その構成は形式的には次の（図２－１）のようになる。

テーブルT

	A 1	B 1
	1	
	2	
	3	
P 2 →	4	
	5	
	6	
	7	
P 1 →	8	

スタックS

A 2	B 2
10	
11	
12	
13	
14	

図２－１

Tを参照する時，もしTの固有のポインタ変数がＰ１，Ｐ２が別のポインタ変数であれば

Tのみ書いたときには，Ｐ１で示されるTの項目すなわち，Ａ１として8の入っている項目をとり出すことになる。

Ｐ２.Tとポインタ変数を指定しているときには，このポインタ変数の指す項目，ここではＡ１として4の値をもつ項目をTとして参照することになる。

スタックSにおいては，もし第５番目の項目まで使われているとするとSあるいはＡ２，Ｂ２と書けばこの第５番目の項目を指すことになる。

スタックの項目をスタック独自の方法でとり出したり，新しい項目を作ったりする時にはPUSHおよびPOPステートメントを用いる。

さらにスタックの一番上から，いくつ目かの項目を参照する方法もあるが，詳しくは〝変数の参照方法〟として後述する。

テーブルの項目をとり扱かうのには，SET，ERASE，SEARCHの特別なステートメントがある。

2.2 データの型

LSDでとりあつかうデータの型は，ふつうのデータとプログラム制御用データの２つに大別される。

2.2.1 ふつうのデータ

ふつうのデータとは数値，ビットストリングおよび文字ストリングデータのことである。

(a) 数値データ

数値データは値として符号つき，あるいは符号なしの整数を持つもので次のものである。

＜整数＞::＝＜数字＞｜＜整数＞＜数字＞

＜符号なしの整数＞::＝＜整数＞

＜符号つきの整数＞::＝＜整数＞｜＋＜整数＞｜－＜整数＞

整数は$2^{35}-1$および　-2^{35}の範囲内になければならない。

(1)　数値定数

数値定数は符号つき整数のことである。

(2)　数値変数

数値変数は値として符号つき整数を持つような変数で，宣言文ではNUMERICと宣言するか，あるいは型に関する属性を省略したものおよびDECLAREステートメントにもよらず前後関係によっても解釈されない名前をもつものである。

(b)　ビット・ストリング・データ

ビット・ストリング・データは1個以上のビットの列である。ビット・ストリング定数は，2進表示のほか8進，16進数として表わすこともできる。

(1)　ビット・ストリング定数

①　2進表現

2進表現のビット・ストリング定数は1個以上の2進数（0 or 1）の列を引用符で囲みその前にBをつけたもので，2進数字は36個以内でなければならない。

例　2－1　　B▼101▼　　B▼10101▼

②　8進表現

8進表現のビット・ストリング定数は，3ビットを1個の8進数で表わしたもので，1個以上12個以下の8進数字を引用符で囲みその前にOをつけたものである。

例　2－2　　O▼0473▼ … B▼000100111011▼ の速記形

O▼01101▼… B▼000001001000001▼の〃

③　16進表現

16進表現のビット・ストリング定数は4ビットを1個の16進数で表わしたもので1個以上9個以下の16進数字を引用符で囲みその前にXをつけたものである。

なお，１６進数では１０進数１０をあらわすのにＡ，１１をＢ，１２をＣ，１３をＤ，１４をＥ，１５をＦで示す。

例 2－3　X▼1234▼ …B▼0001001000110100▼　の速記形

X▼049AC▼…B▼00000100100110101100▼の　〃

(2) ビット変数

ビット変数は値としてビット・ストリング定数をとるような変数のことで宣言文でＢＩＴ と宣言し，さらにビット数を指定する。ビット数は３６以下の整数で，指定しなければ３６とみなされる。

(c) 文字ストリング・データ

文字ストリング・データは値として１個以上の文字の列を持つもののことである。

(1) 文字定数

文字定数は１個以上２５６個以下の任意の文字の列を引用符で囲んだもののことである。文字定数の中で引用符を表現したい場合には▼▼と２つ重ねて書いて１個の▼を表わす。

例 2－4　▼COMPILER－GENERATOR▼

▼CHARACTER▼▼X▼▼デス▼

(2) 文字変数

文字変数は値として文字ストリングを取るような変数であって宣言文でCHARACTER と宣言し，さらに文字数を指定する。文字数は１以上２５６以下の整数で，指定しなければ４とみなされる。

2.2.2 プログラム制御データ

プログラム制御データにはラベル・データ，ポインタ・データ，アドレス・データの３種がある。

(a) ラベル・データ

ラベル・データは値としてラベル定数すなわちステートメント・ラベルをとるもののことである。

(1) ラベル定数

ラベル定数とは，プログラム中にステートメント・ラベルとしてあらわれている名前のことで，この名前は出現によって定義される。

(2) ラベル変数

ラベル変数は値としてラベル定数（ステートメント・ラベル）をとるような変数のことで，宣言文でLABELと宣言することによって定義できる。

(b) ポインタ・データ

ポインタ変数はテーブル内の項目を指定するのに用いる変数で宣言文で独立にPOINTERと宣言するか，あるいはテーブルの宣言の中でCONTROLLEDに続く句の中に書くことにより定義する。後者の場合には，この変数はそのテーブルの固有のポインタ変数となる。

(c) アドレス・データ

アドレス変数はデータのアドレスを値としてもつような変数で宣言文でADDRESS と宣言することに定義する。

アドレス変数に何かのデータのアドレスをセットする時には組込関数ADDR を用いる。

また，アドレス変数の内容である特定のアドレスに値を代入したり参照する場合には，仮想変数を用いてアドレス変数の修飾型を作る。

2.3 変 数 の 分 類

LSDの仕様書の中では変数に関していろいろな呼び方を用いているので，これについて説明を行なうことにする。

(1) 単純変数

ただひとつその変数名を書くことによって識別できる変数名でスカラ・データである。

単純変数は構造体，テーブルあるいはスタックの要素であってもよい。

(2) 添字付変数

配列要素のことである。

(3) 仮想変数

アドレス変数の値として持っているアドレスの中に値を入れる時に，その値の持つ属性を明示するために仮想変数を用いる。

この変数は型に関する属性だけを備えたもので，この変数に値を代入することはできないし，仮想変数のみを参照することもできない。

仮想変数は常にアドレス変数と共に出現してそのアドレス変数の内容の属性をあらわすのに使われる。

仮想変数は単純変数とはみなさない。

(4) ポインタ変数修飾型

テーブルの項目，およびその項目の要素を識別したり，参照したりする場合に，特定のポインタ変数によってその位置を指定することがある。

これをポインタ変数修飾型と呼ぶことにし，形式は

ポインタ変数 . テーブル　あるいは

ポインタ変数 . テーブル項目の要素

とする。

注) この時のポインタ変数は単純変数，添字付変数である。

(5) アドレス変数修飾型

アドレス変数の内容を参照したり，これに値を代入する時には，前述の仮想変数を用いてその属性を明らかにする必要がある。

これをアドレス変数修飾型と呼び

アドレス変数 . 仮想変数

の形で指定する。

注) このときのアドレス変数は単純変数，添字付変数およびこれらをポインタ変数によって修飾した形であって，配列，ポインタ変数修飾型の配列であってはならない。

(6) ビット変数修飾型

ビット・ストリング・データの場合，そのスカラ変数の何番目のビットか

を指定して参照したりｏｎ／ｏｆｆにセットしたりすることができる。これをビット変数修飾型と呼ぶことにし

ビットストリング型　　ビットを示す

スカラ変数　.　整　数

の形とする。

この形はＳＥＴＢ，ＲＥＳＥＴＢ，ＩＦステートメントの中だけで用いられる。

(7) スカラ変数

単一のデータ項目のことで次のものを含む

- 単純変数
- 添字付変数　（配列要素）
- ポインタ変数修飾型でテーブル項目の要素が単純変数
- 〃　〃　添字付〃
- アドレス変数修飾型

(8) 擬　変　数

組み込み関数ＳＵＢＳＴＲは，代入ステートメントの左辺に出現し得るので擬変数（ｐｓｅｕｄｏ ｖａｒｉａｂｌｅ）と呼ぶことにする。その一般型は次のとおりである。

ＳＵＢＳＴＲ（Ｓ，ｉ〔，ｋ〕）

Ｓ　：　ビットあるいは文字型のスカラ変数

ｉ　：　抽出するビット・ストリングあるいは文字ストリングのはじめの位置を示す。符号のない整数あるいは数値型の単純変数

ｋ　：　抽出するビット・ストリングあるいは文字ストリングの文字数を示す。符号のない整数あるいは数値型の単純変数

2.4 変数の参照方法

ＬＳＤで用いる変数の参照方法は，次のとおりである。

2．4．1　単純変数

①　テーブルの要素でない場合

単にこの変数の名前を書くことにより参照する。

②　テーブルの要素となっている場合

①と同じ形あるいはポインタ変数による修飾型を用いる。ただし，後者の場合はスカラ変数ではあるが，全体としては単純変数とはみなさない。

例2－5　　X

P1．X

2．4．2　添字付変数

配列要素は，添字式をカッコでくくったものを配列名のあとに書くことにより参照する。

添字式の形は次のとおりである。

＜添字式＞＝i＊j＋k｜i＊j－k｜

i＊j｜j＋k｜j－k｜＋k｜－k｜k

ここでi，j，kは数値型の単純変数あるいは整数であって単純変数は構造体，テーブル，スタックの一部であってもよいが，ポインタ変数によって修飾される形であってはならない。

また，アドレス変数修飾の形であってはいけない。

例2－6　　Aを数値型の配列とする

A（I1＊I2）

A（1－I3）

添字付変数全体としては，ポインタ変数修飾の形をとることができる。

例2－7　　P2．A（I1＋3）

2.4.3 配　　列

代入ステートメントや入出力ステートメントなどで，配列全体を1度にとり扱かう場合には配列名だけを書くことによって代入したり，参照したりすることができる。

さらに配列がテーブル要素である場合はポインタ変数による修飾型を作ることもできる。

A，Bを配列とする時，次の代入ステートメントは有効である。

例2－8　　A＝P．B；

2.4.4 構　造　体

構造体を1度にとりあつかう時には，その構造体名だけで参照したり，値を入れたりすることができる。

例2－9　　PUSH　S1　INTO　S2；

ただし　S1…構造体
　　S2…スタック

2.4.5 テーブル

テーブルを参照したりこれに値を入れたりする場合，2通りの意味に分かれる。

① テーブル内の1項目を指す場合で，通常はこの意味でテーブル名を用いる。

例えば，T1というテーブルがあり，固有のポインタ変数がP1であるとすれば，単に

T1

と書くと，これはP1．T1に等しく，テーブルT1内のP1で示される項目を指すことになる。

この形のものにはポインタ変数修飾型を作ることができる。

② テーブル全体を指す場合で，入出力ステートメントで用いられる場合である。

これらの意味の解釈は個々のステートメントにより異なるが，特に指定がなければ①の意味と解釈してよい。

2.4.6 スタック

スタックを参照したりこれに値を入れたりする場合にも，テーブルの場合と同じく2通りの場合が考えられる。

① スタック内の1項目を指す場合で，単にスタック名を書いてもよし，スタック名のあとに（＊〔－整数〕）をつけてもよい。例えばスタック名をSとすると

S　　は　スタックの一番上の項目
S（＊）　は　同じく　一番上の項目
S（＊－1）は　スタックの上から2番目の項目

を指す。

② スタック名によって，スタック全体を意味する場合である。

これらの意味の解釈は個々のステートメントにより異なるが，特にことわらないかぎり①の意味とみなす。

注) スタックの項目の要素に対して（＊〔－整数〕）を指定することはできない。

2.4.7 アドレス変数修飾型

アドレス変数の内容に値を代入したり参照したりする場合には，アドレス変数修飾型を用いる。

この形式は

アドレス変数．仮想変数

であって，ここでいうアドレス変数は

- 単純変数
- 添字付〃

- ポインタ変数（単）の修飾型
- ポインタ変数（添）の　〃

のいずれかである。

2.4.8 ポインタ変数修飾型

テーブルの項目およびその項目の要素を識別したり参照したりする場合には，ポインタ変数修飾型を用いる。

この形式は

ポインタ変数．テーブル　あるいは

ポインタ変数．テーブル項目の要素

であって，ここでいうポインタ変数は

- 単純変数
- 添字付変数

のいずれかである。

3. 式

式はある値を計算するのに使われる算法である。

式は，演算の種類によって算術式，論理式，関係式および連結式となる。これらの中で用い得るオペランドの型は式の種類によって定まっている。

3.1 算術式

算術式は算術演算を行なう式であって，算術データから成るオペランドと算術演算子によって作られるものである。

ＬＳＤでは算術演算子は＋，－，＊，／の４種，

すなわち，四則演算を行なう演算子である。

次にＬＳＤの算術式をＢＮＦで記述してみる。

＜算術式＞::＝＜因子＞｜＜算術式＞＋＜因子＞｜＜算術式＞－＜因子＞｜
＋＜算術式＞｜－＜算術式＞

＜因　子＞::＝＜１次子＞｜＜因子＞＊＜１次子＞｜＜因子＞／＜１次子＞

＜１次子＞::＝＜符号のない整数＞｜＜算術型スカラー変数＞

演算の順位は＊，／の方が，＋，－よりも高く，同順位の場合には左から右へと算定する。

ここで算術型スカラー変数とは，算術型の値をとるスカラー変数のことであって，すなわち単純変数あるいは添字付き変数のいずれでもかまわない。

3.2 関係式

関係式は２つのオペランドを比較演算子でつないだもので両オペランドの型は等しくなければならない。

いずれの場合にも関係式の結果は長さ１ビットのビット・ストリング・データとなり，true の場合１（on）false（off）となる。

＜比較演算子＞::＝＜比較演算子１＞｜＜｜¬＜｜＜＝｜＞＝｜＞｜¬＞

＜比較演算子１＞：：＝　＝｜¬＝

関係式をＢＮＦで書くと次のようになる。

＜関係式＞：：＝＜算術型関係式＞｜＜ビット型関係式＞｜＜文字型関係式＞
｜＜ポインタ型関係式＞

＜算術型関係式＞：：＝＜算術型オペランド＞＜比較演算子＞＜算術型オペランド＞

＜算術型オペランド＞：：＝＜算術変数＞｜＜符号つきの整数＞

＜ビット型関係式＞：：＝＜ビット型オペランド＞＜比較演算子１＞＜ビット型オペランド＞

＜ビット型オペランド＞：：＝＜ビット変数＞｜＜擬変数＞｜＜ビット定数＞
｜＜８進定数＞｜＜１６進定数＞

＜文字型関係式＞：：＝＜文字オペランド＞＜比較演算子＞＜文字オペランド＞

＜文字オペランド＞：：＝＜文字変数＞｜＜擬変数＞｜＜文字定数＞

＜ポインタ型関係式＞：：＝＜ポインタ変数＞＜比較演算子１＞＜ポインタ変数＞｜＜ポインタ変数＞＜比較演算子１＞
NULL

算術型の関係式は，算術変数あるいは符号つきの整数を比較演算子で結んだものである。

ビット型では，ビット変数やビット・ストリング型の定数，あるいは擬変数ＳＵＢＳＴＲでビット型のものをオペランドとし演算子で結ぶ。ただしこの場合，大小関係の比較はゆるさず等しいか否かの関係だけを調べる。
左右のオペランドのビット数が異なる場合には，最左端のビットをそろえ，不足分にゼロをつめて右に延長する。

文字型では，文字型の変数，文字オペランドおよび擬変数ＳＵＢＳＴＲの文字型のものをオペランドとし演算子で結ぶ。照合順序は
９＞…＞０＞…＞Ｚ＞…＞Ａ＞特殊文字とする。

左右オペランドの文字数が異なる場合は，最左端の文字をそろえ，不足分に空白をつめて右に延長する。

ポインタ型の関係式は，両オペランドがポインタ変数で等しいか否かを調べるものである。特別な形として

＜ポインタ変数＞＜比較演算子１＞NULL

がある。NULL は何も入っていないことを示すポインタの値で，キーワードである。

3.3 論　理　式

論理式は論理和，論理積，論理否定を行なう演算で，オペランドとしてはビット・ストリング・データあるいは関係式を取るものである。その構文をＢＮＦで示すと次のようになる。

＜論理式＞::＝＜論理因子＞｜¬＜論理因子＞

＜論理因子＞::＝＜論理因子＞⊥＜論理１次子＞｜＜論理因子＞&＜論理１次子＞｜＜論理１次子＞

＜論理１次子＞::＝＜ビット定数＞｜＜８進定数＞｜＜１６進定数＞｜＜ビット変数＞＜ビット型擬変数＞｜＜関係式＞

オペランドのビット長が異なる場合には，最左端のビットをそろえて不足分にゼロをつめて右に延長する。

演算の順位は，関係式の中の比較演算が一番高く，次に¬（ＮＯＴ），その次に｜（OR）あるいは＆（ＡＮＤ）であるとし，同一順位のものでは左から右へと算定することにする。

3.4 連　結　式

連結式は文字型のオペランドに対して連結演算（concatenation)を行なうものである。

＜連結式＞::＝＜文字オペランド＞｜＜連結式＞‖＜文字オペランド＞

＜文字オペランド＞::＝＜文字変数＞|＜擬変数＞|＜文字定数＞

連結式においては，文字数の合計が２５６文字を越えてはならない。

4. 宣　　言

ＬＳＤのプログラムで用い得る名前は，いろいろな対象を表わすことができる。例えばファイルをあらわしていることもあるし，文字データのことも算術データのこともある。名前によって表現される対象を規定する性質を属性(attribute)という。

属性には，データの型に関するもの，有効範囲に関するもの，初期値の設定に関するものなどがある。

この名前と属性を結びつけるのが宣言であるが，この宣言にはDECLAREステートメントによって陽に示すもの，暗黙のうちにきまるもの，前後関係によってきまるものがある。

4.1 DECLARE ステートメント

DECLAREステートメントは名前の属性を指定するために用いられる非実行ステートメントである。

一般型は次のとおりである。

{DECLARE / DCL} DCL要素，DCL要素，…；

ＤＣＬ要素にはoption 1とoption 2の2種類がある。

option 1

名前　〔属性〕……〔ＩＮＩＴＩＡＬ属性〕

option 2

名前　〔（次元属性）〕属性〔属性〕……，

＊名前〔属性〕……〔ＩＮＩＴＩＡＬ属性〕〕

〔，＊名前〔属性〕……〔ＩＮＩＴＩＡＬ属性〕〕

(1) option 1は単純変数，配列，ファイルの定義のときに用いる。

(2) option 2は構造体，テーブル，スタックの場合に用いる。

このときには，まず構造体，テーブルあるいはスタックの名前を書き，その後に，全体としての属性（スタック属性，構造体属性，テーブル属性）を指定し，コンマで区切って構成要素を列記する。

構成要素はアスタリスクのあとにつづけて書き，さらに要素の属性を指定する。構造体，テーブルの場合には要素の属性の後に，INITIAL属性を指定することができる。

スタックには，INITIAL属性は指定できない。

個々の属性については後述する。

① 属性はそれが結びつけられる名前の直後にブランクで区切って任意個つづけることができる。

② DECLAREステートメントによる宣言は，その名前に対する代入や参照が行なわれる以前になければならない。

③ 一つの名前に対する宣言は同一external procedureでは一度しか書けない。

④ 構造体，テーブル，スタックの要素に指定することのできる属性は

次元属性

NUMERIC　属　性

BIT　〃

CHARACTER　〃

POINTER　〃

ADDRESS　〃

LABEL　〃

のみである。

4.2 前後関係による宣言

(1) ステートメントの前にコロンで区切ってつけられている名前は，ステートメントラベル定数である。

(2) テーブルに関する宣言の中でCONTROLLEDに続く括弧の中の名前は，ポインタ変数とみなす。

(3) external procedureの名前，あるいはinternal procedureに含まれないENTRYステートメントによって規定される2次的な入口名はEXTERNAL属性が与えられる。

(4) SYSPRT，SYSINは宣言されなければ，ファイル名であるとみなす。

4.3 省略時の解釈

LSDでは，DECLAREステートメントにもよらず，前後関係によっても解釈できない名前が出現すると算術型の単純変数とみなす。

4.4 属　　性

属性には，いろいろな種類のものがある。例えば配列の大きさを指定するものは次元属性であり，データの型が算術型であるかビット型であるかを示すのは型に関する属性である。これらの属性には組み合わせて宣言することの可能なものと排他的なものとがある。以下個々の属性について述べることとする。

(a) 次元属性

次元属性は配列あるいはテーブル，スタックの上限を定義するものである。

一般型は　　（符号のない整数）

であり，この属性は名前の直後に他のすべての属性に先立って書かなければならない。

この宣言のあとにTABLEあるいはSTACK 属性が指定されなければ配列とみなす。

(b) データの型に関する属性

(1) 算術データ

変数はNUMERIC と指定されたとき，あるいは他のデータの属性が全

く指定されない場合に算術データであると宣言される。

(2) ビット・ストリング・データ

ビット・ストリング・データに対しては，データの長さを付けてＢＩＴと宣言することによってその属性を規定する。

ＢＩＴ〔（長さ）〕

① 長さは符号のない整数（３６以下）である。

② 長さを指定しなければ３６とみなす。

(3) 文字ストリング・データ

文字ストリング・データはCHARACTER と宣言し，そのあとに文字数を指定することによってその属性を規定する。

{CHARACTER / CHAR} 〔（長さ）〕

① CHARACTERはCHAR と略して書くことができる。

② 長さは符号のない整数（２５６以下）である。

③ 長さの指定を省略すると４とみなす。

(4) ポインタ属性

ポインタ変数であることを宣言するのにはPOINTER と宣言すればよい。

POINTER

① テーブル宣言のCONTROLLED のあとに出現した名前もポインタ変数とみなされ，この場合にはＰＯＩＮＴＥＲという宣言は行なってはならない。

(5) アドレス属性

変数名はADDRESS と宣言されることによってアドレス変数という属性を与えられる。

ADDRESS

(6) ラベル属性

変数名はLABELと宣言されることによってラベル属性を与えられる。

LABEL

(c) データの構成に関する属性

(1) 構造体属性

変数が構造体であることを示すには，STRUCTUREと宣言する。

この属性とともに書くことのできる属性はEXTERNAL属性とINIT−IAL属性であり，構造体はDCL要素のoption2に従う。

(2) テーブル属性

変数がテーブルであることを示すには，次元属性のあとにTABLEと宣言し，さらにこのテーブルに固有なポインタ変数をCONTROLLEDのあとに指定しなければならない。

TABLE CONTROLLED（変数名）

この場合もDCL要素のoption 2に従う。

テーブル属性とともに指定できるのは次元属性，INITIAL 属性ある。

EXTERNAL属性は暗黙のうちに持つ。

(3) スタック属性

変数がスタックであることを示すには，次元属性と共にSTACKと宣言する。

STACK

スタックの場合にもDCL要素のoption 2に従う。

EXTERNAL属性は暗黙のうちに持つ。

(d) 範囲の属性

範囲の属性は，宣言された名前の有効範囲を指定する属性である。

EXTERNAL

EXTERNAL と宣言した変数は，どんなprocedure からも参照できる。

EXTERNAL 属性を宣言した場合には，他のすべての属性が一致しなけ

ればならない。

(e) VIRTUAL属性

VIRTUAL属性は，この属性を宣言した変数を仮想変数とする。

仮想変数は，なんら実体のない変数でアドレス変数修飾型の場合に，アドレス変数にそのアドレスが入っているデータの属性を明らかにするために用いる。

VIRTUAL 属性を宣言した場合の他の属性との関係

① データの型に関する属性を必ず宣言しなければならない。

② EXTERNAL 属性は宣言できない。

(f) INITIAL 属性

INITIAL 属性はデータに初期値（定数）を与えるのに用いる。

$$\left\{\begin{matrix}\text{INITIAL}\\ \text{INIT}\end{matrix}\right\}\ (\text{item}[\ ,\ \text{item}]\cdots)$$

① ここで定数というのは

算術定数，文字定数，ビット・ストリング定数，

ラベル定数であって，ここで宣言している名前の型（データ属性）と一致していなければならない。

② INITIAL属性はポインタ・データ，アドレス・データ，スタックに指定することはできない。

③ item は次の形である。

｛定数｜くりかえし定数＊定数｜none｜くりかえし定数＊none｝

ここでnone というのは，特に指定をしないことで空白でもよいし，全く何もなくてもよい。

④ 変数名が配列の場合は，はじめから順に指定したい個数のitem を書けばよい。

例４－１

```
DCL  A(10) NUMERIC
INIT(1,3*0,5,2*,0);
```

この場合Ａには次の値がセットされる。

Ａ（１）　…　１

Ａ（２）　…　０

Ａ（３）　…　０

Ａ（４）　…　０

Ａ（５）　…　５

Ａ（６）　…　不定

Ａ（７）　…　不定

Ａ（８）　…　０

Ａ（９）　…　不定

Ａ（10）　…　不定

⑤　ラベル変数にＩＮＩＴＩＡＬ属性で書くことのできるラベル定数は，そのprocedure内で定義されているステートメントラベルでなければならない。

⑥　ＥＸＴＥＲＮＡＬ属性を持つ変数にもＩＮＩＴＩＡＬ属性は指定できるが，二つ以上のプロシージャーで同じ変数に異なる値をＩＮＩＴＩＡＬ属性で入れていると結果は保証しない。

⑦　テーブル構造体を宣言している場合には，ＩＮＩＴＩＡＬ属性は各要素のあとに書く。

例４－２

```
DCL  A(10)  TABLE  CONTROLLED(P),
  *B  CHAR(4)  INIT  ('ABS',……),
  *C  BIT  INIT  (O'1',……),
  *D  INIT(1,……);
```

(g) ファイル属性

変数がファイルであることを示すのには，ファイル属性を指定する。ファイルは常にEXTERNAL 属性を持っており，ファイル属性と共に他の属性を指定することはできない。ＬＳＤで用いるファイルはカード・イメージの入力およびライン・プリンタへの出力用のファイルSYSIN，SYSPRT以外は，すべてDECLARE ステートメントによって宣言しなければならない。

一般型は次のとおりである。

ファイル名　FILE (RCDSIZE = {符号のない整数 [, WORD / CHAR] / 〃 , CHAR},

BLKSIZE =符号のない整数$_2$〔RECORD〕

〔, TMOD = {6 / 8 / 9}〕)

① RCDSIZE はこのファイルのレコードサイズで，WORDあるいはCHARをつけることによって何語分あるいは何字分であるかを指定する。

② BLKSIZE はこのファイルが何レコードを１ブロックとしているかを指定するもので，RECORDは省略してもよい。

③ TMODは転送モードのことで，指定がなければ８とみなす。

④ ファイルは自動的にEXTERNAL 属性を持っているので１つのプログラムを構成する２つ以上の external procedureで同一ファイルに関して異なった宣言を行なうと結果は保証しない。

4.5 EQU ステートメント

同じエリアに異なる名前をつけたいとき（データの型は異なってもよい）には，ＥＱＵステートメントを用いて変数を結びつけることができる。一般型は次のとおりである。

ＥＱＵ（変数名１，変数名２〔，変数名３〕…）

〔，（変数名１，変数名２〔，変数名３〕…）〕…；

ここで変数名1と変数名2……は次の組み合わせである。

	変数名1	変数名2…	
①	単純変数名 配列名 構造体名	単純変数名 配列名 構造体名	但し 変数名1のTOTAL WORD数か1項目の占有WORD数が変数名2…のそれと等しいか大きくなければならない。
②	テーブル，スタックの要素名	単純変数名 配列名	
③	テーブル名 スタック名	①と同様	

5. 各ステートメントの説明

LSD で使用可能なステートメントは次のとおりである。

1. PROCEDURE ステートメント
2. END 〃
3. ENTRY 〃
4. Assignment 〃
5. SETB 〃
6. RESETB 〃
7. SET 〃
8. ERASE 〃
9. SEARCH 〃
10. PUSH 〃
11. POP 〃
12. OPEN 〃
13. CLOSE 〃
14. WRITE 〃
15. READ 〃
16. SCAN 〃
17. DO 〃
18. GO TO 〃
19. IF 〃
20. CALL 〃
21. STOP 〃
22. RETURN 〃
23. ENTER 〃

24. EXIT　　ステートメント

25. NULL　　〃

26. Debug　　〃

27. INCREASE　　〃

以下に個々のステートメントの説明を述べることにする。

5.1 PROCEDURE ステートメント

PROCEDUREステートメントはそれ以下のステートメントがLSDのProcedureであることを示すと同時にその入口名を定義する。

一般型は次のとおりである。

label : { PROCEDURE / PROC } 〔OPTIONS(MAIN)〕

〔(仮パラメタ・リスト)〕;

① OPTIONS(MAIN)のある場合は，これがメイン・プログラムのexternal procedureであることを示し，そうでなければサブ・プログラムとみなす。internal procedureにはOPTIONS(MAIN)と指定することはできない。

② OPTIONS(MAIN)を指定した場合には仮パラメタ・リストは指定できない。

③ 仮パラメタ・リストは仮パラメタを2個以上あればコンマで区切って並べたもので，仮パラメタとなり得るものはスカラ・データ，配列，構造体である。

5.2 END ステートメント

END ステートメントは2種類あり1つはpocedureを終了させるものであり，もう1つはDOループを終了させるものである。

一般は次のとおりである。

(1) procedure を終了させるENDステートメント

END procedure 名 ;

(2) DOループを終了させるENDステートメント

〔 label:〕 END 〔ラベル名〕;

① DOループを終了させるENDステートメントでは，ラベル名を指定してもしなくてもよい。ここでラベル名というのはDOステートメントにつけてあるlabelのことでDOが入れ子になっている場合には外側のDOステートメントのlabelをENDのあとに書くことによって1つのENDステートメントで複数のDOループを終了させることができる。

例5－1

```
L1 :  DO……;
      :
L2 :  DO……;
      :
      END L1 ;
```

② DOループに対応するENDステートメントでEND；のみの形の場合には，各DOステートメントに対応してENDがなければならない。換言すればEND；は1番最近出現したDOループを終了させるものである。

例5－2

（例5－1）をEND；の形で書くと次のようになる。

```
L1 :  DO……;
      :
L2 :  DO……;
      :
      END;
      END;
```

③ procedureを終了させる場合のENDステートメントには必ずprocedue名を書かねばならない。

5.3 ENTRY ステートメント

ENTRY ステートメントは procedure の2次的な入口名を指定するもので一般型は次のとおりである。

入口名 ： ENTRY 〔（仮パラメタ・リスト）〕；

① 仮パラメタ・リストに関してはPROCEDURE ステートメントを参照のこと。

② MAIN−PROCEDURE内では，指定してはいけない。

5.4 代入ステートメント

5,4,1 一般形

代入ステートメントは式を計算してスカラ変数，配列などに値を代入するのに用いる。一般型は次のように分類できる。

Option 1 （スカラ変数への代入）

{スカラ変数 / 擬変数} ＝式；

Option 2 （配列への代入）

{配列 / 擬変数の配列} ＝ {配列 / スカラ・データ} ；

Option 3 （構造体への代入）

構造体 ＝ {構造体 / テーブル / スタック} ；

Option 4 （テーブル，スタックへの代入）

{テーブル / スタック} ＝ {構造体 / テーブル / スタック} ；

Option 5　（ラベル変数への代入）

$$\text{ラベル変数} = \left\{\begin{array}{l}\text{ラベル定数}\\ \text{ラベル変数}\end{array}\right\};$$

Option 6　（ラベル変数の配列への代入）

$$\text{ラベル変数の配列} = \left\{\begin{array}{l}\text{ラベル定数}\\ \text{ラベル変数}\\ \text{ラベル変数の配列}\end{array}\right\};$$

Option 7　（ポインタ変数への代入）

ポインタ変数＝ポインタ変数；

Option 8　（ポインタ変数の配列への代入）

$$\text{ポインタ変数の配列} = \left\{\begin{array}{l}\text{ポインタ変数}\\ \text{ポインタ変数の配列}\end{array}\right\};$$

① Option 1のスカラ変数は算術，ビットあるいは文字型のものである。左辺と右辺の型が異なる場合には，右辺の式の算定結果を変換ののち左辺に代入する。

② Option 2における配列は，算術，ビットあるいは文字型のものであって，右辺の配列についても同様である。両辺の型が異なる場合は上記①と同様の変換を行なう。

Option 2，Option 6，Option 8で両辺が配列で両配列の大きさの異なる場合は次のようになる。

（左辺の配列の大きさ）≧（右辺の配列の大きさ）のとき

右辺に対応する左辺の要素すべてに代入を行ない，

左辺の余ったものには代入を行なわない。

（左辺の配列の大きさ）＜（右辺の配列の大きさ）のとき

右辺に対応する要素の存在するところまで左辺全体に代入を行なう。

さらに，Option 2，6，8で右辺がスカラ・データ（Option 2では，算術，ビットあるいは文字型でなければならない，Optin 6で

はラベル・データ，Option 8 ではポインタ数である）の場合には，左辺の配列の要素すべてに右辺の同じ値を代入する。

③ Option 3 では両辺の構造体の構成要素の順序，型，大きさはすべて等しくなければならない。

④ Option 4 では，テーブルやスタックの1項目と右辺の構造体の構成は，全く等しいものでなければならない。

テーブルの場合には，このテーブルと共に宣言されているポインタ変数の示している項目に対して代入を行なう。代入ステートメントの実行後もこのポインタ変数は変らない，スタックの場合にも一番上の項目に代入が行なわれる。

⑤ Option 3，4 以外の場合に，両辺に出現する変数がスタックやテーブルの要素であることも可能であるがこの場合は，テーブルならばこのテーブルと共に宣言したポインタ変数の指している項目に属するもの，スタックならば一番上に入っている項目の中のものが用いられる。

5.4.2 代入の際の変換規則

(1) 算術変数＝算術式；

変換は行なわず，右辺の式を算定し左辺に代入する。

(2) 算術変数＝論理式；

右辺の式の算定結果のビット・ストリングを右シフトして左辺に代入する。右辺のビット数が36ビットであれば変換は行なわない。

例5－1
```
DCL B1 BIT(10);
B1 = B'1110001100';
XX = B1;
```
この結果XXには B'000……01110001100' が入る。

（000……0 は 26 ビット）

(3) 算術変数＝連結式；

右辺の文字ストリングの値としては文字表現の〔＋｜－〕数値定数の形しかゆるさない。ただし，前後の空白はあってもかまわない。この場合には，右辺の値を内部表現に変換ののち左辺に代入する。

例５－２　DCL　C1　CHAR(５)；

C1=▼+10▼；

XX=C1；

この結果　XX には内部表現の１０すなわち

B▼0…01010▼が入る

32ビット

(4)　ビット変数＝算術式；

右辺の算術式の内部表現を下から有効ビット数分だけ取り出す。内部的には左シフトを行なうことになる。

例５－３　DCL　B1　BIT(１０)；

XX＝１２；

B１=XX；

この結果XX には１２，B１にはB▼0000001100▼が入る

(5)　ビット変数＝論理式；

変換は行なわず，右辺の値を算定し，左辺に代入する。左辺のビット数の方が短かい場合には右辺の頭からそのビット数分が取られ，右辺の方が足りない場合には，後に必要な個数だけ０をつけ加える。

(6)　ビット変数＝連結式；

右辺の文字ストリングが１，０から成る集合であればそのままこれをB▼▼の中の値と考えて変換し左辺に代入する。文字ストリングの文字数が左辺のビット数を越える場合には，頭からそのビット数分だけを代入し残りは捨てる。また，左辺のビット数が足りない場合は文字ストリングの後に必要な個数だけ０を補って考える。

空白は０とみなす。

例５－４　DCL　B1　BIT(10),　B2　BIT(3);

DCL　C1　CHAR(8);

C1=▼11001010▼;

B1=C1;

B2=C1;

この結果B1にはB▼1100101000▼

B2にはB▼110▼

が代入される。

(7)　文字変数＝算術式；

右辺の数値変数の値を文字コードに変換して必要ならば－（マイナス）を付け加えて左辺に代入する。変換の結果文字ストリング（左辺）の方が長ければ頭に空白をつめる。短かすぎる場合は結果は保証されない。

例５－５　DCL　C1　CHAR(10), C2　CHAR(3);

XX=1521;

C1=XX;

C2=XX;

この結果C1には▼␣␣␣␣␣␣1521▼，C2は不定となる。

(8)　文字変数＝論理式；

右辺の算定結果のビット・ストリングの値を文字コードに変換して左辺に代入する。文字数がそろわない場合は(9)に準じる。

例５－６　DCL　B1　BIT(5);

DCL　C1　CHAR(8);

B1=　B▼10111▼;

C1=　B1;

この結果C1には文字コードの▼10111␣␣␣▼が入る。

(9)　文字変数＝連結式；

右辺のストリングの文字数が左辺と一致していればそのまま代入する。

左辺の方が短かい場合には，右辺の頭から有効文字数を取って残りを捨て長い場合には後に空白をつめる。

例５－７　DCL C1 CHAR(10), C2 CHAR(3), C3 CHAR(5);

C1＝'ABCDEFGHIJ';

C2＝'XYZ';

C3＝C1;

C3＝C2;

はじめの代入でC3には'ABCDE'が入り，あとの方では'XYZ␣␣'が入る。

5.5 SETB ステートメント

SETB ステートメントは，指定された変数の指定されたビットをONにするステートメントである。

一般型は次のとおりである。

〔label:〕SETB ビット変数．符号のない整数

① ここでビット変数というのは，ビット属性を持ったスカラー変数のことである。

② 符号のない整数とは，第何ビット目をONにするかを示すもので第１ビット目を１とし順に右の方へと番号をつけていく。

③ ビット変数のビット数はONにしようとするビットが含まれているように宣言されていなければならない。

5.6 RESETB ステートメント

RESETB ステートメントは，SETB ステートメントと対をなすもので指定された変数の指定されたビットをOFFにするステートメントである。

一般型

〔label:〕RESETB ビット変数．符号のない整数

5.7 SETステートメント

SETステートメントはテーブルの一番はじめの空き領域に1項目をセットするためのステートメントで一般型は次のとおりである。

〔label：〕SET 構造体 INTO テーブル名

〔〔WHEN OVERFLOW 〔THEN〕〕単純型GOTOステートメント〕；

① SETの次に指定するものは構造体であるがこれはテーブルの1項目の構成と一致していなくてはならない。

② このステートメントの実行の結果，テーブルの中の空いている項目のものの中に構造体がセットされ，このテーブルと共に宣言したポインタ変数にはこの項目を示す値が入れられる。

③ WHEN OVERFLOW…以下の節がある場合には，テーブルがオーバーフローしているか否かをチェックしてオーバーしている時には，指定したラベルにJumpすることになる。

④ WHEN以下を指定しなければテーブル・オーバーフローは起らないという前提があるものとしてチェックは行なわない。

5.8 ERASE ステートメント

ERASEステートメントはテーブルの項目を削除するためのステートメントで一般型は次のとおりである。

〔label：〕ERASE テーブル〔FROM ポインタ変数〕；

① FROMポインタ変数を指定すると，そのテーブルの中でそのポインタ変数が示している位置からさいごまでの項目を削除する。

② ①の場合そのポインタ変数の値は変わらない。

③ FROM以下を省略するとテーブル内のすべての項目を削除する

④ いずれの場合にもこのステートメントを実行するとそのテーブルと共に宣言されたポインタの値は不定となる。

例５－８

いま次のようなDCLステートメントを持つテーブルがあるとする。

```
DCL T1(10) TABLE CONTROLLED(P1),
  * A1 CHAR(8),
  * A2 POINTER;
```

そしてさらにＰ２もポインタ変数で，このテーブルの中が次のようになっているとする。

T1

	A1	A2
	A	5
	B	4
	C	6
	D	7
P2→	E	9
	F	4
	G	3
	H	2

（図５－１）

Ｐ２が第５番目の項目を指しているとするとこの時

```
ERASE T1 FROM P2;
```

を実行するとＥ～Ｈで始まる項目が削除されＰ１は不定となる。

5.9 SEARCH ステートメント

SEARCHステートメントはテーブルの中から指定した変数あるいは定数と一致する項目をさがしてそのポインタをセットするステートメントで一般型は次のとおりである。

〔label:〕SEARCH テーブル要素 KEYED { スカラ変数 / 定数 }
〔FROM ポインタ変数$_1$〕〔TO ポインタ変数$_2$〕
{ BACKWARD / FORWARD } 〔INTO ポインタ変数$_3$〕

{〔IF EXIST 〔THEN〕〕 / IF NOT 〔THEN〕} 単純型GOTOステートメント；

① テーブル要素は，このSEARCHステートメントによって探し出そうとするデータの入っている可能性のあるテーブル要素のことでスカラ変数である。（配列名を指定することはできない）

② テーブル要素とスカラ変数あるいは定数（KEYEDの次に指定するもの）との型は一致していなくてはならない。

③ KEYEDに導かれるスカラ変数あるいは定数の値と一致するものがテーブル要素の中にあれば，ポインタ変数$_3$ の指定されているときはこのポインタ変数に，もし指定がなければテーブルと共に宣言されたポインタ変数に，この項目を示すポインタの値をセットしてIF以下を実行する。

④ テーブル要素の中にさがしているものがない場合には，ポインタ変数の値をNULLにセットする。

⑤ FROM ポインタ変数$_1$ およびTO ポインタ変数$_2$は；このSEARCHステートメントの対象がテーブル全体ではなく，これらのポインタ変数によって示される範囲に限定することを指定するものである。FROMを省略するとテーブル内の第1番目の項目，TOを省略するとテーブル内の一番さいごの項目が指定されているものとみなす。

FROM，TOに導かれているが，これらはテーブル内の位置を指しているのであって，SEARCHを始める位置を示すのではない。このため，ポインタ変数$_1$ ≦ポインタ変数$_2$ でなければならない。

⑥ BACKWARDあるいはFORWARDはテーブル内を探すsearchの方向を指定するものでBACKWARDの場合にはテーブルをさいごの方からはじめの方へとsearchを行ない，FORWARDは，はじめの方からさいごの方に向かって行なう，省略時にはFORWARDとみなされる。

⑦ IF EXISTを指定すると，一致したものがあればGOTOステートメントを実行し，IF NOTであれば該当するもののないとき，GOTOス

テートメントの実行を行なう。

⑧ IF EXISTは省略可能である。

例5－9

次のようなテーブルがあってその内容として図のような値が入っているものとする。

```
DCL TAB1 (10) TABLE CONTROLLED (P),
    * A CHAR
    * N NUMERIC ;
```

	A 1	1
P 1→	A 2	2
	B 1	3
	B 2	4
P 2→	B 3	5
	A 1	6
P 3→	A 3	7
	B 4	8

（図5－2）

今ポインタ変数P1，P2，P3が図のようにテーブルの項目を指しているとする，この時NAMEという文字変数をキーとしてsearchを行なう。

```
SEARCH A KEYED NAME FROM P1
INTO P IF EXIST THEN GO TO L1;
```

NAMEに▼A1▼が入っているとするとテーブルの第6番目に入っているA1が探し出され，Pにこの項目を指す。ポインタの値がセットされ，L1へコントロールを移す。また，

```
SEARCH A KEYED▼B1▼ FROM P2 TO P3 INTO Q
IF NOT GO TO L2;
```

ならばテーブル内の第5の項目から始めて第7まで探し，存在しないので

L 2 に行く。このときポインタ変数 Q には NULL がセットされる。

SEARCH N KEYED 1 FROM P1 TO P2 BACKWARD GO TO L3;

この場合にはテーブルの第5番目の項目から先頭に向かって P 1 の指している2番目のものまでを探して該当するものがないのでこのステートメントの次のステートメントにコントロールを移す，このテーブルと共に指定されたポインタ変数 P には NULL がセットされる。

5. 10 PUSH ステートメント

PUSH ステートメントはスタックを Push down して指定した1項目を入れるステートメントで一般型は次のとおりである。

〔label:〕 PUSH 構造体 INTO スタック

〔〔WHEN OVERFLOW 〔THEN〕〕単純型GOTOステートメント〕;

① ここで用いる構造体はスタックの項目と同じ構成でなければならない。

② WHEN OVERFLOW…以下の節があれば，スタックがオーバーフローしているか否かをチェックしてオーバーしている場合には，GOTOステートメントを実行することになる。

③ WHEN OVERFLOW…以下を指定しなければオーバーフローのチェックは行なわない。

5. 11 POP ステートメント

POP ステートメントはスタックから1項目を pop up して指定したエリアに移すもので一般型は次のとおりである。

〔label:〕POP 構造体 FROM スタック名

〔〔WHEN NOTHING 〔THEN〕〕単純型GOTOステートメント〕;

① 構造体はその構成においてスタックの1項目と等しくなければならない。

② WHEN 以下の節を指定した場合にはスタックが空になってとり出すも

ののない場合が起ると，GOTOステートメントを実行する。この場合指定した構造体への値のセットは行なわない。

③ WHEN 以下を指定しない場合，ユーザーがスタックの空になる状態はあり得ないと考えているとみなしてチェックは行なわない。

5.12 OPEN ステートメント

OPENステートメントは，ファイル名とデータ・セットとを結合させ，ファイルに対する属性を指定するものである。

一般型は次のとおりである。

〔label:〕OPEN ファイル名 〔INPUT |OUTPUT〕;

① INPUT, OUTPUT の指定を省略するとINPUT とみなされる。

② OPENステートメントが省略されると，最初のREAD/WRITEステートメントでファイルの open を行なう。

5.13 CLOSE ステートメント

CLOSE ステートメントは open されているファイルを，結びつけられていたデータ・セットから切り離すもので一般型は次のとおりである。

〔label:〕 CLOSE ファイル名;

① SYSIN, SYSPRT のファイルを用いるときはCLOSE ステートメントはなくともよい。

② 上記以外のファイルを用いた場合必ずCLOSE しなければならない。

5.14 WRITE ステートメント

WRITEステートメントはレコードを内部ストレージ内の変数から出力用エリアへmove し，ブロッキングを行なって指定されたファイルへ書き出すものであって一般型は次のとおりである。

〔label:〕WRITE ファイル名 FROM 変数

〔IMMEDIATE〕;

① WRITEステートメントでは，データの転送の際に変換は行なわない。したがってユーザーは1レコード分のエリアを用意してそのエリアの変数を介してレコードの転送を行なう。

② WRITEステートメントはSYPRT に対して行なうこともできる。この場合出力がライン・プリンタに出るわけであるがそのためにはユーザが文字型の値にデータを変換して出力エリアの形式をととのえなくてはならない。

③ レコードサイズの小さいもの，レコード数の少ないものを write する場合にはIMMEDIATE と指定すること。

5.15 READ ステートメント

READステートメントはレコードを入力用エリアから内部ストレッヂの指定エリアに転送する。（ファイルから入力用エリアへの転送を行ない，そののちデブロッキングを行なう）.

