普 及 課

48－S 003

コンパイラ・ジェネレータの開発とその評価

昭 和 49 年 3 月

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48年度に実施した「コンパイラ・ジェネレータの開発とその評価」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とその評価」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，その都度開発する労力は多大なものであります。かかる労力を軽減し，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良く知られております。

しかし，コンパイラ・ジェネレータへの入力方法および作成されたコンパイラの効率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，47・48年の2ケ年にわたって行なわれたもので，47年度の成果は当財団報告書「コンパイラ・ジェネレータの開発」（47-S002）にまとめられております。48年度においては47年から着手したコンパイラ・ジェネレータの実用性の評価について検討を行ないました。

本報告書が，この方面に興味を持つ方々に広く利用され，情報処理技術の発展の一助として寄与できることを念願いたす次第であります。

昭和49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

目　　次

ま　え　が　き

本研究開発は，47・48年の2ケ年で行なったもので，本年度は47年度において作成に着手したコンパイラ・ジェネレータJPAC（JIPDEC Automatic Compiler generator）の完成と，その実用性の評価を行なった。

JPACは実用性を重視したコンパイラ・ジェネレータで，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LSD言語及び標準ライブラリとして用意されているルーチンを用いる。

実用性の評価はJIS3000レベルのFORTRANコンパイラをJPAC，PL／I，アセンブラで作成し，作成コスト及び作成されたコンパイラの性能を比較することによって行なった。

評価結果によると，JPACでコンパイラを作成した場合，アセンブラによった場合と比べて作成労力は60％減，作成時使用計算機時間は25％減となり，作成時コストの大幅な軽減が可能であることが確認された。

JPACによって作成されたコンパイラの性能に関しては，アセンブラによるものと比べて，同一プログラムのコンパイルに17～19倍のCPU時間を要するが，その主原因はLSD言語のコンパイラの文字処理関係のオブジェクト効率の悪さに起因することが確認されたので修正を行なう予定である。

本報告書は2章からなり，第1章はJPACの概要を記述し，第2章には評価の詳細を述べてある。また付録として作成コンパイラの性能評価に用いたテスト・プログラムを掲載した。

なお，JPACの内部および外部仕様の詳細は47年度の報告書「コンパイラ・ジェネレータの開発」を参照されたい。

１．コンパイラ・ジェネレータＪＰＡＣの概要

I. コンパイラジェネレータＪＰＡＣの概要

1.1 コンパイラ記述とＪＰＡＣ

従来コンパイラ作成の方法として

(1) アセンブリ言語で記述する

(2) 汎用言語（FORTRAN，PL／Ｉ）で記述する

(3) コンパイラ記述言語を作成して，これで記述する

(4) Compiler Generator で作成する

等がある。現在稼動しているCompiler の多くはアセンブリ言語で記述されているが，この方法は

① 設計よりも製造の面に時間と労力が余計に使われ，こまかいコーディング技術を駆使することによって得られるプラス面以上に，大きな設計上の失敗が見過ごされる可能性が多い。

② プログラマの単位時間当りの平均生産量はほぼ決まっているので，compiler の規模が大きくなればなるほど要員の数も増す。要員の数が増加するにつれインタラクションも増加し，効率が低下すると同時に意志疎通による誤解も生じやすい。

③ アセンブリ言語でコーディングするには概念的な流れ図の他にコーディング・チャートが必要になるが，この保守にはかなりの労力が必要である。しかしコーディング・チャートの保守が完全に行なわれていない場合は，時後の修正・変更はかなり困難であり，とくに保守者と開発者が異なる場合は更に大きな問題となる。

等の欠点が目立つ。

次の段階として，汎用言語で記述する方法が試みられている。ALGOLのサブ

セットやFORTRANに必要な表現能力を加えた言語またはPL／Iのサブセット（たとえばBPL）等を用いてcompilerを記述するが，

① ハードウェアの機能を十分使いこなすことができないため，アセンブリ言語に比較してでき上がったcompilerの効率が悪くなる。

② マシンに依存する部分は結局アセンブリ言語にたよることになる。

等の点が指摘される。

次に各種のcompilerを記述するための専用言語を使用する方法がある。これはコーディングの労力を節減しデバッグを容易にしコンパイラ作成期間を短縮することが出来るが，でき上がったcompilerの処理がアセンブラ記述のものと比べて一般には遅い。

以上の様にいくつかのコンパイラ作成方法にはそれぞれ一長一短がありどれが最適とはいえない状態である。残る方法としてCompiler Generator によるものがある。近年コンパイラ作成の自動化ということで脚光をあびてきているが，いまだ実用に供しうるものはきわめて少ない。今回当センターで開発したCompiler Generator JPAC はいくつかのCompiler Generator の現状をふまえた上で，比較的機能をおさえむしろ実用性を高めることを主眼としたものである。

JPACはFACOM230－60のMONITOR-Vの管理下で作動するCompiler Generatorであり，ジェネレートされるcompilerもFACOM230－60用のものである。

このシステムで使用している構文解析手法はDe RemerのSLR(k)･L(m)R(k)であり，Syntaxの記述にはBNF，Semanticsの記述にはLSD言語（新しく開発したシステム記述言語）で書かれたセマンティックス・ルーチンを用いている。

1.2 ＪＰＡＣの構成

ＪＰＡＣは（図１－１）に示すようにcompilerのSyntax（BNFで記述）を

入力して遷移テーブルを作り出すSTAGE，セマンティックス・ルーチン（ＬＳＤ言語で記述）をコンパイルするＬＳＤの２つのステップと，ＦＡＳＰ・ＬＩＤＥのＦＡＣＯＭ２３０－６０ＭＯＮＩＴＯＲ－Ｖとの共通ステップより構成されている。

ＳＴＡＧＥ・ＬＳＤの２つのステップでの出力オブジェクトはアセンブリ言語（ＦＡＳＰ）であり，次のＦＡＳＰステップによりリローケータブル形式に変換され，更にシステム・ライブラリあるいはユーザ・ライブラリがＬＩＥＤステップにより結合され，実行形式の compiler が作成される。

作成された compiler の動作概略を（図１－２）に示す。コントローラは最初にユーザが own-coding した Initiator に制御をわたす。Initiator においてはセマンティックス・ルーチンの初期設定を行なう。制御が 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

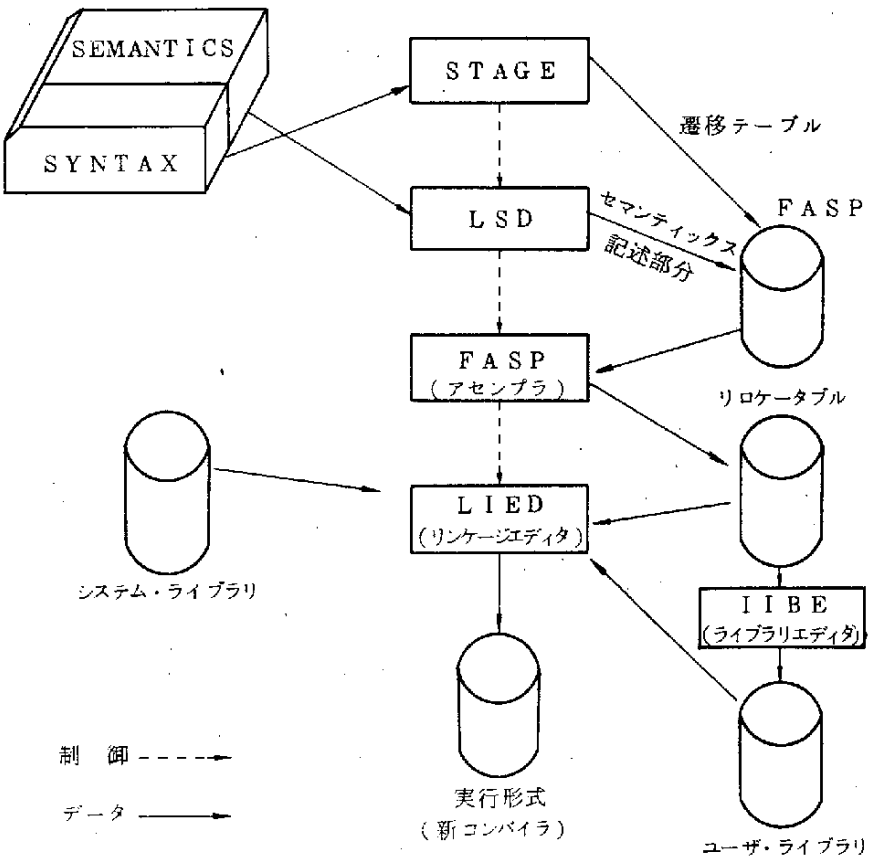


図１－１　　ＪＰＡＣの概略

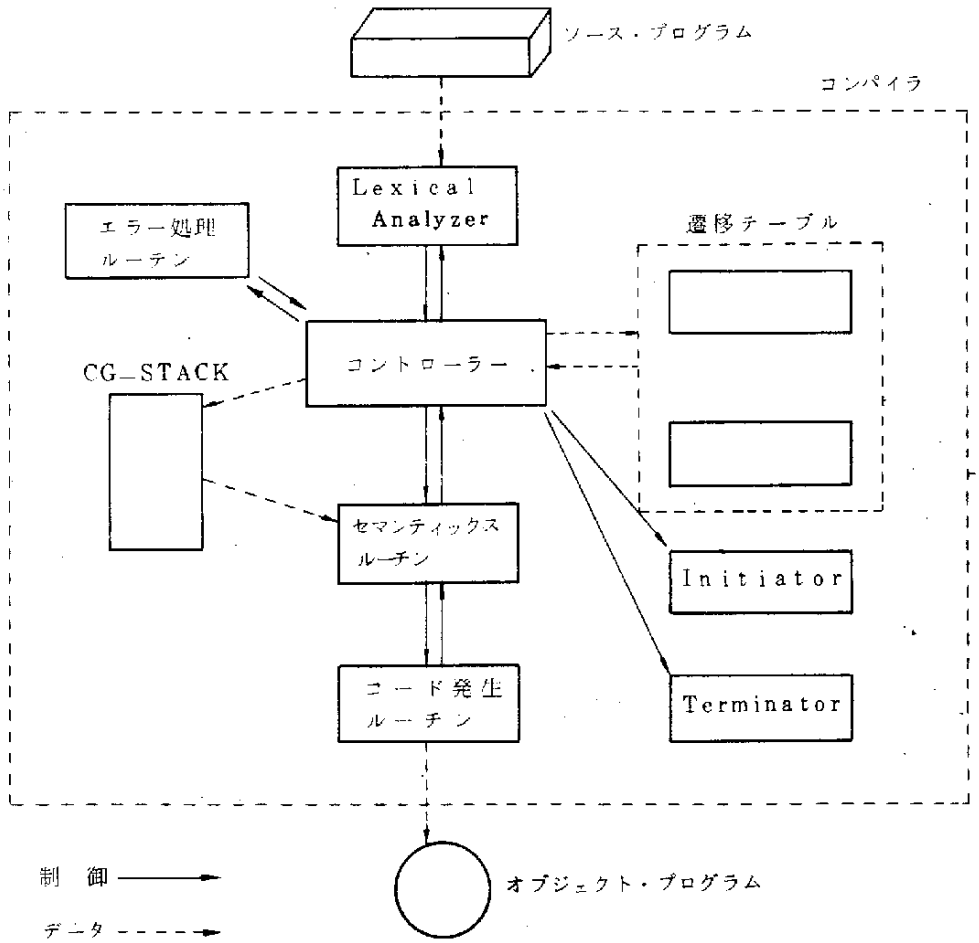


図１－２　コンパイラ・ジェネレータによって作成されたコンパイラ概略

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，セマンティックス・ルーチンの呼び出し指定があると，（図１－３）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ィックス・ルーチンに制御をわたす。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行なう。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，必要があればシステム・ライブラリとして用意されてい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・メッセージを出力し，ソースプログラムの読みとばしを行な

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もユーザがown-codingするものであり最終処理を行なう。

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る。その場合，

GO TO CG__IN;

とすれば制御がコントローラへ移る。ここでCG__INはラベル変数でありコントローラの入口名が入っている。

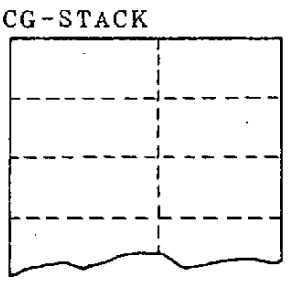


図１－３

1.3 JPAC（JIPDEC Automatic Compiler-Generator）

ＪＰＡＣの入力となるSyntax及びSemantics記述部の言語仕様とそれらを処理するSTAGE･LSDの動作を，それぞれＪＰＡＣの外部仕様・ＪＰＡＣの内部仕様として以下に示す。

1.3.1 ＪＰＡＣの外部仕様

ＪＰＡＣは次の２つの定義部より構成される。

SYNTAX DEFINITION;

SEMANTICS DEFINITION;

各定義部を示す前にいくつかの基本的な定義を行なう。

<Alphabet> :: = A | B | C | …… | Z

<Numeric> :: = 0 | 1 | 2 | …… | 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記述中下記の左側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。

A・SYNTAX DEFINITION

この定義部においてコンパイラ・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 DEFINITION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，この番号をコントローラに引き渡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Lexical Analyzer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でき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

<Basic 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nenterminal 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。

(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遷移テーブルが作成される。この遷移テーブルはコンパイラが動作する時にコントローラ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，コントローラは〔生成規則の右辺の項数－１〕をpop upして，スタックの最上位に還元されたnon-terminal 記号をセットして遷移テーブルの解読をつづける。

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，1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べ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つの形のいづれかである。

＜E＞::=＜E＞+＜T＞:EXP;

＜P＞::=＜id＞;

前者の場合，＜E＞+＜T＞という形が現われた時，コントローラはEXPという名前のセマンティック・ルーチンに制御を渡し，その処理が終ると＜E＞+＜T＞を＜E＞に置き変える。後者の場合，＜id＞が現われた時＜P＞に置き変えることのみを行ない，セマンティックス・ルーチンには制御を渡さない。

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

B．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・ルーチンの指定を行なった。すなわち，ある生成規則の適用が行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制御が渡される。

例えば，

＜E＞::＝＜E＞+＜T＞：EXP；

という生成規則があり，EXPルーチンに制御がわたった場合に，コントローラからの連絡情報は，CG_STACKにあり，次のような形をとる。

＜T＞	user define
+	同　上
＜E＞	同　上
CG_STACK	

system reserve	user define

CG_STR

セマンティックス・ルーチンにおいては，必要な処理を行ない＜T＞・+・＜E＞をpop upし，CG_STRの第2要素へ＜E＞に関する情報をセットして re-

turnする。コントローラでは，ＣＧ＿ＳＴＡＣＫの頭にＣＧ＿ＳＴＲを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言語の外部仕様にしたがう。

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言語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行なわれる。

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

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

C。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 Sym-

bol を使用した場合，ユーザは自分の責任で own-coding し以下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 が使用できる。しかしながら，でき上がったコンパイラの大きさ，コンパイルスピード等はBasic Symbol を使用した場合に比べ著しく悪くなる。

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。

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

D．エラー処理ルーチン

ＪＰＡＣで作り出されたコンパイラは，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 own-coding のエラー処理ルーチンを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，必ずユーザが own-coding しなければならない。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，エラー情報をパラメータとして，このルーチンをコールするようになっている。又，セマンティックス・ルーチン

の中では，エラーの検出にともなって直接このルーチンをコールするように，セマンティックス・ルーチンを作成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行ないリターンすればよい。これによりコントローラは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。

エラー・ルーチン名

CG_ERROR

パラメータ

COND：エラーを起した記号の存在場所を示すスイッチ。

COND＝1の時，CG_BFの内容

COND＝0の時，CG_STACKの内容

属性：NUMERIC

E・LSD言語

(1) 概　要

LSD（Language for Semantics Description）言語はコンパイラ作成に有効であるスタック操作・テーブル操作・ビット操作・文字操作が容易にできるように考慮された言語である。

(2) LSD言語の機能

LSD言語の主たる機能としては下記のものがあげられる。

① ニューメリック，ビット，文字，アドレスが取り扱える。

② ビット型，文字型に対してSUBSTR擬変数が使える。

③ 配列，構造体，テーブル，スタックが取り扱える。

④ 算術演算，論理演算，連結演算機能を持つ。

⑤　シーケンシャル・ファイルの取り扱いができる。

⑥　デバック機能を有す。

⑦　プログラム中にＦＡＳＰによるコーディングも可能である。これはＬＳＤ言語で記述したのでは効率が悪くなる部分をアセンブラで記述することを可能にするものである。

以上の機能はコンパイラの動作を記述する言語として必要な機能を満足していると思われる。

(3)　ＬＳＤステートメント一覧

①　プログラムの定義

ＰＲＯＣＥＤＵＲＥ，ＥＮＤ，ＥＮＴＲＹ

②　データの定義

ＤＥＣＬＡＲＥ．ＥＱＵ

③　テーブル操作

ＳＥＴ，ＥＲＡＳＥ，ＳＥＡＲＣＨ，ＩＮＣＲＥＡＳＥ

④　スタック操作

ＰＵＳＨ，ＰＯＰ

⑤　ビット操作

ＳＥＴＢ．ＲＥＳＥＴＢ

⑥　ファイル及び入出力

ＯＰＥＮ，ＣＬＯＳＥ，ＷＲＩＴＥ，ＲＥＡＤ，ＳＣＡＮ

⑦　制御の変更

ＣＡＬＬ，ＧＯＴＯ，ＲＥＴＵＲＮ，ＳＴＯＰ

⑧　判定，くり返し

ＩＦ，ＤＯ

⑨　デバッグ文

ＳＮＡＰ，ＴＲＡＣＥ

⑩　アセンブラとのリンク

ＥＮＴＥＲ，ＥＸＩＴ

⑪　その他の実行文

代入文，ＮＵＬＬ

⑫　組み込み関数

ＳＵＢＳＴＲ，ＡＤＤＲ

ＬＳＤ言語の仕様の詳細は日本情報処理開発センターの４７年度報告書「コンパイラ・ジェネレータの開発」に掲載されている。

1.3.2　ＪＰＡＣの内部仕様

ある言語のSyntax記述部を入力として遷移テーブルを出力するＳＴＡＧＥとSemantics記述部を処理するＬＳＤの処理過程を以下に示す。

Ａ・ＳＴＡＧＥ

(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を出力するものである。

ＳＴＡＧＥの処理の流れは（図１－４）で示されるとおりである。

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

表１－１　ＳＹＮＴＡＸ　ＤＥＦＩＮＩＴＩＯＮの例

```
SYNTAX DEFINITION;
BASIC SYMBOL;
<IDENTIFIER>;
SYNTAX RULE;
<S>::=<E>; EXEC0;
<E>::= <E>+<T>; EXEC1|<T>; EXEC2;
<T>::=<P>↑<T>; EXEC3|<P>; EXEC4;
<P>::=<IDENTIFIER>|(<E>); EXEC5;
SYNTAX RULE END;
```