一般型は次のとおりである。

〔label:〕 READ ファイル名 INTO 変数
〔AT END 単純型GOTOステートメント〕;

① AT END以下を指定するとEnd of File になったとき制御の行く先を示すことになる。

② SYSINファイルからの read の場合には，ブロックサイズ15レコード，レコードサイズ80バイトのファイルとみなす。

③ READステートメントでは，データの転送の際に変換は行なわない。従って変換はユーザーが自分で行なわなければならない。特にSYSINからの readでは，データは文字コードでエリアに入るものとする。

5.16 SCAN ステートメント

SCANステートメントは実行時にSYSIN ファイルからデータを読み込み，その中から指定された文字数のデータを指定エリアにセットするもので一般型は次のとおりである。

〔 label : 〕 SCAN [{符号のない整数 / 単純変数}] INTO 変数 ;

① 符号のない整数は読み込む文字数を示すもので省略時には1と仮定する。

② 変数は文字変数でこのステートメントで指定している文字数を満たすものでなくてはならない。

③ 読み込まれるデータはカードの形式に基づいているものとし，80カラムのカードのどの部分を有効なものと考えるか，および1枚のカード毎にレコードがきれていると考えるか，あるいは 特定のカラムが次のカードの何カラムかに続いているものとするかは，ユーザーがラン・タイムのコントロール・カードのパラメタで指定する 。

BOUND＝(n_1 , n_2)　　n_1 …開始カラム　　n_2 …終了カラム

RECORD／STREAM　　RECORD…カード毎に区切る

STREAM…続けて考える。

このパラメタを省略すると BOUND＝(2, 72), RECORD とみなされる。

④ End Mark は1文字分全ビットONの O▼777▼である。

⑤ SCAN ステートメントをくりかえし実行している間でSYSINファイルに対してREADステートメントを行なうことを禁止する。

⑥ SCANの次に単純変数を指定してもかまわない。この場合も読み込みの文字数を示しているものであるから数値変数で1以上の値をとるものでなければならない。

5.17 DO ステートメント

DO ステートメントはDOループの開始を規定するステートメントで一般型

は次のとおりである。

Option 1　DO　制御変数＝{スカラ変数 / 定　数} TO {スカラ変数 / 定　数}

〔BY{スカラ変数 / 定　数}〕〔WHILE（論理式）〕；

Option 2　DO WHILE（論理式）；

① Option 1の場合，制御変数とは算術型の単純変数であり，変数および定数も算術型のスカラーデータである。

② 制御変数＝の次に書いてある値を初期値，TOのあとのものを終値，BYのあとのものを増分と呼ぶことにすると，初期値≦終値であってかつ，WHILE以下があれば，この論理式が真でないとDOループを1度も実行しない。DOループを1回実行するごとに増分を制御変数値に加え変数の値が終値よりも大きくなるか，あるいは論理式の値が偽になるまでくりかえしこのループを実行する。

③ 増分は特に指定しなければ1である。

④ Option 2では，論理式の値が真のときDOループを実行し，偽になると実行をやめてDOループの次のステートメントに制御を移す。

5.18 GO TO ステートメント

GO TO ステートメントは制御を指定されたステートメントへ移させるものであり，一般型は次のように単純型のもの（Option 1）と計算型（Option 2）の2種類がある。

Option 1（単純型GO TO ステートメント）

〔label:〕{GO TO / GOTO}　{ラベル変数 / ラベル定数}；

Option 2

〔label:〕 {GO TO / GOTO} 単純変数,

（ラベル定数〔，ラベル定数〕…）；

① ここでラベル変数というのは，ラベル属性をもつスカラ変数のことである。

② Option 2の単純変数というのは，算術型の添字のない変数のことである。

③ Option 1においては，指定されたラベル変数の内容，あるいはラベル定数へ制御を移す。

④ Option 2においては，指定された算術型スカラ変数の値に従って，ラベル定数に制御を移す。

例5－10

```
DCL LBL LABEL ;
    :
II=2
LBLO:GO TO II,(LBL1,LBL2,LBL3) ;
LBL2: II=3 ;
      GO TO LBLO ;
LBL3: LBL=L1 ;
      GO TO LBL ;
      :
L1:
```

この例では最初にLBLO に入るとII の値2に従ってLBL2 に制御が移り，次にLBLO に来るとII が3なのでLBL3 に行く，ここでLBL というラベル変数にL1 という値をセットしGO TO LBL の実行ののちL1 にJump する。

⑤ Option 2では，label定数の並びの個数は，算術型のスカラ変数のとり得る最大値と等しくしなければならない。もし，算術変数の値が大きいかあるいは0以下の場合にはこのGO TOステートメントの直後のステートメントに制御を移す。

5.19 ＩＦ ステートメント

ＩＦ ステートメントはプログラムの流れを式の値によって変えるもので一般型は次のとおりである。

Option 1

ＩＦ 論理式 〔THEN〕 ＩＦ，DO以外のステートメント；

Option 2

ＩＦ 文字ストリング ＩＳ { NUMERIC / ALPHA / ALPHANUMERIC / ENDMK }

〔THEN〕 ＩＦ，DO以外のステートメント；

Option 3

ＩＦ スカラ・データ＝スカラ・データ〔，スカラ・データ〕…

〔THEN〕 ＩＦ，DO以外のステートメント；

Option 4

ＩＦ ビット変数．符号のない整数 ＩＳ { ON / OFF }

〔THEN〕 ＩＦ，DO以外のステートメント；

① Option 1 では論理式が true の値を取ればＩＦ，DO以外のステートメントを実行する。

② Option 2では，文字ストリング変数は値として文字を持つような1文字の文字変数（スカラー変数）でなければならない。内容の文字コードが数値であるか，英数字か，英字かあるいはエンド・マークO▼777▼であるかを調べ，条件を満足すると後半のステートメントを実行する。

③ Option 3では，両辺のデータの型は関係式に準ずる。ただしスカラー変数でなければならない。

左辺が右辺のリストのどれかに該当していたら後半を実行する。

④ Option 4ではビットのON／OFFによって後半を実行するか否かをきめる。

5.20 CALL ステートメント

CALLステートメントはprocedureを呼び出してprocedureの指定された入口へ制御を移すのに用いる。

一般型は次のとおりである。

〔label:〕CALL 入口名（実パラメタ・リスト）;

① 入口名として書くことのできるものは，同じexternal procedure内のprocedure名，あるいはENTRYステートメントによって定義される2次的な入口名，あるいは他のexternal procedure名および入口名である。

② 実パラメタ・リストは実パラメタをカンマで区切って並べたもので，実パラメタとしてゆるされるものは，スカラデータ，配列，構造体，である

③ recursive call はゆるさないのでユーザーが注意すること。

5.21 STOP ステートメント

STOPステートメントはユーザープログラムの実行を終了させるステートメントで一般型は次のとおりである。

〔label :〕 STOP ;

5.22 RETURN ステートメント

RETURNステートメントはそのRETURNステートメントを含むprocedure を終了させ，制御を呼び出し側の procedure に返すもので一般型は次のとおりである。

〔 label:〕 RETURN ;

5.23 ENTER ステートメント

ＥＮＴＥＲステートメントはこれ以降ＥＸＩＴステートメントまでがユーザのアセンブラによる own ― coding であることを示し，一般型は次のとおりである。

〔 label:〕 ENTER ;

ユーザーのown－codingに関しては，ＬＳＤではcheckを行なわない。

5.24 EXIT ステートメント

ENTERステートメントに対応し，ユーザーのown－codingを終了させこれ以後のステートメントが再びＬＳＤ言語で書かれていることを示す。

一般型 EXIT ;

① EXIT ステートメントにはlabelはつけられない。

5.25 NULL ステートメント

ＬＳＤでは何も行なわないNULL ステートメントの使用をゆるしている。

一般型：

〔 label:〕 ;

5.26 Debug ステートメント

ＬＳＤではプログラムのデバッグを容易にするためのステートメントを設けている。これには指定した範囲内のラベルの履歴をたどるものと，指定した変数

の内容を出力させるものとの2種類があり，次のステートメントに分類できる。

(1) TRACE START ステートメント

(2) TRACE END ステートメント

(3) SNAP ステートメント

これらのステートメントに関しては，次に述べるが，TRACEの場合はSTARTからENDまでの間でコントロールの移ったラベル名を出力するステートメントでこの有効範囲は，スタティックに決まる。すなわちプログラムの物理的な順序によって決まり，論理的な流れにはよらない。またSNAPステートメントは，変数の内容を出力するものである。

いずれの場合にもこれらのステートメントに関連する出力はSYSPRTファイルに対して行なわれる。

(1) TRACE START ステートメント

TRACE START ステートメントは，このステートメントからTRACE END ステートメントあるいはこのTRACE STARTステートメントを含むprocedure のEND ステートメントまでに出現する人口名，あるいはステートメント・ラベルを出力するものであって一般型は次のようになっている。

TRACE START ;

① TRACE START ステートメントにはラベルをつけることはできない。

(2) TRACE ENDステートメント

TRACE ENDステートメントは上記のTRACE STARTステートメントの効果を終らせるもので次の形である。

TRACE END ;

① TRACE END ステートメントにはラベルをつけることができない。

(3) SNAPステートメント

SNAPステートメントはこの位置での変数の内容を示すもので一般型は次のとおりである。

SNAP 変数のリスト ;

① 変数として指定できるものはスカラ変数，配列，構造体，テーブル，スタックであって出力形式は

ビット，ポインタ型ならば		8進定数
文字型	〃	文字モード
数値型	〃	10進数

の形である。変数のリストは変数をカンマで区切って並べたものである。

② テーブルを指定した場合にはそのポインタ変数の示す項目，スタックならば一番上のものを出力する。

③ SNAP ステートメントにはラベルをつけることはできない。

5.27 INCREASE ステートメント

INCREASEステートメントは，指定されたポインタ変数の値を BY以後に示される項目数だけ increase する。

〔 label : 〕 INCREASE ポインタ変数〔 OF テーブル名〕
〔 BY { 定数 / 変数 } 〕 ;

① ポインタ変数が固有ポインタである場合は，OF テーブル名は省略可能である。

② 定数は数値定数，変数は数値型のスカラー変数である。

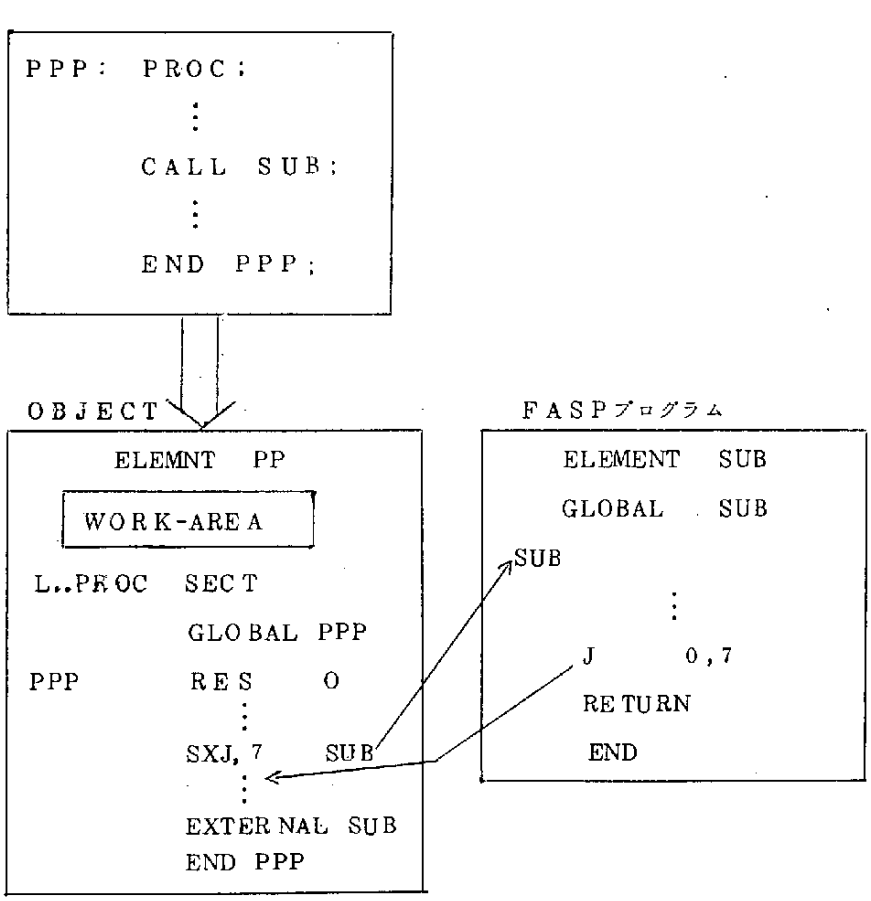
③ BY以後省略の場合は，1とみなす。

6. ＬＳＤとＦＡＳＰのリンク

6.1　入口名のリンク

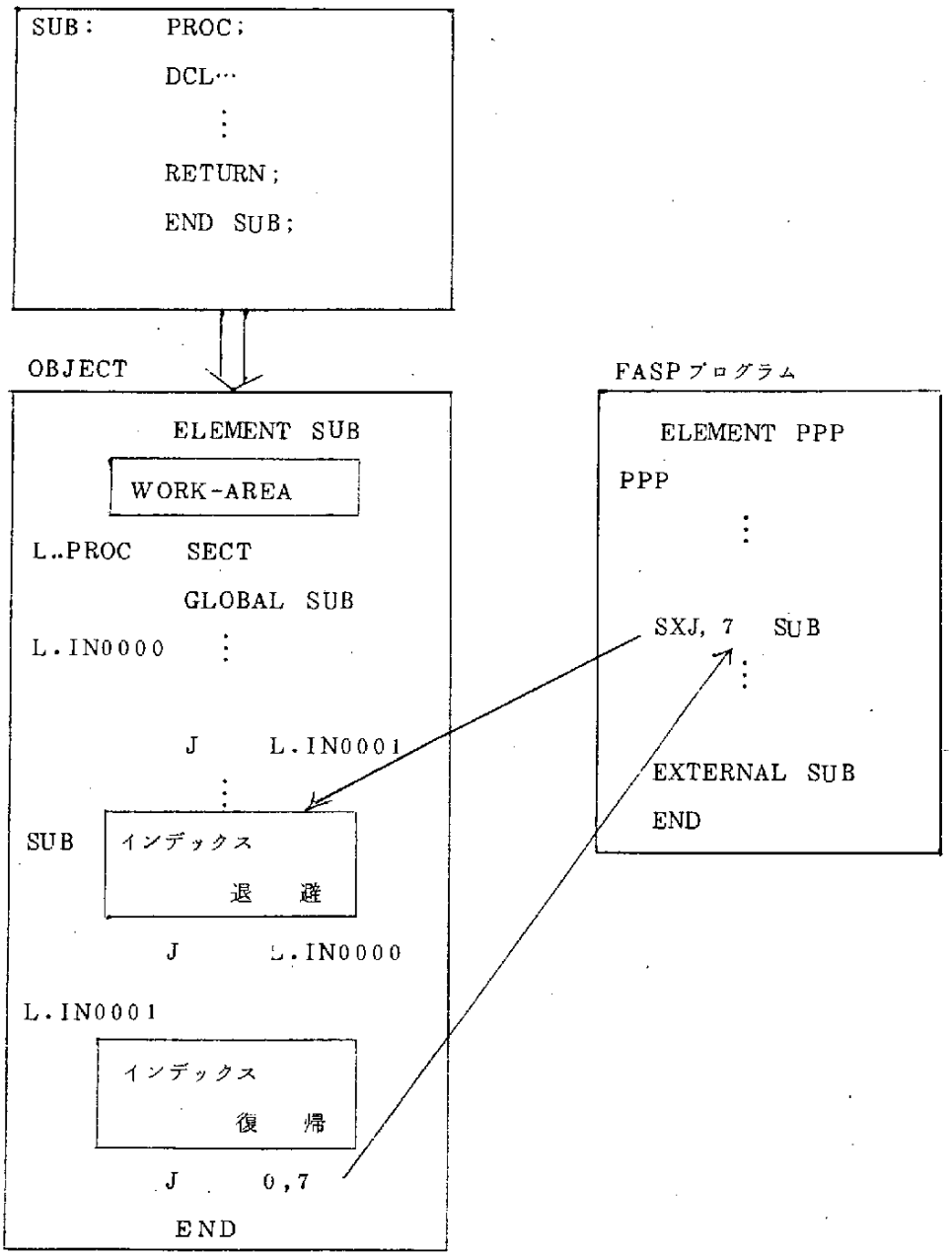
ＬＳＤ プログラムにおいてexternal procedure の入口名，および external procedure 内で internal procedure に含まれない部分にあるＥＮＴＲＹステートメントによる２次的な入口名は，ＧＬＯＢＡＬ宣言がなされているのでＦＡＳＰ プログラム内からＬＳＤプログラムを参照する場合，その入口名をEXTERNAL として宣言しなければならない。又ＦＡＳＰプログラム内の入口名をGLOBAL宣言しておくと，ＬＳＤ プログラムからFASP プログラムを参照できる。

例６－１



例６－２

LSD プログラム



6.2 LSDプログラムにおける変数の占有領域と参照の方法

6.2.1 単純変数

(1) 単純変数の占有領域

数値型	1語
ビット型	
ポインタ	
アドレス	
ラベル	
文字型	（文字数－1）／4＋1語

上のように領域がとられ，局所セクションの型でまとめられる。

例6－3

LSDプログラムのDCL ステートメント

```
DCL  A, B  BIT(3)  INIT('0'7'), C POINTER,
       D ADDRESS, E LABEL, F CHAR(4) INIT ('AAAA');
```

OBJECT

```
L.. WORK        SECT
A               RES         1, !!△
B               OCT         700000000000
C               RES         1, !!△
D               RES         1, !!△
E               RES         1, !!△
F               OCT         301301301301
```

(2) 単純変数を参照する方法

① LSD プログラム内でEXTERNAL 宣言を出したものを，FASP

プログラム内で参照するときには，そのプログラム内では，COMMON SECTIONを用いる。

例６－４

LSDプログラムのDCLステートメント

```
DCL A EXTERNAL ;
```

OBJECT

```
A COMMON
A RES   1,▼△
```

FASPプログラム

```
A COMMON
A RES   1,▼△
```

②．LSDプログラムのCALLステートメントの引数として引渡す。

例６－５

LSDプログラム

```
DCL A ;
      ⋮
   CALL SUB(A);
      ⋮
```

OBJECT

```
L..WORK SECT
A       RES 1,▼△
          ⋮
L..PROC SECT
          ⋮
        SXJ,7 SUB
        NOP   A,,1
          ⋮
      EXTERNAL SUB
```

FASPプログラム

```
   ELEMENT   SUB
   GLOBAL    SUB
A  OCT       0
          ⋮
SUB インデックス
        退  避
   LR     =O/70777777
   LA     0,7
   KSTA   *+1
   LA     *-*
   STA    A
```

6.2.2. 配　列

(1) 配列の占有領域

1要素の占有語数に配列の大きさをかけたものである。

例6－6

LSDプログラム

```
DCL  A(10), B(10);
```

名前は先頭のエリアにつけられる。

OBJECT

```
L..WORK  SECT
A        RES  10,▼△▼
B        RES  10,▼△▼
```

(2) 参照方法

6.2.1(2)と同じである。

6.2.3 構　造　体

(1) 構造体の占有語数

6.2.1(1)と6.2.2(1)の各要素の語数を加えたものである。

例6－7

LSDプログラム

```
DCL   STR  STRUCTURE,
      *A,
      *B  CHAR(5)  INIT(▼A***A▼),
      *C(2)  BIT  INIT(O▼7▼,O▼6▼);
```

OBJECT

```
L..WORK   SECT
STR       RES       0
          EQU       A=STR+0
          EQU       B=A+1
          EQU       C=B+2
          RES       1,△
          OCT       301134134134
          OCT       301100100100
          OCT       700000000000
          OCT       600000000000
```

(2) 参照方法

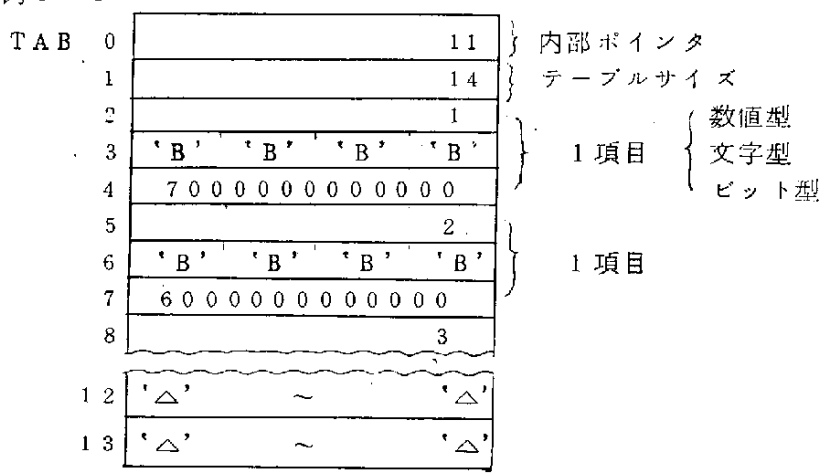
6.2.1 (2)と同じである。

6.2.4 テーブル

(1) テーブルの構成

同じ構成の構造体を複数個集めたものである。

例6－8



（例６－８）をＬＳＤで記述すると

```
DCL  TAB(4)  TABLE  CONTROLLED(P)
   *A  INIT(1,2,3),
   *B  CHAR(4)  INIT(▼AAAA▼,▼BBBB▼,▼CCCC▼),
   *C  BIT  INIT(O▼7▼,O▼6▼,O▼5▼);
```

となる。

(2)　テーブルの占有領域

6.2.3(1)と同じく計算した１項目分に，ＳＩＺＥをかけ２語を加えたものである。

例６－９

ＬＳＤプログラム

```
DCL  TAB(2)  TABLE  CONTROLLED(P)
     *A  INIT(1,2),
     *B  CHAR(5)  INIT(▼AAAAA▼,▼BBBBB▼);
```

OBJECT

```
TAB    COMMON
P      OCT        0
TAB    OCT        2…(*1)
       OCT       10…(*2)
       EQU       A=TAB+0
       EQU       B=A+1
       ORIGIN    TAB
       OCT       10
       ORIGIN    *+1
       OCT       1
       OCT       301301301301
       OCT       301100100100
       OCT       2
       OCT       302302302302
       OCT       302100100100
```

COMMON領域

P	0
TAB=A	8_{10}
B	8_{10}
	1
	▼A▼ ▼A▼ ▼A▼ ▼A▼
	▼A▼
	2
	▼B▼ ▼B▼ ▼B▼ ▼B▼
	▼B▼

（＊1） 内部ポインタ

（＊2） テーブルサイズ

(3) 参照方法

（例6－9）のようにLSD プログラムにおいては，テーブルはCOMMON SECTION としてエリアが確保されるため，参照する側でもCOMMON SECTION としてエリアを確保しなければならない。

例6－10

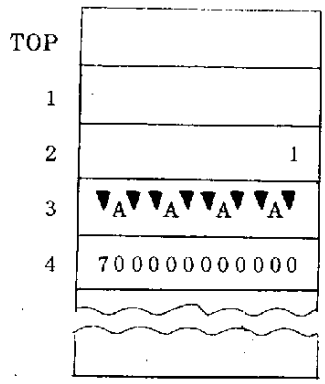
FASPプログラム

```
TAB    COMMON
P1     RES        1,▼△▼（ポインタ領域）
TAB1   RES        8
```

(4) FACPプログラム内でテーブル・ポインタへ値をセットし，そのポインタをLSDプログラムで用いる場合。

テーブル・サーチ等で一致する要素が見つかった時，その要素が示すTOPからの相対値をポイタへセットするのではなく，その要素が属する項目のTOP からの相対値をポインタへセットする。

例6－11



文字列（▼AAAA▼）が指定された時相対値は3であるが，テーブルポインタへは，文字列（▼AAAA▼）が属する項目の相対値2がセットされる。

6.2.5　スタック

(1)　スタックの構成

6.2.4(1)と同じである。

(2)　スタックの占有領域

6.2.4(2)と同じである。

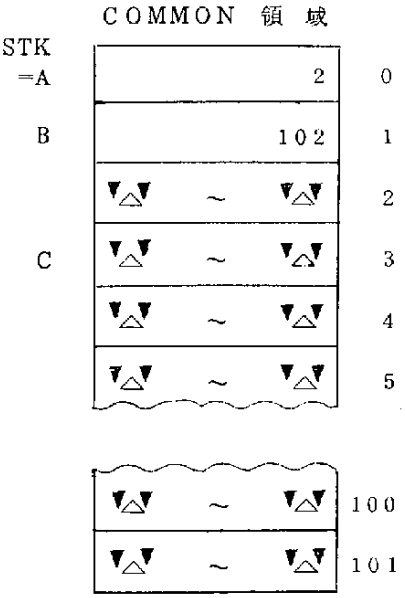
例6－12

LSDプログラム

```
DCL   STK(20)  STACK,
      *A,
      *B  CHAR(7),
      *C(2)  BIT  ;
```

OBJECT

```
STK  COMMON
STK  OCT    2 …(*1)
     OCT    146…(*2)
     EQU    A=STK+0
     EQU    B=A+1
     EQU    C=B+2
     RES    100, ▼△▼
```



（＊1）　内部ポインタ

（＊2）　テーブルサイズ

(3)　参照方法

6.2.4(3)と同じである。

スタックの項目あるいは項目の要素を参照したり，これに値を入れる

時には常に現在使われているスタックの一番上の項目を対象とするものである。

参照文献

参 照 文 献 一 覧

1. J.W. Alsop

   A canonic translator

   MAC-TR-46 1967

2. Brooker and Morris

   A general translation program for phrase structure languages

   J.ACM 9 Jan. 1962 PP.1～10

3. ——— ———

   Trees and Routines

   Comput .J. 5 1962 PP.33～47

4. ——— ———

   The Compiler - compiler

   Annual Review in Automatic Programming Vol. 3 1963 PP.229～275

5. Brooker, Morris, Rohl

   Compiler - compiler facilities in Atlas Autocade

   Comput. J. 9 1967 PP.350～352

6. ——— ——— ———

   Experience with the Compiler - compilar

   Comput. J. 9 1967 PP.345～349

7. C.S.Carr, et.al.

   The TREE META Compiler - compiler system : A META

   Compiler system for the Univac 1108 and the Genral Electric 645

   TR 4-12 (RADC-TR-69-83)

8. A.Colmerauer

   Total Precedence Relation

   J.ACM Vol .17 No.1 1970 PP.14～30

9. F.L.Deremer

   Practical Translators for LR(k) Laguages

   MAC - TR -65

10. ————

Simple LR(k) Grammars

C.ACM Vol.14 No.7 1971 PP.453～458

11. P.J Dixon , J. Sable

DM-1 a generalized data management system

STCC 1967 PP.185～198

12. J.J. Donovan , H.F. Ledgard

A fomal system for the specification of the syntax and translation of computer languages

FJCC 1967 PP.553～569

13. R.K.Dove

Design highlights of ABAC - a compiler-compiler

FJCC 1968 PP.1321～1328

14. J.A. Feldman

A Formal Semantic for Computor Languages and its Application in a Compiler-compiler

CACM Vol.19 No.1 1966.PP.3～9

15. J.A. Feldman , D. Gries

Translator writing systems

C.ACM Vol. 11 No.2 1968

16. E.N. Ferentzy , J.R. Gabura

A syntax directed processor writing system

FJCC 1968 PP.637～647

17. R.W. Floyd

Syntactic analysis & operator precedence

J.ACM Vol.10 Tan. 1963 PP.316～333

18. A descriptive language for symbol manipulation

J.ACM Vol.8 Oct. 1961 PP.579～584

19. 二村良彦他

PL/IW による TSS用コンパイラ・コンパイラの作成

第12回プログラミングーシンポジウム報告集 1971 PP.43～53

20. 藤原宏,渡辺勝正

Compiler 記述言語：COL

情報処理 Vol.9 No.4 1968 PP.187～196

21. ———— ————

COL で書かれたCompiler について

情報処理 Vol.10 No.6 1969 PP.375～383

22. J.D.Ichbiah , S.P.Morse

A Technique for Generating Almost Optimal Floyd - Evans Production for Precedence Grammars

C.ACM Vol.13 No.8 1970 .PP.501～508

23. 井 上 謙 蔵

順位言語の右順位解析

情報処理 Vol.11 No.4 1970 PP.231～233

24. ————

右順位文法

情報処理 Vol. 11 No. 8 1970 PP. 449～456

25. ————

コンパイラ・コンパイラ

産業図書

26. 井 上 他

Compiler－compiler における意味記述言語について

コンパイラ，コンパイラ シンポジウム報告集1971 PP 158～163

27. D.E.Knuth

On the translations of languages from left to right

Inf.Contr. Vol.8 Oct.1965 PP.607～639

28. 小 久 保 靖 世 , 佐 谷 鉄 夫

コンパイラ記述言語BPL

情報処理 Vol.11 No.6 1970 PP.342～349

29. W.M. Mckeeman , et.al.

A Compiler Generator

Prentice - Hall Inc.

30. ———— ————

The XPL Compiler Generator system

FJCC 1968 PP.617～635

31. W.M. Mckeeman

An Approach to Computor language design

Stanford Univ. Technical Report No. CS 48 1966

32. L. F. Mon dshein

VITAL Compiler - compiler System reference manual

Lincoln Laboratory Technical Note 1967 - 12

33. 日本電子工業振興協会

超高性能高子計算機のソフトウエアに関する調査報告書 昭和42年3月

34. J.T ONeil Jr.

META-PI An On-line interactive compiler - compiler

FJCC 1968 PP.201～218

35. P. J. Pankhurst

GULP-A compiler - compiler for verbal and graphic languages

Proc. ACM 1968 PP.405～421

36. M. Resnich , J. Sable

INSCAN: a system directed language processor

Proc. ACM 1968 PP.423～432

37. D. T. Ross

The AED approach to generalized computor - aided design

Proc. ACM 1967 PP.367～385

付　　録

Translator Writing Systems

Translator Writing Systems

Jerome Feldman,David Gries 共著
（カリフォルニア州，スタンフォード大学）

—概念と原理の研究—

この報告書は，プログラミング言語のトランスレータの作成（Translator Writing）を自動化することに関する最近の研究成果を批判的に検討しようとするものである。

第2章ではトランスレータ作成のためのシンタックスおよびその応用に関する形式についての研究を述べ，第3章ではTranslator Writingにおけるpost-syntax（すなわちセマンティックス）の自動化へのさまざまな試みについて述べることにする。さらにこれらに関連するトピックスについて第4章でとりあつかうことにする。

目　　次

I Introduction

「…なぜならば，そのこと総ては，私や他の人々が今まで簡単には理解することのできなかった長い詩の中に含まれているからだ」とショーの「人間と超人」の中で悪魔が語っている。

コンパイラ・ライティングということは，プログラミング部門で長い間注目を集めてきた分野であり，また長く言い伝えられてきた夢物語であった〔Knu 62, Ros 64b〕。

最近では，コンパイラ作成者の仕事のいろいろな部分を自動化する方法へと研究者の注意が向きつつあるのである。そこで，この報告書では これらの研究の成果を批判的に概説してみようと思う。この同じ研究に関して前に発行され，広く配布されている報告書であるStanford Computer Science Report CS69（1967年6月発行），およびこれに対して我々のもとに寄せられた沢山の行き届いたコメントが，この報告書を正確かつ概念上明快たらしめるのに計り知れない程の助けとなったことをここで述べておきたい。

個々のシステムについて述べる前にtranslator writing に関する一般的な問題について，二，三述べておきたい。我々が特にコンパイラに着目している理由はコンパイラがアセンブラやインタープリータに含まれる基本的な問題を含んでいるためである。compiler writingの研究には多大な努力がはらわれているのに，この問題に関して発表された書類は比較的少ないのである。そのためTranslator Writing System（以下TWSと略す）の設計者たちは，それまでに注意を払って記述されたことのない方法をひとつひとつ形式化して行かざるを得ないのである。さらに困難なことは，トランスレータの性能に関しては効率というような合言葉があるだけで一般に認められた評価基準のない事である。コンパイラの効率はトランスレータやオブジェクト・プログラムの実行に要する時間とスペースによって決まってくる。使いやすさ，エラーの検出と回復の能力，編集機能，そしてリコンパイル（recompile）の速さなどが効率に重要な影響を与えるのである。これらは互いに矛盾する点も含んでいるので，コンパイラの効率についての絶対的な尺度を期待することはできない。ここで考えているTWSの設計者たちは多様性に富んだ独自の評価規準を採用することにしている。

compiler writingというものは，多くの観点から成る大規模なプログラミング作業なのでコンパイラ作成者を助けるために多種多様な方法が提唱されてきているのも不思議はない。実際大きなプログラムを作る時に役立つシステムの特性（たとえば，トレースやエディット）は，すべてcompiler writingの道具となるものである。このことは，我々がcompiler writingに関していろいろなシステムの特性を調査する上で関係を持ってくるのである。術語として一般に認められているものがないので，ここでTranslator Writing System（TWS）という語

は，ここで考えているところのプログラムおよび提唱されているプログラムを表わすものと定義する。TWSで用いられているトランスレータという言葉は，インタープリータ，コンパイラ，インクリメンタル・コンパイラ，アセンブラのいずれであってもよい。

この報告書は　TWSの研究に関する歴史や紹介にはふれていない。この節の終りに記した参考文献の目録は　概説書としてどのような文献があるかを示すためのものである。また　ここでは個々のシステムやいろいろな方法の比較を行なってはいるが，この報告書は Translator Writing System の使用者のための案内書を目的とするものではない。この報告書の目的は将来の調査に備えて統一された科学的基礎を作るべく，現在行なわれている研究に注意深い考察を加える事にある。このような事を目的とするため　我々はある特定のプログラムの実行に役立つシステムの特徴よりは，むしろ種々のTWSの設計の知的内容に重きを置くものである。

TWSの商業用コンパイラへの応用は，最近ようやく普及してきた。過去三年間の研究の進展は　決して遅々としたものであるとは言えないが　はかばかしいものでもなかったので，そのため　総てのTWSの開発を軽視する向きもあるようである。またある特定のTWSが　特定のコンパイラの特性に依存するということもありうるわけであるが，これはTWSを使用する上において大きな障害にはならないと思われる。ここで注目したいのは，TWSの成功例がすべて企業に追従しない人々によってなされた研究にあらわれているという点である。さらに　重要なことは，TWSの研究に固有な柔軟な言語（flexible language）という概念が，企業が強調する規格化への動きとは全く反対の方向に進んでいくものであるという事である。従って　TWSが商業用のコンパイラの作成者に採用されはじめたとはいえ，これがユーザーの手に委ねられるようになるまでには幾多の小さな改変を経なければならないであろう。

不幸にしてTWSに関する文献を読もうとするものは，かなりな注意を払って　これに当たらねばならない。書物に形式的に表わされたシステムは，それを実際に使う段になるとかなり不備な点が目につくからである。また　シンボリックな記法を用いることによって，TWS関係の書類の有用性を高めようとする数学主義（mathematism）の流れもある。しかし　各研究者の間の連絡はほとんどなく，この分野に関する限り　ひとつの発見が重複して行なわれることが多く，他の研究者の行なった研究を参照する事が少ないように思われる。そして　このような状況と　評価するものの寛大さとがあいまって，他の点に関しては申し分のないTWSの研究の歴史に汚点を残しているのである。

このTWSに関する調査は，シンタックスとセマンティックスのふたつの項目に大別することができる。自動構文法に関する研究は，TWSに関する研究の中で最も古く　また最もよく理解されてきた部門である。この構文法は，さらに　II章Bで述べるシステムに使われている限定

された一般論と，II章Cに述べるような理論的には有力であるが　まだ実現の段階には到っていない方法とに分けられる。

一方　セマンティックスを三つの部分に分けているが，この分け方には　まだ議論の余地があるように思われる。先ず，III章Aの syntax－directed symbol processor は一般に最も良く研究されてきた問題の特殊ケースとして，ここに取り上げたトランスレータの研究の一翼をになっていると言えよう。またIII章Bのコンパイラ・コンパイラはプログラムの post-syntactic な処理の機械化に多大な助力を与えんとするものである。最後のIII章Cでは，従来からあるmacro－assembler をさらに推し進めて，TWSに拡張しようとする動きに関連した二つの企画を扱っている。

IV章で扱かっている関連するトピックスは，各章をふりかえって批評を加えようとするもので詳しい事は述べていない。すなわち，syntax－directed technique の他の利用法，および関係のある数理研究について書き添えておいたのは，TWSの研究とのつながりを明確にするためである。最後に　将来のTWSの開発に関係の深い，現在の実り多き研究活動にもふれておいた。なお　参考文献はアルファベット順に並べ，各章で参照した文献については　その章の終りに記したことを述べておく。

〔参考文献〕

REFERENCES FOR I

The *Communications of the ACM* and to a lesser extent the *Computer Journal* of the British Computer Society are the major journals for publications on translator writing. See especially *Comm. ACM* 4 (Jan. 61), 7 (Feb. 64), and 9 (Mar. 66).