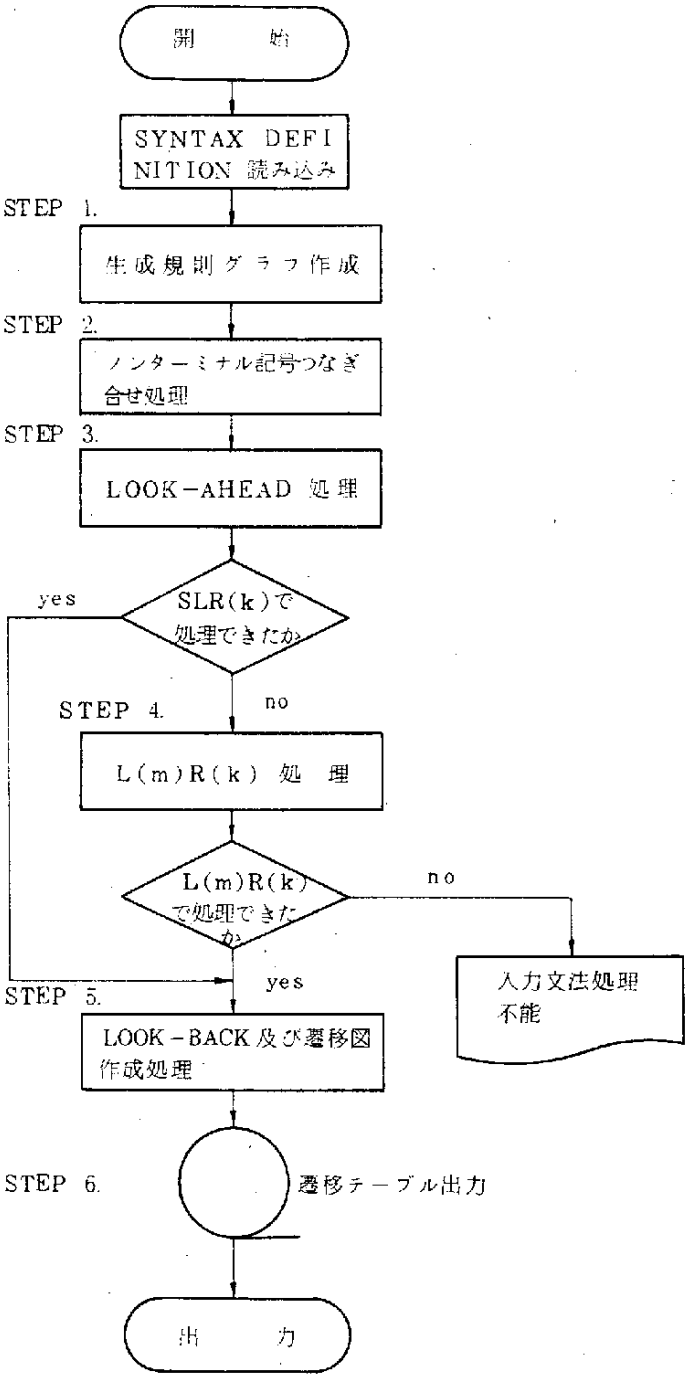


図１－４　ＳＴＡＧＥの処理の流れ

(2)　生成規則グラフ作成処理

この処理は（図１－４）のStep１に対応するものであり，与えられた生成規

則からnodeとarrowで表現した生成規則グラフを作成する。（表１－１）の場合は（図１－５）のようになる。

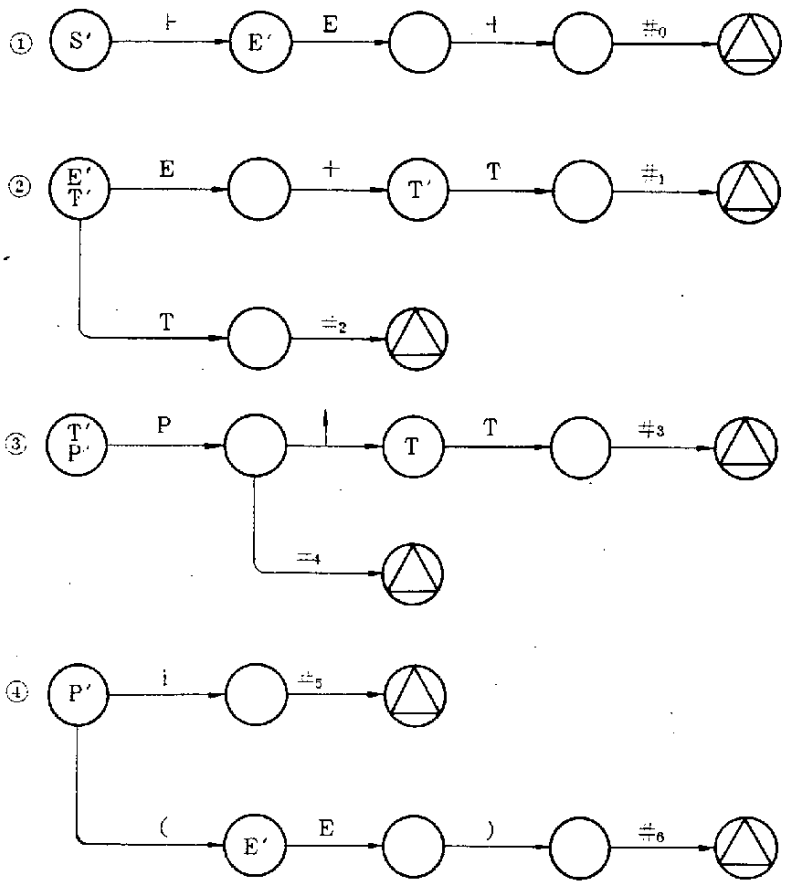


図１－５　生成規則グラフ（ｉ：＜ＩＤＥＮＴＩＦＩＥＲ＞を示す）

(3)　Nonterminal記号つなぎあわせ処理

この処理は（図１－４）のStep 2に対応し，各生成規則グラフをつなぎ合わせ，一連のグラフにするものである。つなぎ合わせは，nonterminal 記号を持つ全てのnodeに対して，同一のnonterminal記号を持つnodeにより始まる生成規則グラフをつなぎ合わせることにより達成できる。

（図１－５）の生成規則グラフをつなぎ合わせると（図１－６）となる。これはＣＦＳＭグラフと呼ばれる。

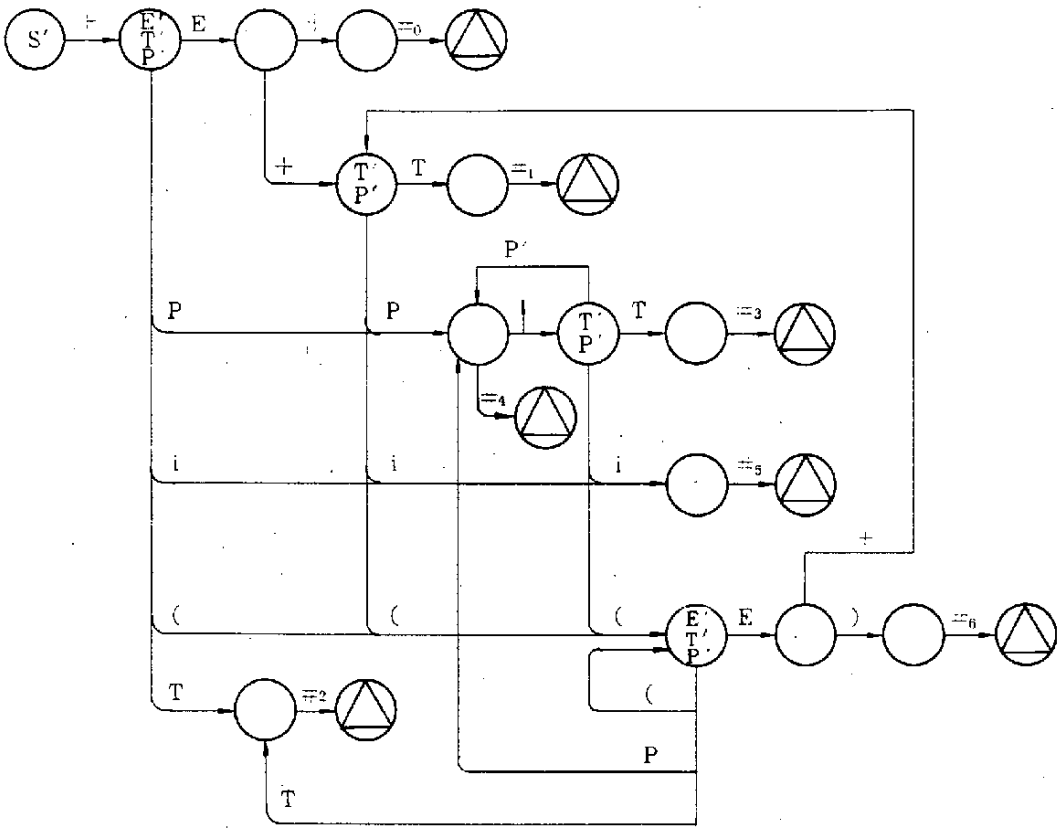


図１－６　　ＣＦＳＭグラフ

(4)　ＬＯＯＫ－ＡＨＥＡＤ処理

この処理は（図１－４）のStep 3に対応し，inadequate状態に対処するものである。

（図１－６）をみると同一nodeから≠4 arrowと他のarrowがでている場合があるが，この様な状態をinadequate状態といい，１個以上の記号を先読みすることにより，どちらのarrowを選ぶかを決定する。