Other general references : Che 64a, Flo 64b, Hals 62, Knu 62, Ran 64, Ros 64b, Weg 62, Wil 64b.

Formal descriptions of various programming languages: Bac 59, Ber 62, Brook 61, BroS 63, EvA 64, Gor 61, IBM 66, Naur 60, 63b, Rab 62, Samm 61, Shaw 63, Tay 61, Wir 66b, 66c.

Ⅱ　SYNTAX

A. Terminology

TWSに関する文献の内容を読む場合にわずらわしいことのひとつに　一定した言葉の定義（notation）に欠けている点がある。この報告書を　より読みやすくするために　いろいろな著述者たちの使用している記号や，時にはシンタックスも自由に変更するなどのことを行なった。シンタックスについて論ずる場合には，我々はGinsburgによって用いられた記法を採ることにした。〔Gin 66a，pp.8,9〕しかし　これとは矛盾しないようにしながら，構文的な超言語であるBackus－Naur　Form（BNF）の記法　特に記号 ::＝の使用などの記法は，随時併用して読みやすくなるように努めることにした。

この報告書では，多くの言葉を形式的な意味と非形式的な意味の両方に用いている。たとえばシンタックスに関するこのセクションでは　形式的な意味で用い，第Ⅲ章や第Ⅳ章ではそれ以外の意味に用いるというようにしている。

▼シンタックス▼や▼セマンティックス▼のような用語の形式的な定義に関しては，一般的には意見がまとまっていない。これらの定義については，セクションⅣでより詳しく話を進めるつもりである。非形式的な意味では，シンタックスとは構造を記述する機構を通常含んでいる　ある言語のよく整ったステートメント（well－formed　statement）を詳しく吟味することであり，セマンティックスとはこのstatement　をいかにして実際のあるいは理論上のコンピュータによって実行するかについて詳しく調べていくことであろう。

この報告書全体を通じて次のように定義する。すなわち　集合a^*はアルファベットaのsymbolから成るすべての有限記号列の集合であり，言語£はこのa^*の部分集合である。　どのような記号列が言語£の中にあるか（£のシンタックス）はsyntax metalanguage　で記述する。syntax　metalanguage　は手続きであり，そして£の記号列を作り出すためのアルゴリズム（synthetic　syntax），あるいは　a^*の任意の要素が£に含まれることを認識するアルゴリズム（analytic　syntax）を記述するものである。syntax　metalanguage　の中のstate－mentはふつうproductionと呼ばれる。

有効な（nontrivialな）analytic　syntaxを利用するプロセスはすべてsyntax－directedであると言われる。

アルファベットaに属するsymbolは端記号（terminal　symbol）と呼ばれ，第Ⅱ章ではこれらをT，T_1，T_2などというように表わすことにする。syntax　metalanguage　はアルファベットaに属さない非端記号（nonterminal　symbol）ヵの集合を含みうる。非端記号は

その言語を定義するために用いられる。この非端記号はALGOL report では常に括弧＜と＞に囲まれて登場し，同じ report の formal syntax rule 中に出ているものである。

この章ではシンタックスについて もっと形式的にとり扱いたいので，上記の括弧を省略し nonterminal をあらわすのには ラテン大文字のU，V，およびZを用いることにする。さらにシンタックスに関するこれらの章では，以下に述べるような，やや広義の術語を必要としている。

語彙Uは，terminal symbol の集合 a と nonterminal symbolの集合ηの和集合として定義する。記号S，S_1，S_2 等々はUの構成要素を示すのに用い，一方 記号の列（要素を1つももたない空な列Λを含む）は ラテン小文字のu，v，w，…を用いて表わすものとする。
Z＝xy がひとつの記号列である時 xはZの head であり，yはZの tail であるという。

Ui→ui の形の productionの有限集合が次のような性質を持つものとして定義できるとき，その文法Gを句構造文法（phrase structure grammar）と呼び，これによってある言語$\mathfrak{L}_G$を規定することにする。その性質とは，

(1) 各Uiが空でない任意の記号列であって，その記号列に含まれる記号が語彙Uに属している。
(2) 各Uiは任意の1個の nonterminal symbol であってUi∈ηである。
(3) distinguished symbol Zと呼ばれるUiが唯一個あり，これはいかなるuiの中にも現われない。

Ui，uiはそれぞれ production Ui→ui における left part，right part と呼ばれる。図1(a)は Z＝＜program＞というひとつの文法の例である。図1(b)は 同じ句構造文法に対して Backus Naur Form（BNF）の表記法を用いたものである。ここでは▼→▼のかわりに▼::＝▼を使用し，metasymbol ▼｜▼は 同じ left part に いくつかの異なるright part が対応する時 これらを分けるのに用いられるのである。

＜program＞	→	⊥ E ⊥	＜program＞	::＝	⊥ E ⊥
E	→	T	E	::＝	T ｜ E＋T
E	→	E＋T	T	::＝	P ｜ T＊P
T	→	P	P	::＝	I ｜ (E)
T	→	T＊P			
P	→	(E)			
P	→	I			

(a) formal な記法　　(b) BNFの記法

Nonterminal symbol : ＜program＞ E T P

Terminal symbol : I ( ) + * ⊥

図1　句構造文法の例

いくつかの研究をふりかえってみると　常にそうであるとは言えないが，right part が空であるようなproductionが認められていることがある。

句構造文法がどのようにして　ある言語を定義するかを示すためには，より詳細な定義を与える必要がある。w＝xUyで　かつ　v＝xuyであるような記号列x，yがある時　（x，yは空であってもよい），U→uというproductionの適用によってvはwのdirect derivationであるという（w⇒ vと書く）。

▼⇒▼の transitive closureは ▼$\overset{*}{\Rightarrow}$▼という記号で表わされ，$w = w_0$ ，$w_0 = w_1$ ，…，$w_{i-1} \Rightarrow w_i$ ，$w_i \Rightarrow v$という記号列w_0 ，w_1 ，…，w_i（$i \geq 0$）が存在する時，$w \overset{*}{\Rightarrow} v$となる。この時　vをwのderivativeと呼び，$w = w_0$ ，$w_0 \Rightarrow w_1$ ，…，$w_{i-1} = w_i$，$w_i = v$という連続を　wからのvのderivation　（導出）という。

distinguished symbol Zからのderivativeを Sentential formという。言語$\mathfrak{L}_G$は文（sentence）の集合として定義される。例えば　terminal symbol だけから成るsentential form の集合は，

$$\mathfrak{L}_G := \left\{ x \mid Z \overset{*}{\Rightarrow} x \quad \text{and} \quad x \in a^* \right\}$$

と表わされる。

図1の文法Gにおいて$\mathfrak{L}_G$は算術式の集合

（演算子として＋と＊，括弧として（と），operand としてIを使用）である。この算術式の始めと終りは記号⊥によって明示してある。

（図1の文法によれば）⊥P＋T＊P⊥というsentential form は次に示すような少なくとも2つのderivation を持っている。

＜program＞ ⇒ ⊥E⊥ ⇒⊥ E＋T⊥ ⇒⊥ T＋T⊥

⇒⊥ P＋T⊥ ⇒⊥ P＋T＊P⊥　　（2.1）

＜program＞ ⇒⊥ E⊥ ⇒⊥ E＋T⊥ ⇒⊥ E＋T＊P⊥

⇒⊥ T＋T＊P⊥ ⇒⊥ P＋T＊P⊥　　（2.2）

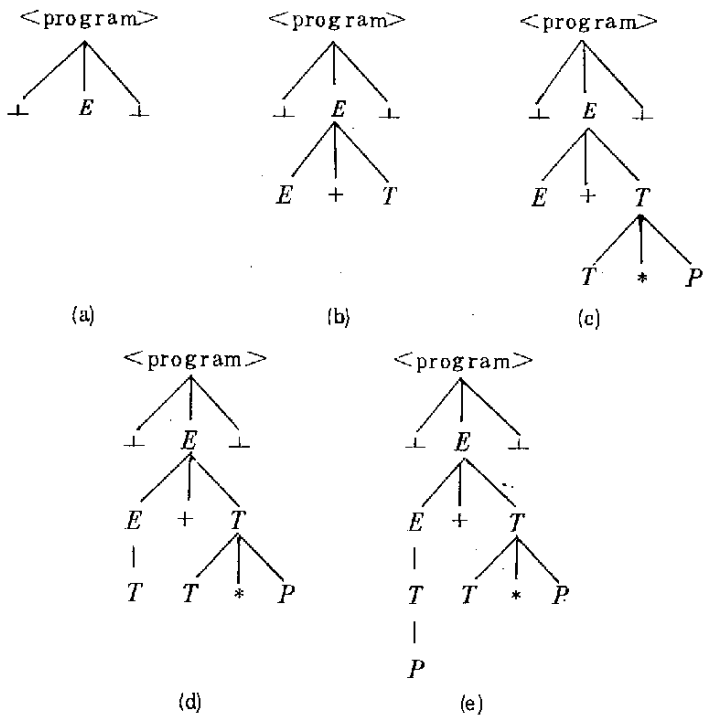


図 2　Syntax tree

derivationはsyntax treeによって図示することができる。例えば derivation (2.2) を書きあらわしてみると次のようになる。

記号＜program＞から始まって枝(branch)は図2(a)のように描かれる。枝とは node ＜program＞ から 下向きにのびている線の集りで，これらの線の終りの node (label) を含めたもののことである。連結されたこれらの node は，derivation において ＜program＞ と置き換えられた記号列 ⊥E⊥ を形成する。このようにして続けていく。すなわち production U ::＝ x (各 direct derivation) を適用することにより，置き換えられた node U より 枝を描く。この枝はその node が置き換えるべき記号列 x を形成しているものである。このことは図2が derivation (2.2) に対する一連の syntax tree を図示するものであり，特に図2(e)が derivationを完了した形を表わすものであることを示している。ここで注意すべきことは，連結されている時に treeの end node (下方へ枝の出ていないもの) が 最終的な sentential form を作るということである。

⊥P＋T＊P⊥ はふたつの derivation を持ち，ふたつとも同じ syntax tree を持っている。このとき，derivation は production が direct derivation として適用される順序

についてのみ　異なっているのである。文法Gは，$\mathfrak{L}_G$ の文の中で　Gに関して2つ以上のsyntax　treeを持つようなものがひとつでも存在する場合に▼あいまいである▼（ambiguous）と言われる。

任意の総合的な句構造文法が与えられる時，derivation によって　その言語の文を自由に作り出すこともでき，また　distinguished　symbol　Zからsyntax　treeを作ることもできる。ところがコンパイラは，これとは反対の問題をかかえている。それは　ある文xと文法Gとを与えられて　xのderivation を作り上げ，対応するsyntax　tree を見い出すということである。このことは　文を解析する（parsing）とか，認識する（recognizing ―この言語はrecog―nizer に関連して出てくる）とか，分析する（analyzing）と呼ばれる。

parsing の方法には top－down と bottom-up と呼ばれる二つの異なる方法があり，このふたつは時として混同されることがある。その理由は，ひとつには　人によってsyntax　treeの書き方が異なるからである。例えば図2(b)の　syntax　tree　（これが我々の表記法であるが）は，図3のようにも書けるのである。そして　いまひとつの理由は　前記のふたつの方法が，recognizer が複雑になるにつれて，実際にひとつのものになってきたからである。このことについては，この節のこのあとの部分と　第Ⅱ章C 5においてふれることにする。これらのふたつの方法のどちらも共に　left - right と呼ばれる。というのは　文の中で記号を処理する順序が，可能なかぎり左から右へと進行するからである。

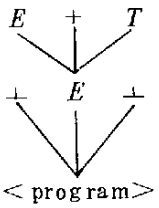


図3　Upside-doun syntax　tree

純粋な top - down recognizer は，完全に goal - oriented である（goal に依存している）。main goal は　もちろん　distinguished　symbol　Zであり，predictionは認識される記号列が実際にある文になるように作られている。したがって　先ず第一に，記号列をある　production　$Z \rightarrow S_1 S_2 \cdots Sn$ の　right part　の$S_1 S_2 \cdots Sn$ に還元（reduce）することができるか否かを確めなければならない。

このことは次のようにして行なう。すなわち　S_1 が terminal symbolであれば，その production の適用が有効なものであるためには　その記号列は terminal　S_1 から始まらな

くてはならない。S_1 がnonterminal である場合は，subgoal を設け，これをテストして，その記号列の head が S_1 に還元できるかどうか調べる。そして これが正しいことがわかると，S_2 をテストし，同じ方法で S_3 以降を調べていく。ある Si に適合するものが見つからない場合には，別の production $Z \rightarrow S_1' S_2' \cdots Sm'$ を適用して試してみる。

同じようにして subgoal Uもテストする。すなわち production $U \rightarrow S_1 S_2 \cdots Sn$ が適用できるか否かをテストするのである。このように，新しい subgoal が適当でない場合は，失敗であったことを もうひとつ高いレベルに報告し，この高い方のレベルで別のものを試みるようにする。

left - right top - down recognizer においては時々左回帰（ left recursion ）が生じて困まることがある。すなわち $U_1 ::= U_1$ x という形の production は無限のループをひき起すことがありうるのである。この理由は，U_1 が新しい subgoal になった場合，次のステップで新しい subgoal として再び U_1 を作り出すことである。しかし この左回帰の問題は，時には 文法を変えることにより，また時には（次のように）recognizer を修正したりすることによって解決されることもある。同一の left part に対応するいくつかの異なる right port のテストの順序がここでは重要な役割りを果すことになる。左回帰を含む right part がひとつしかない場合には，テストの際にこれを一番最後にすればよい。しかし こうすると 他の順序の問題，例えば right part の短いものを一番さいごにテストするというような問題に矛盾することもありうるのである。

top - down recognizer という名前は syntax tree が構成される方法に由来している。すなわち 解析のあらゆる点においても（おそらくは正しくないものもあるであろうが），一番上の node から出て その記号列に至るというように tree を作り上げていくのである（図4(a)）。このような recognizer は各段階で作られる関係を予知（ predict ）しようとするので，しばしば predictive と呼ばれる 〔Kun 62 を参照〕。

top - down recognizer は，回帰的なサブルーチンや1個の stack を用いて動く単一のルーチン，その他 多くの異なる方法でプログラムすることができる。ここで重要な特徴は，これらが goal - oriented であるという事である。

反対に 純粋な bottom - up recognizer は（もちろん，implicit な goal Z は除くが）基本的にはいかなる長期的見通しにたった goal も持っていない。記号列は production の right part である部分記号列（ substring ）としてさがされるのである。

次に これらの記号列は，それに対応する左側のものと置き換えられるのである。図4(b)はこのことを示すものである。ここで 後の節で必要な術語を導入するため，少々詳しく述べよう

と思う。

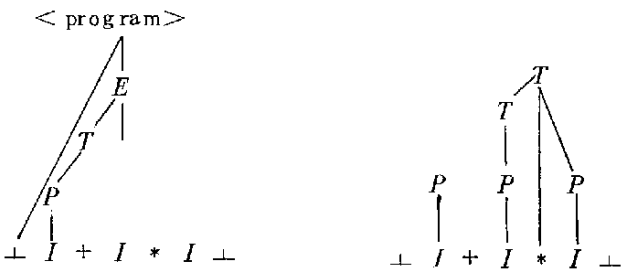


図4　(a) top - down 解析の一部　　(b) fottom-up 解析の一部

syntax tree は 我々がある文の解析を始めようとするときには まだないのであるが，bottom - up 解析の背後には それが存在するという概念が在ることは明らかである。したがって，sentential form S とそのsyntax tree があると仮定する。S の phrase とは，その syntax tree のある subtree の end node の集合であると定義する。また ある syntax tree に関係のある S の handle（把手）とは，それ自身の他には phrase を含まないような最左端の phrase であるとする。例えば，図2(c)の syntax tree には P，T＊P，P＋T＊P，⊥P＋T＊P⊥の4つの phrase がある。この中で P と T＊P のふたつは他のいかなる phrase も含んでいない。hanale とは，P のような最左端の phrase を指すのである。

次のアルゴリズムは，left - right bottom - up 解析の背景にある一般的な規則を指すものである。sentential form $S = S_0$（およびそれに対する syntax tree）から始めて，$S_n = Z$ が作り出されるまで i＝0，1，…，n に対して次の各ステップをくりかえす。

1. （Si についての syntax tree を見ることによって）Si の handle を見つける。
2. 対応する枝に名前をつけ，sentential form S_{i+1} を生み出すような node の label で Si の handle を置きかえる。
3. handle をそれから取り去ることによって tree を刈り込む。

すると $Z = S_n \Rightarrow S_{n-1} \Rightarrow \cdots \Rightarrow S_1 \Rightarrow S_0$ という一連のつながりは，S_0 の derivation である。例えば sentential form $S_0 = \perp P + T * P \perp$ と図2(c)のその syntax tree が与えられる時，handle P を T に置き換え，$S_1 = \perp T + T * P \perp$ とそのsyntax tree 図2(d)ができる。図2(d)，2(c)，2(b)，2(a)における syntax tree に対する handle は，それぞれ T，T＊P，E＋T，⊥E⊥ である。図2(c) の syntax tree は2(d)の handle を刈りこむことによって生じ，2(b)は2(c)の handle を刈りこむことにより，そして2(a)は同じ

ようにして2(b)から生じる。この連続は，derivation (2.2) を生み出すのである。

純粋の top - down recognizer と同じように 純粋の bottom - up recognizer は，通常正しくない結果を返すreduction(あるいは結合) をもたらすことがある。これは次のふたつの異なる方法のいずれかで操作しうる。第1の方法は別の可能性のあるものを試しうる点まで逆もどりすることである。(backup, backtracking) これは その記号列の一部を前の形に戻したり，作られていた結合のいくつかを消去することを含むものである。第2の方法は すべての可能な解析列(parse) を並行して作るものである。それらのいくつかが" dead end "(もう reduction や結合が不可能になること) に陥いると，それらを捨て去るのである。

COGENT (III章A．3) は top - down の方法の中でこのやり方を用いている。(〔 Kun 62 〕を参照のこと)

まちがった reduction の可能性を減ずるために，より複雑な recognizer が開発されてきた。例えば ある修正を施した recognizer では，新しい subgoal から出発する前に ある sub－goal のいくつかの derivative が，実際に今問題となっている部分記号列の最初の記号から出発しうるかどうか(look ahead)，あるいは調べようとしている subgoal がずっと先まで形成された tree の一部にできているものかどうか(memory) をみるために，前以って作られたテーブルに目を通すことをする。

修正した top - down recognizer の例は 〔 Ir 61, May 61, War 64 〕にある。第III章Aの syntax - directed symbol processor の多くは，修正した top - down recogni - zer を用いている。

同様に 修正した bottom - up recognizer は，決定の際に助けとなる可能性のある handle のまわりの文脈を見る。実際 これらの recognizer は，文法にある種の限定をほどこしているので逆戻りや並行に進む方法(parallelism) が不用であるが，そのかわり 大変に複雑なものである。

このような修正は，これらふたつの概念の連合 (混同) に役立つものである。ある recognizer については，bottom - up とか top- down とかの区別をすることが大変にむずかしいことがある。例えば Earley のアルゴリズム (第II章B．5) によって作られた production language は，両方の性質を持っている。ある recognizer が出会うはずの explicit な goal や subgoal を持っている時には，これを (修正した) top - down であると言いたい。その理由は goal - oriented であるからである。この問題について詳しくは第III章C.5を参照のこと。

残された術語の多くはコンピュータ・サイエンスの一般的な知識を有する人々には耳新しいものではないだろう。定義の必要な二，三の data structure の用語を用いるので 次に示しておく。

リスト構造（ list structure ）システムという語は，ポインター（ links ）とダイナミック・アロケーションを使用する いくつかのプログラミング・システムを述べるのに用いる。任意の二つの node の間に 1つしか道（ path ）のないリスト構造が tree である。一般的な結合が explicit に許されていて，各々の要素がいくつかの（ 大ていは 2 つの ） link を含む storage のあつまりとなっているようなリスト構造は plex とよばれる。我々は逐次的な情報をとりあつかうための一般的規則としてＬＩＦＯ(last - in - first - out) とＦＩＦＯ(first - in - first - out) という言葉を用いる。

〔 参 考 文 献 〕

REFERENCES FOR II. A

Che 64c, Chom 63, Flo 64b, Gin 66a, Ir 61, Kun 62, Naur 60, War 64.

B. Automatically Constructed Recognizers

この節では　文法に定められた言語の文を解析(paring)，あるいは　認識(recognizing)するためのいくつかの実用的な方法を述べ　評価する。"実用的な"方法とは，コンパイラを作成するのに今まで用いられてきた，あるいは現在用いられているもののことである。このようなrecognizerは，基本的なルーチンの集合によって使われるテーブルの形　あるいは　ある言語によるプログラムの形であろう。ここで述べられているrecognizerは，任意の適当な文法からそれを作り出すためにconstructorと呼ばれる関連するアルゴリズムを持っている。結局，これらはすべて1個のＬＩＦＯスタッフを持って働き，逆戻りの機能を持たないleft - right recognizerである。

automatic　generation という特徴はコンパイラ作成者にとって大変に重要なものである。多くのconstructorでは，実際にrecognizer を作成する前に　その文法にあいまいさがあるかを調べる(これは，はっきりした利点である)。コンパイラの大部分の automatic　construction はまた　大した仕事量でなくなり，code の optimizationのようなことを考えるのに十分な時間ができるのである。さらに　automatic　constructionはそのrecognizer が形式的なシンタックスを追いかけることを保証するものである。

不幸にして　これらのrecognizerとそれらのconstructorとは，すべての問題を解決できるというわけにはいかない。第1に　現在存在しているシンタックスの形式的概念は，多くのプログミング言語のシンタックスを完全に記述する習慣がない。第2に　セマンティックスというもののプログラミング言語において果す役割がより大きく　より複雑である。― セマンティックスに正しく適合するようにその文法や，生成されたrecognizer を変更しなければならないことがよくあるのである。第3に　ある方法が理論的に正しいと思われても，その使用に必要な制限がそのプログラミング言語の慣習的文法に本質的な変更を要するかもしれないのである。

我々は，ついでに数種のrecognizerの"効率"についてGriffithsとPetrick が比較を行なったことを記しておく。〔 Grif 65 〕 この比較は　理論的な面白さはあるが，各recognizerに対応する Turing machine　の効率を主としているので，コンパイラの作成においては実際的な価値を持っていない。我々は　次のような点について　recognizerを比較したことには興味を抱いた。

(a)　recognizerの使用するスペースはどの位か。

(b)　その　recognizerの速さはどの位か

(c)　その　constructor　に受け入れられるために普通使っている文法をどれ程変更しなければならないか。

(d) 一旦 recognizer が作られたあとで，その文法によって表現されていなかったセマンティックスやシンタックスの特性をどのようにしてそれに追加できるか。

読者は 我々の比較が形式的に適用されるときの，これらの recognizer をもとにしていることおよびこれらが一般的な観察であるということに注意されたい。例えば bit - pushing や工夫をほどこした速いリストのサーチの方法で，特定 implementation の効率をいちじるしく上げることができるのである。上記の特徴はまた 問題の言語を作り出すコンパイラのタイプなどにも依存している。特に問題(d)に対する答えは ソース・プログラムの内部的な形 またはマシーン・コード自体が生成されるかどうかに依っているし，そのコンパイラで有効なセマンティックな処理の能力にも依存している。

逆戻りをゆるす bottom-up reognizer を作ることは可能であるにもかかわらず，ほとんど行なわれていない。その文法のあいまいでないことを保証し，任意の sentential form について一意的な handle を効果的にさがし出し，reduction を行なうことのできることを保証するために，通常は 制限を設ける。

一方 今日使われている top - down recognizer の多くは，逆戻りを許しているか あるいは 可能な解析をすべて並行して行なっているかである。唯ひとつの制限は左回帰をゆるさないことである。（第II章Aを参照） 現在存在する top - down recognizer はそのために より広いクラスの文法を受けとることができるが，余り効率の良くない傾向がある。文法が不完全な場合には，逆戻りを許すことは大変に効率のよくない recognizer を作ることになる。

純粋な top - down recognizer については 第II章Aで簡単に述べたので，ここでは論じないことにする。修正した top - down recognizer を用いるコンパイラについて詳しくは〔War 61〕を参照されたい。さらに〔Che 64c〕はコンパイリングにおける top - down recognizer の使用に関する指導的な論文であり，〔Flo 64b〕もこの方法についての十分な内容を含んでいるものである。

ここに記述する recognizer のあるものは 多くのコンパイラの中で多くの人々によって用いられているものであるが，ここではこれらのすべてを参照できるように列挙することはできない。各 recognizer に対して，その recognizer と constructor が述べられている論文名を挙げておくことにする。ここで述べられているシステムの形式的性質と同様に機械的に組み立てられうる理論的に面白い recognizer については第II章C節で簡単に述べることにする。

第II章Aの図1の文法を例として この節全部を通して用いることにするので，読者は第II章A節の定義を簡単に見直しておかれる方がよいかもしれない。

B. 1. Operator Precedence (Floyd (Flo 63))

この文法は，最初 演算子文法に限定する。すなわち 任意の記号列xおよびy， nonterminal U_1， U_2 に対して，$U \rightarrow x\,U_1\,U_2\,y$ の形となる production があってはならない。このことは，2つの隣りあった nonterminal を含むような sentential form はないということを意味している。これは決して重大な制限ではない。なぜなら 多くのプログラミング言語の文法はすでにこの形になっているし，この形でないプログラミング言語の文法の大部分のものをその言語の構造を本質的に混乱させることなく 演算子文法に変えることができるからである。

ある演算子文法があるとして，Sがその sentential form であるとする。Sの prime phrase とは，自分自身以外の phrase を含まず 少なくとも1つの terminal T を含むような phrase であると定義する。(これを第II章Aの phrase と handle の定義と比較してみること)。例えば 図5(a)において phrase は P， T＊P， P＋T＊P， ⊥P＋T＊P⊥であり，prime phrase は T＊Pである。同様に図5(b)において prime phrase は P＋Tであり，図5(c)では ⊥E⊥ である。ここで作られる recognizer は，各ステップにおいて handle ではなく，最左端の prime phrase を還元 (reduce) する。しかし これも基本的には左から右へと進むので bottom－up， left－right recognizer と呼ぶことにする。

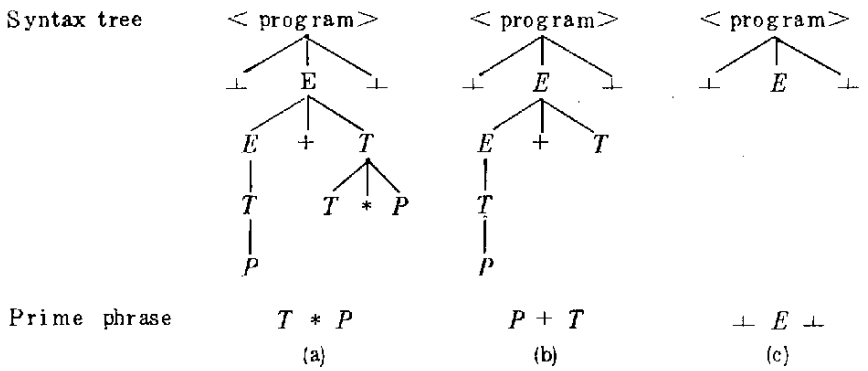


図5 Operator Precedence を用いた解析

同様に xが少なくともひとつの terminal を含み かつ $U \rightarrow x$ という production が存在するか，あるいは $x' \overset{*}{\Rightarrow} x$ であって derivation $x' \overset{*}{\Rightarrow} x$ において適用される唯ひとつの production の形が $U_i \rightarrow U_j$ である時 $U \rightarrow x'$ という production が存在するかのいずれかである場合，またこの場合にかぎり xは少なくともひとつの sentential form S から成る prime phrase である。

文 T_1 …Tmの解析の間中，ひとつのＬＩＦＯスタックが部分的に還元した記号列 S_0 S_1 … $SiTjTj+1$…Tmの記号 S_0 S_1 …Siを含んでいる。各ステップにおいて，(1)Siがそのスタック中の最左端の prime phrase の tail symbol であるかどうか，あるいは(2)Siが tail ではなく Tjがスタック中に push されなければならないかどうかを記号 Si－1，Si，Tjを見て答えることができる必要がある。

このことを行なうために，演算子文法の terminal symbol T_1 と T_2 の間に次の３つの関係を定義する。

1. $U \rightarrow x\,T_1\,T_2\,y$ あるいは，$U \rightarrow x\,T_1\,U_1\,T_2\,y$ の production が存在する時 $T_1 \doteq T_2$ である。ここで U_1 は nonterminal である。
2. production $U \rightarrow x\,U_1\,T_2\,y$ があり かつ あるzと U_2 に対して derivation $U_1 \Rightarrow zT_1$ あるいは $U_1 \overset{*}{\Rightarrow} zT_1\,U_2$ が存在する時 $T_1 \lessdot T_2$ である。
3. production $U \rightarrow x\,T_1\,U_1\,y$ が存在し かつ あるzと U_2 に対して devivation $U_1 \Rightarrow T_2\,z$ または $U_1 \overset{*}{\Rightarrow} U_2\,T_2\,z$ が存在する時 $T_1 \gtrdot T_2$ である。

terminal symbol T_1，T_2 の任意の順序対（ordered pair）の間にたかだか１個の関係（relation）が保たれている時，その文法を演算子順位（operator precedence）文法とよび，その言語を演算子順位言語という。

演算子順位言語においては これらの一意的な関係は，還元され得る部分記号列（すなわち prime phrase）を見出すために用いられる。$T_0\,x\,T$ が sentential form $S = x_1\,T_0\,x\,Tx_2$ の部分記号列であり，この部分記号列x内の terminal symbolは 順番に T_1，T_2，…，Tn（n≥1）であるとする。また T_0，T_1，…，Tn および Tの間に次の関係があると仮定する：

$$T_0 \lessdot T_1 \doteq T_2 \doteq \cdots \doteq Tn \gtrdot T$$

（xの nonterminals は，ここでは何の役割も果していないことに注意すること）。

この時xは prime phrase である。さらに あるUに対するxの reduction は，常に sentential form $x_1\,T_0\,U\,T\,x_2$ を生み出すように実行されるのである。

文（またはプログラム）の解析は，大変素直なものである。（図６を参照） terminal スタックの一番上にある記号Tと 次に入ってくる記号Tとの間に Tn ≫ Tの関係が存在するまで，記号をそのスタックに push down する。その記号列が確かにその言語の文であれば，スタックの一番上の要素は 前述のように記号列 T_0 xを持っている。この関係を用いて，T_0 とxのはじまりを見つけるためにスタックを逆向きにサーチする。そこでxは prime phrase でありスタック内に T_0 Uを作り出しながら，あるUに還元される。T_0 とTとを比較することによっ

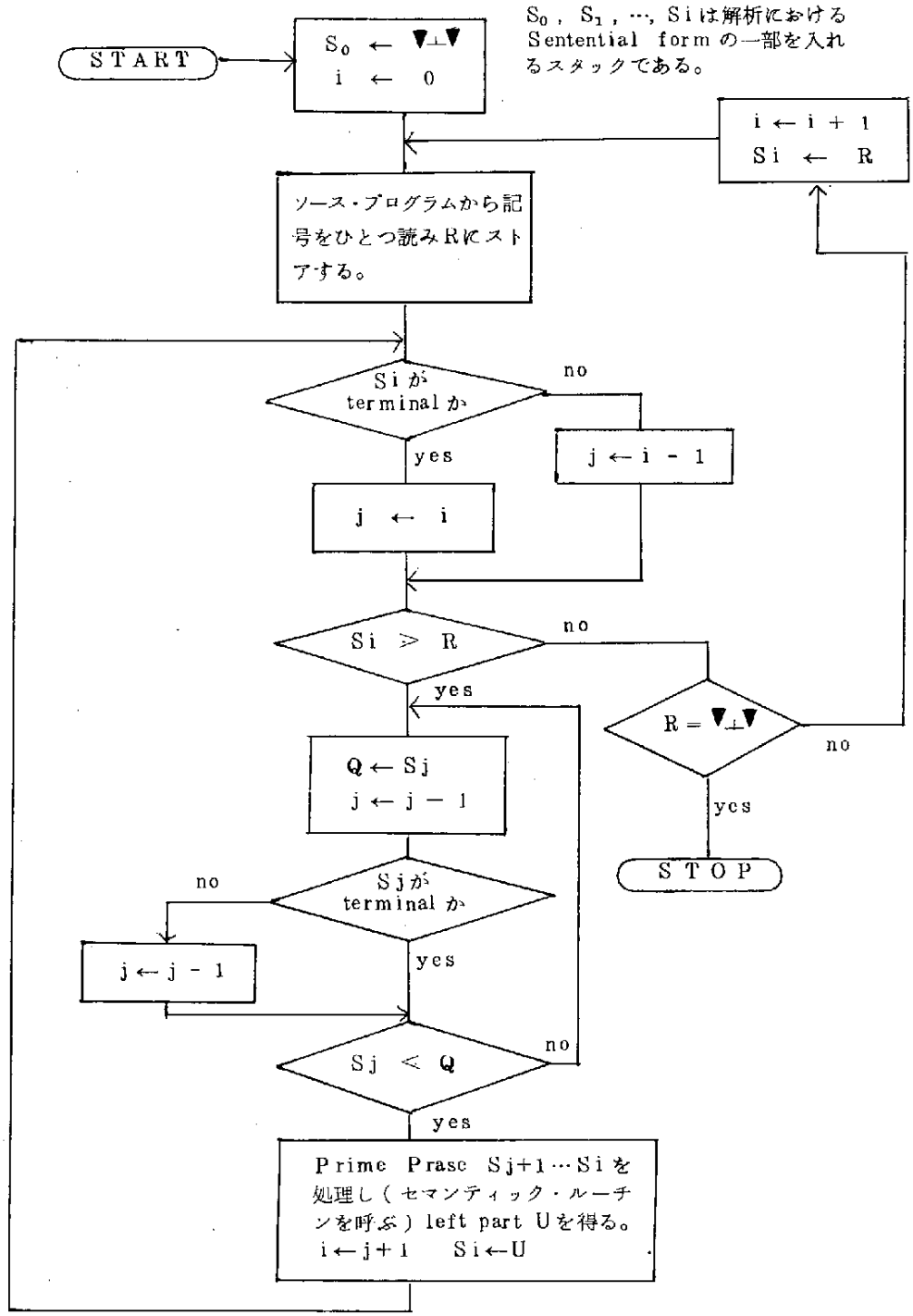


図6　演算子順位を用いた recognizer

てこの過程をくりかえす。

例を挙げれば，sentential form ⊥P＋T＊P⊥は，図5に示されるように（図1の文法を用いて）解析される。ここでは 各treeは，その直前の syntax treeの prime phraseを刈りこむことによって導出される。

表　I

T_1 \ T_2	(	I	*	+	⊥	)
)			⋗	⋗	⋗	⋗
I			⋗	⋗	⋗	⋗
*	⋖	⋖	⋗	⋗	⋗	⋗
+	⋖	⋖	⋖	⋗	⋗	⋗
(	⋖	⋖	⋖	⋖		≐
⊥	⋖	⋖	⋖	⋖	≐	

T	$f(T)$	$g(T)$
)	5	1
I	5	6
*	5	4
+	3	2
(	1	6
⊥	1	1

いま lをこの文法の terminal symbol の個数とすると，⋗, ≐, ⋖ からなる関係は l × l の行列の中に置くことができる。（〔Fl$_0$ 63〕によればＡＬＧＯＬ式の言語に対するこの行列は，35 × 35 ぐらいである。）行は スタックの一番上にある terminal symbol に対応し，列は入ってくる記号に対応するものと定義する。すると 行列の要素の間の関係をテストすれば，前述の比較を行なうことになる。

もし，$T_1 \lessdot T_2$ が $f(T_1) < g(T_2)$，$T_1 \doteq T_2$ が $f(T_1) = g(T_2)$，$T_1 \gtrdot T_2$ が $f(T_1) > g(T_2)$ を含むような二つの integer precedence function $f(T)$ と $g(T)$ が見つかれば，この関係に必要なスペースは長さ l のふたつのベクトルにまで減らすことができる。

Floydは 順位関係（precedence relation）の行列を見出すためのアルゴリズム およびそれらが存在するならば，そしてそれらが存在するときに限り，fとgを見出すアルゴリズムの概略を述べている。図1の言語に対する順位行列と関数とは表1のように作られる。

関数fとgを用いる時には，よいエラーの復旧法を図示することは かなりむずかしい。というのは，エラーは可能性のある prime phraseがひとつではないという場合にのみ 検出されるからである。したがって この関数はこれに先立つ passで完全なシンタックス・チェックを行なったとき はじめて用いることができる。（あるコンパイラでは，実際にプログラムを2回解析（parse）する。第1回の解析では 完全なシンタックスのチェックを行ない，また変数やブロックなどについての global な情報を集めることもできるようにする。第2回の解析では

効率のよい演算子順位の方法 — すなわち 関数 — を用い，第1の解析で集めた情報を code generation のために用いる。しかし コードがより簡単に generate でき，第2の解析を不要とするために syntax - checker にソース・プログラムの変化した形（逆ポーランドや tripl など）を作り出させる方法を採る傾向がある。）

この方法についてのひとつの障害は，この言語が まだ あいまいな文を含みうることである。解析樹（ parse tree ）の構造（ structure ）は，その文法が演算子順位であるかぎりあいまいではないが，node の名前はあいまいでないというわけにはいかない。production の right part が一意的であるという制限がないので，ひとつの prime phrase x に対して それを還元できうる二つ以上の nonterminal が存在しうるのである。この障害は nonterminal が通常 セマンティック・ルーチンによってとりあつかわれ，シンタックスでは それ程 扱かわれないということによって大部分解決できる。すなわち シンタックスは構造を定義するものであり，ある node が例えば< integer expression >と名付けられているか < real expression > といわれているかは セマンティックスの話である。

セマンティック・ルーチンは prime phrase が認識され，それが還元されうる場合に呼び出される。異なった prime phrase の各々を処理するために，別々のルーチンが書かれるのである。このことはもちろん 実行されるべきセマンティックな処理に依存して，時には文法の変更を必要とすることもある。例えば production < cond > → if < be > then < expr > else < expr > は，test や jump がセマンティック・ルーチンによって 正しい位置に挿入できるように次のごとく変更する：