先読みすることをlook－aheadと称し，先読み記号列の集合をlook－aheadセットという。

（図１－７）はlook－ahead処理後のＣＦＳＭグラフである。

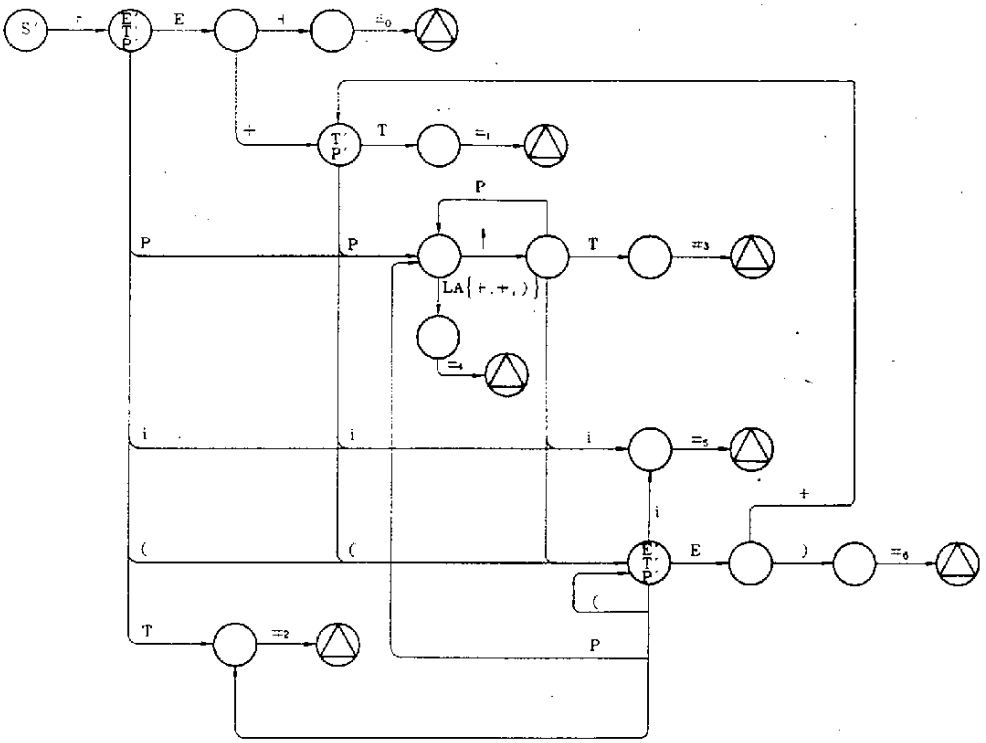


図１－７　look-ahead処理後のＣＦＳＭグラフ

(5)　Ｌ(ｍ)Ｒ(ｋ)処理

この処理は(図１－４)のStep 4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。

look-ahead処理でinadequate状態を処理できなかった時Ｌ(ｍ)Ｒ(ｋ)で処理できるか否かを試みる。その方法はinadequate状態のnodeからの遷移が左のｍ個の記号と右のｋ個の記号によって決定できるか否かを調べることであり，その結果遷移がユニークに決定できるなら該当inadequate状態はＬ(ｍ)Ｒ(ｋ)で分離可能となる。

Ｌ(ｍ)Ｒ(ｋ)で分離可能の場合，該当inadequate状態のnodeをいくつかのnodeに分割しlook-aheadセットのみで遷移(arrow選択)が決定できるようにＣＦＳＭグラフを作り変える。

ＪＰＡＣではＬ(２)Ｒ(１)まで行なっている。

Ｌ(ｍ)Ｒ(ｋ)でも分離不能の場合，処理不能のメッセージとともに,原因となった生成規則を表示して，処理を終了する。

(6) 遷移図作成処理

この処理は（図１－４）のStep５に対応しＣＦＳＭグラフをコントローラが処理できる遷移図に作り変える。その場合必要があればlook-back セットを求める。

この処理の結果得られる遷移図はＦＡＳＰソースの形式で出力される。

Ａ．遷移図作成の方法

＜Ｔ＞をnonterminal記号，＃ｎを＜Ｔ＞を生成する生成規則と仮定する。

ＣＦＳＭグラフにおいて＜Ｔ＞を遷移記号として持つarrowを，＃ｎを遷移記号として持つarrowの終端node（ＣＦＳＭグラフ例では△で表わしてある）からのarrowで置き変える。その時＜Ｔ＞を遷移記号として持つarrowが複数本あるならば終端nodeからのarrowも複数本あり，コントローラはその経路選択を行なわねばならない。その為に＜Ｔ＞以前の履歴（ＣＧ＿ＳＴＡＣＫ中に残って

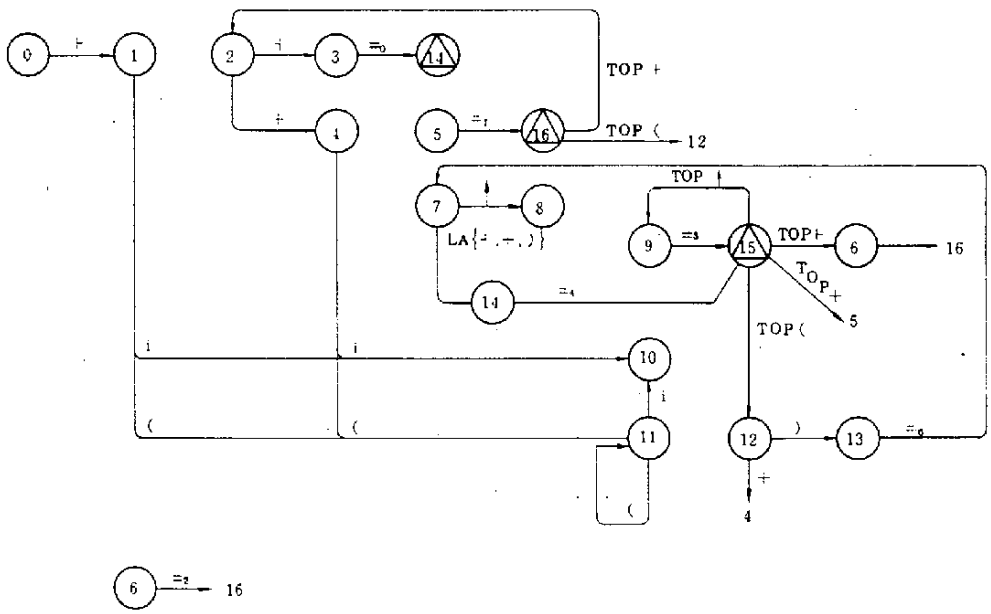


図１－８　遷移図

いる）と照合する必要があり，その履歴の集合がlook-backセットである。

上記の操作をＣＦＳＭグラフ上のnonterminal記号を遷移記号として持つすべてのarrowに対して行なう。

（図１－７）のＣＦＳＭグラフに上記の処理を行なってできた遷移図を（図１－８）に示す。

Ｂ．ＬＳＤ

ＬＳＤは（図１－９）に示されるようにSEMANTICS DEFINITION以後に指定されるＤＣＬステートメントの集合をSEMANTICS BODY部で示される各semantics proecdureにさし込み，ＬＳＤコンパイラを用いてＦＡＳＰソースの形で出力する。

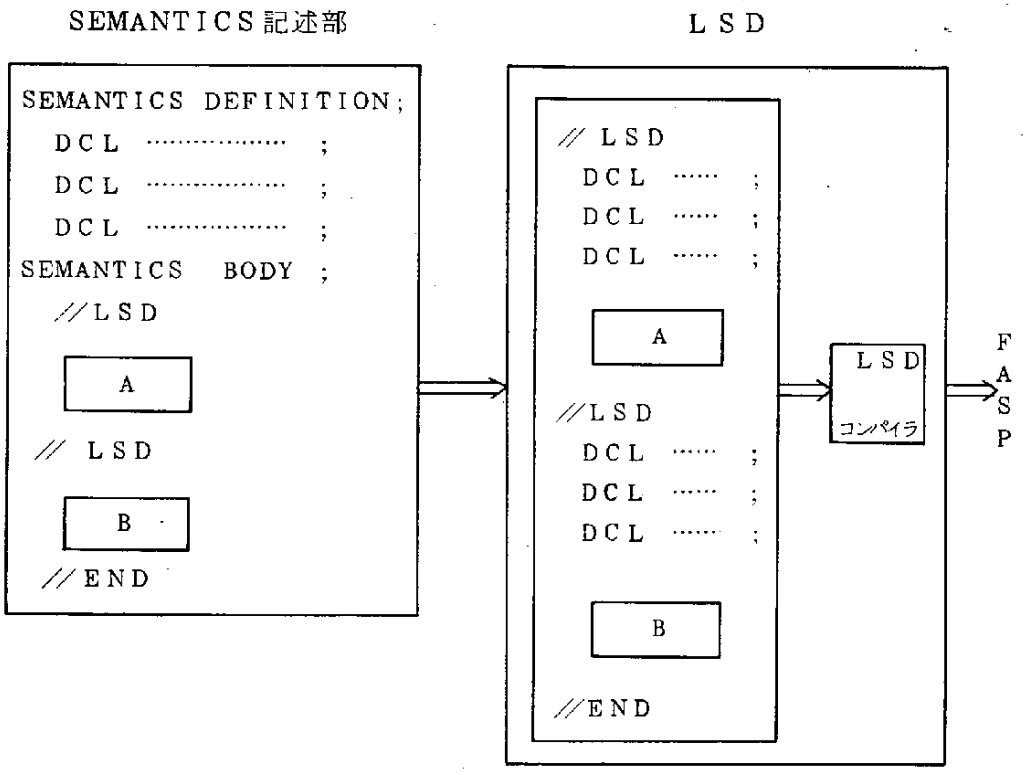


図１－９　ＬＳＤ

1.4 ＪＰＡＣの標準ライブラリ

ＪＰＡＣシステムでは下記のルーチンを標準ライブラリとして用意している。ＪＰＡＣを使用してコンパイラを作成するユーザは，これらの標準ライブラリを使うことにより作成時の労力を省くことができる。

① ソース・プログラムを読み込むルーチン（READ）。

② ソース・プログラムをSYSPRTへ出力するルーチン（WRITE）。

③ READルーチンで読み込んだソース・プログラムを走査するルーチン（SCAN）。

④ オブジェクト・プログラムをディスクあるいはテープに出力するルーチン（MTWRITE）。

⑤ ジョブ制御文EXEC（FACOM230－60MONITOR－Vのコントロール・カード）のPARAMパラメータを解析するルーチン（CG_PARAM）。

ただし，これらのルーチンはＬＳＤ言語で作成されたプログラムよりCALLされるものである。

1.4.1 READ

(1) 機　　能

SYSINファイルからソース・プログラムを読み込む。読み込むためのバッファはＣＧ＿ＢＵＦＦでなければならない。ＣＧ＿ＢＵＦＦはレコード長の2倍の大きさの領域であり，レコードを読み込む前にその後半部を前半部へ移送し，そののちＣＡＬＬする。レコードは後半部に読み込まれる。

(2) コーリングシーケンス

ＣＡＬＬ　ＲＥＡＤ（Ｐ）

Ｐ：読み込んだレコードの有効文字数がセットされる。

属性：NUMERIC

(3) ユーザ定義領域

① CG_BUFF

属性： CHARACTER(80)，配列(2)，EXTERNAL

（例）DCL CG_BUFF(2) CHAR(80) EXTERNAL ;

機能： ソース・プログラムが読み込まれるバッファである。

② CG_EOF

属性： LABEL，EXTERNAL

（例）DCL CG_EOF LABEL EXTERNAL ;

機能： ソース・プログラム読み込み終了時に，JUMPするラベルをセットする。

(4) 使 用 例

```
      PROC ……  ;
       DCL   CG_BUFF(2)…… ;
       DCL   CG_EOF…… ;
       CG_EOF=LAB_2 ;
LAB_1:CG_BUFF(1)=CG_BUFF(2) ;
       CALL READ;
         .
         .
         .
       GO TO LAB-1;
LAB_2:STOP ;
       END;
```

1.4.2 WRITE

(1) 機 能

SYSPRTにCG_BUFF(2)あるいはMESの内容を出力する。CG_BUFF(2)については1.4.1 READを参照すること。

(2) コーリングシーケンス

① CALL WRITE

CG_BUFF(2)の内容をSYSPRTに出力する。

② CALL WRITEM(MES,L)

長さLより成るMESの内容をSYSPRTに出力する。

MES： 出力情報をセットする。

属性： CHARACTER

L ： MESの長さである。

属性： NUMERIC

③ CALL WRITEINT

WRITEルーチンの初期設定を行なう。

(3) ユーザ定義領域

① CG_DATE

属性： CHARACTER(6),EXTERNAL

機能： 西暦・月・日（各2桁）がWRITEINTでセットされる。

② CG_LB1

属性： CHARACTER(90),EXTERNAL

機能： 各頁の頭に出力する見出し1をセットする。

③ CG_LB2

属性： CHARACTER(105),EXTERNAL

機能： 各頁の頭に出力する見出し2をセットする。

④ CG_IC

属性： NUMERIC,EXTERNAL

機能： 出力後の改行数をセットする。

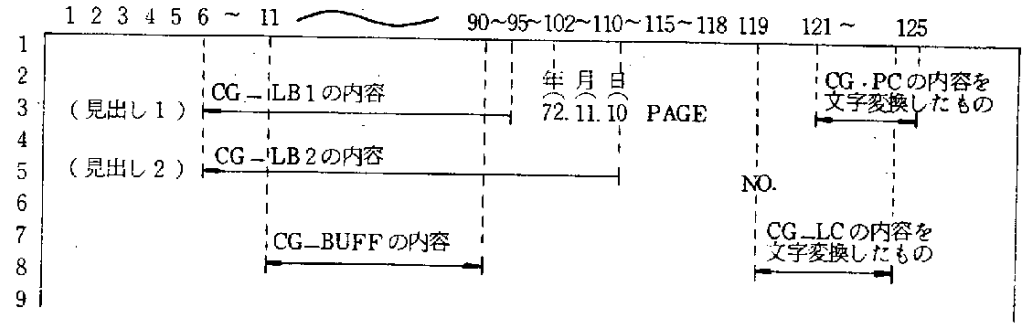
⑤ CG_TC

属性： NUMERIC,EXTERNAL

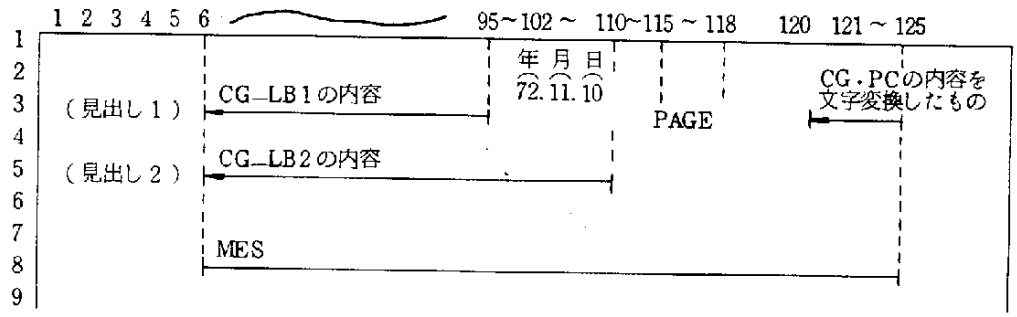
機能： 1頁内の出力行数をセットする。

(4) 出力フォーマット

① WRITEの場合



② WRITEMの場合



(5) 使 用 例

```
PROC ……… ;
  DCL CG_DATE …… ,
      CG_LB1 …… ,
      CG_LB2 …… ,
      CG_TC …… ,
      CG_IC …… ,
      CG_BUFF …… ,
      CG_EOF …… ,
      MES CHAR(7),INIT('**END**');
LAB 1: CG_LB1='PROBLEM-NAME AAA';
       CG_LB2='SOURCE LIST';
       CG_TC = 50;
       CG_IC = 2;
```

```
          CG_EOF=LAB_3;
          CALL WRITEINIT;
LAB_2:    CG BUFF(1)=CG_BUFF(2);
          CALL READ;
          CALL WRITE;
            .
            .
            .
          GO TO LAB_2;
LAB_3:    CALL WRITEM(MES,7);
          STOP;
          END ;
```

1.4.3 SCAN

(1) 機　能

ソース・プログラムの走査および移送等を行なう。

(2) コーリングシーケンス

① CALL INITP(P)

CG_BUFF(2)に新しいレコードを読み込む等の初期設定を行なう。

P：1回で入力するソース・プログラムの文字数をセットする。

属性：NUMERIC

内部ポインタ*：1

② CALL SCAN(CHR,P)

パラメータCHRと等しい文字を内部ポインタ以後のCG_BUFF(2)中にさがす。

CHR：　1文字をセットする。

属性：CHARACTER(1)

P　：　CHRと等しい文字位置がセットされる。

内部ポインタ：P＋1となる。

③ CALL SCANEXC(CHR,P)

パラメータＣＨＲと等しくない文字を内部ポインタ以後のCG_BUFF(2)中にさがす。

ＣＨＲ： １文字をセットする。

Ｐ ： ＣＨＲと等しくない文字位置がセットされる。

内部ポインタ： Ｐとなる。

④ CALL SCANSP(P,CONST)

特殊記号の文字を内部ポインタ以後のCG_BUFF(2)中にさがす。

Ｐ：（文字位置＋CONST）にセットされる。

CONST： 増減する値をセットする。

属性： NUMERIC

内部ポインタ： 特殊記号の文字位置となる。

⑤ CALL GET(CHR)

内部ポインタが示すCG_BUFF(2)中の１文字をＧＥＴする。

ＣＨＲ： １文字がセットされる。

内部ポインタ： ＋１される。

⑥ CALL SAVEP(P)

内部ポインタの保存を行なう。

Ｐ： 内部ポインタがセットされる。

内部ポインタ： 不変

⑦ CALL LOAD(P)

内部ポインタの変更を行なう。

Ｐ： 変更する値をセットする。ただし１≦Ｐである。

内部ポインタ： Ｐとなる。

⑧ CALL CHANGEP(P,P1)

Ｐの値を変更する。

Ｐ：（Ｐ＋Ｐ１）となる。

P１：　増減値をセットする。ただし１≦Ｐ＋Ｐ１≦内部ポインタである。

属性：NUMERIC

内部ポインタ：　不変

⑨　CALL　MOVE(CHR1,P,P1,MOJISU)