< ifcl > → if < be >

< if − then > → < ifcl > then < expr >

< cond > → < if − then > else < expr >

しかし 変形された文法は，多分 その言語のもともとの参照される文法とは本質的に異なってはいない。（例えば 〔 Flo 63 〕の Floyd の言語を見てみること。）

我々の知るかぎりでは，どんなコンパイラも機械的に組み立てられたこの種の recognizer を持ってはいない。けれども 順位（ precedence ）の方法自体は，ALGOL，MAD，FORTRAN のコンパイラの中で 度々用いられている。この方法はわかりやすく，柔軟で大変に効率のよいものである。

B．2. Precedence （Wirth と Weber 〔 Wir 66c 〕）

Wirth と Weber とは，Floyd の順位の概念に変更を加えた。その文法は 必らずしも演算

子文法に制限することなく，その関係⋖，≐，⋗が　任意の記号S_1，S_2 のすべての対の間に成り立つ。

1. production $U \rightarrow x\,S_1\,S_2\,y$ がある時，$S_1 \doteq S_2$ である。
2. production $U \rightarrow x\,U_1\,S_2\,y$（または　$U \rightarrow x\,U_1\,U_2\,y$）が存在し，ある z，w に対する derivation $U_1 \overset{*}{\Rightarrow} zS_1$（そして $U_2 \overset{*}{\Rightarrow} S_2\,w$）がある時，$S_1 \gtrdot S_2$ である。
3. production $U \rightarrow x\,S_1\,U_1\,y$ とある z に対する derivation $U_1 \overset{*}{\Rightarrow} S_2\,z$ の時，$S_1 \lessdot S_2$ である。

記号S_1，S_2 の任意の対の間にたかだかひとつの関係が成り立つとき，そしていかなる production も同一の right part を持たない場合に，この文法を順位文法（precedence grammar）とよび　その言語を順位言語（precedence language）という。順位言語の文は，一意的な syntax tree を持つ。解析の途中で　スタックの一番上の記号 Si と入ってくる記号 T の間に　⋖か≐かの関係が成り立つ時には，Tをスタックに push down する。Si ⋗ T の場合には，

$$Sj\text{-}1 \lessdot Sj \doteq \cdots \doteq Si\text{-}1 \doteq Si$$

の状態をさがして，下向きにスタックをサーチする。この時 handle は Sj…Si であり，一意的な production U ::＝ Sj … Si の left part Uによっておきかえられるのである。このやり方と Floyd のものとの主な違いは，この方法では　その関係（relation）は terminal symbol 間に成り立つだけでなく　任意の２つの記号の間に成り立ちうるという点である。それゆえに，prime phrase ではなく　handle が還元されるのである。順位に関する行列と関数 f，g を作り出すアルゴリズムは Floyd の場合とよく似ている。これについては〔Wir 66c〕を参照されたい。

表　II

S_2 / S_1	E'	E	T	T	P	(	I	*	+	)	⊥
E'										≐	≐
E									≐	⋗	⋗
T'									⋗	⋗	⋗
T								≐	⋗	⋗	⋗
P								⋗	⋗	⋗	⋗
)								⋗	⋗	⋗	⋗
I								⋗	⋗	⋗	⋗
*							⋖	≐	⋖	⋖	
+						⋖	⋖	⋖	⋖	⋖	
(						≐	⋖	⋖	⋖	⋖	⋖
⊥						⋖	⋖	⋖	⋖	⋖	⋖

S	$f(S)$	$g(S)$
E'	1	1
E	2	2
T'	3	2
T	3	3
P	4	3
)	4	1
I	4	4
*	3	3
+	2	2
(	1	4
⊥	1	1

図1の文法については，$+\ominus T$，$+\lessdot T$，$\perp \ominus E$，$\perp \lessdot E$，$(\ominus E$，$(\lessdot E$ の関係が成り立っている。このような矛盾は，文法を次のように変えることによって処置することができる。

$$
\begin{aligned}
\langle \text{program} \rangle &\rightarrow \perp E \perp \\
E' &\rightarrow E \\
E &\rightarrow T' \\
E &\rightarrow E + T' \\
T' &\rightarrow T \\
T &\rightarrow P \\
T &\rightarrow T * P \\
P &\rightarrow (E') \\
P &\rightarrow 1
\end{aligned}
$$

この文法に対する順位行列と関数を表IIに挙げておく。実際に　任意の句構造文法をその句構造（ phrase　structure ）を曲解することなく，任意のふたつの記号の間に　たかだかひとつの順位関係が存在するように修正することができる。production の右側が一意的でない時には，あいまいさが目だつのである〔 Michael　Fisher，ハーバード大学〕。　右側が複数個あるという問題は，その文法があいまいでない時でさえ　このやや実際的でない問題をひきおこすのである。

Floyd の recognizer の場合と同じく，この場合も順位行列と関数 f, g　のいずれをも用いることができる。ここで使う行列は　任意の2つの記号間に関係が成立するので，Floyd の場合よりも　ずっと多いのである　（ALGOL に対して 70 × 70 ）。ここでもhandleが見つかった場合だけ，セマンティック・ルーチンが呼び出される。

理論的には，この方法は非常に安全で　効率のよいものである。その関係が任意の2つの記号間に成り立ちうるということからFloydの場合よりも信頼性が高いように思われる：すなわち順位文法においては，各文に対して一意的な canonical　parse （正準解析列）が存在することがわかっているからである。しかしながら　実際問題としては，一意的な right part に関する制限がまだ残っている。二つ以上のやり方で還元されうる1個のhandleに対する各セマンティック・プロセッサは，文脈とグローバルな情報から　xを置きかえる正しいnonterminalを決定しなければならない。このことは次のような production が必要なことを示している。

＜array identifier＞ → ＜identifier＞

＜procedure identifier＞ → ＜identifier＞

さらに　ある文法が順位文法である以前に，それが平均的なプログラミング言語であるとして巧みにとり扱かわねばならない。その理由は，順位関係を決定するのに　十分な文脈を用いられないことである：すなわち　二つの記号の間に二つ以上の関係が存在することが多々あるからである。したがって　（上の例のように）中間的なproductionを挿入するとか，時には　その文脈に依存する異なった記号（例えばカンマ）を用いたりすることが必要である。後の場合には，prescannerをその文脈に注意するように変更し，どのような内部記号（internalsymbol）を各カンマに対して用いたらよいかを決めなければならない。最終的な文法は，その言語に対する参照（reference）として　プログラマに示すことができなかった。

B．3. Extended Precedence （Mckeeman〔Mckee 66〕）

Mckeemanは順位行列を2つのテーブル — すなわち　ひとつはhandleのtailをさがすためのものと，もうひとつは　handleのheadを見つけるためのもの — に分け，次に　順位の矛盾がほとんど起らないようにするために　もっとよく文脈を調べるrecognizerを作ることによってWirthの概念を拡張した。したがって　このconstructorは，より広いクラスの文法を受けとることができるのである。

(a) スタックの一番上から2個の記号Si－1，Siおよび次に入ってくる記号をTとし，Tをスタック内に置くべきかどうかを決定するため，そしてSiがhandleの tailであってreductionを始めるべきかどうかをきめるために用いる。

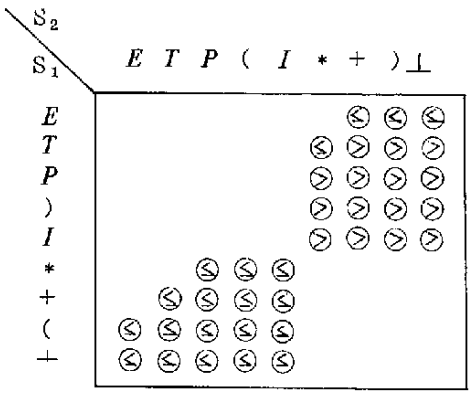
(b) 同様に　handleの最初の記号を見つけるために　スタック内に戻るときにも，2つではなく　3つの記号が用いられる。

この方法は　Eickel et al.が提唱した方法〔Ei　63〕と比較すべきである。第II章C.1を参照のこと。実際には　これは異なるtripleの個数が多すぎ（PL1のひとつのdialect（方言）に対して10000以上），また多くの場合 すべきことを一意的に決定するのには2つの記号で十分である。

Mckeemanのrecognizerは，可能なかぎりWirthの2個のアーギュメントの順位を用い必要があればtripleに切り替える折衷的なものである。スタックがhandleを含むかどうかを調べるために右側を見る場合には，エントリ⋖（⋖または≐），⋗，および⋙（⋗および≐または⋖）をもつ行列MATRIX1が用いられる。スタックの一番上の記号Siと入ってくる記号Tとの間に⋙が成り立つとき，3つのアーギュメントを持つ次のような関数の値を見出すためにtripleのリストをサーチする：

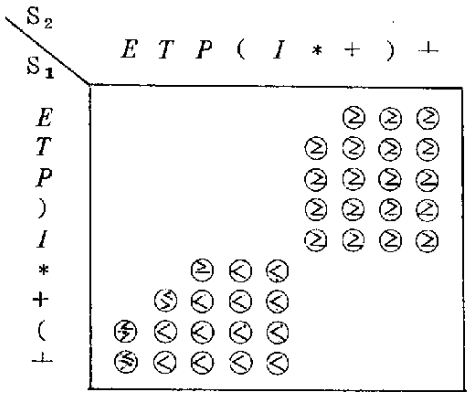
表　III

MATRIX1



⋖ という矛盾が生じていないので関数P1は不要である。

MATRIX2



関数P2

（いくつかのsentential formの有効な部分記号列をも形成している必要なtripleのみ列挙する）

$P2[\perp, E, +] = \mathrm{TRUE}$
$P2[\perp, E, \perp] = \mathrm{FALSE}$
$P2[(, E, +] = \mathrm{TRUE}$
$P2[(, E,)] = \mathrm{FALSE}$
$P2[+, T, *] = \mathrm{TRUE}$
$P2[+, T, +] = \mathrm{FALSE}$
$P2[+, T,)] = \mathrm{FALSE}$
$P2[+, T, \perp] = \mathrm{FALSE}$

$$P1(Si\text{-}1, Si, T) := \begin{cases} \text{true} & \text{文脈 } Si\text{-}1\ Si\ T \text{ において } Si \gtrdot T \text{ の場合。} \\ & (Si \text{ は handle の tail である}) \\ \text{false} & \text{文脈 } Si\text{-}1\ Si\ T \text{ において } Ti \lessdot S \text{ が成り} \\ & \text{立つとき。} \end{cases}$$

もちろん　この関数は　任意のtripleに対して値を持たねばならず，constructorはこのことをチェックするのである。エントリ ⋗, ⋖ および ≐ （⋖ および ⋖ または ⋗）を持つ 同じような行列　MATRIX2と関数P2はスタックのそのhandleの最初の記号を探す時に用いる。

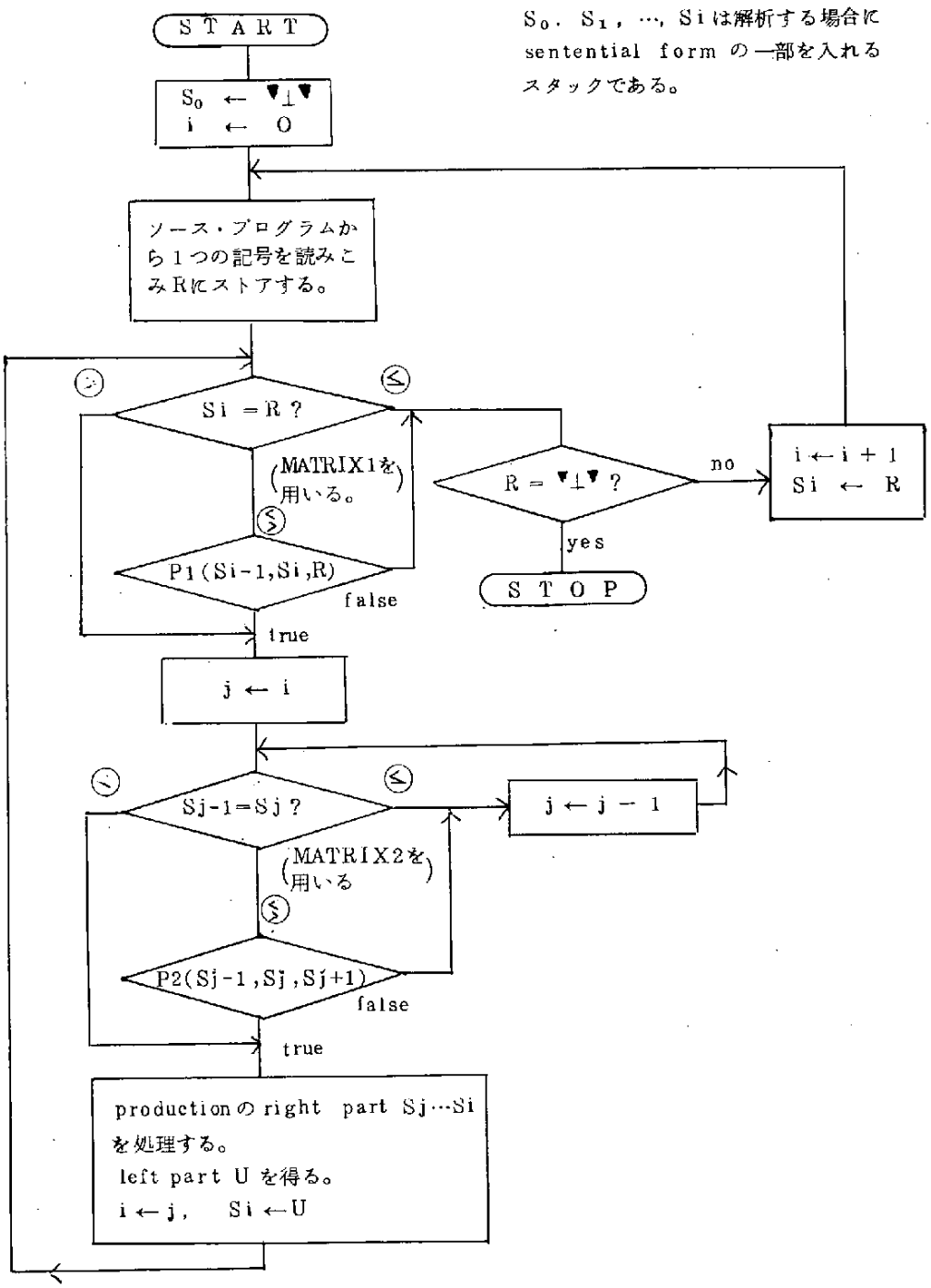


図7　Wirth の順位と Mckeeman の triple を用いた recognizer

$$P2(Sj-1,Sj,Sj+1)=\begin{cases}\text{true} & \text{文脈 } Sj-1\ Sj\ Sj+1 \text{ において } Sj-1 \lessdot Sj \\ & \text{の場合。(Sjは handle の head である)} \\ \text{false} & \text{文脈 } Sj-1\ Sj\ Sj+1 \text{ において } Sj-1 \gtrdot Sj \\ & \text{が成り立つとき。}\end{cases}$$

図1の文法に対する行列と関数は表IIIのように作ることができる。その recognizer は図7に示すごとくである。

triple の使用は，順位文法で遭遇する不都合な点を避けるのに役立つのである。しかし ここでも 同じ理由で文法を変更する必要があるかもしれないので，handle が見つかった時のみセマンティック・ルーチンが呼び出されるであろう。

Mckeeman はこの方法を用いてIBM 360 のPL/1の dialect に対するコンパイラを作成している。行列MATRIX1は約90×45，MATRIX2は90×90（各行列の要素は2ビットの長さである）で，約450個の triple が必要である。今考えられている別のアプローチは，handle の head を見つけるために用いる90×90の行列を捨ててしまうことである。この時には，スタックの中に handle が存在する時 適用するのに正しい production を決めるためにすべての可能な right part をスタックの内容と比較することになるであろう。

B．4 Transition Matrices（遷移行列）

（Samelson と Bauer〔Sam，60〕，Gries〔Grie 67a〕）

文を解析する（parsing）ためのこの方法は，Samelson と Bauer によってはじめて紹介されたものである。彼らの叙述では，2つのスタック — operator のスタックと operand のスタック — を用いている。文を左から右へと処理し／入ってくる identifier は operand スタックに push down する一方，入ってくる operator（+，／，begin など）は operator スタックの一番上のものと比較する。reduction が実行できない場合には，入ってきた operator を operator スタックに push down する。reduction が実行できる場合には，operator スタックの一番上の要素と operand スタック内のオペランドとを用いてサブルーチンを実行し，使ってしまったこれらの要素を消去して，結果のオペランドを operand スタックに push down する。演算子順位の方法に似ていることに注目されたい。すなわち 2つの terminal (operator) が実行すべき過程を決めるのに用いられ，nonterminal (operand) はここでは何も役割を果さないのである。特別な operand スタックというものは，terminal と nonterminal を別々に参照するのを容易にするためにのみ 用いられるもので，これは理論的なものではなく，

実際的な考え方である。唯ひとつの本当の相異は，演算子順位の方法では順位による行列を用いるのに対して，遷移行列の方法では　サブルーチン名の行列を用いることである。operator スタックの一番上の要素と入ってくる記号によって，実行すべきサブルーチンの名前である行列要素を決定する。次に　このサブルーチンは，必要な reduction を行なったり　operator あるいは operand スタックに記号を push down したりする。このようにして　productionは，行なうべき reduction　を決定するために各ステップにおいてサーチされる必要がなくなるのである。

遷移行列は，数多くのコンパイラの中で分析的な構文言語(anlalytic　syntax　language)として用いられてきている。Gricsは大きなクラスの演算子文法についての遷移行列の recog - nizer を組み立てる constructor を作成した。組み立てた recognizer が手書きの recognizer に似るように　演算子文法に制限が加えられた。

この constructorは，以下のようなやり方でprime　phrase (handle ではない)の始めを見つけるためにテストしなければならないスタック上のいくつかの要素を還元することを始める。いま，

＜cond＞→ if ＜be＞ then ＜expr＞ else　＜expr＞　　(4.1)

がこの文法のひとつの production であるとする。これは　次の４つの production　に変えることになる。

" if "　→　if

" ibt "　→　"if" ＜ be ＞　then

" ibtee"→　"ibt" ＜expr＞ else

＜ cond ＞→ " ibtee "　＜expr＞

これらの中間的な reductionは３つの記号 if　＜be＞ then から成る表現" ibt "　をスタックに push down するのを許すためにのみ用いる。同様に if　＜be＞　then ＜expr＞ else は "ibtee" で表わす。これらの新しい　引用符で囲んだ　nonterminal は，V, V_1 , V_2 , …などで表わすことにする。

解析の任意の段階において，そのスタックは　引用符で囲まれた記号である Vj　(j=1,…, i) ―あるもともとの production の right part の head についての表現 ― および多分最終的な nonterminal　operand のUから成るはずである。したがって　一連の

V_1 V_2 … Vi　(U) Tj Tj+1 … Tm

は，sentential form ではなく　ひとつの表現(　representation)である。(Uのまわりの括弧は，Uがそこにあってもなくてもよいことを示している。) 一般に　解析の各段階にお

いて存在する可能性のあるactionは次の3つである：

1. V_1 V_2 …Vi Vi+1 Tj+1 … Tm というreduction を生ずる場合には，(U) Tj は 他のright part の head を形成している。
2. V_1 V_2 … Vi-1 U Tj … Tm あるいV$_1$ V_2 …Vi-1 U_1 Tj+1 … Tmのいずれかを作り出す時，Vi (U) および おそらくTiは還元されうるright part を形成している。
3. V_1 V_2 … Vi-1 Vi′ Tj+1 … Tm というreductionを生ずる時，Vi (U) Tj はあるright part の head を形成している。

constructor はそれぞれのV，Tの対およびV，U，Tから成る各tripleを上記の3つの可能性があるかどうかを見るためにチェックする。各対やtriple に対して，このうちのたかだかひとつが存在し 実行されるべきreduction が一意的である場合には，遷移行列とサブルーチンが次のように組み立てられる。各行は引用符で囲まれた記号Vのそれぞれに対して割り当て，各列は可能なterminal symbol T の各々に割り当てる。行列要素の各々において もしVがVi，Tが入ってくる記号であるとするならば，実行するサブルーチンの個数がMv, T にストアされる。対応するサブルーチンは，nonterminal Uの順位をcheckし 適当なreduction を実行するために組み立てられる。我々は唯ひとつのスタックV_1，V_2，…，Viを用い，Uを location U 1におくことにする。図8は基本的なrecognizer である。図1の文法から作られる行列とサブルーチンは表4のごとくである。

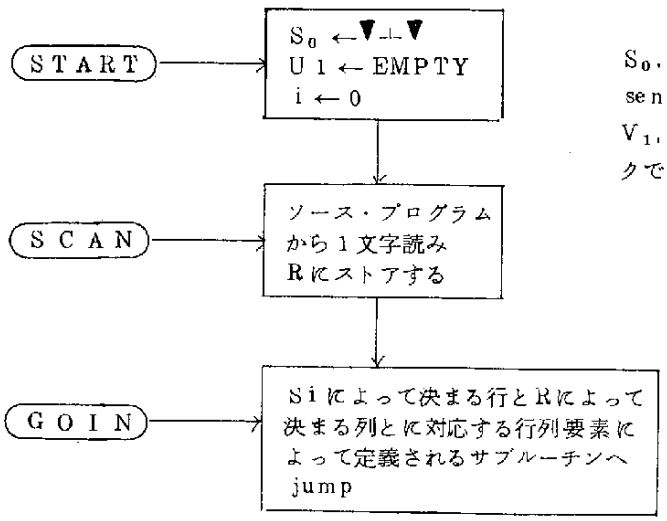


図8　遷移行列の recognizer

表　IV

Vi \ Tj	⊥	+	*	(	)	I
"⊥"	1	4	5	6		8
"E+"	2	2	5	6	2	8
"T*"	3	3	3	6	3	8
"("		4	5	6	7	8

1： if $U1 = E$ or $U1 = T$ or $U1 = P$
then $SUCCESS\ \ EXIT$ else $ERROR$;

2： if $U1 = T$ or $U1 = P$
then begin $i \leftarrow i-1$; $U1 \leftarrow E$; go to GO IN end else $ERROR$;

3： if $U1 = P$
then begin $i \leftarrow i-1$; $U1 \leftarrow T$; go to $GO\ IN$ end else $ERROR$;

4： if $U1 = E$ or $U1 = T$ or $U1 = P$
then begin $i \leftarrow i+1$; $S_i \leftarrow$ "E+"; $U1 \leftarrow$ empty; go to $SCAN$ end else $ERROR$;

5： if $U1 = P$ or $U1 = T$
then begin $i \leftarrow i+1$; $S \leftarrow$ "T*"; $U1 \leftarrow$ empty; go to $SCAN$ end else $ERROR$;

6： if $U1 =$ empty
then begin $i \leftarrow i+1$; $Si \leftarrow$ "("; $U1 \leftarrow$ empty; go to $SCAN$ end else $ERROR$;

7： if $U1 = E$ or $U1 = T$ or $U1 = P$
then begin $i \leftarrow i-1$; $U1 \leftarrow P$; go to $SCAN$ end else $ERROR$;

8： if $U1 =$ empty
then begin $U1 \leftarrow P$; go to $SCAN$ end else $ERROR$;

ALGOLに対する行列は約50×40であり，約500のサブルーチンを持つ。U1＝emptyであるか否かのチェックは，行列のいくつかの行を二重にすることによって削除されうる。（〔Grie 67a〕を参照）。一旦 recognizerが作り出されるから いくつかの変更が通常必要であるが，セマンティックスは解析の任意の段階で挿入されるはずであって（表IVの1～8の任意のサブルーチン内），right partが認識された時に限りはしない。この文法は多くの変更を要したりはしないが演算子文法でなければならない。このconstructor自体は，コンパイラでまだ用いられていないが，しかし ここでgenerateされるrecognizerに大変よく似たrecognizerが同じ方法を手書きで行なうことによって作られている。（〔Grie 65〕）

この方法は多分一番速いものである。一般に スピードが大切な場合にはいつも，テーブルを

切り替えること（Switching table）が用いられる。なされるべきreductionを決め 実行すべきセマンティック・ルーチンを決定する度ごとにproductionをサーチする必要はないことに注意すること。

この弱点は，用いられるスペースおよびこの方法をインプリメントするのに必要なサブルーチンの個数の多いことである。

B．5. Production Language

（Floyd〔Flo 61〕，Evans〔Ev 64〕，Earley〔Ear 65〕）

production languageは，Floydによって紹介され Evansによって修正されたコンパイラ作成のための分析的な（analytic）syntactic metalanguageである。これは，プロダクションの集合から成り，（語production（構文規則）とは，違った使い方であることに注意すること）。例えば次のようなものである：

$$LO : S_3\ S_2\ S_1\ |\rightarrow S_2'\ S_1'\ |*G1$$

これに対するもっと普通の名前はreductionである。というのは，それは記号列を還元あるいは解析する方法を示すものであるからである。

PROGRAMO	$\perp$	|		|*E0
	σ	|		|ERROR EXIT
E0：T0：P0：	(	|		|*E0
	I	| →	P	|*P1
	σ	|		|ERROR EXIT
$E1$：	$\perp E \perp$	|		|SUCCESS EXIT
	(E)	| →	P	|*P1
	$E+$	|		|*T0
	σ	|		|ERROR EXIT
$T1$：	$T*$	|		|*P0
	$E+T\sigma$	| →	$E\sigma$	|E1
	$T\sigma$	| →	$E\sigma$	|E1
	σ	|		|ERROR EXIT
$P1$：	$T*P\sigma$	| →	$T\sigma$	|T1
	$P\sigma$	| →	$T\sigma$	|T1
	σ	| →		|ERROR EXIT

図9　Production languageのrecognizer

スタック上にある文の最初の記号⊥を置くことから，その文の解析を始める。次に，スタックの一番上のものを第1の"｜"の左隣の記号S_1，S_2，…と比較しながらプロダクションをずっと進んでゆく。

等しいものが見つかると，そのスタック内の一致した記号S_1，S_2，…を記号S_1' S_2'によって置き換える（もし，置き換えを行なわない場合には，矢印"→"と記号S_1'，S_2'があらわれていない）。記号αがSiとして現われていれば，これはスタック上のどの記号とも一致することを示している。第2の"｜"のあとに"＊"があれば 次の入力記号（input symbol）がscanされ，スタックにpush downされることを示す。そのあと そのproductionの右にある名前を付けられているプロダクション（ここではG1）に行き，再びスタック上の記号との比較をはじめる。任意のプロダクションにラベルをつけることができる。Earleyは，適当な（総合的syntheticな）句構造文法からproduction languageで書いたrecognizerを作り出すconstructorを作成している。

図1の文法からgenerateされるproduction languageのプログラムを図9に示す。

セマンティックスは，あるプロダクションがgenerateされたあとで，そのプロダクションの任意の行の第2の"｜"の直後にあるEXEC iの形の"action"を挿入することにより導入される。ここでiは，あるセマンティック・ルーチンの番号である。

production languageの記述は，言語についての決定論的（deterministic）な記述であることを理解することは重要なことである。それは実際に，ある言語の文を解析するrecognizerを作成するためのひとつの言語である。これは普通の句構造文法をもつ場合ではないのである。

production languageやその変形は，多くのシステムの中で用いられてきている。実際にこれを使ってみると，プログラムするのに大変に自然で 柔軟性のある言語である。

プログラマはそれでコンパイラを書くのを比較的簡単に学ぶことができる。今までどんなコンパイラも機械的に組み立てられたrecognizerを用いて作成されたことはない。

EXEC actionは任意のプロダクションに入れることができるので，通常は文法にほとんど変更を加える必要がないのである。そして このrecognizerは より多くの文脈を用いることができるので，より多くの文法が他の4つのconstructorよりもこのconstructorによって受け入れられることができそうである。

我々は，これでTWSのひとつの部分がほぼ完全であると考える。限定された文脈を用いて，文を解析するための全く異なるたかだか有限個のleft－right recognizerを発明することは可能である。ここに述べたはじめの4個のrecognizerは，使用するプログラミング方法が異な

っているだけである　―すなわち，理論的には，これらは皆第２章Ｃの限定文脈（bounded context）を用いている。

演算子順位の方法は，これらの中で最もよく知られているものである。これは言語の一部　ふつう算術式や論理式を認識するのに　度々用いられているもので，IBM 360（H― level）のFORTRAN コンパイラでも使用されている。この方法を用いている　この他のコンパイラのドキュメンテーションについては〔Ar 66，Grie 65〕を見ればよい。〔Gall 67〕にも，これについて述べてある。遷移行列の方法は（といっても，そのconstructorについてではないが）いくつかのALGOLコンパイラを書くのに用いられてきている。特にNELIAC コンパイラやCO―NOテーブルを用いたALCORグループによるもの〔Hals 62，Mas 60〕などである。上記の両方法のいずれも，まぎれもなく数多くのコンパイラの中で使われてきているのである。production language は　ALGOLコンパイラで用いられてもいる〔EVA 64〕，けれども他の沢山のコンパイラを作成してきている〔Rov 67，It 66〕２つのコンパイラ・コンパイラで重要な部分を占めている〔Feld 66，Mond 67〕。

他のふたつのコンパイラ・コンパイラのプロジェクトがこの言語を用いており〔Fie 67，Grie 67b〕，その独立した変形が〔Che 65〕その他で用いられてきている。precedence およびextended precedence の方法は，主にWirth〔Wir 66a，Wir 66b〕とMcKeeman〔McKee 66〕によって用いられたものである。

operator precedence，precedence，extended precedence の recognizerにおいてはhandle（もしくは，prime phrase）が認識されるごとに，還元されるべきhandleおよび実行されるべきセマンティック・ルーチンに対する記号を見つけるためにproductionをサーチしなければならない。同様に　Floyd と Evansの analytic production languageを用いる場合には，あるreduction が実行されうる前にそのスタックがいくつかの可能性に一致していなければならない。したがって　これらの方法はすべて効率のよいテーブル・サーチングが実行されない限り　非常に遅いものとなる。

遷移行列の方法は，可能性のあるreductionのそれぞれに対して別々のサブルーチンをコーディングし　switching table　を用いることによって　この問題を解決している。このやり方の欠点は，行列とサブルーチンに費すスペースが多すぎることである。

理論的な考え方の読者に対して，次に　もっと一般的で，強力で，かつ複雑な（したがって，余り効率のよくない）left―right recognizerについて議論をすすめて行くことにしよう。形式言語（formal language）の理論についての基本的な参考文献は次の節の終りで与えることにする。

〔 参 考 文 献 〕

REFERENCES FOR II.B

Operator precedence: Ar 66, Gall 67, Grie 65.

Precedence and extended precedence: McKee 66, Wir 66a, Wir 66b, Wir 66c.

Transition matrices: Grie 65, Grie 67, Hals 62, Mas 60, Sam 60.

Production language: Che 65, EvA 64, Feld 66, Fie 67, Grie 67b, It 66, Mond 67, Rov 67.

C. Formal Studies of Syntax

C. 1. Bounded Context Grammars （限定文脈文法）

（Eickel〔Ei 63, 64〕, Floyd〔Flo 64a〕, Irons〔Ir 64〕, WirthとWeber〔Wir 66c〕）

任意の文法は，そのhandleがそれの左にあるm個の記号とその右にあるn個の記号によって常に一意的に決定される時　またその時にかぎり，（m, n）位限定文脈文法（（m, n）bounded context grammar）と呼ばれる。解析の間中　left－right recognizerは，可能性のあるhandleの左については（スタックの中にある）たかだかm個の記号　右については n 個の terminal symbol を各段階で考慮しながら，（m, n）位限定文脈文法の sentential form の handle を見出すことができる。

第II章Bで論じたはじめの4種の文法は，本質的には　（1, 1）位限定文脈文法である。ある意味では，（m, n）位限定文脈文法Gに対する（bottom-up）recognizer を組み立てることは，G から同値な文脈依存文法（context sensitive grammar）G1 を組み立てることである〔Gin 66a, P9〕。

文脈依存文法とは，x U y→xuy の形の production（生成規則）を持つ文法である。したがって　このような文法においては，Uによってuを置き換えるということは　xがuの左にありyがuの右にある場合にかぎり　実行できるのである。その recognizer を作る場合には，その文法が　どんな文脈の中で各 reductionがなされうるのかを 明らかに，そしてあいまいでなく述べることができるようになるまで文脈を各production の中に左から右へとつけ加えていく（そして　このためにより多くのproductionを組み立てることになる）。

m＞1，n＞1に対する（m, n）位限定文脈文法の recognizerは，コンピュータの時間とストレージのスペースを法外に要求をするらしい。したがって（m, n）位限定文脈文法は，コンパイラの中ではほとんど用いられてはいない。

（1, 1）位限定文脈文法に対する recognizer の作成に関しての初期の論文のひとつに〔Ei 63〕がある。この組立てのアルゴリズムの第1段階は，productionの right part の長さを1つか2つにまで変えるために　中間的なproductionを挿入することである。従って，production

$U_1 \rightarrow abcd$,　$U_2 \rightarrow abd$

は機械的に次のように変更されるのである。

$U_3 \rightarrow ab$,　$U_4 \rightarrow U_3\,c$,　$U_1 \rightarrow U_4\,d$,　$U_2 \rightarrow U_3\,d$

（第Ⅱ章B．4で用いた同様の方法とこれとを対照させてみること）。さて recognizerがスタックの一番上でhandle を見つけた時には スタックの一番上の二つの記号Si とSi-1 および入ってくるterminal symbol Tjが そこでとるべき段階を一意的に決めなければならない。そのため 各triple（S_1，S_2，T）について次の4つの状態のうちのひとつが必らず成り立たなくてはならない。

1. $S_1 S_2$ が handle であって，U ::＝ S_1 S_2 というひとつのreduction が実行されるはずである。
2. S_2 がhandle であって，U ::＝ S_2 というひとつのreduction が実行されるはずである。
3. Tがスタックにpush down されねばならない。
4. S_1 S_2 Tは sentential formの部分記号列としては現われない。

tripleとそれに対応するactionを作り出すアルゴリズムは，〔Ei 63〕に 例を挙げて出ている。このアルゴリズムと作り出されるrecognizerは，プログラムされテストされているがコンパイラを書くのに用いられてはいないのである。

一般的な限定文脈文法については，3つの主な論文が書かれている。そして それぞれの定義する"限定文脈"はわずかずつ異なっているのである。しかし それらの背景にある考えは同じなので，ここではその差異について論じようとは思わない。限定文脈に関するFloyd の論文〔Flo 64a〕と構造的結合（structural connection）についての Irons の論文〔Ir 64〕は限定文脈の秘密をより深く探究することに興味ある読者には一読をおすすめするものであるが，いずれも実際にrecognizerを組立てるためのアルゴリズムを与えてはいない。Eickelのもの〔Ei 64〕には，そのrecognizer と作り方について詳しく書いてある（そしてそのために この論文の方が前の2つよりも読みにくいものである）。

このrecognizerは通常のスタックを用いる。〔Ei 63〕によれば その文法は，先ず すべてのproductionが長さ1か2を持つようなものに還元される。次に5つの組（5－tuples）

（x；S；y，k，U）

を作り出す。ここでx，yは長さ(x)≦mで長さ(y)≦nのような記号列であり，Uはひとつのnonterminal，kは正の整数である。例えばスタックが

$S_0 \cdots Si-1\ Si$

を含み，

$Tj\ Tj+1 \cdots\ Tj+l-1\ \ Tj+l \cdots Tm$

が残りの入力記号（input symbol）でl＞0であるとする。個数lは入力記号列の最初のl

個の記号がその点における右側への文脈として必要とされることを意味している。5－tuple はS＝Si でxがS_0 …Si－1 の tail，yがTj…Tj+ℓ－1 のhead であるようなものが見つかるまでサーチされる。

取られるべき段階は，対応するkとUとに依存するものであり，次のごとくである：

k	action
0	stop … シンタックス・エラー
1	handle Si をUでおきかえる。（U→Siというreductionを行なう。）
2	i←i－1 Si←U（handle Si－1 Si をUでおきかえる）
3	if ℓ＝0 then ℓ←1 else begin i←i+1 ; Si←Tj ; 入力記号列からTjを削除する ; ℓ←ℓ－1 ; end
4	ℓ←ℓ＋1 （右側にもっと文脈が必要である）

文法が（m，n）位限定文脈であっても 右にn個の記号が常に必要であるとは限らない，ということに注意すること。recognizerは，取られるべきaction を決めるのに必要なだけの文脈のみ 用いるのである。これが前記の個数ℓを使った理由である。

Eickel はこのconstructorとrecognizerとをプログラムし テストしてきているが，この方法を用いてコンパイラを作ってはいない。この constructorは，xとyの長さを1に制限してスタートし すべての可能な5－tuple を作り出している。

もし 同じx，yとSおよび異なるk（または，同じx，y，S，kと異なるU）に対して二つ（またはそれ以上）の5－tuple が存在すれば，その文法は（1，1）位限定文脈文法ではない。このような5－tuple に対しては，矛盾が解決するか 最大値mかnになるまで，前述のようにさらに文脈（と5－tuple と）を加えながら xとyの長さを増加するのである。

Wirth と Weber〔Wir 66c〕は，記号列xとyに対する記号XとYの間の順位の概念を拡張した（II章B．2を参照）。このため，長さ(x)≦m および長さ(y)≦nにおいて，x⋖y，x≐y，x⋗yを得る。（m，n）位順位文法は もちろん我々の定義する（m，n）位限定文脈文法でもある。第II章B．2によれば，順位文法は（1，1）位順位文法である。

C．2 Deterministic Pushdown Automata

（決定論的プッシュダウン・オートマシン）

（Ginsburg と Greibach〔Gin 66b〕）

ＤＰＤＡとは，ひとつのスタックを伴なって働くleft－right recognizerの概念に関するオートマトン理論の形式化のことである。出発の状態（start state）So を含む状態（state）の有限集合 $S=\{So, \cdots, Sn\}$，入力（input）の集合 a （terminal symbol），出発記号（start symbol）Ｚを含む集合 η（nonterminal symbol に対応している）および写像（mapping）δ があるとする。ここで δ は次のとおりである。

δ :（states ×（nonterminal symbols）×（input symbols））→（states ×（nonterminal symbol の記号列））

あるいは

$\delta : (S \times \eta \times (a \cup \{\Lambda\}) \rightarrow (S \times \eta^{*}))$

この写像 δ は（ひとつの値を持つ）関数でなければならない。このほかに input のどこに現われてもよい空の記号（empty symbol）Λ をとりあつかうために，制限が加えられる。各段階において我々は triple

$$(\underbrace{Sp}_{\text{state}},\ \underbrace{U_1 \cdots Ui}_{\text{stack}},\ \underbrace{Tj \cdots Tm}_{\text{inputの残りのもの}})$$

を得る，初期の triple は（So，Z，$T_1 \cdots Tm$）である。各ステップで $n \geq 0$ の時 （Sp，Ui，Tj）→（Sq，$U_1' \cdots U_i'$）という写像の助けを借りて，その triple を次のように変えることができる：

（Sq，$U_1 \cdots Ui-1\ U_1' \cdots Un'$，$Tj+1 \cdots Tm$）

（入力の）記号列は 最終状態（final state）Sm が最終状態の集合Ｆの要素であれば，受け入れられる（accept される）。

任意の言語（何らかの文法から導かれる入力記号から成る記号列の集合）は，それがあるＤＰＤＡによって受け取られる時 deterministic（決定論的）であるという。これは deterministic な言語が，あるマシーン ― ＤＰＤＡ ― によって定義されるということであって，その言語を定義している文法の特別な性質によって定義されるということではないということに注意すること。Ginsburg と Greibach は，DPDA や deterministic な言語についていくつかの興味深い特徴を証明している。ここで我々にとって意味のあることは Knuth の LR(k) 言語との関係である。（第Ⅱ章Ｃ．３）

遷移行列を用いてＤＰＤＡをインプリメントできることに注意すること。この場合には 可能な state Sp の各々を行で表わし，各 terminal T を列で表わす。各段階において行列要素 $M_{Sp, Tj}$ は，そのスタックの一番上にある Ui に従って適当な写像を実行するサブルーチンを決定するのである。

このように遷移行列の方法は，かなり一般的である。しかし Griesによって書かれたconstructor（第II章B.4）は，（1，1）位限定文脈文法のサブセットしか受け取ることができないのである。