ＰとＰ１で示される範囲のCG_BUFF(2)をＣＨＲ１へ移送し，その文字数をMOJISUへセットする。

ＣＨＲ１：　移送される領域

属性：　CHARACTER

Ｐ，Ｐ１：　範囲を示す値をセットする。ただしＰ≦Ｐ１である。

MOJISU：移送された文字数がセットされる。

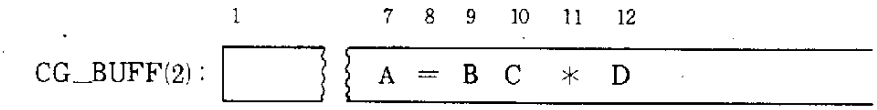
属性：　NUMERIC

内部ポインタ：　不変

＊内部ポインタとはCG_BUFF(2)の桁を示すものである。

(3)　使　用　例

CALL　INITP(80)終了時の状態を以下とする。



	結果	内部ポインタ
CALL SCANEXC('',P);	P＝1	7
CALL GET(CHR);	CHR='A'	8
：		
P=9,P1=10;		
CALL MOVE(CHR1,P,P1,MOJISU)	CHR1='BC' MOJISU=2	
CALL CHANGEP(P,-1)	P＝8	

1.4.4 MTWRITE

(1) 機　　能

FASPCHAR・FASPNUM・FASPSTRの内容を編集し，カード・イメージのFASP命令をソース形式でOBJECTファイルへ出力する。

(2) コーリングシーケンス

① CALL FASPOPEN

FASPNAMEにエレメント名をセットして最初にCALLする。OBJECTファイルのOPENと領域の初期設定を行ない，エレメント文の出力もする。

② CALL FASPOUT1

出力する命令のオペランド欄が１２文字以内である時使用され，FASPCHARに情報をセットしてCALLする。

③ CALL FASPOUT2

出力する命令のオペランド欄が数値型である時使用され，FASPNUMに情報をセットしてCALLする。

④ CALL FASPOUT3

出力する命令のオペランド欄が１３文字以上である時使用され，FASPSTRに情報をセットしてCALLする。

⑤ CALL FASPCLOSE

OBJECTファイルのCLOSEを行なう。

(3) ユーザ定義領域

① FASPNAME

属性：　CHARACTER(8)，EXTERNAL

機能：　エレメント名をセットする。

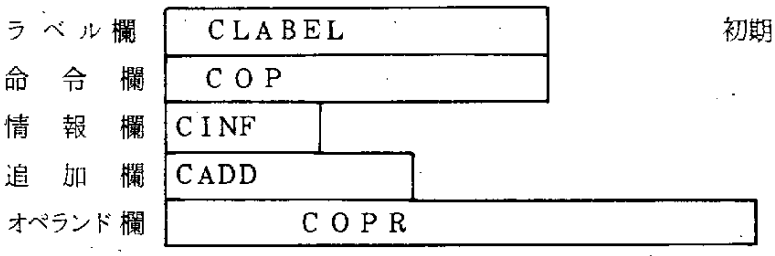
② FASPCHAR

属性：　STRUCTURE，EXTERNAL

機能：　オペランド欄が１２文字以内である場合に使用される。

```
DCL FASPCHAR STRUCTURE EXTERNAL,
```

```
        *CLABEL  CHAR(8),
        *COP     CHAR(8),
        *CINE    CHAR(3),
        *CADD,
        *COPR    CHAR(12);
```



初期値：スペース
スペース
スペース
ゼロ
スペース

○ CLABEL

FASP命令のラベルをセットする。

○ COP

FASP命令の命令をセットする。

○ CINF

命令追加情報	オペランド追加 INDEX	オペランド追加 BASE

○ CADD

FASP命令のオペランド追加情報をセットする。

○ COPR

FASP命令のオペランドをセットする。

③ FASPNUM

属性： STRUCTURE,EXTERNAL

機能： オペランド欄が数値型である場合に使用される。

```
DCL  FASPNUM STRUCTURE EXTERNAL,
       *NLAB  CHAR(8),
```

＊NOP　CHAR(8),

＊NINF,

＊NOPR;

ラベル欄	NLAB	初期値：スペース
命令欄	NOP	スペース
情報欄	NINF	スペース
オペランド欄	NOR	ゼロ

○NLAB，NOP，NINFはFASPCHARと同様である。

○NOP

FASP命令のオペランド（数値型）をセットする。

ただし，セットした値が0≦セット値≦262143である時，

LA　→　LAI
LR　→　LRI
AX　→　AXI
LX　→　LXI
LXC　→　LXIC
A　→　AI
S　→　SI
SX　→　SXI

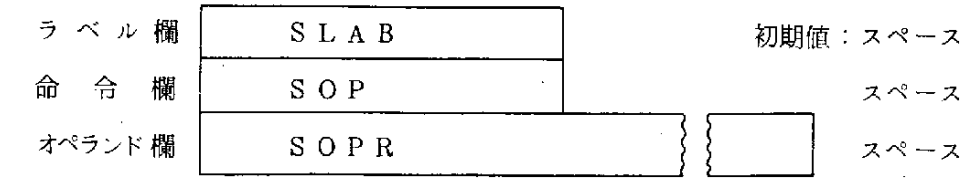
と変更される。上記以外の時は'＝0／03…'という形で出力される。

④ FASPSTR

属性：　STRUCTURE，EXTERNAL

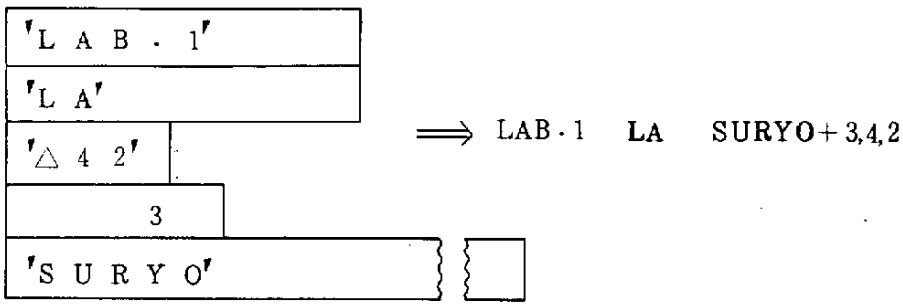
機能：　オペランド欄が13文字以上の時使用される。

```
DCL   FASPSTR STRUCTURE EXTERNAL,
        ＊SLAB  CHAR(8),
        ＊SOP   CHAR(8),
        ＊SOPR  CHAR(120);
```

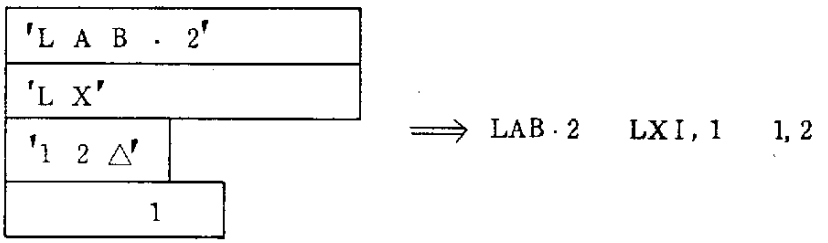


(4) 使用例

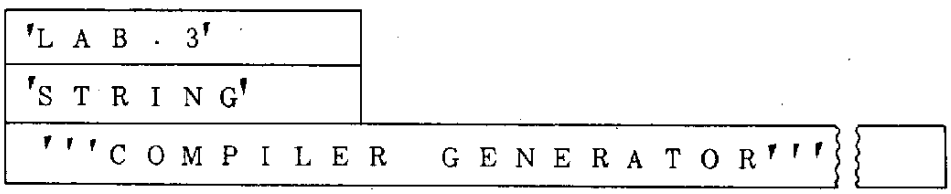
FASPCHAR



FASPNUM



FASPSTR



⟹ LAB・3 STRING 'COMPILER GENERATOR'

1.4.5 CG_PARAM

(1) 機　能

ジョブ制御文EXECのPARAMパラメータで与えられた，プログラムに対するパラメータ情報の読み込みと解析を行ない，特定領域のビットをON or OFFにしたり値をセットする。ユーザはこの領域をソース・プログラム等の出

力時のスイッチに使用できる。

① PARAパラメータの記述形式

¥EXEC ルーチン名，PARAM='list1,list2,……, listn'

② CG_PARAMが用意しているlistは(表1－2)であり，これ以外のlistを使用する時はユーザ自身がそれらの処理を行なう。

表1－2

list	処理
LIST	CG_LISTの1ビット目 ON
NOLIST	CG_LISTの1ビット目 OFF
FASP	CG_LISTの2ビット目 ON
MAP	CG_LISTの3ビット目 ON
DEBUGC =B(01…)	B(01…)の内容をCG_DEBUGにセットする。( )内はBIT列。
BOUND =(n,m)	CG_BOUNDの配列にn，mをセットする。n；mは数値定数。

(2) コーリングシーケンス

CALL CG_PARAM

(3) ユーザ定義領域

① CG_LIST

属性： BIT,EXTERNAL

機能： PARAMパラメータの内容に従って対応するビットがONされる。

② CG_DEBUG

属性： BIT,EXTERNAL

機能： PARAMパラメータで'DEBUGC＝B(101……)'が指定されている場合，B(101……)で示されるビットパターンに従って対応ビットがON or OFFされる。

③ CG_BOUND

属性：　NUMERIC，配列(2)，EXTERNAL

機能：　PARAMパラメータで 'BOUND＝(2,72)' が指定されている場合，CG_BOUND(1)に2，CG_BOUND(2)に72 がセットされる。

2．ＪＰＡＣの評価

2　ＪＰＡＣの評価

ＪＰＡＣを作成した目的は実用性のあるコンパイラ・ジェネレータが作成可能であるか否かを検討することである。この評価は実際にＪＰＡＣを用いてある言語のコンパイラを作成しその作成コスト及び性能を，手書きのコンパイラ（アセンブラ及びＰＬ／Ｉで作成）と対比することによって行なった。

2.1　評価の目的と方法

評価の目的はＪＰＡＣの実用性を調べることである。このようなシステムの実用性を論じる場合その視点は２つある。第１点はＪＰＡＣを使用することによりコンパイラ作成コスト（労力＋計算機使用時間）が手書きと比べてどの程度節減できるかということである。第２点はＪＰＡＣにより作成されたコンパイラが，コアサイズ・コンパイル速度において手書きのものとどれ程差があるかということである。以上の２点を評価するために，ＪＩＳ３０００レベルのFORTRANコンパイラをアセンブラ・ＰＬ／Ｉ・ＪＰＡＣの３方法を用いて作成し，それぞれに要した労力・計算機使用時間の比較と，作成されたコンパイラの性能を比較した。そしてこれらの結果を検討しＪＰＡＣの実用性に関する評価を行なった。

なお，作成するコンパイラ言語をＪＩＳ３０００レベルのFORTRANとしたのは，

① ある限られた期間内に３つのコンパイラを作成する為には，かなり言語仕様をしぼる必要があった。

② FORTRANは最もポピュラーな言語であり，この評価目的の例として取り上げるには適している。

③ ３通りの方法で一応コンパイラを作成したが，期間の制約からランタイム・

ルーチンは既存のFORTRANコンパイラのものを利用する。

等である。

手書き用言語としてアセンブラ及びＰＬ／Ｉを選んだ理由は次のとおりである。

① アセンブラは現在までのところコンパイラ記述の中心言語であり評価の物差しとなり得ると考えられる。

② 最近は汎用言語でコンパイラを記述することが多くなってきており，その中でもＰＬ／Ｉはシステム記述に最適と思われる。

2.2 ＦＯＲＴＲＡＮコンパイラの機能と言語仕様

2.2.1 機 能 概 略

FORTRANコンパイラの機能は（図２－１）で示されるように，ＳＹＳＩＮファイル（ＳＹＳＩＮ形式）からカード・イメージのFORTRANソース・プログラムを読み込み，ＯＢＪＥＣＴファイルにFASPソースの形式でオブジェクトを出力する１パス１フェーズのコンパイラである。

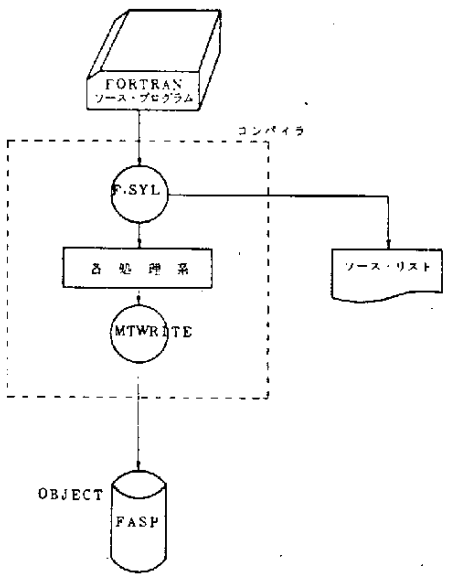


図２－１　コンパイラの概略

2.2.2 言語仕様

FORTRANはＪＩＳ（日本工業規格）３０００の水準である。

但し以下の制限及び変更を設ける。

(1) 制限点

① ＦＵＮＣＴＩＯＮ文を除く。

② 文関数を除く。

③ ＥＱＵＩＶＡＬＥＮＣＥ文を除く。

④ ＳＵＢＲＯＵＴＩＮＥ文に引数は書けない。

⑤ ＰＡＵＳＥ文を除く。

⑥ 継続行は１９行までとする。

⑦ 入出力文の機番は１～８までとする。

⑧ 空白はセパレータになり得る。

⑨ 予約語を設ける。

⑩ ＲＥＷＩＮＤ文を除く。

⑪ ＢＡＣＫＳＰＡＣＥ文を除く。

⑫ Name Table、定数Table，COMMON Tableの大きさは夫々最大１０００個まで，ＤＯのネストは最大２０までとする。

(2) 変更点

① ＤＯ型並びの制御変数に配列要素も許す。

② 配列要素の添字式に一般の整数型の式を許す。

③ 算術式の項はＪＩＳでは「因子か　または　項／因子　または　項＊項」となっているが，これを「因子か　または　項／因子　または　項＊因子」と変える。

(3) 予約語

ＤＩＭＥＮＳＩＯＮ

ＣＯＭＭＯＮ

ＲＥＡＤ

ＷＲＩＴＥ

FORMAT

GO TO

DO

CONTINUE

IF

STOP

CALL

SUBROUTINE

RETURN

END

関数名

2.3 ＪＰＡＣによるコンパイラの作成

FORTRANコンパイラの作成にあたり，FORTRANのSyntax をできるだけ規模の小さい範囲で収まるようにし，又Lexical-Analyzer は Basic Symbol を定義する方法を用いて，実行時の性能がよいコンパイラを作成することに最大の努力を払った。

2.3.1 FORTRANコンパイラの構造

ＪＰＡＣによって作成されたFORTRANコンパイラの構造は（図2－2）で示すとおりである。

図上□で囲まれたモジュールはＳＴＡＧＥにより作成されたものである。⬭で囲まれたモジュールは他のコンパイラ（アセンブラ，ＰＬ／Ｉ）と共通のユーティリティーであり，それ以外はＪＰＡＣのセマンティックス・ルーチンである（CG_MAINは除く）。詳細は後節で述べる。

次にセマンティックス・ルーチン以外のモジュールの機能概略を示す。

① CG_MAIN……ＪＰＡＣにおけるコントローラである。

② CG_SENI ……遷移テーブルが集って１モジュールとなっている。

③ F・SYL ……Syllable Read ルーチンでＦＯＲＴＲＡＮソースを区切り記号で分離したり，必要があれば加工したりする。

④ MTWRITE ……ＦＡＳＰ命令をgenerate してTape あるいはDisc へ書き込む。

⑤ F・ERROR ……エラー処理ルーチンでエラーメッセージの出力，不必要なソースデータの読み飛ばし等を行なう。

⑥ FTERM ……ユーティリティーの最終処理（File のClose等）を行なう。又エラーの数を出力する。

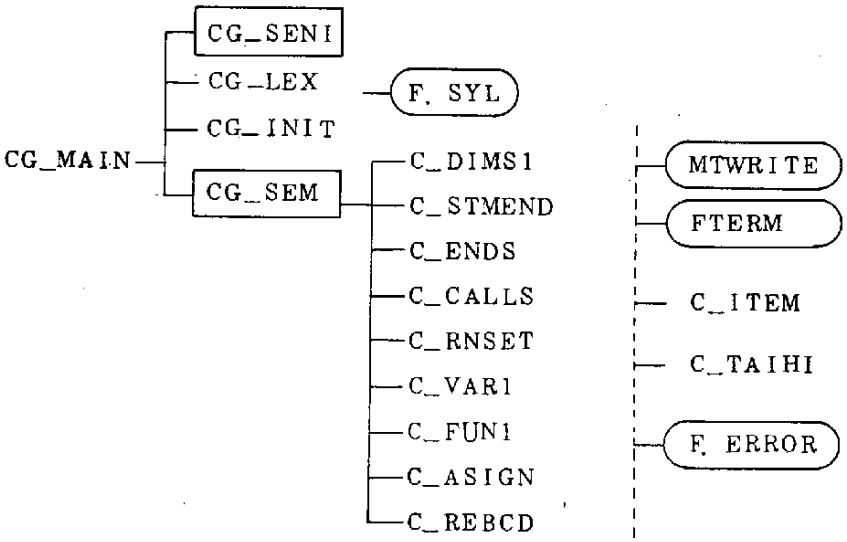


図２－２　コンパイラの構成

2.3.2　シンタックス記述部

ＪＰＡＣのシンタックス記述部は，作成コンパイラの構文を示すものであり，BASIC SYMBOL，BASIC STATEMENT，SYNTAX RULEの３節から構成されている。各節の詳細は1.3を参照されたい。

A． BASIC SYMBOL

BASIC SYMBOLとしてはFORTRANのKey語（DO,GO TO…etc）以外に次のものを宣言した。

① ＜START＞……コンパイルのはじまりを示す。

② ＜FORM LIST＞……FORMAT list。

③ ＜FUN NAME＞……関数名。

④ ＜LABEL＞……文番号。

⑤ ＜IDENTIFIER＞……名前。

⑥ ＜RNUMBER＞……実定数。

⑦ ＜INUMBER＞……整定数。

⑧ ＜STM END＞……ステートメントend。

⑨ ＜BEKI＞……ベキ演算子。

B． BASIC STATEMENT

BASIC STATEMENTは文を意味する＜STATEMENT＞を指定した。その理由は，FORTRANにおいてエラーが検出されたとき該当する文を読みとばせば処理は継続できるからである。

C． SYNTAX RULE

SYNTAX RULEはnonterminal記号41個，生成規則84個で構成されている。構文作成にあたっては次の2点に留意した。

① 構文記述をキメ細かくセマンティックス・ルーチンの作成を容易にする。

② 実行時の速度を向上させるため，look-back個数look-ahead回数を減らす。

その結果，生成規則，nonterminal記号の増加が生じた。ここでとりあげたFORTRANとJIS FORTRANの差異はすでに2.2.2で述べたが，そのうち特に構文上の差異は次の3点である。