C. 3. LR(k) Grammars（LR(k)文法）

（Knuth〔Knu 65〕）

任意の文法は handle がその左の記号列全体とその右側のk個の terminal symbol によって，常に一意的に決定される時に限り LR(k) であるといわれる。これに対応する言語がLR(k)言語である。したがって スタックを用いて文を解析する場合，left－right recognizer は完全なスタック（the complete stack）（そしてその中の一定個数の記号とは限らない）と，その文でそのあとに続くk個の terminal symbol を調べればよいのである。これは，それに対してその文法から機械的に作り出されうる効率のよい left－right，bottom－upな recognizer が存在する文法の最も一般的な形である。事実 論究された他の constructor のどれによっても受け取られる文法は，あるkに対するLR(k)である。したがって LR(k)の条件はまた，今有効であるあいまいさのないことについての最も強力な一般的なテストでもあるのである。

Knuth はある文法が与えられたkに関してLR(k)であるか否かを決定するために2つのアルゴリズムを与えている。第2のアルゴリズムもまた―もし その文法がLR(k)であれば―本質的には（上述の）DPDAの形でrecognizerを組み立てるものである。これは一見 不思議に思えるかもしれない。というのは スタック中に現われうる無限個の記号列S_1，S_2，…，Siがあり 従って 解析の決定を行なうのに用いなければならない無限個の記号列があることになり，一方DPDAはスタックの一番上の要素と state と入力記号のみを必要としているからである。

しかしながら production の個数が有限であることから，解析の各段階において スタック中に存在する記号S_1，S_2，…，Si－1には関係なく 有限個の可能な action が存在するだけである。このようにして 可能な記号列S_1，S_2，…，Si－1を" state " Spと呼ばれるクラスに分けること，およびあるkに対してその文法が LR(k) であるならば 必要なひとつの値をとる（single valued）写像

（Sp，Si，Tj）→（Sq，S_1'…，Sn'）

を記述することのみ必要である。

もちろん 我々はここで，その概念をわかるようにその過程を非常に簡略化しているのである。

ＤＰＤＡのスタック中の記号Ｓｉはもともとの文法の記号であるばかりでなく，それ自身その必要なｓｉｎｇｌｅ ｖａｌｕｅｄな写像を決めることができるような複雑な集合であるかもしれないのである。

Ｋｎｕｔｈはまた ＤＰＤＡによって受け取られる言語£のそれぞれに対して，£を定義するＬＲ(1)文法が存在することをｃｏｎｓｔｒｕｃｔｉｏｎによって証明している。したがって任意のＬＲ(k)言語は，同じ文法を持たないにもかかわらず ＬＲ(1)でもある。Ｅａｒｌｅｙ〔Ｅａｒ 67〕は ＬＲ(k)文法に対するｃｏｎｓｔｒｕｃｔｏｒを書いたが，そのアウトプットは Ｆｌｏｙｅｄ－Ｅｖａｎｓのｐｒｏｄｕｃｔｉｏｎによく似た形のｐｒｏｄｕｃｔｉｏｎである。

Ｃ．４ Ｒｅｃｕｒｓｉｖｅ Ｆｕｎｃｔｉｏｎｓ ｏｆ Ｒｅｇｕｌａｒ Ｅｘｐｒｅｓｓｉｏｎｓ

（Ｃｏｎｗａｙ〔Ｃｏｎ 63〕，Ｔｉｘｉｅｒ〔Ｔｉｘ 67〕）

Ｃｏｎｗａｙは，多くのｔｒａｎｓｉｔｉｏｎ ｄｉａｇｒａｍをそのシンタックスが指定されるコンパイラについて論じている。そのｔｒａｎｓｉｔｉｏｎ ｄｉａｇｒａｍとは，図10のように一定のｎｏｎｔｅｒｍｉｎａｌから導かれ得る記号列を認識するものである。

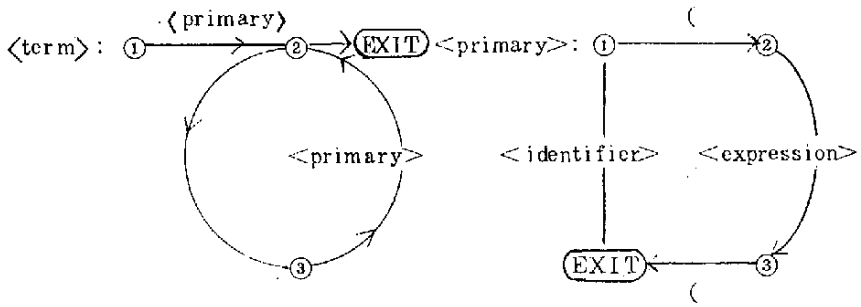


図10　２つのｔｒａｎｓｉｔｉｏｎ ｄｉａｇｒａｍ

文の解析の各段階において，あるｃｕｒｒｅｎｔ ｄｉａｇｒａｍの ｃｕｒｒｅｎｔ ｎｏｄｅが存在する。ｌｅｆｔ－ｒｉｇｈｔ ｒｅｃｏｇｎｉｚｅｒによって実行されるａｃｔｉｏｎは（ｄｉａｇｒａｍや ｐｕｓｈ ｄｏｗｎスタックを用いて）次のように決定される：

(a) 次の入力記号（１個のｔｅｒｍｉｎａｌ）がｃｕｒｒｅｎｔ ｎｏｄｅから出発する線（ｌｉｎｅ）のひとつの名と一致する場合には，新しいｃｕｒｒｅｎｔ ｎｏｄｅまでその線を進み 次の入力記号をｓｃａｎする。

(b) (a)で一致しない場合で ひとつの線がｎｏｎ－ｔｅｒｍｉｎａｌの名前を付けられている時には，

その線をたどり current diagram の名前と新しい current node とをスタック上に push down し current node をUと名付けられたその diagram の最初の node に変更する。

(c) (a)でも(b)でもなく ラベルを付けられていない線があれば それをたどる。

(d) current node がEXITである時には，その current diagram で記されたnonterminal から導きうる記号列が認識できたわけである。したがって そのcurrent node (と diagram) をスタックの一番上の要素によって指定されるものに変え，スタックからその要素を削除する。

これは 逆戻りを伴なわない top－down left－right recognizer である。(b)の段階において，Uから導きうる記号列が認識されるであろうという prediction (予言) に注意すること。この方法は，各サブルーチンに含まれる文字集合 (character set) が大変小さなものであるために，構文解析を効果的に小さな簡単な部分に分け スペースを節約するものである。各 transition diagram が回帰的に他の有限状態のオートマトンを呼ぶこのできる個々の有限状態のオートマトンを表わしているということは明らかである。

Tixierは，この概念を彼の学位論文の中で独立に形式化している。彼は production Ui → xi を Ui = xi という regular expression であると考えた。ここでは union (+), product (=), closure (*) という演算の集合が用いられている。

このようにして production

< identifier >→< letter >

< identifier >→< identifier >< letter >

は, < identifier >=< letter >+< identifier >< letter >

あるいは < identifier >=< letter >< letter >*

と等しく書くことができる。

Tixierは Euler の 120 の production [Wir 67c]を7個の変数から成る7つの関数として書き直している。そのうちの3つをここに示すが，記号"("と")"は bracket set expressionに対するメタシンボルとして用いている：

$program = block$

$block$ = begin ((new id + label id) ;)*(id :)* $expr$ (; (id :)* $expr$)*
end

$expr$ = (cut + if $expr$ then $expr$ else + id ([$expr$] +)* ←)*
(go to $primary$ + $block$ + $catena$)

Tixier の constructor は，main goal が有効な有限状態オートマトンを通して できるかぎり解析を行なう時，もし必要ならば最少数の方程式になるように任意の文法の方程式（もしくは production）を操作することができる。次に適切な文法に対して　そのconstructorは　その言語の文をあいまいさがないように，そして逆戻りせずに，互いに回帰的に呼び出しあう（各方程式に対してひとつの）有限状態オートマトンの集合の形において　解析できる（modified top－down） recognizer を作り上げるのである。ここで最も困難なことは，他の有限状態オートマトンが呼び出されるかどうかを current node と入力記号からあいまいでないように決定し，もし　そういうものがある場合にはどれであるかを一意的に決めることである。受け付けられる文法とその言語とは共に regular context free（RCF）と呼ばれる。

この constructor は実際に効率のよい制限されたＤＰＤＡを作るものである。したがって　ＲＣＦ言語はＬＲ(1)言語である。

C．5. Summary

図11は，この報告書で論じた個々の constructor によって受け付けられる文法のクラスを inclusion tree の形で示すものである。この tree は　いくつかの場所でまぎらわしくなっている。 Floyd の production language のようなある種の metalanguage は 強力なものではあるが，このような metalanguage を用いる recognizer の特定の constructor は　受け付けることのできる文法を制限するかもしれない。参考文献が書いてあれば，その node はこの論文で定義されている言語（または constructor ）を指すものである。図11について次のようなことに注意すること。

(a) （1，1）文法と extended precedence 文法とはいずれも triple を用いるが，（1，1）文法の長所は本質的に　より広範囲の文脈を許容する自動的な中間的 reduction（automatic intermediate reduction）を実行することである。もちろん　任意の constructor が，これらの中間的 reduction を含むように変えることができるであろう。

(b) 遷移行列（transition matrix）文法は（1，1）と（0，1）位限定文脈文法の間のどこかに落ちることになる。

(c) 演算子順位の条件が，右側が同一であることを禁ずるように増大されていると仮定する。そうでないと，この inclusion は成り立たない。演算子順位における行列の方法の利点は，(a)と同様に　automatic intermediate reduction を用いることである。

(d) Feldman は〔Feld 64〕の中で，各ＤＰＤＡが production language で書くことのできる何らかのプログラムに同値であることを示している。

ここでしばらくの間，recognizerをtop－downとかbottom－upとかに分類するという問題に立ちもどってみよう。Tixierのrecognizer(第Ⅱ章C・4)は明らかにtop-downである，すなわち新しい有限状態オートマントに切り替える際に，prediction(goal)が作られることがわかるのである。

Knuthの組み立てたLR(k)文法のrecognizerはtop－downであろうか？ある人々は「否」というであろう。このrecognizerは右側のk個の記号と左側に対応するスタック内の記号とを文脈に従ってreductionを行なうだけである。LR(k)自体 そのrecognizerも含めて (m,n)位限定文脈recognizerを拡張したものであり，この(m,n)位限定文脈recognizerは 通常bottom－upである。さらに，実行されるべきreduction を決めるための助けとなるように余分の"state"がスタックにつけ加えられている。しかしまた ある人々はこれらのstateを用いることによってpredictionとかgoalとかが実際に導入されているとも言うのである。このためにtop－downであるTixierのrecognizerも，Knuthのrecognizerも共にDPDAである。ここで これらの概念がひとつになってしまい，ある場合にはどちらかになりうるということが はっきりと言えるのである。

コンパイラ作成に関係のあるシンタックスの研究において これまでにいくつかの概念を要約してきたが，沢山のものを省略してきている。Gilbert〔Gil 66〕について少し言及するならば，彼はreductionを行なう際に(もし いくつかの可能性があるならば)どのproductionを用いたらよいかを(sentential formに基いて)示すsleetion function (選択機能)を文脈依存文法につけ加えたのである。このように selection functionを用いることによってsynthetic(総合的な)文法をanalytic(分析的な)文法として用い得るのである。Gilbertは これらの文法とselection functionのいくつかの特徴を証明し，(宣言文の比較と変数の使用を含む)FORTRANやALGOLのような言語のシンタックスを完全に記述することができることを示している。

top－down recognizerについては〔Che 64c〕や〔Flo 64b〕によく出ているので，ここでは詳しく述べないことにした。文脈独立言語(context free language)におけるあいまいさの問題については，それがTWSに関連する時だけ簡単にふれてきた。オートマトンの理論の分野についても関連が深いけれども ここでは述べなかった。

事実，我々は文脈独立文法，オートマトンの理論やマシーンについてとり扱かっている多くの論文をこの報告書の最後の参考文献から除外しなければならなかった。これらの多くのものや，それ以外のものに対するリファレンスは，J. ACM, Information and Control および〔Gins 66a〕に出ている。

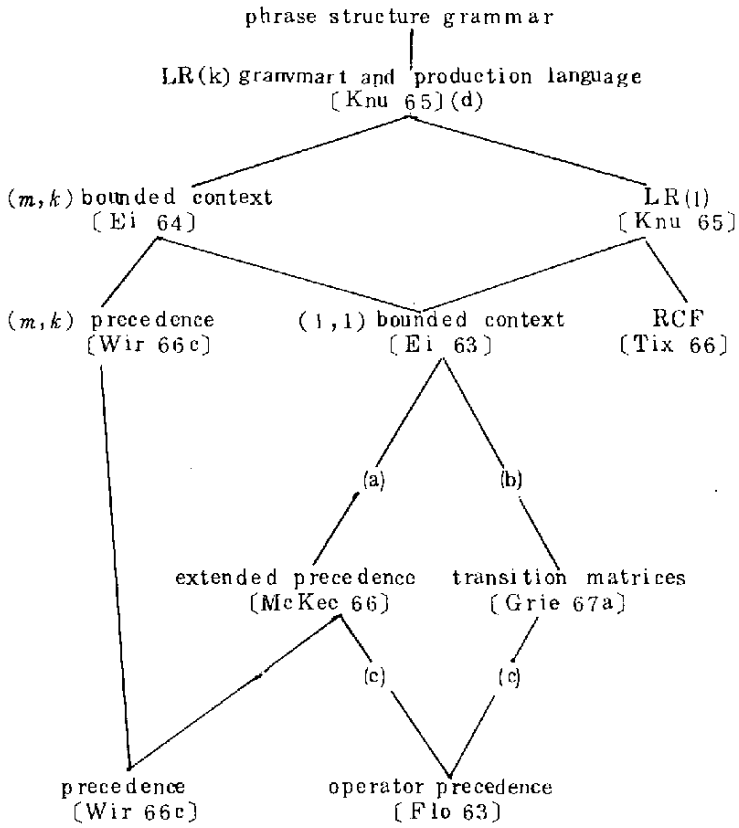


図11　Inclusion tree

〔参考文献〕

REFERENCES FOR II.C

Introduction to the theory of formal languages: Bar 64, Gins 66a.

Pure or modified top-down algorithms: Barn 62, Br 62a, Che 64c, GraR 64, Ing 66, Ir 63a, Kir 66, Kun 62, Rey 65, Scho 65, War 64.

Construction of efficient recognizers — sufficient conditions for unambiguity: Ea 65, Ea 67, Ei 63, Ei 64, Flo 63, Flo 64b, (iil 66, Gins 66b, Grie 67a, Ir 64, Knu 65, McKee 66, Paul 62, Wir 66c, Tix 67.

Surveys, tutorials on recognizer techniques: Che 64c, Flo 64b, GraR 64.

Ambiguity in context free languages: Can 62, Flo 62a, Flo 62b, Gin 66a, Gor 63, Lang 64, Ross 64,

Thirteen different ways to define languages: Gorn 61.

Ⅲ Sementics

A. Syntax – Directed Symbol Processor

この章で述べるプログラムは，正確には コンパイラ・コンパイラとは呼ばないものであるがコンパイラ作成に用いられてきたものである。記号処理の仕事としてのこれらの共通のコンパイラ作成のやり方は，これらのプログラムをＴＷＳより以上のものにも 以下のものにもしている。ＴＷＳの初めのうちの努力の大部分は この種のものであり，最も特筆すべきものは〔 Ir 61〕である。このようなシステムは より一般的であるので，第Ⅳ章Ａのいろいろな nontranslator の仕事によく用いられる。実際その goal が 最初からもっと一般的であるために，ＡＥＤについて（〔 Ross 67 〕）は その節に譲ることにした。

A・1 ＴＭＧ （ McClure〔McCl65〕）

ＴＭＧシステムは Texas Instruments で，シンボリックなアウトプットを作り出す。簡単な１パスのコンパイラを作成する手段として開発されたものである。シンタックスの処理方法は逆戻りを伴なう簡単な top-down scan である。しかし そのセマンティック・ルールの埋めこみは，セマンティックな面において いくつかのシンタクティックに可能な goal を削除することによって，その recognizer がより効率のよいものとなるようにしているのである。

基本的なＴＭＧステートメントの形は，スペースによって分離された action の列である。任意の action の先頭に＜label＞をつけることができ，action の後には その action が失敗であった場合に取られる＜destination＞をつけることができる。＜action＞は syntax recognizerに対する中間的な goal，入力に関する記号列の計算（string computation on the input），組み込まれたステートメント（built-in）のいずれかである。

これらの action は すべて 中間的な tree を作り上げる際に，トランスレータによって実行されるものである。

実際のコードの出力は，以下に述べるような別のルーチンの集合によってなされる。

primitive ＭＡＲＫＳとＩＮＳＴＡＬＬを用いて，入力記号列から作り出される文字をベースとしたシンボル・テーブルがあるとする。このとき 次の例について考える，

ＩＮＴＥＧＥＲ：ＺＥＲＯ＊ＭＡＲＫＳ ＤＩＧＩＴ ＤＩＧＩＴ＊ＩＮＳＴＡＬＬ

action ＺＥＲＯ＊はすべての leading zero を scan することであり，ＭＡＲＫＳは 入力記号列のポインタの現在の値を記憶することである。action ＤＩＧＩＴ ＤＩＧＩＴ＊は＜digit＞というクラスに属するすべての文字を scan することである。ＩＮＳＴＡＬＬの実行によって，

MARKSのポインターで始まる記号列がシンボル・テーブルに入れられ，それへのreferenceが中間的なtree(intermediate tree)に入れられる。このテーブルに許されているこのほかの情報は，identifierの属性を記述するのに用いられる宣言されたFLAGS（論理変数）の集合だけである。

組み込みルーチンは　条件つきの算術式，数値変換　そして2，3の入出力関数を含んでいる。また　入力ポインタを入れておくJ，入力された最近の記号列の長さを入れておくSYMNRMのようなsystem cellもある。出力もcharacter-orientedであって，例えば次のようである。

LABELFIELD：LABEL＝$（$P1／BSS／0//$）

このステートメントは，ある言語のlabelの処理に用いるものであろう。"＝"記号は，"$（"と"$）"で囲まれる出力ルーチンを知らせるものである。この出力ステートメントの本体はアセンブラのコード　　value(P1)　BSS　0

から成る1行を形成する。

記号$P1は＝の左にある第1の要素，たいていは　labelのシンボリックな名前を算定するコマンドである。／はタブ(tab)を入れることと　BSSおよび0が，その文字自体を表わしていることを示している。最後に//は出力の中にキャリッジ・リターンを置くことを意味する。出力ルーチンはプログラムの中間的なtreeの上から下へと働きかける。したがって　出力ルーチンの中の$Pnはsubtrecを参照してもよく，このとき$Pnの算定は　他の出力ルーチンの回帰的な呼び出しを含むことになる。valueによってパラメタを内部のルーチンへ受け渡すことも可能である。この論文では　これらの機能の例をいくつか上げており，また　TMGのエラーのrecoveryの能力についても簡潔に述べている。

このTMGの仕事は　試験的なプロジェクトであり，その体裁のよくないシンタックスは　定着する(fix)のが容易であった。これはすでに多くのコンパイラを作成するのに用いられており関連システムTROLは，Knuthによってコンパイラ記述を教えるのに用いられたものである。MULTICSで使ったEPL（Early PL/I）は，TMGの定義を用いて　一層よいコードを得るように2パスシステムとして作成された。TMGシステムは　以下で考えるようなシステムのあるもの程　筋の通ったものではないように思われるが，他のiterationから利益をうけているものである。

A．2　The　META　Systems

（Schorre，〔Schor 64〕，Schneider and Johnson〔Sch 64〕）

METAシステムは，ロサンジェルスのSIGPLANのsyntax directed compilerに関す

る作業グループの作り出したものである。そのもともとの仕事は多様性に富んだものであったが，現システムは大部分はSchorr が開発したMETAⅡとして知られるモデルに基づくものである〔Schor 64〕，このモデルの中では　ある言語についての解析と翻訳の過程は　BNFのような規則の集合として　すべて述べられている。これらの規則はインプリメントされた時に　埋めこまれるcode generator と共に回帰的なrecognizer となるものである。この規則は　その代りにprefix iteration operator $を用いているので，左回帰を禁止している。terminal symbolは引用符で囲み，システムの記号は"．"を先頭につける。そして特別な印のない記号はユーザーのnonterminal である。right part の選択するものをまとめるには括弧を用いる。次の規則は，論理式を翻訳するのに用いる。

1. UNION=INTER('OR'.OUT('BT' *1) UNION .LABEL *1|.EMPTY);
2. INTER=BPRIMARY('AND' .OUT('BF'* 1) INTER .LABEL *1 | .EMPTY)
   ;
3. BPRIMARY=.ID .OUT('LD'*) | '('UNION')';

最後の規則は，算法言語における論理1次子を認識するための手続きを定義するものである。"="の前にあるBPRIMARYという語は　この規則の名前を定義し，規則のright part は入力ストリームが論理式として正しいかどうかをテストするアルゴリズムと　(.ID)が見つかった場合のcode generator である。

上記の規則は，多くのMETAコンパイラで用いられている3つの基本的な要素を含んでいる例である。他の規則の名前　この場合にはUNIONについて，記述は　呼び出されるべきrecognizer の回帰的な呼び出しをひきおこす。リテラル・ストリング"("が現われることは，入力ストリームが左括弧であるかどうかのテストされるはずであることを意味している。出力ステートメント.OUTは　テキスト1行を作り出し，ここでは　"*"は常にprimitive なnonterminal .IDによって認識された最近の項目(item)を指している。

(上記の規則1のような)規則中にある最初の*1はlabel のgeneration とそのlabel の出力の両方を意味している。同一の規則の中で次の*1は同じlabel を出力することである，すなわち新しい規則が現われるごとに新しいlabel がgenerate され得るのである。これらのlabel は，その規則がactive である時にかぎり存在するものである。他の規則の呼び出しがなされる場合には　このlabel は，スタックにpush downされる。呼び出した規則から帰ってくると，その前のlabel がrestore される。action .LABEL　*1は*1に対応するlabel が書き出されることを示している。.EMPTYは　入力に関しては効果を持たないけれども，常に満たされているか　真(.true)であるかのいずれかのprimitive なnonterminal である。

入力ストリーム"（A OR B)AND(C OR D)"については，次のようなコードが作り出されるはずである。ここでLD，BT，BFはそれぞれ Load，Branch True，Branch False を示すニーモニック・コードである。

```
        LD    A
        BT    L1
        LD    B
L1
        BF    L2
        LD    C
        BT    L3
        LD    D
L3
L2
```

META－Ⅱの有用性は　逆戻りの機能のないことと　出力を並べ直すこと（reordering)によって，著しく制限される。METAの方法を完全なTWSへと拡張しようとする試みが度々なされてきている。META－3〔Schor 64〕は　基本的なMETA－Ⅱの概念を拡張しようとしたものであり，これによってALGOL 60 を 7090 でコンパイルすることができたのである。これはセマンティックなテストとレジスタの操作について　いくつかの機能を与えているが，その付け加えたものは　必らずしも適当であるとは証明できない。META－5は既に多くのフォーマット変換やソース言語からソース言語への翻訳に用いられてきているが，コンパイラのためには使われていない。最も最近　開発されたものは　TREE　METAであり，これは中間的な syntax tree の複雑な処理を用いるマルチパス（multipss）システムである。METAコンパイラのスピードの遅いことと　効率のよくないことは　それらの製作者も認めるところのものであるが，インプリメンテーションのしやすさ，boot－strapping の機能　それに　とりあつかうことのできる言語のクラスの広いことなどが，彼らの発展しつつある　この仕事を正当化するのに役立つものとなっている。

A．3　COGENT　（Reynolds〔Rey 65〕）

COGENTシステムはArgonne National 研究所で設計され　CDC 3600 に対してインプリメントされたものであって，BrookerとMorris（第Ⅲ章B．3）や　Irons〔Ir 61，May 61〕の考え方に強い影響を受けている。そして　LISP　COGENTは，大変によく考えられ

かれたもので　この節で記する他のシステムよりも比較的わかりやすいものである。COGENTコンパイラは，完全にそれ自身の言語で書かれている。boot－strappingを3回行なうことによって，それ自身のコンパイルのスピードは　6の倍数で速くなっていったのである

COGENTで書かれたシステムは，2つの部分から成っている。すなわち　シンタックスとgeneratorとよばれる処理ルーチンの集合とである。そのシンタックスは，総合的な句構造文法で与えられる。左回帰の文法を含めてほとんどすべての文脈独立文法が受け付けられうる。わずかの制限としてright partが空なものはだめである。この文法を用いるrecognizerは，並行に処理していく各段階で代わりのもの(alternative)を持っている。修正されたtop－downの形である。任意の記号列は，recognizerが　それに対する一意的なsyntax treeを見つければ，受け付けられ得るのである。

構文解析(syntactic analysis)は，中間的なtreeを表現するためのリスト構造を作り出す。例えば　次のproduction

＜term＞::＝＜term＞＋＜factor＞

を用いることによって，＜term＞と＜factor＞についてのsubtreeに対するポインタを持つリスト・エレメント＜term＞が作り出されるのである。

generator（semantic routine）の名前によって任意のproductionを処理することができる。このgeneratorは　そのproductionがtreeを作り上げるのに用いられる時，次に実行されるものである。（alternativeを並行に処理しているために）2つ以上の可能なsyntax treeがある場合には，この局所的なあいまいさが解決し　唯ひとつのtreeが残るまで，これらのgeneratorの実行を遅らせ　構文解析を続ける。そのあと　すべてのgeneratorが，正しい順序で呼び出される。

例として，次のlabelをつけたproductionを考える。

Processterm／　＜term＞::＝＜term＞＋＜factor＞

＜term＞に対するsubtreeがroot（根）として完全に形成される時，アーギュメントとして＜term＞と＜factor＞についてのsubtreeを持つようなgenerator ▼processterm▼ が呼び出される。▼processterm▼は　これらのsubtreeを操作したり，それらを削除したり，それらに対応するコードを作り出したり　などするのである。

generatorの言語は，リスト処理のオペレーションと失敗のメカニズム（mechanism of failure）を基礎としている。リスト・エレメントは　他のエレメントへのポインタを持つことができ，そのポインタの個数は可変である。リスト・エレメントのタイプは　数値（固定小数点もしくは浮動小数点），generatorの入口のポインタ，dummy element，identifier element

およびパラメタ・エレメントである。固定小数点数は　任意の大きさであってよく，その大きさを表現するのに十分なワード数を取ることができる。この性質は　ＣＯＧＥＮＴの記号数学的な応用に（symbolic mathematics applications）便宜を与えている。

普通の代入ステートメントに加えて，generatorでは　リスト構造の総合と分析を記述するために，総合的および分析的な代入ステートメントを用いることができる。総合的代入ステートメント（synthetic assingnment statement）は，

＜identifier＞／＝＜template＞，＜expression list＞

の形で，ここで＜template＞は　パターン・マッチングのために用いるものであって"::＝"のかわりに"／"を伴なう括弧内のproductionと思われる。このステートメントは＜identifier＞が＜template＞のコピーであるようにし，この中では第i番目のパラメタ（nonterminal）が＜expression list＞内の第i番目の式の値によって置きかえられるようにする。

例えば　総合的な代入ステートメント

Ｚ／＝（＜term＞／＜factor＞＊＜factor＞），Ｘ，Ｙ

を実行する。ここでＸは（＜factor＞／ＡＢＥ）の値，Ｙは（＜factor＞／ＢＥＤ）の値を持っているとすると，この実行により　（＜term＞／ＡＢＥ＊ＢＥＤ）のコピーがＺに代入される。

同様に　分析的代入ステートメントの形は

＜test expression＞＝／＜template＞，＜identifier list＞

であって，これは式を分解するのに用いるものである。＜test expression＞を＜template＞に対して照合させる。もしそれが一致すれば，そのtemplateの第iパラメタ（nonterminal）に対応する値が＜identifier list＞の第i番目のidentifierに代入される。したがって　いま　Ｚが（＜term＞／ＡＢＥ＊ＢＥＤ）という値を持つ時，次のステートメント

Ｚ＝／（＜term＞／＜factor＞＊＜factor＞），Ｘ，Ｙ

は，Ｘに（＜factor＞／ＡＢＥ）という値，Ｙに（＜factor＞／ＢＥＤ）という値を与えるのである。

もし＜test expression＞と＜template＞が一致しなければ，その分析的代入ステートメントは失敗である（fail），失敗（failure）というものはＣＯＧＥＮＴにおいては，分岐の方法である。もし　すべての条件ステートメントが失敗するようなactionを含んでいない場合には，そのgenerator全体が失敗である。この失敗は，何らかの条件ステートメントが現われるまでgeneratorの呼び出しのchainをたどって進んでいくのである。

ＣＯＧＥＮＴのプログラムは，任意個のシンボル・テーブルを用いることができる。action label $ＩＤＥＮＴ　nは，そのproductionの結果（文字ストリングのはずである）がシンボ

ル・テーブルnに入れられねばならないことを指定するものである。それがすでにそこに在れば，その古いコピーに対するポインタが返される。すなわち，任意の指定されたテーブル内で，すべてのidentifierは，一意的な文字ストリングを持っている。テーブルの各エントリは，そのidentifierおよび，通常そのidentifierの属性を指すポインタ・エレメントから成っている。

出力は　1個のパラメタ――出力行に置かれるべき文字内部コード――を持つＰＵＴＰルーチンを呼び出すことによってなされる。1行が一杯になると，それは外に書き出され　新しい行が始まる。しかしＯＵＴＰを呼び出して，ひとつの行が完全に一杯になる前に外にプリントすることも可能である。このほかのprimitiveなgeneratorＳＴＡＮＤＳＣＮ(Ｘ，ＰＵＴＰ)は，リスト構造Ｘをそのend nodeによって表わされる記号列Ｓに写像する。ＳＴＡＮＤＳＣＮは　Ｓ内の記号を見つけ，それらを1度に1つづつＰＵＴＰに送るものである。

ＣＯＧＥＮＴは明らかに実験的なものであり，いくつかの欠点も持っている。generatorに対する言語の構造には，ＡＬＧＯＬが持っているような言語としてのスマートさはない。入力中にシンタックス・エラーがあれば，それは致命的なものとなる。リスト処理は，plex が単にそのシンタックスに基づくだけでなく　任意のplexの創造を含むように一般化されるべきである。ＣＯＧＥＮＴは，記号的な数学における諸問題に適用されてきた。

Reynoldsは データ構造のよりよい理論の展開を未解決のままにしてＣＯＧＥＮＴに関する仕事を一時中止していたが，現在は　他の人々と共に研究を続けている。

（第Ⅳ章Ｃを参照のこと）

Ａ．4　ＥＴＣ（Garwick〔Gor 64〕，Gilbert〔Gil 66〕，Pratt〔Pra 65〕）

この章では，syntax−directed symbol processorとして記るすのによい3つの仕事について記述してみようと思う。いろいろな理由で　これらのシステムは，前述のシステムの影響を受けてはおらず　詳しくは述べられていないのである。

ＧＡＲＧＯＹＬＥシステム〔Gar 64〕は，ノルウェーのDefense Establishmentにおいて，Jan GarwickがControl Data 3600に対して開発したものである。内部報告書としては　かなり完全なものがあるが，論文として発行されたもの――これが初めて書かれたものであるが――は，ＧＡＲＧＯＹＬＥの大まかな輪かくしか与えていない。

ＧＡＲＧＯＹＬＥは，多くの面でＴＭＧ（第Ⅲ章Ａ－1）に似ている。すなわち　そのrecognizerとは，サーチの方向づける機能を備えたtop−downである。シンタクティック，およびセマンティックなステートメントはすべて＜label＞＜else＞＜next＞＜link＞＜action＞の5つのエントリからなるテーブル形式で記述される。（逆戻りが暗黙のうちに行なわれたりす

るので），次にどこに進むかという規則（ sequencing rule ）は　大変に複雑であって，通常は"steering routine"によってなされるのである。

逆戻りのメカニズムもいくつかの変数をスタックすることや，そのほかのものをコピーすることなどを含んだ複雑なデータのとりあつかいを要するのである。５つのエントリはすべて　いくつかの方法で用いられる。例えばlabelは，その行がどのように翻訳されるかをコントロールするコードの代わりになるものである。

次の例は，＜代入ステートメント＞の処理に用いられるものである。

⟨label⟩	⟨else⟩	⟨next⟩	⟨link⟩	⟨action⟩
R		*ASSIGN*		
L				*VAR*
0		*VARIABLE*	1	
1		*NEXT*	2	**if** *SYMB*=*COLEQ* **then** *VAR*←*WORD*
2	3	*ASSIGN*	4	
3		*EXPR*	4	
4		*RETURN*		*OUTTEXT* (*"STO"*, 10); *OUTFIELD* (*VAR*, 20);

第１行目は　このルーチンＡＳＳＩＧＮのheaderであり，ＶＡＲのそのローカルな変数であって，第２行目で宣言されている。ＶＡＲＩＡＢＬＥが＜変数＞を見つけると，その次の記号が，ＣＯＬＥＱ（＂：＝＂）であるか，このルーチンが失敗であるかのいずれかである。２という行は，多重に出現した左辺を扱かうためにＡＳＳＩＧＮを回帰的に呼び出すものである。最初の逆戻りは　（右辺をコンパイルする）ＥＸＰＲに導き，そして続いて起るリターン（ return ）は多重の代入（muliple assigment）において，各変数について１回だけ　ステートメント４を実行する。ステートメント４の中の＜action＞ＶＡＲの続きの値の各々にストアを行なうテキストを作り出すものである。

＜action＞の列にある言語は，部分的なword operatorや テーブル・サーチや入出力などのいくつかの決まったルーチンを含むものである。ＧＡＲＧＯＹＬＥのユーザーは，しばしば　アセンブラ・ステートメントを組み込むことも予想されるし，＜action＞言語全体は　〔wir 66a〕の高レベルの機械語にもよく似ているように思われる。ＧＡＲＧＯＹＬＥは　その製作者が　複雑な言語〔 Gar 66 〕をインプリメントするのを助けるために用いられてきているが，広く利用されるのは　もう少し何かすっきりした設計と　もう少しよい出版物が必要である。

GilbertとMc Lellan〔Gil 67〕によるＴＷＳは，Gilbertのシンタックティックなアイディアを基として，ＵＮＣＯＬ〔 Ste 61 〕の概念を復活させようと試みたものである。ソース・プログラムは，機械に独立な言語ＢＡＳＥに翻訳される。このＢＡＳＥという言語は，多くの機械語に翻訳できるものである。最初の翻訳は　いくつかの決まったデータ構造の約束を持つマクロ

・アセンブラ言語である中間言語で記述され，第２の翻訳は　ＴＭＧのようなストリングの形で書かれる。この論文には　沢山のよいアイディアが含まれているが　これらはこの章の他のものとそれ程異なっていない。しかし　いくつかの悪いアイディアがユニークに説明されている。

machine　oriented でありながら，機械には　独立な言語であるというＵＮＣＯＬの概念は，いつも　言語とコンピュータの多様性において　挫折してきている。Gilbert と Mc Lellan は，ＢＡＳＥにおいて新しいオペレータを定義することと　ＢＡＳＥから機械語への各トランスレータにそれを無条件に送ることを許すことによってこれを避けようと試みている。このことは　機械に独立なことを示しているが，どのマクロを選ぶかという基本的な問題は　触れずに残している。この製作者たちは，また　彼らのシステムは"厳密に立脚している"という点を強調している。このことから　彼等は，そのシステムの姿全体を説明するプログラム言語の例として"$A^mB^nA^mB^nCCC$"という記号列の集合を用いる気になったのであろうと思われる。結局　まちがって置かれた厳密の典型的な例として，彼らは　ループと go to ステートメントを禁止することによってそのシステムから無限ループを除外したのである。この論文〔Gil 67〕の中で唯ひとつ参照されている他のＴＷＳの文献は〔Ir 61〕である。

Pratt と Lindsay によって開発されたＡＭＯＳシステム〔Pra 65, Pra 66〕は，ＴＷＳのリスト処理の概念をそのまま応用したものである。ソース言語は，中間言語（Ｉ言語）に変換（translate）され，この中間言語がＡＭＯＳによってインタープリートされる。このＩ言語は簡単なリスト処理語であって，記号列処理演算と荒っぽい形の算術 ── 例えば，"＊ＡＤＤ＊Ｐ１，Ｐ２，Ｐ３"は"Ｐ１とＰ２を加え，その結果をＰ３に入れる"ということを意味するを含んでいる。

このＩ言語は　最小の形に設計され，マクロの流儀で拡張できるように設計されたものである。シンタックスは　回帰的呼び出し（recursive call）に重点をおいて production language（第Ⅱ章B.5）の変形によってとり扱われ，セマンティックスはＩ言語で書かれる。ＡＭＯＳの最も面白い特徴は，プログラムのみならずデータ構造の変換も与えようとしていることである。ＡＭＯＳは　リスト構造のとり扱かいには余り成功しているとは言えないが，この問題は　もっと多くの注目を受けるのに値するものである。

〔参考文献〕

REFERENCES FOR III.A
Ab 66, Gar 64, Gar 66, Gil 66, Gil 67, Kirk 65, Ir 61, McCl65, Met 64, Pra 65, Pra 66, Rey 65, Sch 64, Schm 63, Schor 64.

B. Compile－Compiler

TWSの中でこの集合に属するもののきわ立った特徴は，トランスレータの作成のpostsyntacticな面の多くを自動化しようと試みている点である。このようなシステムはcompiler writing systemと呼ぶのが適当であろう。というのは metalanguageのプロセッサと同様に，翻訳時にも 実行時にも常駐する有意味なプログラムを含んでいるからである。この節のプログラムは前に述べたものの大部分のものよりも もっと複雑であって，同じようなタイプの仕事に前にたずさわっていたことのある人でないとインプリメントすることに成功したことのないものである。

次にFSL（Formal Sematics Language）〔Feld 66〕に関する論文からの抜すいを載せる。これは 基本的な原理の概観を述べ，コンパイラ・コンパイラに関する我々の議論に対する適当な序論として役立つであろう。

ある言語Lに対するコンパイラが要求された時には，次のような段階がふまれる。最初にsyntactic metalanguageで述べられた£の形式的シンタックスがシンタックス・ローダーに与えられる。このプログラムは，言語£の中のプログラムの認識と解析をコントロールするテーブルを作り出す。次に semantic metalanguage で書かれている£のセマンティックスがセマンティック・ローダーに与えられる。このプログラムは もうひとつのテーブルを作り出すが，こちらのテーブルは £の中のステートメントの意味の記述を含むものである。さいごに 図12の二重の線の左側のものがすべて捨てられ，£に関するコンパイラだけが残るのである。

結果として得られたコンパイラは，1個のLIFOスタックを用いるrecognizer に基づくtable－driven translatorである。このmain stack の中の各要素は2つのマシーン・ワードから成っている。ひとつはシンタクティックは構成要素であり，もうひとつは 対応するセマンティックな記述を持つものである。

特定の構成要素が認識されるとその semantic wordとsemantic tableが，トランスレータが何をしたらよいのかを決定するのである。

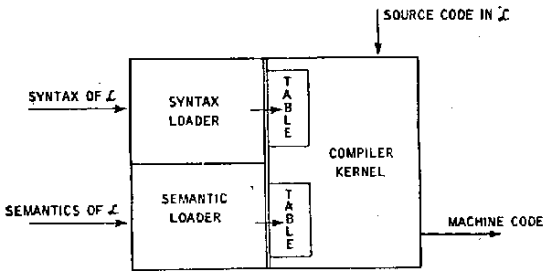


FIG. 12. A compiler-compiler

図 12 コンパイラ・コンパイラ

コンパイラの核心は，入出力code generationルーチン　およびすべてのトランスレータで使われるそのほかの機能を含んでいる。

B.1　ＦＳＬand Its Descendents（Feldman 〔Feld 66〕）

もともとのＦＳＬの仕事が直面した問題は，postanalytic（セマンティック）な処理を記述するための言語を開発することであった。適切なセマンティックmetalanguageによって，ソース言語の記述ができるかぎり自然であるようにできるはずである。他の人々が定義されているソース言語の意味を理解できるためには，それが読みやすくなければならない。効率のよい自動コンパイルができるためには，記述が十分厳密で完全でなければならない。そしてさいごに　そのmetalanguage は，特定のコンパイラの特徴に依存してはならないのである。

ＦＳＬで用いられるsyntax metalanguagc は，第Ⅱ章B.5で論じたproduction languageに近いものである。このシンタックス言語の中のステートメントは，nとラベル付けられたセマンティック・ステートメントを呼び出す"ＥＸＥＣ　n"という命令を含んでいる。このほかには　シンタックスとセマンティックスの相互作用は，main stack上のシンタクティックおよびセマンティックな記述の対を作っていることだけである。

semantic metalanguage ＦＳＬは，　この仕事の主な焦点であった。ここでは　そのいくつかの点を詳しく論ずることにする。ＦＳＬにおける考え方は　あとで述べるＴＲＡＧＥＮがcode のoptimizationを目的としたのと逆にmachine independence（機械独立）であるということであった。その計画は，コンパイラの核心にうめこまれたprimitives の大きな集合によって操作されるtranslation の機械に依存する面を持たせることにより機械独立であるようなmetalanguage を作ることであった。このmeta－language のステートメントは，（コンパイラ・コンパイラで）主にprimitive なルーチンの呼び出しから成る機械のコードにコンパイルされる。このアプローチを説明するのに役立つように例をいくつか挙げることにする。

シンタックス・フェーズがＲＥＡＬの宣言を処理中であり，スタックの第２番目の位置（ＬＥＦＴ２）の中で宣言された　identifier と共にセマンティック・ルーチン１を呼び出すものとする。

```
1：ENTER〔SYMB；LEFT2,（STORLOC|DOUBLE）, REAL, LEV〕；
    STORLOC←STORLOC+2
```

ここで　その変数についての記述は，シンボル・テーブルＳＹＭＢ内に置かれている。その変数についての項目はその名前，tagをつけたアドレス，ＲＥＡＬという語および現在のブロックのレベルである。最後に　ＳＴＯＲＬＯＣは，倍精度の変数に対しては２個のcell を割り付ける（allocate)ために２だけ増加する。

identifier が算術ステートメントの中でscan されると，セマンティックルーチン２が呼び出される。

２：ＩＦ　ＮＯＴ　ＬＥＦＴ１　ＩＳ　ＣＯＮＳＴＡＮＴ　ＴＨＥＮ　ＩＦ　ＳＹＭＢ
〔ＬＥＦＴ１，ＴＹＰＥ〕＝ＲＥＡＬ　ＴＨＥＮ
ＲＩＧＨＴ１←ＳＹＭＢ〔ＬＥＦＴ１，ＳＥＭＡＮＴＩＣＳ〕
ＥＬＳＥ　ＦＡＵＬＴ１

セマンティック・ルーチン２では，その identifier（ＬＥＦＴ１の中にある）が 定数であれば　このルーチンを終了する。もしそうでないときは　その identifier は 変数であり，シンボルテーブル内から探し出されねばならない。テーブルの参照（table lookup）はＦＳＬでは，

＜table name＞〔＜operand＞，＜position name＞〕

の形の独特なテーブル・オペランドを通してなされるものである。テーブル・オペランドについてのこの例では，その第１欄がＬＥＦＴ１の内容に等しい項目（もしくは row）を探すためにテーブルＳＹＭＢをサーチすることから始まる。次に　等しかった row の指定の位置（ＴＹＰＥ）が選び出され，string construct　ＲＥＡＬと比較される。これらが等しい時には，その変数はＲＥＡＬとして宣言されたものですべてがうまくいくのである。この場合には　ＳＹＭＢ内の一致した row の（tag をつけたアドレス）ＳＥＭＡＮＴＩＣＳが，その real 変数のセマンティックスとして代入される。その変数が　ＲＥＡＬ型でないか　テーブル中に全くない場合には，ステートメントＦＡＵＬＴ１が実行される。これは　コンパイル中のソース言語のプログラムのリスト上にエラー・メッセージをプリントするものである。

最後にそのシンタックスが，コンパイルされる加算（addition）を認識し　セマンティック・ルーチン３を呼び出すものとする。

３：ＲＩＧＨＴ２←ＣＯＤＥ（ＬＥＦＴ４＋ＬＥＦＴ２）

code bracket "ＣＯＤＥ（"および"）"は，それらの中のステートメントが翻訳時に実行されるのではなく，オブジェクト・コードにコンパイルされるべきものであることを示している。このステートメントの実行は，加算に対する一連のコードをコンパイルするために，スタックの第２と第４番目の位置にあるセマンティックな記述（semantic discription）を用いる code generating routine への呼び出しを作り出すものである。このセマンティックな記述とは　データの型，符号，index 属性，オペランドの現在の location を含むものであって，これらは　トランスレータの状態と相まって局所的によいコードを作り出すのに十分である。加算の結果はそれ自身式であり，そのシンタックスは　シンタクティックな記号Ｅをスタックの第２の位置においてあると仮定している　（図９のＴ１＋１行目を参照）。

ＲＩＧＨ２への代入は　結果のセマンティックに（例えばアキュムレータ内のＤＯＵＢＬＥ）そのセマンティックな記号を結びつける。

ＦＳＬシステムでは，ほとんどすべての要素（construct）をcode bracketの内側（実行時に行なうため）にも　code bracket　の外側（翻訳時に行なうため）にも，書くことができるのである。

semantic　metalanguage ＦＳＬは，コンパイラの作成者が　トランスレータを作る際に，さまざまなデータ構造を宣言し用いることを許している。例に示したtableやcellのほかにstack，mask，stringがある。このシステムは　翻訳時にも，実行時にも利用できる数多くの補助的なルーチン（例えば，フォーマットや，ファイル処理）を含んでいる。Formula　ＡＬＧＯＬコンパイラは　大部分ＦＳＬで書かれたもので，このインプリメンテーションの記録〔It 66〕はＦＳＬの長所と短所のよい研究になるものである。

ＦＳＬの弱点としては，いくつかの便宜の不足と多くの基本的な構造上の欠点のあることが挙げられる。便宜の不足とは　index変数，回帰的なサブルーチン，アセンブラ言語の埋めこみ，デバックの手段などのないことであり，その開発の際の主題ともなるひとつの課題である，後のシステムでは　矯正されてきているものである。構造上の欠点は，主に機械独立であることを保とうとすることの結果として起ったことである。

ＦＳＬシステムは，コンパイラ作成者の必要とするものが　semantic languageの機能で満される範囲まで有効である。しかし　これは，適切な概念の妥当な公式化が存在するところにおいてのみ可能である。したがってⅣ章Cに列挙したresearch problem（例えば　データ構造，ページング，並行処理）はすべて　どのＦＳＬの中でも問題である。ＦＳＬは　あるconstructを認識すると　ただちに作り出すべきコードを要求するものであるというのは，ひとつの一般的なまちがいである。code generationは　無期限にひき延ばすことができるが，現在動いているシステムは　グローバルなコードのoptimizationやマルチ・コンパイラに対して特に優れた機能は備えていないのである。

これらの問題は，現行のＦＳＬのような　いくつかのプロジェクトでとり組んでいるものである。しかしながら　機械独立な方法で得ることのできるコードのoptimizationには限度がある。どんなＦＳＬシステムであっても失敗するであろうという観念がある。すなわちいつも方法が形式化されるということを十分よく理解される以前に，用いられてしまうのである。それでもなおこのようなシステムは　大変に役立つものであり，metalanguageの表現をサーチすることは新しい方法を注意深く研究することにつながるのである。さらに　特定のインプリメンテーションは，通常まだ形式化されていないconstructを扱うための非公式な方法（例えばアセンブラ言

語）を含んでいるのである。

このほかに完成し終っているＦＳＬ型のシステムは，Lincoln Laboratory のＶＩＴＡＬ〔Mond 67〕だけである。ＶＩＴＡＬタイムシエアリングのもとで動くもので，主にシステムの特徴において　ＦＳＬとは異なっている。これらのことは　（この記述の中で用いられている）多くの著しい改善とあいまって　ＶＩＴＡＬをより使い易いものにしているが，反面　理論的な面白さの少ないものにしてもいる。より有意義な変更の中に，シンタックス・テーブルの大きさを約４分の１に減らし　スピードを増す実行可能な（executable）syntactic class name がある。テキストはすべて，dictionary pointer でつながったブロックに退避される。これは　行単位の編集や，再コンパイルの時間を約半分に減らすのに役立つものである。persistent storage とコンパイル時における実行の組合せは，インクリメンタル・コンパイラを作成する時に　助けとなるものである。ユーザーは　レジスタの割り付け（register allocation）で　かなりの融通性を与えられるが，もともとのシステムにおけるように　この責任を回避することを選ぶこともできる。小さいけれども　賢明で重要な変更は，production language にＴＥＳＴというsyntactic ＜action＞ を加えたことである。このＴＥＳＴは，セマンティックスによって　変数の集合に依存するものである。これはＢＮＦの伝統に違反するものであるが，あるトランスレータには必要なものであることがわかっており，その他のものにも多大な便宜を与えるものである。

ＦＳＬシステムは　Ｇ－20やTX－2などという一般的でない機械でインプリメントされたために，明らかに　ハンディキップをつけられている。このことを補うため，現在　ＩＢＭ360シリーズで別々に３つのインプリメンテーションが進められている。カーネギーのＣＡＢＡＬグループ〔Fie 67〕は，最小限度ＡＬＧＯＬを拡張したセマンティック言語を用いてマルチパス・コンパイラに対するシステムを設計中である。スタンフォードのGries〔Grie 67b〕のもとでもマルチパス向けであるが，特殊問題向きセマンティック言語を用いる仕事を行なっている。J. Curry によるLincoln Laboratoryの仕事は，多分，ＶＩＴＡＬによく似たものとなるであろう。これらのプロジェクトはすべてＦＳＬと次の節のＴＧＳの良いところを組びつけようと努力しているように思われる。

B.2　ＴＧＳ

（Plaskow と Schuman〔Plas 66〕，Cheatham〔Che 65〕）

ＴＷＳの研究における最も生産的なグループのひとつは，現在はApplied Data Research の一部となっている Massachusetts Computer Associates（ＣＯＭＰＡＳＳ）という　小さな

コンサルティングの会社であった。彼等のＴＷＳは　幾多の変更を受けたけれども，その基本的な観念と彼等の仕事の目的は　ほとんど変わることなく存在している。彼等はコンパイルすることを６つの過程として定義した。すなわち　語句（lexical analysis），構文解析，解析列の翻訳，optimization，コードの選択　および出力である。彼等の仕事の主な動機は，実行時の効率を追求することであった。ただし　そのほかの考えも　その時々で大切な役割を果してはいたのであるが。Computer Associates の現在のＴＷＳの関するこの仕事は，コンパイル時の各段階に対して　１個の言語ＴＲＡＮＤＩＲを用いている。ＴＲＡＮＤＩＲは，基本的には　算術部門，パターン・マッチングの部門（第Ⅱ章B.5）　および多くの組み込み関数から成っている。彼等の仕事の別の面は，ＴＧＳ内の拡張可能なコンパイラの計画に関して第Ⅱ章C.5 で論じている。

ＣＯＭＰＡＳＳにおけるＴＷＳ問題の初めの試みは　ＣＧＳと呼ばれるもので，彼等のこの仕事とは全く異なったものであった。彼らは　このアプローチをやめてしまったけれども，これは周期的に再認識されているように思われるので，ここで簡単にＣＧＳについて述べることにする。ＣＧＳシステムは，あとの解析で用いるような syntax tree を作り出す top－down recognizer をもとにしている。このフエーズに対する入力は，基本的には　ＢＮＦであった。第２のフエーズは，ＧＳＬという tree－matching の言語を用いて中間的なコードを generate することであった。実際のコードを選択する過程は，第３の言語ＭＤＬで書かれた。この仕事が放棄された理由は tree を作り出すのに時間がかかり，それに対してパターン認識を行なうのがむずかしいことがわかったためである。

ＣＧＳとは異なり　ＴＧＳシステムは，この節の他の言語と同じように　コンパイラのすべてのフエーズを記述するのに１つの言語を用いるものである。この言語ＴＲＡＮＤＩＲは，ＴＲＡＮＧＥＮインタープリータによって処理される interpretive code にコンパイルされる。もし図12のシンタックスとセマンティックスのローダーを組み合せると，ＦＳＬのモデルが　ＴＧＳにうまくあてはまる。事実　これらの２つの仕事の間には緊密なコミュニケーションがあり，それらは　著しい程度に一致してきた。しかし　そのコミュニケーションは完全とは言えなかった。２つの同時期のインプリメンテーションであるＴＧＳとＦＳＬとは，2,300 ヤードの間で　何の接触ももたずに実施されてきたのである。

ＴＲＡＮＤＩＲ言語は，ＦＳＬで用いられたシンタックス言語（第Ⅱ章B.5）と本質的には同じ　パターン・マッチングのサブセットを含んでいる。このＴＧＳの version は　さまざまのスタックを用いることができるとか，記号の同一性とは別な特性を照合することができるなどの一層柔軟なものである。パターン・マッチングの特徴は，いろいろなコードの optimization や

構文解析で用いられうるのである。

ＴＲＡＮＤＩＲのそのほかの特徴は，ＦＳＬのsemantic languageに酷似している。ＦＳＬにおける"semantic word"に類似したものとして，こちらには　各シンタクティックなconstructに関連づけられた"symbol description"（ＳＤ）がある。そのトランスレータによって用いられるために　宣言のテーブル，マスク　等々に関して，かなり精巧な機能がある。operatorに結びつけられた　これらのいろいろなstorage methodは，効率のよいコードをコンパイルするために必要な情報を記憶し　アクセスするのに非常に柔軟性のある手段を与えている。code bracketについてのＦＳＬの概念は，ＴＧＳでは　コンパイラ作成者が出力コードを作り出すのを助ける一連のsymbol manipulation primitives によって　おきかえられている。ＴＧＳコンパイラのオペレーションは，ほぼ完全な例を用いて記述するのがよいであろう。

次の例は〔Plas 66〕の中に述べられているミニチュア的な算法言語はℒtoから取ったものである。ここで選ばれた基本的なコンパイル方法は，ＣＯＭＰＡＳＳコンパイラ〔Che 66〕に共通な　テーブル状の中間コードを用いることである。

Ｚ ← Ｘ ＊ Ｙ

に対する典型的な中間コードへの変換は

① ＴＩＭＥＳ　　ＸＹ

② ＳＴＯＲＥ　　Ｚ①

である。中間コードは，後でアセンブルする時のために最終的な出力を生み出すcode selectionフェーズで　処理されるのである。

最初に，次のＴＧＳステートメントを考える。

…ＶＡＲＡＥ／／ＥＭＩＴ（ＳＴＯＲＥ，ＣＯＭＰ(1)，ＣＯＭＰ(0)）；

ＥＸＣＩＳＥ；ＴＲＹ（ＥＮＤＳＴ）．

このステートメントの（／／までの）左の部分は，＜variable＞（変数）と＜expression＞（式）というタイプのパターンであり，これは　main stack（ＳＹＭＬＩＳＴ）と比較される。一致するものが得られた時には，このステートメントの残りの（action part）が実行される。ＥＭＩＴというactionは，一致したスタック内の第０番目と第１番目の要素をオペランドとしてＳＴＯＲＥという中間的な命令を作り出すものである。結果として生じるsemantic description（ＳＤ）がないので，ＥＸＣＩＳＥというactionは　スタックから既に一致した２つの要素を削除するのに用いられる。最後にＴＲＹ（ＥＮＤＳＴ）というactionは，TRANGENにENDSTというラベルのついたパターンと一致するかを試すようにさせるものである。

乗算を認識するためには，もう少し複雑なルーチンが用いられる。

…VALS＊VAL／／PHRASE（SYMRES（TIMES，COMP(2),
COMP(0)））；
AESET；SYNTYP（COMP(0)）＝AE；TRY（AE1）

"$＊"がterminal symbol"＊"を示すことがわかれば，このステートメントの左の部分は明らかである。SYMRESというactionは，同じパラメタについてEMITを実行し，SDとしてその値を返すようなルーチンへの呼び出しである。このSDはPHRASEのパラメタとなってそのスタックの一致した部分を置き換えるために用いられる。AESETというラベルのついたactionは，新しい一番上の要素のシンタクティックなタイプにAEという値を代入するということを生じる。最後にTRY（AE1）というステートメントは，さらに 式の処理へ進むものである。

以上2つのTGSステートメントは，逆の順序に実行されれば Z←X＊Yを中間言語にコンパイルするものである。実際の世の中では 典型的なステートメントは，テーブルのオペレーション，sting commands，条件およびもっと複雑なそのほかのTRANDIRの構成要素を含んでいるのである。さらに トランスレータを読みやすく，また書きやすく改善するかなり複雑な＜procedure＞という特徴もあるのである。

どのような場合でも中間コード自体は，code selectorと呼ばれるTRAGENルーチンのほかの集合により処理される。これらは，前述のsyntax routineと同じ形で書かれるものである。例えば

／／TIMES INMEM INNEM…
LOADMQ（XM＋1）

このステートメントは 照合するための記号ではなく，スタックのエントリのpredicate のINMEM（in memoryを意味する）を含むパターンを持っている。（delimiter"／／"と"…"は，そのパターンが中間コードのスタックに対して照合されるべきであることを示している。）LOADMQというサブルーチンは，パラメタとして スタックの第2番目のオペランドに対するポインタを付けて呼び出される。このユーザーの書いたルーチンは，必要ならばLOADMQ命令をアセンブルし 現在MQレジスタにひとつのオペランドが在ることを考えて スタック上のSDを調整する。同じようなルーチンが適当な乗算をコンパイルするのに用いられる。その結果はアキュムレータにあり，TRANGENは 結局はこのステートメントに一致するのである。

／／STORE＊＊ ＊INAC…
IF SIGN（SYMBOL（ACHOLDS））THEN

```
EMIT(CHS);
EMIT(STO, ARG(1));
LINE(TEMPS)=0;
ACHOLDS=0;  MQHOLDS=0;
TO(STEP).
```

このパターンに含まれる"**"は 常に一致するものであることを示し，"*"は間接的な参照を意味している。ACHOLDSと記されるアキュムレータ内のオペランドが負である場合は"補数を取る"（CHS）命令が出される。置き換えるべき変数に関してテストをしない場合はいつもストア命令が出される。これに続くactionは，temporaries を取り返し ACとMQのレジスタを解放してトランスレータの状態に影響を与える。最後に STEPというラベルのあるaction に制御を移し，中間コードを通して続けていく。

TGSシステムは既にいくつかのコンピュータについてインプリメントされ，多様なコンパイラの製作に用いられている。そのコンパイラの作成者たちは 既に専門家となっていて，形式的なシステム内に留まっている必要がなくなっているのである。TGSの使用は，かれらが それを商業用コンパイラにひきつづき用いていることから，COMPASSにとって 十分価値のあるものである。つい最近の〔Che 66〕によれば，Cheathamは 多分（meta＋meta）プロセッサによってTRANDIRのprocedureに変換されるべきものであることを意味する宣言的(declarative) metalanguage £D を用いることを提唱している。

言語 の変換は，WirthとEarlyの機械的なconstructorの組み合さった概念に基づいている（第Ⅱ章Bを参照）。より強力な言語を考慮して （タイプのチェックなどの）predicateを添け加えたり，宣言的なシンタックスに対しても任意の計算を行なうことができる。最後にシンタックスの規則に付いている中間コードを出力するための規則が存在する。宣言的な言語（declarative language）は まだ インプリメントされていないが，cheathamは それがTRANDIRコンパイラの最初の形式化に役立つことが証明できたと主張している。このことは多分事実であろうが，procedualな形への変換が 現在 機械的な処理でないということが予想される。さらに ユーザーとしては，£D の方がTRANDIRよりも多少とも複雑でないとは考えられない。

TGSとFSLの主な違いは，企画 目的において端的に現われている。すなわち TGSはコンパイラ作成者からより詳しい情報を得ることによって，より柔軟性のあるものとなっている。スタンフォードのGriesとカーネギーのFierst〔Fie 66〕の仕事は，簡単なcode bracket ステートメントとマルチフェーズ処理を許すことによって双方の長所を備えようと努めている。

VITAL〔Mond 67〕と最も最近のTGS〔Plas 66〕は相互作用的で，複雑なトレース，編集およびデバッギングの特徴を持っている。

B．3　CC

（Brooker，　Morris 他〔Brook 67a，b，c〕）

マンチェスタ大学で始められたCC（Compiler−Compiler）のプロジェクトは最も古く，また最も孤立したTWSの仕事のひとつである。CCシステムは　既に時々動かされ，いくつかの算法言語をインプリメントする〔Cou 66，Kerr 67〕のに用いられているけれども，発行された書類が不十分で　CCは一般的には理解されていない。

このCCの仕事は，セマンティックスの問題に中心を置いている。構文解析は，memory と1記号前を見ること（one symbol look−ahead）を伴なった top−down である。構文解析の結果として，セマンティックス・フェーズで用いられる完成した syntax tree を得る。もちろん　これは処理が遅く，非公式には他の方法の用意がしてある。我々はここで，彼等の記法を勝手に変えながら正式な操作を示すことにする。

シンタックス・フェーズへの入力は　BNFに類似しており，これに（option であることを示す）angle bracket 内に出現しうる"？"と（左回帰を置き換える）repeat operation "＊"をつけ加える。次のステートメントは，算術和を取る代入ステートメントのシンタックスを指定するのに用いられるものである。

1. FORMAT〔SS〕=＜variable＞←＜sum＞
2. PHRASE＜sum＞=＜sign？＞＜term＞＜terms＞
3. PHRASE＜term＞=＜variable＞｜＜number＞｜(＜sum＞)
4. PHRASE＜terms＞=＜sign＞＜term＞＜terms＞｜＜empty＞

行1は　format definition と呼ばれ，行2～4の補助的な phase definition を利用するものである。SSは　この format のクラスを指定するもので，これについては後で述べることにする。format definition も phase definition も共に top−down recognizer の"production（生成規則）"として用いられる。そのちがいは，format definition に対応する中間的な tree が完全に作られると　関連する format（セマンティック）routine が　それを処理するために呼び出されることである。行1に関連する format routine は　次のように書かれうる。

with Line 1 would be written as follows：

5. ROUTINE〔SS〕≡＜variable＞←＜sum＞
6. Let ＜sum＞=＜sign ？＞＜term＞＜terms＞

7. ACC←<sign ?><term>

8. L1:GO TO L2 UNLESS<terms>=<sign><term><terms>

9. ACC←ACC<sign><term>

10. GO TO L1

11. L2: STORE ACC IN<variable>

12. END

行5は，このroutineが行1のformatに関連し，そのシンタックスが行1に一致した時に呼び出されて 適当な中間的なtreeを作り出すということを示している。行6は<sign ?>，<term>および<terms>の値として，中間的なtreeからdescriptorに代入する。アキュムレータをloadするコードがコンパイルされたあとで(行7)，その<sum>が 二つ以上の<term>を持つかどうかを知るために このtreeを調べる。もしなければ このルーチンはストア命令をコンパイルし(行11)，そしてこのルーチン終了する。もっと複雑な<sum>については 行8，9，10でとり扱かう。実際のコードの出力は インプリメンテーションに依存し，通常は 簡単な記号列の処理ルーチンでとり行なわれる。

CCにおいて用いられるステートメントには，basic (BS)，master (MP)およびsource (SS)の3つのクラスがある。BSという部分言語は FSLやTGSのsemantic sublanguageに対応するもので，これは算法言語におけるcode generation，リスト処理，語句の解析ルーチンを含むものである。これらのBSステートメントは，さらに前にコンパイルされているステートメント(precompiled statement)(例えば 上の行6，8，10，12)と，FORMAT〔BS〕ステートメントとそれらに関連するROUTINEによって定義されるBSステートメントのトランスレータ指定のcompilation(例えば行7，9，11)の2つに分類できる。BSステートメントはformat routine内のみに出現できる。

MPのクラスのステートメントは FORMAT，PHRASE，ROUTINEステートメントなど自体(行1～5)および編集やシステム・ダンプ命令を含んでいる。これらの構成要素は いずれもformat routine内に出てきてはならない。最後のステートメントのクラスSSは，ソース言語のステートメント自身を含むものである。これらは組み合わされて，BSとMPステートメントで 事実 第Ⅲ章Cの意味で強力な拡張可能なコンパイラにCCをするのである。CCシステムはもともとこのようにして働くように設定されてはいるが，実際には 多少異なっている。

例えば FORTRANの定義をFORMAT〔SS〕とROUTINE〔SS〕のステートメントの集合として書き，CCがこれらをテーブルにコンパイルするとする。次に CCのこの

更新されたコピーを　伝統的なＦＯＲＴＲＡＮコンパイラを生み出すようなＳＳステートメントだけを受け取るようにＣＣをセットしたスイッチと共に記録する。この方法を用いるとＣＣは，スタックではなく中間的なtreeによって，シンタックスとしてのＦＯＲＭＡＴおよびＰＨＲＡＳＥステートメント，そしてセマンティックスとしてのＲＯＵＴＩＮＥを結びつけた形として　図12のモデルのようになりうるのである。ＣＣがＦＳＬやＴＧＳのように中間的な記号に対してdescriptor　をとり扱かう機能は持っていないということに注意すること。ＣＣはtop－down recognizer　を用いているので，構成要素は　それらが処理されるにつれて用いられる。これは中間的なdescriptorを削除するが，これによってＡＬＧＯＬコンパイラをひとつの大きなＲＯＵＴＩＮＥとして書かねばならないようになるのである。

ＣＣグループは最近，かれらのシステムの使用と実行に関する報告書を作っている。これらはＴＷＳを手書きのコンパイラと比較するという始めての試みを含んでいる〔Brook 67b〕。

Brookerは　1年以内で，Atlas Autocode compilerの手書きによるコーディングの　スペースに関して1.6倍，時間に関して1.7倍まで減らすことを可能にした。これらの結果は　より多くの情報がないと解釈することはむずかしいけれども，多くの人々が想像している程　コンパイラ・コンパイラが　必らずしも莫大なスペースと時間とを要するものではないということを示唆するものである。このことはKerrの結果によっても言うことができる〔Kerr 67〕。

ＣＣは，ＡＬＧＯＬのような言語であるAtlas Autocodeの中に首尾よくうめこまれてきた〔Brook 67a〕。ＳＰＧ（System Program Generator）と呼ばれる。このシステムの応用はマンチェスターのMorrisによって　現在開発中である。ＳＰＧは，その基礎となるメカニズムに関する知識を持っているシステム・プログラマを対象としている。ＣＣのインプリメンテーションは　現在Atlas　ＫＤＦ－９，Ｇ－21に在り，Carnegie－MellonではＩＢＭ360／67で仕事が始まっている。

〔参考文献〕

REFERENCES FOR III.B

Design: Brook 60a, 62a, 63, 67b, 67c, Che 64a, 64c, 65, Cou 67, Feld 64, 66, 67, Fie 67, Grie 67b, Mond 67, Mor 67, Nor 63, Plas 66, Ros 64a, War 61, 64.

Uses: Brook 67a, 67b, Cou 66, It 66, Kerr 67, Nap 67, Rov 67.

C. Meta Assemblers and Extensible Campilers

これらの形のＴＷＳは，これらが共にマクロの概念を高いレベルのプログラミング言語に拡張しようとしている点で類似している。マクロ・プロセッサの基本的な考え方は，ある記号をそれらに結びつけられているテキストの一部で直接おきかえるということである。ほとんどすべての最近のアセンブラは　大変複雑なマクロの機能を備えているけれども，この考え方について最もよく書かれたものは　Strachey〔Str 65〕およびMooers と Deutsch〔Moo 65〕による一般的な論文に載っている。メタ・アセンブラと拡張可能なコンパイラ（meta assembler と extensible compiler）は　どのようにしてマクロを高いレベルの言語に拡張するかという点で，異なる二つの概念に基づいているのである。すなわち　メタ・アセンブラは　コンパイラをアセンブラの特殊な場合と考えて近づいて行き，一方　拡張可能なコンパイラは　標準的なコンパイラにテキストの置き換え能力を加えたものとして近づいて行く。これらのアプローチは共に　広く普及しているものであって，ここで考えているものよりも後で多くの論文が発表された。そのためこれらは，この節の終りに列挙しておくことにする。

C.1　General Discussion and METAPLAN (Ferguson〔Fer 66〕)

Ferguson の論文は　カルフォルニア州サンディマスにあるＡＣＭ Programming Language Conferenceから 1965年に提出されたもので，メタ・アセンブラについての優れた紹介を含んでいる。その基本的な考え方は，すべてのアセンブラが　多くの性質を共通して持っていることを観察することから生まれた。

シンボル・テーブルや location counter やマクロのようなものをとり扱うために，procedure を作ることによって　特定のアセンブラの作成のスピードを上げることができる。特定の機械に対するアセンブラを組み立てるためには　ワードの大きさ，数値の表現　その他を指定するはずである。各機械についての出力は　format statement を用いてプログラムされ，それは relocation やシンボリックなデバッグ情報を容易に含みうるのである。このようなシステムはアセンブラを書くのに便利で　全く有用なものに思われる反面，どのようにしてこれをＴＷＳに拡張するのかは　明らかではないのである。

ＴＷＳとしてのメタ・アセンブラの使用は，前に述べた　コンパイラというものが　マクロ・アセンブラの特殊な場合であるという前提に基づくものである。この前提に対する論議は，マクロ対高レベル言語の論争の生まれ変りのような感がある。マクロ・アセンブラ側は，保守の側に立ち　数においてまさり　そのために議論上ではより激烈なのである。そしてこの議論は，アセ

ンブラの定義が定まっていない事から　さらに複雑になるのである　（〔 Fer 66〕 の論文を参照されたい）。この例は，我々の論文の目的からすれば　ひとつ掲げるだけで十分であろう。

Fergusonは，メタ・アセンブラが　コンパイラに似たステートメントを　いかに操作するかを次のように表わしている。

IF F(A) PLUS 5 EQ G(B) GO TO L

彼は，many−many macro とよばれる方法を用い　IF, PLUS, EQ, GO, TO を(prafix)オペレータとして定義している。このmany-many macor は　テキストの置き換えが行なわれている間，state情報を用いたり　更新したりする性質を持っている。さらにこれは，マクロの一般的用法における主な問題 —— すなわち，アセンブルの過程におけるグローバルな性質（state情報）をいかにして効果的に使用するかという問題——に取りかかろうとするものなのである。many−many macroはかなり柔軟性に富んでいるので　現在知られているいかなるコンパイラをもインプリメントすることができるが，本当の問題は　このmany−many macroがそうすることがよい方法なのかどうか　ということである。これに対する答は　state情報を記録し　利用するメカニズムに依るもので，これらはこの論文には論じられていない。

C.2　PLASMA

（Graham と Ingerman〔Gra M 65〕）

Graham と Ingermanの　メタ・アセンブラの仕事は，置換および接合（binding）の問題にその中心を置いている。彼等は，Halpern（次節で述べる）よりはシンタックスに重きをおいていない。なぜならば　彼等は，syntax−directed を彼等のシステムにおけるコンパイラ作成のための糸口と仮定しているからである。（〔 Ing 66〕を参照）

このメタ・アセンブラに対する基本的な入力は，リストのリストである" line"である。第1のリストは一般化されたlabelで，これには任意の記号，それらがactiveである高いレベルの数値，同じくそれらがactiveである低いレベルの数値が含まれる。第2のリストはオペレーションを含み，第3のリストはオペランドを含むものである。入力はtreeに変換され，tree構造のもとで置換が行なわれる。記号的（Symbolic）あるいは　数値的（numeric）な置換を行えるようにするようにすることによって，テキストの置き換えとアセンブラの機能とを結びつけているのである。

彼等は　この研究をCherry HillのRCAで続けており，恐らくこれに関する論文を　再度発表するであろう。彼等の現在の仕事は，さらに精巧な置換処理を含んでいる。しかしまだ彼らのシステムをどのようにしてコンパイラの基礎として使用するかという点を明らかにするには到っ

ていないのである。

C.3　XPQP　（Halpern〔Hal 64, 67c〕）

XPQPシステムは　IBM 7090 でインプリメントされたもので，よい内部仕様書が作られているものである〔Hal 67c〕。少なくともひとつの言語ALTEXT〔Star 65〕は，このシステムでインプリメントされている。マクロの呼び出し形式が可能であることを示すために，ALTEXTの中で書かれているプログラムから2，3の例も掲げることにする。

```
DO THRU LAB1 I=1 TO T-1 BY 1
IF 1 CHAR AT NAMES I IS EQUAL TO NAMES I+1, GO
  TO LAB2
IF U IS LESS THAN 3, GO TO PUT
IF MATCH GO TO (LAB1, LAB2, LAB3), I
COPY 11 CHAR CARDS 1 TO REJECT 1
```

この言語のステートメントはどれも，IBM 7090のアセンブラ 言語で書かれたマクロに対する呼び出しである。従って　プログラマは自由に，アセンブラ言語を高レベル言語のステートメントの中に使うことができる。XPOPの次のような性質から，上の例のようなステートメントをとり扱うマクロを定義することができるようになるのである。

1. マクロは普通　ステートメントの最初の語（DO，IF，GO，TO）によって認識されるのであるが，ある場合には　マクロの名前がlineのどこにでも出現しうるのである。
2. 各々のマクロの定義の中で　そのステートメントの残りの部分を処理する際に用いられる句読点（punctuation）をそのマクロのパラメタとして定義する事ができた。
3. 各々のマクロの定義の中で　そのステートメントの残りの部分の処理のために，noise word（無視されるもの）とkey word（パラメタを決定するのに用られるもの）を定義することができる。
4. 特定のlabelが現われるまで，ひとまとまりのコードをアセンブルするのを延期することができる。（これは上記のDOステートメントに対する命令をgenerate するのに用いられる。）
5. マクロの定義の中の命令は，たとえばnameのグローバルな属性や条件付きのアセンブラの命令をチエックするような場合はアセンブル時に実行され得る。このことによっては　その命令が実行される度ごとにそれらをアセンブルしなければならないということを除けば，（通常のごとく）コンパイラをアセンブラ言語で作成することが可能になるのである。

6. このシステムは多くの有用やトレースやデバッグを助ける機能が備わっている。

ＸＰＯＰはコンパイラ作成の道具としての見ると，いくつかの短所を持っている。そのひとつは，すべてがアセンブラ言語かあるいは　前以って定義されたマクロで書かれねばならないことである。第２に，変数の属性を入れておくシンボル・テーブルなどをインプリメントする便宜がなく　アセンブラ言語でexplicitに書きあらわさなければならないことである。ＡＬＴＥＸＴはnameを正しく使っているかどうかというチエックはほとんど行なっていないのである。第3には　ＸＰＯＰシステムは，算術式に対して組み込みのコンパイラを持っているけれども　これはすべてが浮動小数点命令であるか　あるいはすべてが固定小数点命令の場合（スイッチに依存する）にのみ　コンパイルを行なう。したがって　型のチエックも行なわないし，mixed modeの式を用いることはできないのである。最後に　高水準の構造を持つＡＬＧＯＬのような言語は容易にはインプリメントできない。というのは　マクロの呼び出しは，簡単なやり方ではnestさせることができないからである。十分なマクロの特性が用意されているかどうか確信できないと言っても過言ではないのである。ある言語に対して新しいステートメントを付け加えることは必然的に他のマクロの特性をもたらしうる。丁度　上記の性質4がＤＯステートメントの処理のためにインプリメントされると同時に。

Halpernは，彼がＡＬＧＯＬのような言語をインプリメントするために用いる他のコンパイラ作成の道具を置きかえるつもりはないと述べている。彼はそのプログラムが英語のようである処理言語に興味を抱いており，彼のシステムはこのためには　適当なものであると信じているのである。

Helpernは最も楽観的でありメタ・アセンブラの支持を言い続けている。メタ・アセンブラにおける彼の仕事は，ＸＰＯＰによって自然言語に"近い"ものをインプリメントすることができるという彼の声明によって，自然言語プログラムに関する彼の議論の余地のある主張に結びつけられている。

Halpernの論文〔Hal 67b〕は，ＸＰＯＰ型のシステムを念入りに擁護するものである。彼はマクロシステムの<オペレータ><オペランドの記号列>という記法は　自然言語や数学的な言語とは反対であるが，プログラミング言語の標準的なシンタックスであると主張している。Halpernはまた，プログラミング言語の研究を３つの部分に分けている。すなわち　関数的（マクロ），記法的（句読点命令の変更　など）　そして　モードに関するもの（アセンブル時の実行）である。

C.4　Extendible Compiler ―― Basic Concepts

（McIlroy〔Mc Il 60〕以来）コンパイラ・システムにマクロの特徴を組み込むという試みが幾度もなされてきている。そのひとつのアプローチは，マクロ・シンタックスの形を保持しながら　コンパイラに似た　組み込みの特性を数多くつけ加えるというものである。例えばＳＥＴシステムは　入出力，記号の操作，テーブルのとりあつかい　それにリスト処理といった特徴を備えたskelton compilerを含んでいるものである〔Ben 64a〕。これらの組み込みルーチンは，ＴＷＳを作ろうとする時には　翻訳時のオペレーション（action operationという）に結びつけられるのである。もっと上首尾のアプローチとしては，拡張への基礎として高レベル言語のstructured syntaxを用いるものがあげられる。

多くの既存のコンパイラ（ＰＬ／Ｉを含む〔ＩＢＭ 66〕）は　簡単な形のマクロ展開を組み込んでおり，その最初のものはＪＯＶＩＡＬであろう〔Shaw 63〕。その最も原始的な形は，パラメタの代入を行なわず，純粋にテキストを置き換えるものである。例えば B5500 ＡＬＧＯＬでは，次のようなステートメントでマクロを定義することができ

```
DEFINE LOOP1 = FORI ← 1 STEP 1 UNTIL N #
```

あとで　次のようにステートメントを書くと

```
LOOP 1N DO A〔1〕← 0
```

展開されて次のようになる。

```
FOR I←1 STEP 1 UNTIL N DO A〔I〕←0
```

次の段階ではパラメタを持つマクロの定義が許される。この便宜は，数あるものの中で特にＡＥＤ－０コンパイラ〔Ross 66〕に含まれている。ＡＥＤ－０においては，次のようなステートメントでマクロを定義できる。

```
DEFINE MACRO LOOP(P1, P2)TOBE
FOR P1←1 STEP 1 UNTIL P2 DO ENDMACRO.
```

この場合　より短いステートメント

```
LOOP(I, N)A〔I〕←0
```

で，上記と同じ結果を得ることができる。

これらふたつの簡単なマクロの形は，いかなる高レベル言語につけ加えても有用な形であり一層複雑なマクロの方法に匹適するメカニズムの展開を考えることもできるのである。ＡＥＤ－０では　パラメタとして任意の記号列を認め　入れ子になった定義（nested definitions）を許しているけれども，条件付きアセンブルのような性質は　高レベル言語においては広く用いられてはいないようである。この理由のひとつは，コンパイラが　通常　テキストの構造に非常に依

存しているためである。次の２つの節では，コンパイラをマクロのやり方で拡張しようとする時に生ずる複雑な問題について述べることにする。

C.5　Definitional Extensions

（Cheatham［Che 66］）

高レベル言語の定義的な拡張（definitional extension）は，Computer Associatesのグループがとり組んでいるＴＷＳにおける最近の問題である。この仕事は，第Ⅲ章B.2に詳細にわたって述べたＴＷＳの仕事に照らして考えると　よく理解できる。

ここで主題となっているこの論文は，性急に書かれたもののようで　準備のために　いくつかの内部仕様書を参照している。これは　明らかにこの線に沿った最初の試みであり　これに続く論文で拡張され　明らかにされるであろう。ここで述べているコンパイラへの拡張は記述的な（descriptive）metalanguage $\mathcal{L}_D$（第Ⅲ章B.2）と一連のマクロの機能という二つの大きなカテゴリーに分かれるのである。

マクロの方法を用いる言語の拡張は　text macro，syntactic macro，computational macroの３つのカテゴリーに分けることができる。text macroはすでによく理解されているものと考え，さらに前述のものに類似したものであると仮定する。cheathamは，syntacticc macroをとり扱う時，コンパイラに対してマクロの概念を適用させるという問題にまじめに直面しはじめるのである。

ひとまとめに考えることのできる２種のsyntactic macroがあって，これら二つの基本的な特徴は　free formatとパラメタに対するタイプの指定（type specification for parameters）である。

```
LET N BE INTEGER
MACRO SQUARE N MATRIX MEANS 'ARRAY[1:N, 1:N].'
```

伝統的な＜オペレータ＞＜オペランドのリスト＞の形よりもfree format　の方が優れていることは明らかである。パラメタに対するタイプの指定は，条件付のアセンブルとよりよいエラー検出ができるようにするものである。このsyntactic macroの呼び出しは，特別な区切り記号（例えば%）によってはじまり　検出可能な終り（detectable termination）を持つことができる。特別な区切り記号の使用を避けるという第２のアプローチは，そのマクロの形をそのままトランスレータのシンタックス・テーブルに加えることである。このやり方でいくと上の例は

```
LET N BE INTEGER
SMACRO SQUARE N MATRIX AS ＜attribute＞ MEANS
```

'ARRAY〔1:N, 1:N〕'

のようになる。ここで< attribute > は，その強調している言語の定義におけるシンタクティックなタイプである。MACROSとSMACROSは共に，適当な実パラメタのdescriptorを代入することによってインプリメントされる（第Ⅲ章B.2を参照）。これらの方法は いずれもTRAGENのインプリメンテーションでは問題にならなかったが，いずれも もしまちがって用いられると最悪の結果をもたらすものである。

syntactic macroを論じる際に，cheathamは マクロの定義に"セマンティックス"を加えるという問題にふれている。これは，many¬many macroやメタアセンブラで用いられるアセンブル時のactionの類型である。このアプローチは不可能であるとするCheathamの結論は，ほとんどすべてをこの問題にかけてきたメタ・アセンブラのアプローチと比較すべきであろう。彼の解答は，多くのprimitiveなオペレーション（例えばテーブルの拡張）を与え，拡張可能言語の影にある 完成されたmetalanguage（TRANGEN）の存在を指摘することである。

マクロの拡張の第3のタイプは，computational macroと呼ばれるものである。このやり方によれば，宣言されたマクロから生ずる中間コードにおいて代入がなされるのである。マクロの本体に対する中間コードは 前以って（仮パラメタを用いて）作り出されているので，このやり方は それに対して中間コードが文脈に独立にコンパイルされるはずの構成要素（construct）に限定されるのである。この条件に合う場合には，computational macroは有用であり 効率のよい道具である。computational macroの簡単な例として最大4×4の左三角行列に対する次のような機能をあげることにする。

```
TAKE I, J AS INTEGER
MAP  M(I, J) = (11 − I) * 1/2 + J − 6 ;
```

ここでTAKEとMAPは，この言語における宣言である。

このコードは配列のアクセスのためのものであるから，これをソース・テキストの中に挿入することはできない。このcomputational macroの形はほぼ適切なものである。Cheatham が指適しているように computational macroは，コンパイラ作成者によって 配列をアクセスするコードを作り出すために用いられてきた。この論文では次節で述べるPerlisとGallerの論文に関する考察でも再び現われる問題である。アクセスの機能（accessing functions）についていくつかの例をあげている。ここで重要な点は，Cheathamは アクセス機能を記述するprocedualな方法を与えているのに対して，PerlisとGallerはprocedualでない記述から コードをgenerateしようとしている点である。

C.6　ＡＬＧＯＬ　Ｃ

（GallerとPerlis〔Gall 67〕）

これは，プログラミング言語の分野における二人の傑出した人物による　非常に長く　難解なそして重要な論文である。この論文には沢山の有意義な面があるけれども，ここでは　拡張可能なコンパイラについてのみとり扱かうこととする。その他の事項については，新しい研究領域の第一歩として　第Ⅳ章Ｃでとり扱かうこととする。

基本概念は，ここでも　高レベル言語にマクロのような機能をつけ加えることである。この目的のために彼等は，拡張するのに適当であると思われるＡＬＧＯＬ　Ｃ　と呼ばれるＡＬＧＯＬのversionを定義している〔Naur 63b〕。ＡＬＧＯＬ　Ｃ　を拡張したものはどれもALGOL Dと呼ばれ，そして　これらの中の任意のプログラムは　機械的にＡＬＧＯＬ　Ｃ　のプログラムと等しいようにreduceできるのである。その拡張は，トランスレータのシンタックス・テーブルにつけ加えるCheathamのＳＭＡＣＲＯＳにやや似ている構成要素を通じて　成しとげられるものである。PerlisとGallerは非常に複雑な方法で　マクロ処理を行ないたいと思っているので，彼等は2，3のきまったカテゴリーにおいてしか　再定義を許していない。基となっている言語のＡＬＧＯＬ　Ｃ　は，配列をとり扱かったり　直接に拡張可能性に関係していたりする多くの特徴を備えている。

拡張可能性に関する特徴の中に，locationとvalueの間の変換に対する演算子がある。すなわち

(a)　整数型の結果を伴なう単項演算子：

loc of　x

ここで，xは＜procedure identifier＞，＜variable＞，＜array identifier＞のいずれかである。loc of xは，基本的にはxという値を含む語のアドレスである。

(b)　ある＜procedure＞，＜variable＞　もしくは＜array＞の"アドレス"を表わす，左のオペランドが＜type＞で　右のオペランドが整数の式であるような　二項演算子：

＜type＞　vc of　x

＜type＞　pic of　x

これらは，それぞれ" value contents of "　と　" procedure identifier contents of "を表わしている。したがって　もしxが＜type＞realの変数であれば

real vc of（loc of x）＝x

である。

(c)　locationとvalueの概念は，適用演算子（application operator）⊕を用いて

＜procedure＞に拡張される。厳密なシンタックスの変更は　配列の約束と密接な関係があるが，＜primary＞と＜function designator＞　の修正した定義はその意味を示しておかなければならないであろう。

＜primary＞::＝＜unsigned number＞|＜variable＞|＜function designator＞|（＜arithmetic expression＞）|loc of ＜procedure identifier＞|＜type＞vc of＜arithmetic expression＞

＜function designator＞::＝＜procedure identifier＞⊕＜actual parameter part＞|（pic of＜arithmetic expression＞）⊕＜actual parameter part＞

このようにしてaddress variableとほとんど同じ方法でprocedureの名前をとり扱かうことができ，例えばprocedure arrayなどを作ることが可能である。ＡＬＧＯＬ Ｃ を作るためのＡＬＧＯＬに対するこれらの追加は　余分なメカニズムのほんの一小部分を構成するにすぎず，その大部分は　ＡＬＧＯＬ Ｄ の多様性のある形の中に埋めこまれている。

すべてのＡＬＧＯＬ Ｄ という言語は，ほとんど同じシンタックスを持っている。すべてのＡＬＧＯＬ Ｄ に対する共通のシンタックスは　＜type＞，＜算術式＞，＜論理式＞，＜代入ステートメント＞を，それらのシンタクティックなタイプに対する特別な形の定義を可能にする規則の集合で置きかえることを除いては　ＡＬＧＯＬ Ｃ に同じである。新しい定義の導入は，任意のブロックの始めの部分にある一連の宣言文として出てくる。この過程についての詳しい説明は非常に複雑なので我々は，例を使って概略を示すことにする。

基本となる趣旨は，新しいデータ・タイプとそれらに関連する演算子の定義を認めることである。これらの演算子に対する記号を見つけるという問題は，太字の英大文字を使うことで解決する。演算子順位文法を仮定する（第Ⅱ章B.1）ことによって，MAD〔Ar 66〕におけるように新しい演算子の順位を既知の順位をもつ演算子に関連づけて定義することができる。演算子に関する残っている問題には　データのタイプがあるが，これについては数センテンスをさくことにする。

新しいデータタイプはＡＬＧＯＬ Ｃ の表現　あるいはmeansステートメントによって，これ以前に定義されたタイプを用いて定義される。これは　仮パラメタを含みうる。この仮パラメタは，もし　それが存在すれば　後の処理で重要な役割を演ずるもので，例えばmatrix（u，v）means array〔1：u，1：v〕などのようなものである。

次に　contextステートメントの集合の中では，演算子とタイプの情報を結びつけることができる。contextステートメントは，ある演算子に対して　そのオペランドのデータ・タイプとそ

の結果とを記述する。それはまた（最終的には）適当なＡＬＧＯＬ Ｃ のテキストに還元できる＜記号列＞を含んでいる。次の例は（擬似）ＬＩＳＰの定義を示すものであるが，これらの概念を明らかにするのに役立つものと思われる。

基本的なＬＩＳＰの predicate atom と eq は，論理的な procedure として定義されていると仮定する。

```
Boolean procedure atom(x); list x;
  atom := cdr x = 0;
Boolean procedure eq(x, y); list x, y;
  eq := car x = car y ∧ atom(x) ∧ atom(y);
```

次の定義は，名前の構造としてのリストを作るのに用いられる。

```
(1)  list means integer array[1:2];
(2)  cons ≒ *;
(3)  car ≫ cons;
(4)  cdr ≒ car;
(5)  of ≪ cons;
(6)  list a cons list b ≡ list 'list(a, b)';
(7)  car list a ≡ list'a[1]';
(8)  cdr list a ≡ list'a[2]';
(9)  loc of list a ≡ integer
(10) integer a := list b ≡ integer'a := loc of b';
```

ステートメント(1)は新しいデータ・タイプ list を 二つの要素を持つ整数型 の配列として定義する。ステートメント(2)から(5)は，ＬＩＳＰの４つの演算子の間の相対的な順位を述べている。ステートメント(6)～(9)は式を定義する。例えば (7)は list "a" の car がモデル化している配列の第１要素であることを定義し，それがひとつの list としてとり扱かわれるべきものであることを指定している。ステートメント(10)は，ある list に整変数を代入する代入ステートメントを定義している。この論文には，このほかにリスト構造に関するＥＶＡＬ関数と各種の sequencing operator の定義も出ている。

この例は，ＬＩＳＰに対しても Galler－Perlis のシステムに対しても正当なものとは言えない。というのは 彼等のシステムの全構想は，一番外のブロックに置かれた同じような定義の集合によって定義されるＡＬＧＯＬ Ｃ を備えているからである。それに続く各ブロックでは，このトランスレータは 局所的な定義の集合を用いるため type table と context table とを作

り上げる。局所的なＡＬＧＯＬ Ｄ のテキストの実際的な処理はきわめて複雑である。これは，文脈が回帰的であること．および ＡＬＧＯＬ Ｃ のテキストが局所的に定義されたテキストに挿入できるということに起因する。この論文におけるその論議は，ＡＬＧＯＬ Ｃ のコードの作成に加えて 計算をオプティマイズしたいという希望によって，さらに複雑になっているのである。

我々は Galler とPerlis の論文の意図を，首尾よくではないにしても慎重にゆがめてきた。彼等は，配列やとりわけ行列の計算におけるスペースを節約することにも 関心を抱いている。彼等が自分自身で論点を分類したのであれば，彼等にとってすべての面で好ましかったであろう。既に述べたとおり この論文は，拡張可能なコンパイラのほかにも重要な問題について論じている。我々の論題へのその貢献は，実用的であるというよりは理論的である。彼等は，非常に複雑なマクロ処理が可能であり それが算法言語に実質のある変化を与えることを示している。しかし 翻訳時の効率のよくないこととプログラミング・エラーに対する感受性が，その適用を大幅に制限するのであろうということが考えられる。さらに どの位の頻度で 高レベル言語を変える要求があるであろうかという，一般的な疑問が存在する。これについては第Ⅳ章Ｃで再びとり上げることにする。

〔参考文献〕

References for Ⅲ．C

Benn 64a,64b Brook 60b,Brop 67,Che 64a,66,Fer 66, Gal 67, Gar 65, GraM 65,Hal 67a,b,c,IBM 66,Lea 66, Mcll 60, Mea 63, Moo 65, Star 65, Star 65, Wai 67.

Ⅳ Related Topics and Conclusions

A. Other Uses of Syntax-Directed Techniques

ＴＷＳの開発が始まってまもない頃から syntax-directedな方法が，広範で種々な問題に対して利用可能であったことが観察されていた。syntax-directed なアプローチは，プログラムに対する入力の形（form）が その内容（content）の重要な一部を含む場合はいつも 考慮されうるのである。syntax-directed な方法についての個々の適用例は，詳しく書きたてられたりはしない傾向がある。ここで扱っている適用例は，主として個人の知識に基づいているものであり 代表的なものではあるが，確かに理解しにくいものである。

第Ⅲ章で述べたＴＷＳは 他の用途に用いられる場合，一緒に使われるものに応じて いろいろに容易に変化する。syntax-directed symbol processorは，最も柔軟性に富み かなり広範な用途に用いることができるのである。このようなシステムのひとつであるＡＥＤ〔Ross 66，67〕は，最初から一般的な目的のプロセッサ（general purpose processor）として設計されたものであった。しかし それに対する姿勢とその術語が特殊であったために，ＡＥＤのプロジェクトは 他のＴＷＳの仕事程 効果を上げることができなかったのである。

ＡＥＤのシンタックスのフェーズは precedence（順位）の方法に基づいており，第Ⅱ章Ｂにあるものに類似している。タイプのチエックや手書きのシンタックス・ルーチンを加えることができる機能を組み入れることによって，ＡＥＤ parserは その根底に流れる理論を犯すという犠牲をはらって さらに強力なものとなるのである。しかしながら 非常に興味深いものは，ＡＥＤステートメントの中間表現（intermediate representation）なのである。これは plex の使用を基礎にしており，この plexとは その要素のそれぞれが多くの link とデータとを持つことのできるようなデータ構造のことである。一方 "modelling plex" についての構成と処理とは，マクロ・ルーチンの集合によって成しとげられる。これらのマクロ・ルーチンは，code generation やコンピュータ・グラフィックスやプログラムされた tool command のためのルーチンを含みうるものである。文献〔Ross 63, 67〕は，いくつかの問題分野における詳しい使用例が出ているので ＡＥＤの紹介として好適なものである。

ＡＥＤシステムの本質的な特徴は，シンタックスにおける順位行列とセマンティックスにおける plex のとり扱かいである。syntax-directed な世界に対する もう少し異なったアプローチは，第Ⅲ章Ｂで論じた一般的なコンパイラ・コンパイラのモデルから展開され得るのである。この計画においては データ構造の選択を含むセマンティックなメカニズム全体が，各適用分野によって異なることが可能である。たとえば，ＶＩＴＡＬ〔Mond 67〕の仕事では 二つの基本的

に異なるデータ構造の言語（共にＶＩＴＡＬで書かれている）が，それ自身ＶＩＴＡＬに基づいているsyntax−directedなgraphics package〔Rob 66〕の中で 比較されているのである。

その他のＴＷＳの適応例の多くは，ある種類 あるいはその他の種類の記号操作の作業に現われている。最初の適用例のうちのいくつか〔Scho 65〕は，記号数学におけるものであった。あるＴＷＳは，式の構造をモデル化する ― おそらくは簡略化したり 分化したりする――のを助けるために用いられることもある。記号数学におけるＴＷＳの使用（とりわけＣＯＧＥ−ＮＴやＭＥＴＡの使用）は，現在 普及しつつあり，特にその目的のために作り上げられたシステム〔Cla 66〕を生じてきている。さらに syntax−directedなモデルが有用であることを見出している純粋数学の学者もいるのである。

ＴＷＳの適用例で formatの変換の問題は，最も普及し余り意外とは言えないものである。これらは 大規模なデータ・ファイルと関連し，密接に関係のあるソース言語間の翻訳から生じてきた。もう一度述べておくが 第Ⅲ章Ａのsyntax−direct symbol processor は最も度々用いられてきているのである。これらのシステムは 論理設計，geometric descriptionの翻訳，シミュレーション，logging routineなどのような多様な作業に利用することもできる。 この他にも 特殊目的のデバイス（装置）に対する一連の命令を作り出すようなＴＷＳの適用例も沢山ある。例えば かなり複雑なＴＷＳ〔Cas 66〕は，人工衛星追跡システムのさまざまな構成要素に対して命令を翻訳するのに用いられた例もあるのである。

ＴＷＳは 幾多の分野に直接適用されているばかりでなく，ほかのいくつかの分野の仕事にも示唆を与えている。これらの分野のうち 特に盛んなのは，pictureのシンタクテイックな記述である。picture 処理に対するsyntax−directedなアプローチは しだいに普及してきたように見えるが〔ShaA 67, An 67を参照〕，その初期のある研究者〔Nar 66〕は このアプローチを（〔Nar 67〕のある章の中で） 否定しているように見える。新しいリスト処理言語に組み入れられたpattern matchingの特性は ＴＷＳによる示唆を一部受けており，artificial intelligenceの関係分野でも 従位的計画として syntax−directedなプロジェクトを備えているのである。

コンピュータ言語学の分野は，その理論面からも実際面からもＴＷＳの研究に密接に関連している。この分野における応用例は 意外に少ないのではあるけれども，大変重要なものである。コンピュータ言語とＴＷＳのシンタクテイックな理論は，いずれも Chomskyの初期の研究〔Chom 63〕に基づいていて 共通な概念も多い。英語のシンタックスのインプリメンテーション（特に〔Kun 62〕）は top−down ＴＷＳと同時に開発されたが，自然言語の仕事は ＴＷＳの研究において開発された有効な進歩をなかなかその中に組み入れないようである。セマンテ

ィックスの応用では言語学に全く新しいアプローチをもたらしたDEACONプロジェクト〔Th 66〕が，TWSの方法の直接的応用であると言えよう（〔Nap 67，Col 67〕を参照）。そして 言語学者たちがセマンティックな理論に検討を加えようとしているこれらの研究部門とnonprocedual な言語に対処しようとするTWSの研究者とが相互に影響を及ぼし合うことが期待される。

最後に これは決してここで考えている用例が最も少ないというわけではないが，教育面に応用されたTWSにふれておく。以上に述べたTWSのうちのいくつかは，トランスレータ作成の課程の基礎として用いられている。これらは教養課程から学部のゼミにまで及び，大変注目されている。また これらの課程は 機械についての問題を除外して講義されているにもかかわらず学生が討論を通じてTWSに容易に近づく時，多大の成功を収めている。

〔参考文献〕

References for IV.A

Ab 66, An 66, Brook 67a, Cas 66, Chom 65, Cla 66, Col 67, Gro 66, Hal 66, 67b, It 66, Kun 62, Mond 67, Nap 67, Nar 66, 67, Rob 66, Ross 63, 66, 67, Scho 65, ShaA 67, Th 66.

B. Formal Studies of Semantics

コンピュータ・サイエンスは　今まで数学に負うところが非常に大きかったが，今やその負債を返す時が来ようとしている。プログラミング言語のシンタックス（第Ⅱ章C）とセマンティックスは共に，形式的なとりあつかいを示唆している。この節では　ＴＷＳに関係の深い開発について簡単にふれ，プログラミング言語の形式的なセマンティックスに関する文献の概要を述べておきたい。

プログラミング言語のセマンティックスの 形式的研究にはいつも，シンタックスをセマンティックスからどのようにして分離するかという問題が起ってくる。プログラミング言語は，（問題を解く）論理から出た概念と（現在では軽視されている）自然言語とを結びつけるものである。そして　プログラミング言語を取り扱う場合，シンタックスが　厳密には　ここで問題となっている　syntactic metalanguage で記述可能な言語の一面と取られることが多い。しかしこの習慣は，metalanguage での定義が変るごとに　シンタックスの定義が変わるという　好ましくない結果を得るのである。

論理になれているコンピュータ・サイエンスの研究者（例えば〔Tix　67〕）は，そこで用いられる定義 —— すなわち，その形（form）の表現によって書きあらわされうる記号列の性質がシンタックティックであるという定義 —— を採ることを歓迎するであろう。このようにして記号列がある計算規則における定理であるか否かは，シンタックスにおけるとうてい結論の出そうにもない問題なのである。このアプローチは　自然言語に対して有効であることを証明しているのではなく，プログラミング言語における直接的な問題であるのである。

例えば

```
X ← Y／0.0
L1：go to L1
```

というステートメントは，シンタックスの面から見て　ＡＬＧＯＬとして正しい形であるだろうか。確かに　データ・タイプを処理することのできるアルゴリズムは，これらのエラーを見つけるであろうが，問題は　どこまでそれが及ぶかという事である。論理に携わる人々の要求を満たした上で，形式の整った決定可能な性質をもつプログラムを残すようなシンタックスの概念を作り出すことができるかどうかは明らかではないのである。

この問題をさらにむずかしくしているのは，総ての主要な言語が形式的シンタックス（formal syntax）だけでは解析できないステートメントを含んでいるという事である。このことについての良い例は，ＰＬ／Ｉ（〔ＩＢＭ66〕）では，ラベルをつけたＥＮＤを使用可能であるが，ＡＬＧＯＬでさえも今だにこのような構成要素を用いることのできないことである。したがって　実

用においては，syntax−directed compilerは　シンタクティックな面に"セマンティック"性質を組み込まねばならないのである。このシンタックスとセマンティックスを分けるという問題についてのひとつの功妙なアプローチは，McCarthyのabstract syntax〔Mc Car 62a〕である。彼は主に　セマンティックスについての研究を行なっており，（分析的な）シンタックスは　ソースの記号列の形から適切な情報を引き出すのに必要なpredicateやfunctionの集合にすぎないと考えている。このシンタックスを定義するという問題を「解決」はしないけれども，シンタックスとセマンティックスについて考えることを可能にするものである。

例のごとくセマンティックスの形式的研究は，プログラミング言語のシンタックスの研究に遅れをとっている。そしてこの問題に関する研究のうちで　最も優れた一般的研究は〔Ste 66〕であり，そしてこれは書物よりもむしろ討議によって形式的セマンティックスの概念規定を与えているものである。現在までに考案されたさまざまな形式化(formalization)は，すべてprodedualなものである。それらは　抽象的な機械のこともあるし，λ−calculus〔Chu 51〕のような命令的形式化(imperative formalisms)のこともある。これは期待しても当然のことなのであるが，現在　存在している数学的モデルの選択をかなり制限することである。

形式化(formalization)がprocedualなことから，プログラミング言語の記述においては"meaning"という言葉よりも"effect"　という言葉の方が好まれるようである。これはセマンティックスの概念をeffectとして支持するという立場をとるのではなく，ただ　物事を見るのに都合のよい方法として　それを採用するのである。この見方によって"environment（環境）に依存して異ったeffectを持つようなプログラムを期待でき，このことは我々の討論に有益であることがわかるのである。また　このことは　semantic metalanguageの選択が，形式的記述の意図的な使用によって影響をうけるという疑いを抱かせるようになるのである。

現在の形式的セマンティックスにおける研究は，プログラムに関する証明に関係のあるものと　コンピュータによるプログラムの処理を解明するのに興味をもつものとの二つに大別できるであろう。またこれは必ずしも必要ではないが，第Ⅲ章Bで述べたsemantic metalanguageを　後者の中に含めることができる。しかしこの中に含めてよいと考えられるものは，多少　抽象化された翻訳モデル(〔Wir 66c〕など)もあるのである。このようなモデルにおいては，ある言語はそのトランスレータがインタプリータであるか　コンパイラであるかによって大変に異なったeffectを持ちうるのである。これはプログラマにとっては理にかなったことのように思えるのであるが，meaningがプログラムではなく　アルゴリズムに存するという見方を好む数学タイプにとっては　ただ混乱を招くだけなのである。上記の研究に関連した開発を掲げると，すべてのプログラミング言語をひとつだけそれよりも高いレベルの言語〔Ste66〕あるいは機械的な言語〔Brat 61, Ste 61〕

に対するreductionによって定義しようとする計画がある。

これまで述べてきた形式化に対するアプローチは　ＴＷＳに関係が深いものであるが，これは余りにも複雑すぎるので　これを利用して証明を行なうには適していないのである。証明が形式化の唯一の目的であると考える人々（そしてそういう人々がこの報告書を読んでいるのであろうが）にしてみれば，これまで述べてきたことなど　まわりくどい言いまわしにすぎないと思うであろう。形式化に基づいている数学的試みの多くは　取り扱いやすいという点に重きを置いており，そのほとんどは　既存の数学に基づくものである。論理（logic）における命令的なシステム（imperative systern）は　2，3　あるだけであるが，そのそれぞれはコンピュータ・サイエンスのある面を形式化するのに用いられているのである。形式的セマンティックスにおける研究の大部分は，churchｃchu 51〕のλ－caluculusやCurryのcombinator caluculusに基づいている。

これらの理論はいずれも　主として変数の役割に関係しており，また　これらがプログラミング言語において成功しているのも　主にこの分野においてである。λ－記法は　ＬＩＳＰにおいて重要な役割を果しており，いろいろなＬＩＳＰに関する書物では　プログラミングの概念として論じられている。これはまた　セマンティックスを形式化しようとする試みに対する最もポピュラーな方法なのである。LandinとStracheyの業績は，彼らがＣＰＬと呼ばれるＡＬＧＯＬ60の拡張型〔Burs 65，Cou 65〕の開発に彼らの研究を組み合せたことによって　特別に興味深いものとなったのである。λ－記法のセマンティックスへの適用は，Landinが最も苦心して追求したものであった。一連の研究論文で　彼は，プログラミング言語（ＡＬＧＯＬ）と増大したλ－記法との間にimperative applicative expressions（ＩＡＥ）と呼ばれる関係を考えている。ＡＬＧＯＬにおける変数の宣言と結合が非常に明らかにモデル化されており，その形式化がＡＬＧＯＬのいくつかの弱点を指摘するのに役立っている。（純粋のＬＩＳＰと同じように）ＩＡＥシステムは，純粋にfunctionalなものであり　そしてその環境に副作用（side effect）を伴うO－adic functionとして　ステートメントを表現しなければならないのである。実際LandinのＡＬＧＯＬについての記述の多くは，ＬＩＳＰにおける"プログラムの性質"〔Mc Car 62b〕の一般化として見ることができるのである。しかし　この段階までででは，これらの研究は必要とされている表記上の明快さを達成しておらず，またオリジナル・プランに関連したλ－caluculusの扱いやすさを保持するのにも到っていないのである。この研究による最もすばらしい利益は，極端な程までに文明化された（civilized）言語ＣＰＬ〔Cou 66〕を持っていることである。そしてＭＩＴには，このアプローチをさらにできる限り推し進めて行こうとしているグループが　現在活動を続けているのである。

McCarthy は，彼がコンピュータに λ－caluculus を導入したのではあるが，形式的セマティックスへの彼のアプローチは多少異なっている。彼の使う“theory of computation（コンピューテーションの原理）”という言葉は，彼が算法言語よりも アルゴリズムに関心を持っていることを示している。彼のアプローチは state ベクトル，およびそれに対するオペレーション，抽象的なシンタックス，そして条件付きの expression を利用するものである。典型的な state 関数は，c（x，d） ― state ベクトル c におけるシンボリックなポジション x の内容 ― および a（x，z，d） ― state ベクトル d において x に z を代入する時の結果の state ― などである。次に 彼は 条件付きの expression を，機械語のコードのようなオペレーション および抽象的なシンタックスによって記述されるような高いレベルの構成要素の定義であるととることができるようにしたのである。この結果として得られる形式化は かなり扱いやすいので，McCarthy とその研究員は 多くの証明〔Mc Car 67〕を完成することができたのである。最近の成果としては，Painter〔Pai 67〕が“コンパイラ”の正統性を証明できたことが挙げられる。

Floyd は，最近 直観的によく満足できるアプローチを開発した〔Flo 67〕。彼は通常の（固定した）プログラミング言語で書かれたプログラムのフローチャートを考えた。その基本的な考え方は，フローチャートの各接続部分（connection）に state ベクトルを適用して proposition を付けることである。この proposition は connection が実行中（翻訳の間を考えるとする）に保たれている限り，成り立つものである。これらの proposition および これに関連したいくつかのメカニズムを用いて，Floyd は「もし初期状態（initial state）が R 1 を満足する時 これが到達するものであれば，最終状態（final state）は R 2 を満足する」という形式の文章を証明する方法を開発したのである。さらに この結末の証明は，例えば そのプログラムが実行されるにつれて減少していく正の整数による何らかの関数を示すことによってとりあつかうことができるのである。また つい最近 限定された言語に対して proposition を生成することと，その正統性を証明することの両方を自動化しようとする研究も進められている。

以上 述べてきたのは 極端な例なのであるが，このほかにもこれらの中間をいくようなセマンティックスの形式化へのアプローチがいくつか行なわれている。そのひとつの例〔Don 67〕をあげると，これは（タイプのマッチング等を含む）シンタックスとプログラムの compilation の双方を記述する post canonical なシステム〔Pos 43〕のひとつの version を用いるものである。ここでなされた定義は 一見合理的に見えるけれども，それらが compilation あるいは証明の中で役に立っているかどうかは疑問である。また 現在の数学を利用する例としては，van Wijngarten と de Bakker〔Bak 65，67，Wij 66〕の研究があげられる。彼らは 単純で

はあるが　有効なルールを読み　翻訳することのできるuniversalな機械を用いて，セマンティックなモデルの複雑さを解消しようとしたのである。そしてこれらのルールは，ソース言語についてのMarkovアルゴリズム〔Mark 59〕の記述にあたるものを定義するのに累積的に用いられるのである。ここで問題となるのは　その形式論が余りにも初歩的なものであるため，ＡＬＧＯＬの記述が逐語的に書かれたもの〔Bak 67〕となるだけで　証明や洞察が効果をあげるには到らないという事である。

また　プログラミング言語のセマンティックスを遂行する抽象的な機械を定義しようとする試みも　数多くなされている。このうちで最も意欲的な研究は〔Elg 64〕のＲＡＳＰであるが，この研究は現在では続けられていないようである。〔Nar 67〕による非常に興味深い論文があるが，これは　以上に論じてきたいろいろな性質を統合した形式論を含むものである。Narasimhanは　フロー・チャート，state変数についての関数，宣言文，演算子の選択やアドレッシングを含む関係の深い形式論において言語や機械を定義しているのである。そして　この研究は有望なものに思われるのであるが　具体的な成果はまだ現われておらず，また彼の根本的前提にかなり問題があるのである。というのは　トレンスレータを可能な限り単純化しようとするあまり　Narasimhanは，シンタックスおよびrecursionが価値を持たないような結論に到ってしまうのである。彼はまた（文献には載っていないが）ＴＷＳの研究は総て失敗しており，そこから得られる興味も衰退の一途をたどっているとも言っているのである。

形式的セマンティックスの研究について　これまで述べてきた事は余りにも　その内容が浅いため，おそらく誤解を招くのではないかと思われる。これらの研究に携わっている人々は　相互に連絡しあっている事もあるであろうし，また　その成果も我々の予想よりも豊かなのかもしれない。そこで　この論文に満足できない方々は，この終りにＴＷＳ関係のすべての傾向にわたる参考文献を掲げておいたので　これを参照されたい。形式的セマンティックス　特に証明に傾いているもの（proof－oriented）は，それぞれ孤立したわずかの研究に影響を与えているにすぎない。たとえばKrohnとRhodesのオートマトンの理論などは，全くといって良い程　どんな研究にも影響を与えていないのである。思うに既存の命令的な論理（imperatiue logic）では，この問題を解明するのには到らないのではなかろうか。そして　プログラミング言語は，数理言語や自然言語がつき当ったのと同じ問題に直面せざるを得ない時がやってくるのではないかと思われるのである。MinskyとPapert〔Min 67〕が，　同様の観測を述べているので最後にこれを記しておく。

優れた理論というものは，良く理解された実際の問題の文脈から離れてしまっては　展開できることはまれである。そして例えば回帰的な関数の数学的展開，オートマトン，形式言語学など

のような「Computationの抽象理論」を作り出す事だけを目的とした研究が，数学という面から見れば いかに優れたものであろうとも 実行という段になると全く惨めな結果しかもたらさなかったとしても 別に驚くにはあたらないのである。

〔参考文献〕

References for IV_B

Bak 65, 67, Braf 63, Burg 64, Burs 65, Cal 62, Chu 51, Cul 67, Cur 58, Don 67, Elg 67, Flo 67, Fr 61, 63b, Knu 67, Landi 63, 65, 66, Luc 65, McCar 62a, 67, Min 67, Nar 67, Org 67, Pai 66, Rig 62, Ste 61, Tars 56, Tix 67, Wir 66c, Zar 67, Zemu 66.

C. Summary and Research Problems

この論文で取り上げたＴＷＳは　多くの傑出したコンピュータ科学者たちの研究の長い歴史に支えられてきた最近の研究成果を代表するものである。また　第Ⅲ章で扱ったカテゴリーは　それぞれ独特の長所と短所とを持ち，それぞれに都合のよい問題の範囲が示されている。そこで種々のカテゴリーの間の関係をまとめた後，将来の研究に対する多くの実り多き分野について述べてみたい。

第Ⅱ章Ｂで述べた recognizer の自動な構成要素（automatic constructor）は　syntax－directed　なアプローチで解明できるような問題ならば　どのようなものにでも使用する事のできる道具である。効率のよい recognizer　を自動的に作り出すことによって，このようなシステムは syntax－directed　な方法の有効範囲を広げることができる。ここで一番大きな問題は　総合的（synthetic）なシンタックスに　セマンティックスな定義をうめこむ手軽な方法を見つけることである。この問題が解決できれば，それは　第Ⅲ章Ａの syntax－directed symbol プロセッサの能力も飛躍的に増大することになるのである。これらのＴＷＳは総て，セマンティックスを導入するのにはかなり便利な方法を備えているのであるが，これらは比較的効率のよくない recognizer を用いているのである。このため，さらに効率の良い recognizer が開発されれば，現在もかなり進んでいるこれらのシステムの応用も拡充される事は明らかである。

第Ⅲ章C.1～C.3で述べたメタアセンブラは，現在のところ　コンパイラの作成よりは　アセンブラの作成に適しているようである。しかし　これらは広範囲に効果をもたらすマクロ言語にいくつかの有意義なものをつけ加えるものである。メタアセンブラの機能を，翻訳時の action に対して拡張したり，シンタックスなフェーズをつけ加えたりすることによって，それらを第Ⅲ章Ａの syntax－directed プロセッサに匹適するものとすることが可能であろう。

第Ⅲ章Ｂのコンパイラ・コンパイラは　専門的なＴＷＳの発展したものとしては最高のものである。専門的であるということによって，それらはコンパイラの作成には　はるかに便利になったのであるが　その代償も大きなものである。すなわち　コンパイラ・コンパイラは　インプリメントすることがむずかしく　コンパイルとは異なる仕事には向かないということがしばしばあるのである。セマンティックス言語が　より複雑なプログラミングの構成要素を含む方向に進んで行くにつれて　このようなシステムはさらに専門化してくる事が予想される。そして　これは言うまでもなく　過度に専門化してくると言う危険性をもっており，ＴＷＳの研究者の中には，ＣＯＧＥＮＴ（第Ⅲ章B.3）のような一般的な syntax－directed プロセッサの方が　将来においては役に立つのではないかと考える向きもあるようである。

拡張可能なコンパイラの研究（第Ⅲ章C.4～C.6）は　ごく最近のものであり　いかなる高レ

ベル言語にも いくつかのマクロの機能を含むようにするべきであるということは明快なようではあるが，これを正確に評価する事は困難である。これよりもさらに風変りなシステムは 利用範囲が狭くなると思われる。理論的に言えば，ある言語をマクロの方式で拡張してから その後その拡張をそのコンパイラに効果的に組み入れることも可能である。CCシステム（第Ⅲ章B.3）は この両方の便宜を備えており，これがその問題を解決するとは言えないが その解決法を試すのには好適な便宜である。

これまで述べてきたTWSは そのどれをとっても それだけで総ての問題が解決されるというわけには行かない。そして我々は そのようなものを期待するのは無理であるということを説明しようとしてきたのである。また いろいろな種類の universal programming システムについてのさまざまの試みのもたらした結果からも これが正しいという事がわかる。しかし全体として見れば，TWSに関する研究が コンパイラ作成にまつわる多くの重要な問題を解決しているという事が言えそうである。現在では 可能性のあるTWSのさまざまな要求を満たすのに十分な方法（ technique ）があり，未解決の問題は特殊な項目にすぎないのである。

TWSのシンタクティックな面は かなり注目を集めており 未解決の疑問は少ない。ここでセマンティックスに深い関係を持ち，また互いに関連する三つの問題が 浮かび上ってくる。この問題のひとつは，「シンタックス」が人間の直観に非常に近いものとなることができるように extra なシンタックスの性質を当てはめるのに満足のいくような方法をどのようにして見つけるかということである。〔 Gil 66, Don 67 〕 また これに関連して，そこでつかわれている recognizer－constructing プログラムに関して詳細な知識を持ち合せていないと 総合的な文法の規則にセマンティクスを適用する適当な方法がないということも述べられている。最後に自動的な recognizer－constructing プログラムが 優雅に退化していっているという問題もある。というのは 可能な限り 効果のある方法を使用するシステムが好まれるのであり，その時使っている方法がその任に耐えられなくなってしまってから これを放棄するよりは，速度の遅い より一般的な計画へと自動的に動いて行くからである。これに加えてシンタックス・エラーの自動復旧という問題は さらに注目を集めるであろう〔 EvA 63, Ir 65 〕。

しかし，TWSのpostsyntactic な面については 余り研究が行なわれてきていない。この「セマンティックス」という問題に対するアプローチとしては，根本的に異なる次の三つがあげられる。第一の方法は，汎用のリスト処理 もしくはシンボル操作の能力（第Ⅲ章Aを参照）を与えることである。第二の方法は，沢山のデータ構造と組み込みのルーチン ことに コンパイラ作成用に設計されたルーチンを供給することである（第Ⅲ章Bを参照）。第三の方法は，これらの機能に code bracket および機械独立な出力の指定方法を結びつける事である（第Ⅲ章B・

1を参照）。マクロやサブルーチンを用いることによって，最初のふたつの方法は　普通のユーザーにとっては　自動化されたシステムのように見えるであろう。この点から見れば，セマンティックスにおける一番重要な問題は　コンパイラ作成の大事な局面を扱う汎用のルーチンをいかにして見出すかという事になるのである。我々は，ＴＷＳのアプローチは可能であることが証明されており　一般的な問題は　現在では　それを展開する段階で考えるべきであると感じているのである。もちろん　現在の範囲では解明できないプログラミング言語（例えば，シミュレーション〔Te 66〕）もいくつかあるが，しかし　それらの各々は先ず研究しなければならないような基本的な概念はわずかしか含んでいないのである。つまり　ＴＷＳの研究の目指すところは　プログラミング言語の未解決な問題を理解する（そして最後には自動化する）事に向かっているわけである。

このように　ＴＷＳの研究について系統的に述べてきた結果，勿論　我々は　各人に対する年間計画は提示してきたことになるであろう。この見解が正しいという事は　metaproblemを考慮することによってもたらされるプログラミング・システムに多大の貢献を与えていることによっても明らかである。そして　その章の残りの部分では　ＴＷＳのアプローチに附随する多くの興味深い問題について論じ，これらの問題に関する文献の概要を述べておきたい。この節の最後に記した参考文献には　ごく最新のもの，包括的なもの，及びこの論文ですでに参照したものが含まれている。

機械語の形式的記述についての問題は　かなり以前から提示されていながら，いまだに解決されていない。この問題が解決されれば，それはtarget machineを表わすTWSへの第三の入力として使う事もできるのである。ところが　この問題は　理論的にも直接的にも研究されていながら，ＴＷＳで使用可能であるような成果は　全くといってよい程上がっていない。parallel processorを利用できるということは　新しい研究分野に新しいレベルの複雑さを加えるものであり，また　これをよりよいものにするものである。parellel processor に対するソフトウェアの仕事の多くは　特定の機械に関連づけられているものなので，この論文の範囲外のものであるように思われる。parallelism の研究の基礎となるような　いくつかの抽象的〔Kar 66〕あるいは具体的〔Shed 67, Sto 67〕理論もある。そして高レベル言語におけるparallelism〔Dij 65〕も注目を集めるようになってきている。

また　コードの選択と増大の理論（optimizationの問題）に関係した問題もかなり古くからである。この理論は弱いばかりでなく　コンパイラの作成者によって一般に使用されるコードの増大（enhancement）の型は6通り程しかないのである。プログラムの能率における飛躍的な進歩の多くは　通常問題に対するアプローチ全体を組み立て直すことによるものである。このことは

広義のoptimizationと呼ばれるが，我々は　これをnonprocedualなプログラミングの一局面として論じることにする。" nonprocedual "は" セマンティックス "と同様に一般に認められた定義がまだ確立されていない。そこで　より高い程度のproblemステートメントに応じてprocedualなステップの選択と組み立て直しを行なうようなプログラミング・システムをnonprocedual と呼ぶことにする。

nonprocedual なプログラミング言語は　宣言的言語（ declarative language),問題向き言語，質問システムなど多くの項目のもとで論じられてきた。しかし　この研究の多くは理論的に余り面白いものとは言えない。というのは，ある人が大きなルーチンを作り　ユーザーは　それにパラメタを与えるだけであるからである。（〔 You 65〕を参照）。また　ある限られた問題の分野に対するかなり優れたnonprocedual システムが　コンピュータ・グラフィックス，関係的言語（ relational languages ）〔 Rov 67〕，配列の処理〔 Gal 67〕，数値解析〔 Ri 66〕において開発されてきている。言うまでもなく　アナログ・コンピュータは常にこの方法でプログラムされ，ハイブリッド・コンピューティングにおいて使われている言語を拡張することによって将来発展の可能性の強いシステム〔 Schl 67〕が開発中である。また　Cheathamは　第Ⅲ章C・5で述べた拡張可能なコンパイラに一般的な種類のnonpracedual な性質をつけ加えるという構想を抱いているのである。一方，自然言語の処理を通じて開発されたより複雑なシンタックスの形や変換を用いるという　別の方面からのアプローチも可能である。ＴＷＳの研究は　自然言語システムと多くの点において同じ興味を持っていると思える程なのである。これには質問言語と呼ばれるもの〔 Com 66〕や，言うまでもなくＣＯＢＯＬ〔 Samm 61〕があるのであるが　これらは　ここで取り上げた問題とは表面的なつながりを持っているにすぎない。しかし　会話的あるいはnonprocedualなプログラミング言語についての最近の興味ある事柄は，syntax−directedな自然言語システム（第Ⅳ章Aを参照）とあいまって，いろいろな考え方の間の情報交換をさらに有意義なものとしていくであろう。

ＴＷＳとエクゼクティヴ・システム（ executive system ）との間の関係には　まだ未解決の問題がいくつか残されている。ＴＷＳの主な利点のひとつは　それぞれのコンパイラがエクゼクティヴに対してインターフェイスをとる労力（しばしば全体の半分以上にも及ぶものである）を削減する事である。しかるに，このパラグラフ以前には「エクゼクティヴ」という言葉が出て来なかったという事が　この分野における過去の研究の実態を示しているものであると言えよう。コンパイラについての"環境の（ environmental ）"問題〔 Leo 66〕に興味を持っている小グループは前々からあったのであるが，それらはタイム・シェアリングという革命が起きるまではほとんど影響を及ぼすには到らなかったのである。つまり，大規模なマルチアクセス・タイム・

シェアリング・システムが使えるようになって（あるいはその要求が高まってきてから）新たなTWSに関する研究問題がもち上ってきたのである。大規模なタイム・シェアリング・システムのエクゼクティヴの主な仕事は リソースの割り当てである。割り当てられるリソースにはいろいろな記憶装置や処理ユニットばかりでなく，コンパイラのようなプログラムも含まれるのである。そこで研究問題は いかにしてシステムの効果をよりよくするように，トランスレータとエクゼクティヴとが情報を交換することができるかという方法を案出することなのである。現存のシステムで早急に必要とされているのは，主記憶に関係したものであり，特定の言語のswapping time を減少させようとする いくつかの計画〔Bob 67，Coh 67，Rov 67〕も試みられている。これに関連する問題は TWS〔Feld 67〕とその結果のオブジェクト・コードにおいて純粋な procedure をoptimal に（すなわちmaximal でなく）使用するという事である。コンパイラとエクゼクティヴの優れたインターフェイスは達成するのが大変にむずかしい反面，理論的には余り興味深くない解決方法が実際面では大いに役立つのである。

次にエクゼクティヴ・システムに関連してふたつの問題について簡単にふれておくことにする。制御言語（control banguage）は，これにシンタックス処理をつけ加える事によって，また理論的に言えばTWSですでに用いられているのと同一のコードを使用する事によって改善することができる。さらに意欲的なプロジェクトは syntax−directed な方法をエクゼクティヴ・プログラム自身を作るのに適用しようとしている。もうひとつのこれに関連する問題はdebugging aidsの問題である。on−line debugging システム〔Evl 66〕については 今までに多大な研究がなされてきたのであるが，その多くはMACプロジェクトを除いては アセンブラ言語のレベルにおけるものであった。すなわち，特定のバッチ用のコンパイラに 記号によるダンプ機能（symbolic dump facilities）の優れたものを備えたものもあるが プリントやTWSに到る道はまだ開拓されていないのである。また 自動的にエラーを検出すること，およびsyntax−directed プロセッサにおける復旧機能に関する研究〔Ir 65〕もわずかではあるがなされているのである。ここでもう一度述べておくが 優れているとはとても言えないようなシステムであってもユーザーにしてみれば大きな価値のあるものなのである。

最後にデータ構造に関する研究分野について論じることにする。この分野は 行列のとり扱いからファイルの処理にいたるまでのすべてを含んでおり，かつコンピュータ・サイエンスの種々の分野に深いつながりを持っているようである。ある意味では このデータ構造こそがコンピュータ・サイエンスで 現在問題となっている最も重要なものであり，このうちの未解決の問題を検討することが かなり意味の深い事であると言えよう。ここではTWSに関連したそのいくつかの面について述べ，ここで扱った他の研究問題において どのようなデータ構造に関係する考

察がなされてきたかを示すことにする。

いかなるＴＷＳにおいても，翻訳時および実行時の双方で組み込まれているデータ構造の選択はひとつの主要な問題となっているのである。第Ⅱ章では翻訳時の構造について述べ，実行時に対しては　組み込みの structure operator を与えるような事は　基本的には何もなされていないことを述べた。つまり　複雑なデータ構造の言語の多くはＴＷＳ（たとえば〔Ab 66，It 66，Rov 67〕）を使って書かれるのに対して，その structure operator はすべて手書きによっているのである。しかし　最近容易にＴＷＳに組み入れることのできる単一の一般的なデータ構造（universal data structure）を開発しようとする試み（たとえば〔Ross 66，ＩＢＭ66，Wir 66b〕もなされている。ここで問題なのは　データ構造が現在適用されている分野においては，既存の提案はすべて大変に効率のよくないものになってしまうということである。ある与えられたアルコリズムに合った構造を選ぼうとすれば　必らずnonprocedual　なプログラミングが問題となってくる。同様に，データ構造を改善することによって　グローバルな optimization や自然言語の処理を大きく進歩させることができるのである。実際，ここで扱ったすべての研究問題，およびその他の多くのものの間には密接な関係が存在しているのである。したがってＴＷＳの問題は　その本来の性質から言っても，プログラミングの研究の最新の分野と常に関連をもっているのである。

最近のＴＷＳの研究を簡単に検討してみようとしたところが　このような長い論文になってしまった。我々はここで，ほとんど独自で研究を続けているかなり多くの優れた人々が　どのようにしてシステム・プログラミングの多くの面について合理的に理解し得たかということを示そうと努めてきたのであった。そして　情報交換が盛んに行なわれ，科学的な規格化がさらに進めば意義深い進展や　研究において開発されたアイディアの迅速な応用も可能となってくる。このような目的があるからこそ，我々はこの論文を書いたのであるし　また読んで下さったものであると確信する次第である。

〔参考文献〕

REFERENCES FOR IV.C

Theory of machine instructions: Brat 61, Bur 64, Car 62, Don 67, Elg 64, Gil 67, Maur 65, Nar 67, Ste 61.

Parallelism: Dij 65, Kar 66, Kuc 67, Mark 67, Shed 67, Sto 67.

Code selection and enhancement, general references: Ar 61, Grie 65, Hill 62, Hor 66, Lucc 67.

Nonprocedural languages: Che 66, Gall 67, Ri 66, Rov 67, Schl 67, Sib 61, Wil 64b, You 65.

Natural language processing: Bar 64, Chom 65, Col 67, Cra 66, Hal 66, Int 63, Kun 62, Nap 67, Nar 67, Th 66.

Executive interface: Feld 67, Le 66, Orch 66, Nob 63.

Paging: Bob 67, Coh 67, Den 65, Rov 67.

Debugging: EvA 63, EvT 66, Ir 65.

Data structures: Ab 66, Brook 67c, Gall 67, IBM 66, It 66, Pra 65, Pra 66, Rov 67, Sta 67, Wir 66b.

謝　　辞

さいごに　John Reynolds氏ほか，示唆や御批判を寄せて下さった多くの方々に感謝の意を表したい。

BIBLIOGRAPHY

Ab 66 — Abrahams, P. W. The LISP 2 programming language and system. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 661-676.

An 66 — Anderson, R. H. A two dimensional syntax for mathematical notation. Unpublished report. Harvard U., Cambridge, Mass., 1966.

Ar 61 — Arden, B. W., Galler, B. A., and Graham, R. M. An algorithm for equivalence declarations. *Comm. ACM 4* (July 1961), 310-314.

Ar 66 — —, —, and —. *Michigan Algorithmic Decoder*. U. of Michigan Press, Ann Arbor, Mich., 1966.

Bac 57 — Backus, J. W. et al. The FORTRAN automatic coding system. Proc. Western Joint Comput. Conf. 1957, pp. 188-198.

Bac 59 — Backus, J. W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. Proc. Internat. Conf. on Information Processing, UNESCO, 1959, pp. 125-132.

Bak 65 — de Bakker, J. Formal definition of algorithmic languages. MR74, Mathematisch Centrum, Amsterdam, May 1965.

Bak 67 — —. Formal definition of programming languages. Mathematisch Centrum Tract No. 16, Amsterdam, 1967.

Bar 64 — Bar Hillel, Y. *Language and Information*. Addison-Wesley Publishing Company, Inc., Reading, Mass., 1964.

Barn 62 — Barnett, M. P., and Futrelle, R. P. Syntactic analysis by digital computer. *Comm. ACM 5* (Oct. 1962), 515-526.

Bart 63 — Barton, R. S. A critical review of the state of the programming art. Proc. AFIPS 1963 SJCC, Vol. 23, pp. 169-177.

Ben 64a — Bennett, R. K., and Kvilekval, A. SET, self extending translator. Data Processing, Inc., March 1964.

Ben 64b — —, and Neumann, D. H. Extension of existing compilers by sophisticated use of macros. *Comm. ACM 7* (Sept. 1964), 541 (actually about assemblers).

Ber 62 — Berman, R., Sharp, J., and Sturges, L. Syntactical charts of COBOL 61. *Comm. ACM 5* (May 1962), 260.

Bob 67 — Bobrow, D. G., and Murphy, D. Structure of a LISP system using two level storage. *Comm. ACM 10* (March 1967), 155-160.

Braf 63 — Brafford, P., and Hirschberg, D. (Eds.) *Computer Programming and Formal Systems*. North-Holland Publishing Co., Amsterdam, 1963.

Brat 61 — Bratman, H. An alternate form of the UNCOL diagram. *Comm. ACM 4* (March 1961), 142.

Brook 60a — Brooker, R. A., and Morris, D. An assembly program for a phrase structure language. *Comput. J. 3* (1960), 168-174.

Brook 60b — —, and —. Some proposals for the realization of a certain assembly program. *Comput. J. 3* (1960), 220-231.

Brook 61 — —, and —. A description of Mercury Autocode in terms of a phrase structure language. *Annual Review in Automatic Programming, Vol. 2*, 1961, pp. 20-66.

Brook 62a — —, and —. A general translation program for phrase structure languages. *J. ACM 9* (Jan. 1962), 1-10.

Brook 62b — — et al. Trees and routines. *Comput. J. 5* (1962), 33-47.

Brook 63 — — et al. The compiler-compiler. *Annual Review in Automatic Programming, Vol. 3*, 1963, p. 229.

Brook 67a — —, Morris, D., and Rohl, J. S. Compiler-compiler facilities in Atlas Autocode. *Comput. J. 9* (1967), 350-352.

Brook 67b — —, —, and —. Experience with the compiler-compiler. *Comput. J. 9* (1967), 345-349.

Brook 67c — —, and Rohl, J. S. Simply partitioned data structures: The compiler-compiler reexamined. *Machine Intelligence I*, Collins and Michie, (Eds.), Oliver and Boyd, London, 1967.

BruP 67 — Brown, P. J. The ML/I macro processor. *Comm. ACM 10* (Oct. 1967), 618-623.

BroS 63 — Brown, S. A., Drayton, C. E., and Mittman, B. A description of the AIT language. *Comm. ACM 6* (Nov. 1963), 649-658.

Burg 64 — Burge, W. H. The evaluation, classification and interpretation of expressions. Proc. ACM 19th Nat. Conf., 1964, p. A1.4.

Burk 65 — Burkhardt, W. Universal programming languages and processors. Proc. AFIPS 1965 FJCC, Vol. 27, pp. 1-21.

Burs 65 — Burstell, R. M. Some aspects of CPL semantics. No. 3, Experimental Programming Rpts., Edinburgh U., Edinburgh, April, 1965.

Can 62 — Cantor, D. G. On the ambiguity problem of Backus Systems. *J. ACM 9* (Oct. 1962), 477-479.

Car 62 — Caracciolo DiForino, A. On a research project in the field of languages for processor construction. Proc. IFIP Congress, Munich, 1962, pp. 514-515.

Car 63 — —. Some remarks on the syntax of symbolic programming languages. *Comm. ACM 6* (Aug. 1963), 456.

Cas 66 — Castle, J. A command program compiler. General Electric MSD, King-of-Prussia, Pa., 1966.

Che 64a — Cheatham, T. E. The architecture of compilers. CAD-64-2-R, Computer Associates, Inc., Wakefield, Mass., 1964.

Che 64b — — et al. Preliminary description of the translator generator system, II. CA-64-1-SD, Computer Associates, Inc., Wakefield, Mass., 1964.

Che 64c — —, and Sattley, K. Syntax directed compiling. Proc. AFIPS 1964 SJCC, Vol. 25, pp. 31-57.

Che 65 — —. The TGS-II translator generator system. Proc. IFIP Congress, New York, 1965, pp 592-593.

Che 66 — —. The introduction of definitional facilities into higher level programming languages. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 623-637.

Chom 63 — Chomsky, N. Formal properties of grammars. In *Handbook of Mathematical Psychology, Vol. 2*, Luce, Bush and Galanter (Eds.), John Wiley & Sons, Inc., 1963, pp 323-418.

Chom 65 — —. *Aspects of the Theory of Syntax*. The MIT Press, Cambridge, Mass., 1965.

Chu 51 — Church, A. *The Calculi of Lambda-Conversion*. Annals of Math. Studies, No. 6, Princeton U. Press, Princeton, N. J., 1951.

Cla 66 — Clapp, L. A syntax-directed approach to automated aids for symbolic math. Summary in *Comm. ACM 9* (Aug. 1966), 549.

Coh 67 — Cohen, J. A use of fast and slow memories in list processing languages. *Comm. ACM 10* (Feb. 1967), 82-86.

Col 67 COLES, S. Syntax-directed interpretation of natural language. Doctoral dissertation. Carnegie-Mellon Inst., Pittsburgh, Pa., 1967.

Conn 66 CONNORS, T. B. ADAM—a generalized data management system. Proc. AFIPS 1966 SJCC, Vol. 28, pp. 193–203.

Con 63 CONWAY, M. E. Design of a separable transition-diagram compiler. *Comm. ACM 6* (July 1963), 396.

Cou 66 COULOURIS, G. F., AND GOODEY, T. J. The CPL1 system manual. PIDI2/GFC, Inst. of Comput. Sci., U. of London, London.

Cou 67 ———. The compiler processor project. Internal Rep., Imperial College, London, April 1967.

Cra 66 CRAIG, J. A., BEREZNER, S. C., CARNEY, H. C., AND LONGYEAR, C. R. DEACON: Direct English Access and CONtrol. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 365–380.

Cul 62 CULIK, K. Formal structure of ALGOL and simplification of its description. *Symbolic Languages in Data Processing*, Gordon and Breach, New York, 1962, pp. 75–82.

Cur 58 CURRY, H. B., AND FEYS, R. *Combinatory Logic, Vol. 1.* North-Holland Publishing Co., Amsterdam, 1958.

Den 65 DENNIS, J. B. Segmentation and the design of multiprogrammed computer systems. *J. ACM 12* (Oct. 1965), 589–602.

Dij 63 DIJKSTRA, E. W. On the design of machine independent programming languages. *Annual Review in Automatic Programming, Vol. 3*, 1963, pp. 27–42.

Dij 65 ———. Solution of a problem in concurrent programming control. *Comm. ACM 8* (Sept. 1965), 569.

Don 67 DONOVAN, J. J., AND LEDGARD, H. F. Canonic systems and their application to programming languages. Mem. MAC-M-347, Project MAC, MIT, Cambridge, Mass., April, 1967.

Ear 65 EARLEY, J. C. Generating a recognizer for a BNF grammar. Computation Center Rep., Carnegie Inst. of Tech., Pittsburgh, Pa., 1965.

Ear 67 ———. An LR(K) parsing algorithm. Carnegie Inst. of Tech., Pittsburgh, Pa., 1967, (mimeo).

Ei 63 EICKEL, J., PAUL, M., BAUER, F. L., AND SAMELSON, K. A syntax controlled generator of formal language processors. *Comm. ACM 6* (Aug. 1963), 451–455.

Ei 64 ———. Generation of parsing algorithms for Chomsky 2-type languages. 6401, Mathematisches Institut der Technischen Hochschule, Munich, 1964.

Elg 64 ELGOT, C. C., AND ROBINSON, A. Random-access stored-program machines, an approach to programming languages. *J. ACM 11* (Oct. 1964), 365–399.

Eng 61 ENGLUND, D., AND CLARK, E. The CLIP-translator. *Comm. ACM 4* (Jan. 1961), 19–22.

EvA 64 EVANS, ARTHUR. An ALGOL 60 compiler. *Annual Review in Automatic Programming, Vol. 4*, 1964, pp. 87–124.

EvT 66 EVANS, T., AND DARLEY, D. On-line debugging techniques: a survey. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 37–50.

Feld 61 FELDMAN, J. A. A formal semantics for computer oriented languages. Carnegie Inst. of Tech., Pittsburgh, Pa., 1964.

Feld 66 ———. A formal semantics for computer languages and its application in a compiler-compiler. *Comm. ACM 9* (Jan. 1966), 3–9.

Feld 67 ———, AND GRIES, J. The compiler-compiler in a time sharing environment. Lecture notes on Advanced Computer Organization, U. of Michigan, Ann Arbor, Mich., 1967.

Fer 66 FERGUSON, D. E. Evolution of the meta-assembly program. *Comm. ACM 9* (March 1966), 190–196.

Fie 67 FIEBERT, J. CABAL Memos. Computer Center Reps., Carnegie Inst. of Tech., Pittsburgh, Pa., 1967.

Flo 61 FLOYD, R. W. A descriptive language for symbol manipulation. *J. ACM 8* (Oct. 1961), 579–584.

Flo 62a ———. On ambiguity in phrase structure languages. *Comm. ACM 5* (Oct. 1962), 526, 534.

Flo 62b ———. On the nonexistence of a phrase structure grammar for ALGOL 60. *Comm. ACM 5* (Sept. 1962), 483–484.

Flo 63 ———. Syntactic analysis and operator precedence. *J. ACM 10* (July 1963), 316–333.

Flo 64a ———. Bounded context syntactic analysis. *Comm. ACM 7* (Feb. 1964), 62–67.

Flo 64b ———. The syntax of programming languages—a survey. *IEEE Trans. EC13*, 4 (Aug. 1964), 346–353.

Flo 67 ———. Assigning meanings to programs. *AMS Symposium in Appl. Math. 19*, 1967.

Gall 67 GALLER, B., AND PERLIS, A. J. A proposal for definitions in ALGOL. *Comm. ACM 10* (April 1967), 204–219.

Gar 64 GARWICK, J. V. GARGOYLE, a language for compiler writing. *Comm. ACM 7* (Jan. 1964), 16.

Gar 66 ———, BELL, J., AND KRIDER, L. The GPL language. TER 05, Control Data, Palo Alto, Calif., 1966.

Gea 65 GEAR, C. W. High speed compilation of efficient object code. *Comm. ACM 8* (Aug. 1965), 483–488.

Gil 66 GILBERT, P. On the syntax of algorithmic languages. *J. ACM 13* (Jan. 1966), 90–107.

Gil 67 ———, AND MCILRAITH, W. G. Compiler generation using formal specification of procedure oriented and machine languages. Proc. AFIPS 1967 SJCC, Vol. 30, pp. 447–455.

Gin 66a GINSBURG, S. *The Mathematical Theory of Context Free Languages*. McGraw-Hill, Inc., New York, 1966.

Gin 66b ———, AND GREIBACH, S. Deterministic context free languages. *Inf. Contr. 9* (1966), 620–648.

Gle 60 GLENNIE, A. E. On the syntax machine and the construction of a universal compiler. Tech. Rpt. No. 2, Computation Center, Carnegie Inst. of Tech., Pittsburgh, Pa., 1960.

Gor 61 GORN, S. Specification languages for mechanical languages and their processors, a baker's dozen. *Comm. ACM 4* (Dec. 1961), 532–542.

Gor 63 ———. Detection of generative ambiguities in context-free mechanical languages. *J. ACM 10* (April 1963), 196–208.

GraM 65 GRAHAM, M. L., AND INGERMAN, P. Z. A universal assembly mapping language. Proc. ACM 20th Nat. Conf., 1965, pp. 109–120.

GraR 64 GRAHAM, R. M. Bounded context translation. Proc. AFIPS 1964 SJCC, Vol. 25, pp. 17–29.

Gran 62 GRAU, A. A. A translator oriented symbolic programming language. *J. ACM 9* (April 1962), 480–487.

Gre 62 — GREEN, J. Symposium on languages for processor construction. Proc. IFIP Congress, Munich, 1962. pp. 513–517.

Grie 65 — GRIES, D., PAUL, M., AND WIEHLE, H. R. Some techniques used in the ALCOR-ILLINOIS 7090. *Comm. ACM 8* (Aug. 1965), 496–500.

Grie 67a — ——. The use of transition matrices in compiling. Tech. Rpt. CS 57, Computer Science Dept., Stanford U., Stanford, Calif., March 1967 and *Comm ACM 11* (Jan. 1968), 26–34.

Grie 67b — ——. Internal notes on the compiler writing system. Computer Science Dept., Stanford U., Stanford, Calif., 1967.

Grif 65 — GRIFFITHS, T. V., AND PETRICK, S. R. On the relative efficiencies of context-free grammar recognizers. *Comm. ACM 8* (May 1965), 289–299.

Gro 66 — GROSS, M. Applications geometriques des langages formels. *ICC Bull. 5* (Sept. 1966), 141–167.

Hal 64 — HALPERN, M. XPOP: a metalanguage without metaphysics. Proc. AFIPS 1964 FJCC. Vol. 26, pp. 57–68.

Hal 66 — ——. Foundations of the case for natural-language programming. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 639–649.

Hal 67a — ——. Toward a general processor for programming languages. *Comm. ACM 11*, (Jan. 1968), 15–26.

Hal 67b — ——. Foundations of the case for natural language programming. *IEEE Spectrum* (March 1967), 140–149.

Hal 67c — ——. *A manual of the XPOP programming system.* Electronic Sciences Lab. Lockheed Missiles & Space Company, Palo Alto, Calif., March 1967.

Hals 62 — HALSTEAD, M. H. *Machine-Independent Computer Programming.* Spartan Books, Washington, D.C., 1962.

Hill 62 — HILL, V., LANGMAACK, H., SCHWARZ, H. R., AND SEEGMÜLLER, G. Efficient handling of subscripted variables in ALGOL 60 compilers. Proc. Symbolic Languages in Data Processing, Gordon and Breach, New York, 1962, 331–340.

Hoa 65 — HOARE, C. A. R. A programming language for processor construction. Notes from NATO summer school lectures, 1966.

Hor 66 — HORWITZ, L. P., KARP, R. M., MILLER, R. E., AND WINOGRAD, S. Index register allocation. *J. ACM 13* (Jan. 1966), 43–61.

Hus 62 — HUSKEY, HARRY D. Languages for aiding compiler writing (panel discussion). Proc. Symbolic Languages in Data Processing, Gordon and Breach, New York. 1962, 187–204.

IBM 66 — IBM System/360 Operating System PL/I Language Specification. Form C28-6571-4.

Ing 62 — INGERMAN, P. Z. Techniques for processor construction. Proc. IFIP Congress, Munich 1962, pp. 527–528.

Ing 66 — ——. *A Syntax Oriented Translator.* Academic Press, Inc., New York, 1966.

Int 63 — International Standards Organization. Survey of programming languages and processors. *Comm. ACM 6* (March 1963), 93.

Ir 61 — IRONS, E. T. A syntax directed compiler for ALGOL 60. *Comm. ACM 4* (Jan. 1961), 51–55.

Ir 63a — ——. The structure and use of the syntax-directed compiler. *Annual Review in Automatic Programming, Vol. 3*, 1963, pp. 207–227.

Ir 63b — ——. Towards more versatile mechanical translators. *AMS Symposium in Appl. Math. 16*, 1963, pp. 41–50.

Ir 64 — ——. Structural connections in formal languages. *Comm. ACM 7* (Feb. 1964), 67–71.

Ir 65 — ——. An error correcting parse algorithm. *Comm. ACM 6* (Nov. 1965), 669–673.

It 66 — ITURRIAGA, R., STANDISH, T. A., KRUTAR, R. A., AND EARLEY, J. C. Techniques and advantages of using the formal compiler writing system FSL to implement a formula ALGOL compiler. Proc. AFIPS 1966 SJCC. Vol. 28, pp. 241–252.

Kar 66 — KARP, R. M., AND MILLER, R. E. Properties of a model for parallel computations: determinacy termination, queueing. *SIAM J.* (Nov. 1966), 1390–1411.

Kerr 67 — KERR, R. H., AND CLEGG, J. The Atlas ALGOL Compiler—an ICT implementation of ALGOL using the Brooker-Morris syntax-directed compiler. *Comput. J.* (1967).

Kir 66 — KIRKLEY, C., AND RULIFSON, J. LOTS, a syntax-directed compiler. Internal Rep., Stanford Research Inst., Menlo Park, Calif., May 1966.

Knu 62 — KNUTH, D. E. History of writing compilers. Proc. ACM 17th Natl. Conf., 1962, pp. 43, 126.

Knu 65 — ——. On the translation of languages from left to right. *Inf. Contr. 8* (Oct. 1965), 607–639.

Knu 67 — ——. The remaining trouble spots in ALGOL 60. *Comm. ACM 10* (Oct. 1967), 611–618.

Kuc 67 — KUCK, D. Programming the ILLIAC IV. Talk given at AFIPS 1967 SJCC. Paper not yet available.

Kun 62 — KUNO, S., AND OETTINGER, A. G. Multiple-path syntactic analyzer. *Information Processing 62* (IFIP Congress), Popplewell (Ed.), North-Holland Publishing Co., Amsterdam, 1962, pp. 306–311.

Lande 62 — LANDEN, W. H., AND WATTENBURG, W. H. On the efficient construction of automatic programming systems. Proc. ACM 17th Natl. Conf., 1962, p. 91.

Landi 63 — LANDIN, P. J. The mechanical evaluation of expressions. *Comp. J. 6* (1963), 308.

Landi 65 — ——. A correspondence between ALGOL 60 and Church's λ-notation. *Comm. ACM 8* (Feb. and March 1965), 89–101, 158–167.

Landi 66 — ——. The next 700 programming languages. *Comm. ACM 9* (March 1966), 157–166.

Landw 64 — LANDWEBER, P. S. Decision problems of phrase structure grammars. *IEEE Trans. EC, 13* (Aug. 1964), 354–362.

Lang 64 — LANGMAACK, H., AND EICKEL, J. Präzisierung der begriffe Phrasenstruktur und strukturelle Mehrdeutigkeit in Chomsky-Sprachen. Rep. No. 6414, Rechenzentrum der Technischen Hochschule, Munich, 1964.

Lea 64 — LEAVENWORTH, B. M. FORTRAN IV as a syntax language. *Comm. ACM 7* (Feb. 1964), 72–80.

Lea 66 — ——. Syntax macros and extended translation. *Comm. ACM 9* (Nov. 1966), 790–793.

Leo 66 — LEONARD, G., AND GOODROE, J. More extensible machines. *Comm. ACM 9* (March 1966), 183–188

Let 65 — LETICHEVSKII, A. A. The representation of context-free languages in automata with a push down type store. Cybernetics (Kibernetika) Vol. 1, No. 2, The Faraday Press, New York (1965), 81–86.

Luc 65 — Lucas, P. Definition of a subset of PL/1 by finite local state vectors. Working paper to IFIP WG2.1, July, 1965.

Lucc 67 — Luccio, F. A comment on index register allocation. *Comm. ACM 10* (Sept. 1967), 572-574.

Mark 61 — Markov, A. A. *Theory of Algorithms*. US Bureau of Standards, OTS 60-51085, available from Clearing House, Springfield, Va.

Mar 67 — Martin, D., and Estrin, G. Models of computations and systems. *J. ACM 14* (April 1967), 281-294.

Mas 60 — Masterson, K.S. Compilation for two computers with NELIAC. *Comm. ACM 3* (Nov. 1960), 607-611.

Maur 65 — Maurer, W. A theory of computer instructions. Mem. MAC-M-262, Project MAC, MIT Cambridge, Mass., Sept. 1965.

May 61 — Mayoh, B. H. Letter to the editor correcting Ir 61. *Comm. ACM 4* (June 1961), 284.

McCar 62a — McCarthy, J. Toward a science of computation. *Information Processing 62* (IFIP Congress), Popplewell (Ed.). North-Holland Publishing Co., Amsterdam, 1962, pp. 21-28.

McCar 62b — — et al. LISP 1.5 programmers manual. Computation Lab Report, MIT (1962).

McCar 67 — —, and Painter, J. Correctness of a compiler for arithmetic expressions. *AMS Symposium in Appl. Math. 19*, 1967.

McCl 65 — McClure, R. M. TMG- a syntax-directed compiler. Proc. ACM 20th Natl. Conf., 1965, pp. 262-274.

McIl 60 — McIlroy, M. D. Macro instruction extension of compiler language. *Comm. ACM 3* (April 1960), 214-220.

McKee 66 — McKeeman, W. M. An approach to computer language design. Tech. Rpt. CS 48, Computer Science Dept., Stanford U., Stanford, Calif., Aug. 1966.

Mea 63 — Mealy, G. A generalized assembly system. RM-3646-PR Rand Corp., Santa Monica, Calif., Aug. 1963.

Met 64 — Metcalfe, H. H. A parametrized compiler based on mechanical linguistics. *Annual Review in Automatic Programming, Vol. 4*, 1964, pp. 125-165.

Min 67 — Minsky, M. L., and Papert, S. Perceptions and pattern recognition. Mem. MAC-M-358, Project MAC, MIT, Cambridge, Mass., Sept. 1967.

Mond 67 — Mondschein, L. VITAL compiler compiler reference manual. TN 1967-1, Lincoln Lab., MIT, Lexington, Mass., Jan. 1967.

Moo 63 — Moores, C., and Deutsch, L. P. TRAC, a text handling language. Proc. ACM 20th Natl. Conf., 1965, pp. 229-246.

Mor 67 — Morris, D., and Wilson, I. A system program generator. Computer Science Dept., U. of Manchester, Manchester, 1967

Nap 67 — Napper, R. B. E. The third order compiler. A context free man-machine communication. *Machine Intelligence 1*, Collins and Michie (Eds.). Oliver and Boyd, London, 1967.

Nar 66 — Narasimhan, R. Syntax-directed interpretation of classes of pictures. *Comm. ACM 9* (March 1966), 166-173.

Nar 67 — ——. Programming languages and computers: a unified meta theory. In *Advances in Computer 8*. Academic Press, Inc., New York, 1967, Chap. 5

Naur 60 — Naur, P. (Ed.) Report on the algorithmic language ALGOL 60. *Numer. Math. 2* (1960), 106-136; *Comm. ACM 3* (May 1960), 299-314

Naur 63a — ——. Documentation problems: ALGOL 60. *Comm. ACM 6* (March 1963), 77-79.

Naur 63b — ——. Revised report on the algorithmic language ALGOL 60. *Comm. ACM 6* (Jan. 1963), 1-17; *Numer. Math. 4* (1963), 420-452; *Comp. J. 5* (1963), 349-367.

Nob 63 — Noble, A. S., and Talmadge, R. B. Design of an integrated programming and operating system, I and II. *IBM Syst. J. 2* (June 1963), 152-181.

Nor 63 — Northcote, R. S. The structure and use of a compiler-compiler system. Proc. Australian Comput. Conf., Dec. 1963.

Opl 62 — Opler, A. "Tool"- a processor construction language. Proc. IFIP Congress, Munich, 1962, pp. 513-514.

Orch 66 — Orchard-Hays, William. Multilevel operating systems. *Comm. ACM 9* (March 1966), 189-190, (abstract only).

Org 67 — Organ, R. J. A mathematical theory of computing machine structure and programming. RC1863, IBM, Yorktown Heights, New York, 1967.

Pai 66 — Painter, J. Semantic correctness of a compiler for an ALGOL-like language. AI Rep. No. 44. Stanford U., Stanford, Calif., 1966.

Par 61 — Parikh, R. J. Language generating devices. Quarterly progress rep. no. 60, Research Lab. of Electronics, MIT, Jan. 1961, 199-212. Reprinted with minor editorial revisions under the title: On context-free languages. *J. ACM 13* (Oct. 1966), 570-581.

Paul 62 — Paul, M. ALGOL 60 processors and a processor generator. Proc. IFIP Congress, Munich, 1962, pp. 493-497.

Plas 66 — Plaskow, J., and Schuman, S., The TRANGEN system on the M460 computer. AFCRL-66-516 (July 1966).

Pos 43 — Post, E. L. Formal reductions of the general combinatorial decision problem. *Am. J. Math. 65* (1943), 197-217.

Pra 65 — Pratt, T. W. Syntax-directed translation for experimental programming languages. TNN-41, Computation Center U. of Texas, Austin, Texas, 1965.

Pra 66 — ——, and Lindsay, R. K. A processor-building system for experimental programming language. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 613-621.

Rab 62 — Rabinowitz, I. N. Report on the algorithmic language FORTRAN II. *Comm. ACM 5* (June 1962), 327-337.

Ran 64 — Randell, B., and Russell, D. J. *ALGOL 60 Implementation*. Academic Press, Inc., London, 1964.

Rey 65 — Reynolds, J.C. An introduction to the COGENT programming system. Proc. ACM 20th Natl. Conf., 1965, pp. 422-436.

Ri 66 — Rice, J., and Rosen, S. NAPSS, numerical analysis and problem solving system. Proc. ACM 21st Natl. Conf., 1966, pp. 51-56.

Rig 62 — Riguet, J. Programmation et theories des categories. Proc. Rome Symposium on Symbolic Languages in Data Processing, Gordon and Breach, New York, (1962), pp. 83-98.

Rob 66 — Roberts, L. G. A graphical service system with variable syntax. *Comm. ACM 9* (March 1966), 173-176.

Ros 64a — Rosen, S. A compiler-building system developed by Brooker and Morris. *Comm. ACM 7* (July 1964), 403-414.

Ros 64b — ——. Programming systems and languages. Proc. AFIPS 1964 SJCC, Vol. 25, pp. 1-15.

Ross 63 — Ross, D., and Rodriguez, J. Theoretical foundations of the computer aided design system. Proc. AFIPS 1963 SJCC, Vol. 23, pp. 305-322.

Ross 64 — Ross, D. T. On context and ambiguity in parsing. *Comm. ACM 7* (Feb. 1964), 131-133.

Ross 66 — . AED bibliography. Mem. MAC-M-278-2, Project MAC, MIT Cambridge, Mass., Sept. 1966.

Ross 67 — . The AED approach to generalized computer aided design. Proc. ACM 22nd Natl. Conf. 1967, pp. 367-385.

Rov 67 — Rovner, P., and Feldman, J. An associative processing system for conventional digital computers. TN 1967-19, Lincoln Lab., MIT, Lexington, Mass., April 1967.

Rut 62 — Rutishauser, H. Panel on techniques for processor construction. Proc. IFIP Congress, Munich, 1962, pp. 524-531.

Sam 60 — Samelson, K., and Bauer, F. L. Sequential formula translation. *Comm. ACM 3* (Feb. 1960), 76-83.

Sam 62 — . Programming languages and their processing. Proc. IFIP Congress, Munich, 1962, pp. 487-492.

Samm 64 — Sammet, J. E. A definition of COBOL 61. Proc. ACM 16th Natl. Conf., 1961.

Schl 67 — Schlesinger, S., and Sashkin, L. POSE: a language for posing problems to a computer. *Comm. ACM 10* (May 1967), 279-285.

Schm 63 — Schmidt, L. Implementation of a symbol manipulator for heuristic translation. Proc. ACM 18th Natl. Conf., 1963.

Sch 64 — Schneider, F. W., and Johnson, G. D. META-3; A syntax directed compiler writing compiler to generate efficient code. Proc. ACM 19th Natl. Conf. 1964, p. D1.5-1.

Schn 65 — Schorr, H. Analytic differentiation using a syntax directed compiler. *Comput. J. 7* (Jan. 1965), 290-298.

Schor 61 — Schorre, D. V. META-II: A syntax-oriented compiler writing language. Proc. ACM 19th Natl. Conf., 1964, p. D1.3.

Schü 63 — Schützenberger, M. P. Context-free languages and pushdown automata. *Inf. Contr. 6* (Sept. 1963), 246-264.

Sh 58 — SHARE Ad Hoc Committee on Universal Languages. The problem of programming communication with changing machines: a proposed solution. *Comm. ACM 1* (Aug. 1958), 12-18.

ShaA 67 — Shaw, A. A formal description and parsing of pictures. Doctoral dissertation, Computer Science Dept., Stanford U., Stanford, Calif., 1968.

Shaw 63 — Shaw, C. J. A specification of JOVIAL. *Comm. ACM 6* (Dec. 1963), 721-735.

Shed 67 — Sheidler, G. Parallel numerical methods for the solution of equations. *Comm. ACM 10* (May 1967), 286-291.

Sib 61 — Sibley, R. A. The SLANG-system. *Comm. ACM 4* (Jan. 1961), 75-84.

Sta 67 — Standish, T. A. A data definition facility for programming languages. Computer Science Rep., Carnegie Inst. of Tech. Pittsburgh, Pa., May 1967.

Stark 65 — Stark, R. *ALTEXT multiple purpose language*. Lockheed Missiles & Space Company Tech. Rep. 6-75-65-15, March, 1965.

Ste 61 — Steel, T. B. A first version of UNCOL. Proc. Western Joint Comput. Conf., 1961, pp. 371-378.

Ste 66 — ——— (Ed.) Formal language description languages for computer programming. Proc. IFIP Conf., Baden, Sept. 1964, North-Holland Publishing Co., Amsterdam, 1966.

Stu 67 — Stone, H. S. One-pass compilation of arithmetic expressions for parallel processor. *Comm. ACM 10* (April 1967), 220-223.

Str 65 — Strachey, C. A general purpose macrogenerator. *Comput. J. 8* (1965), 225-241.

Tar 56 — Tarski, A. *Logic, Semantics, Metamathematics*. Clarendon Press, London, 1956.

Tay 61 — Taylor, W., Turner, L., and Waychoff, R. A syntactical chart of ALGOL 60. *Comm. ACM 14* (Sept. 1961), 393.

Te 66 — Teichroew, D., and Lubin, J. F. Computer simulation—discussion of the technique and comparison of languages. *Comm. ACM 9* (Oct. 1966), 727-741.

Th 66 — Thompson, F. B. English for the computer. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 349-356.

Tix 67 — Tixier, V. Recursive functions of regular expressions in language analysis. Tech. Rpt. CS 58, Computer Science Dept., Stanford U., Stanford, Calif., March 1967.

Tro 67 — Trout, R. G. A compiler-compiler system. Proc. ACM 22nd Natl Conf., 1967, pp. 317-322.

Wai 67 — Waite, W. A language independent macro processor. *Comm. ACM 10* (July 1967), 433-441.

War 61 — Warshall, S. A syntax directed generator. Proc. Eastern Joint Comput. Conf., 1961, pp. 295-305.

War 64 — ———, and Shapiro, R. M. A general-purpose table-driven compiler. Proc. AFIPS 1964 SJCC, Vol. 25, pp. 59-65.

Weg 62 — Wegner, P. (Ed.) *Introduction to Systems Programming*. Academic Press, Inc., New York, 1962.

Wij 66 — van Wijngaarden, A. Recursive definition of syntax and semantics in *Formal Language Description Languages for Computer Programming*, North-Holland Publishing Co., Amsterdam, 1966, pp. 13-24.

Wil 64a — Wilkes, M. V. An experiment with a self-compiling compiler for a simple list-processing language. *Annual Review in Automatic Programming, Vol. 4*, 1964, pp. 1-48.

Wil 64b — ———. Constraint-type statements in programming languages. *Comm. ACM 7* (Oct. 1964), 587.

Wir 66a — Wirth, N. A programming language for the 360 computers. Tech. Rpt. CS 53, Computer Science Dept., Stanford U., Stanford, Calif., Dec. 1966.

Wir 66b — ———, and Hoare, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9* (June, 1966), 413-432.

Wir 66c — ———, and Weber, H. EULER—a generalization of ALGOL, and its formal definition: Part I, Part II. *Comm. ACM 9* (Jan., Feb. 1966), 13-25, 89-99.

Yer 65 — Yershov, A. P. ALPHA—an automatic programming system of high efficiency. Proc. IFIP Congr., New York, 1965, pp. 622-623.

You 65 — Young, J. W., Jr. Nonprocedural languages. 7th Ann. ACM Tech. Symp., S. Calif. Chapter, March 1965.

Zar 67 — Zara, R. F. A semantic model for a language processor. Proc. ACM 22nd Natl. Conf., 1967, pp. 323-339.

Zem 66 — Zemanek, H. Semiotics and programming languages. *Comm. ACM 9* (March 1966), 139-143.

昭和48年 3 月発行

発行所 財団法人 日本情報処理開発センター

東京都港区芝公園3－5－8

機械振興会館内

TEL (434)8211(代表)

印刷所 株式会社 三 州 社

東京都港区芝大門一丁目一番二十一号

TEL (433)1481(代)

47-S 002