① DO型並びの制御変数に配列要素も許す。

② 配列要素の添字に式を許す。

③　算術式の項はＪＩＳでは「因子か　または　項／因子　または　項＊項」となっているが，これを次のように変える「因子か　または　項／因子　または　項＊因子」。

以下に前述の制限を加えたＦＯＲＴＲＡＮに対するＳＹＮＴＡＸ記述部（表２－１）を示す。

表２－１　　ＳＹＮＴＡＸ記述部

```
SYNTAX DEFINITION;
  BASIC SYMBOL;
    <SUBROUTINE>,<COMMON>,<DIMENSION>,<READ>,<WRITE>,<FORMAT>,
    <GO>,<DO>,<IF>,<CALL>,<RETURN>,<STOP>,<END>,<CONTINUE>,
    <TO>,<FORM LIST>,<START>,<IDENTIFIER>,<LABEL>,<RNUMBER>,
    <INUMBER>,<FUN NAME>,<REXT>,<STM END>;
  BASIC STATEMENT;
    <STATEMENT>;
  SYNTAX RULE;

    <PROG>::=<START><STM LIST><END>;C_ENDS;
    <STM LIST>::=<STM LIST><STATEMENT>;C_STMEND(2)
                |<STATEMENT>;C_STMEND(1);
    <STATEMENT>::=<DCL STM>
                |<EXEC STM> |<LABEL><EXEC STM>;C_LBSTMS
                |<FORM STM> ;
    <DCL STM> ::=<SUB STM>
                |<DIM STM>
                |<COM STM>      ;
    <FORM STM> ::=<LABEL><FORMAT><FORM LIST><STM END> ;C_FORMS;
    <EXEC STM> ::=<ASSIGNMENT>
                |<CONTINUE><STM END>
                |<CALL STM>
                |<RETURN STM>
                |<STOP STM>
```

```
                    |<IF STM>
                    |<GO STM>
                    |<DO STM>
                    |<IO STM>    ;
<DIM STM> ::=<DIM DCL><STM END> ;
<DIM DCL> ::=<DIMENSION><DIM ELM>
                    |<DIM DCL>,<DIM ELM>          ;
<DIM ELM> ::=<DIM HEAD><INUMBER>) :C_DIMS1(C_DIMSW)          ;
<DIM HEAD>::=<IDENTIFIER>(          :C_DIMS2(C_DIMSW)
                    |<DIM HEAD><INUMBER>, :C_DIMS1(C_DIMSW)          ;
<COM STM> ::=<COM DCL><STM END>                              ;
<COM DCL> ::=<COMMON><IDENTIFIER> :C_COMS(1)
                    |<COM DCL>,<IDENTIFIER> :C_COMS(2)          ;
<ASSIGNMENT>::=<VARIABLE>=<EXPRESSION><STM END>:C_ASIGN;
<FUN CALL>::=<FUN HEAD><EXPRESSION>): C_FUN1;
<FUN HEAD>::=<FUN NAME>(: C_FUN2
                    |<FUN HEAD><EXPRESSION>,: C_FUN3;
<EXPRESSION>    ::=<TERM>|+<TERM>| -<TERM>:C_EXCNV
                    |<EXPRESSION>+<TERM>: C_ADD
                    |<EXPRESSION>-<TERM>:C_SUB;
<TERM>          ::=<FACT>|<TERM>/<FACT>:C_DIVIDE
                    |<TERM>*<FACT>:C_MULT;
<FACT>          ::=<PRIMARY>
                    |<PRIMARY><REKI><PRIMARY>:C_REKI;
<PRIMARY>       ::=<VARIABLE>
                    |<FUN CALL>
                    |<RNUMBER>:C_RNSET|<INUMBER>;   C_INSET
                    |(<EXPRESSION>): C_PRM;
<VARIABLE>      ::=<IDENTIFIER>:C_VAR1
                    |<VAR HEAD><EXPRESSION>):C_VAR2  ;
<VAR HEAD>      ::=<IDENTIFIER>( :C_VARH
                    |<VAR HEAD><EXPRESSION>,:C_VAR3;
<SUB STM>  ::=<SUBROUTINE><IDENTIFIER><STM END>:C_SUBS;
<CALL STM> ::=<CALL><IDENTIFIER><STM END>: C_CALLS;
```

```
<RETURN STM>::=<RETURN><STM END>:C_RETNS;
<STOP STM> ::=<STOP><STM END>:C_STOPS;
<GO STM>   ::=<GO><TO><INUMBER><STM END>: C_GOS(1)
           |<GO HEAD><INUMBER>,<IDENTIFIER><STM END>:C_GOS(2);
<GO HEAD>  ::=<GO><TO>( :C_GOHEAD(1)
           |<GO HEAD><INUMBER>,: C_GOHEAD(2);
<DO STM>   ::=<DO HEAD><INUMBER><STM END>:C_DOS(1,DOCOND)
           |<DO HEAD><IDENTIFIER><STM END>: C_DOS(2,DOCOND);
<DO HEAD>  ::=<DO><INUMBER><IDENTIFIER>=  :C_DOH1(DOCOND)
           |<DO HEAD><INUMBER>,:C_DOH2(1,DOCOND)
           |<DO HEAD><IDENTIFIER> ,: C_DOH2( 2,DOCOND);
<IF STM>   ::=<IF HEAD><EXPRESSION>)<INUMBER>,<INUMBER>,
           <INUMBER><STM END>: C_IFS;
<IF HEAD>  ::=<IF>(;
<IO STM>::=<IO OP><IO LIST><STM END>:C_IOEND
           |<IO OP><STM END>:C_IOEND1;
<IO OP>:::=<READ OP>|<WRITE OP>;
<READ OP>:::=<READ>(<IDIN>,<INUMBER>):C_REBCD
             |<READ>(<IDIN>):C_RBCD;
<WRITE OP>:::=<WRITE>(<IDIN>,<INUMBER>):C_WEBCD
              |<WRITE>(<IDIN>):C_WBCD;
<IDIN>:::=<IDENTIFIER>:C_VAR1
          |<INUMBER>:C_INSET;
<IO LIST>:::=<IO ELM>|<IO LISTH><IO ELM>;
<IO LISTH>::=<IO LIST>,;
<IO ELM>:::= <VARIABLE>:C_IOS1
           |<IOGH><IOGP>;
<IOGH>:::=(:C_IOGS;
<IOGP>:::=<IO LISTH><DO SPEC>)
          |<IO LIST>):C_DOSE2;
<DO SPEC>:::=<VARIABLE>=<IDIN>,<IDIN>,<IDIN>:C_DOSE11
             |<VARIABLE>=<IDIN>,<IDIN>:C_DOSE12;
SYNTAX RULE END;
```

2.3.3　セマンティックス記述部

ＪＰＡＣのセマンティックス記述部は，シンタックス記述部分にあらわれるセマンティックス・ルーチンに対応し，＂意味＂を与えるプログラムの集合である。セマンティックス記述部はCommon Variable List とSemantics Procedure List より構成されるが，その詳細は1.3を参照されたい。

A．　Common Variable List

ＪＰＡＣで作成したＦＯＲＴＲＡＮコンパイラではセマンティックス・ルーチンのＥＸＴＥＲＮＡＬ属性を持つ領域は原則としてCommon Variable とした。

以下にCommon Variable とした領域の概要を述べる。

① ＣＧ＿ＮＯＮＴ …… nonterminal 記号に関する情報をセットする。

② ＣＧ＿ＮＡＭＥＴ …… 名前及び文番号に関する情報をセットする。

③ ＣＧ＿ＣＯＭＴ …… ＣＯＭＭＯＮ宣言された変数名に対する情報をセットする。

④ ＣＧ＿ＫＡＮＴ …… 関数に対する情報をセットする。

⑤ ＣＧ＿ＤＯＳＴＣ …… ＤＯ文に対する情報をセットするスタッカー。

⑥ ＣＧ＿ＳＴＡＣＫ，ＣＧ＿ＳＴＲ，ＣＧ＿ＢＦ，ＣＧ＿ＳＯＲＴ，ＣＧ＿ＣＣ　Lexical Analyzer 及びコントローラからの情報セットのために用いる。

⑦ ＦＡＳＰＣＭＴ，ＦＡＳＰＲＥＣ，ＦＥＬＮＡＭＥ　オブジェクト出力用情報をセットする。

⑧ ＣＧ＿ＣＯＮＴ …… 定数に関する情報をセットする。

⑨ ＳＹＬＣＯＮ …… Syllable Read の情報をセットする。

．
．
．

etc．

B ・　Semantics Procedure List

Semantics Procedure Listは，オブジェクトをgenerateするセマンティックス・ルーチンの集合からなりＬＳＤ言語で記述されている。セマンティックス・ルーチンの作成にあたっては，関連する処理は１つのEXTERNAL PROCEDUREにまとめるようにした。

以下にセマンティックス・ルーチンの概要を述べる。

① ＣＧ＿ＬＥＸ …… Lexical Analyzer。

② ＣＧ＿ＩＮＩＴ …… 各種テーブルの初期設定。

③ Ｃ＿ＳＴＭＥＮＤ …… ステートメント end 処理。

④ Ｃ＿ＥＮＤＳ …… ＥＮＤ行処理。

⑤ Ｃ＿ＤＩＭＳ１ …… ＤＩＭＥＮＳＩＯＮ文処理。
ＣＯＭＭＯＮ文処理。

⑥ Ｃ＿ＩＴＥＭ …… 項に対する処理。

⑦ Ｃ＿ＶＡＲ１ …… 変数に対する処理。

⑧ Ｃ＿ＦＵＮ１ …… 関数に対する処理。

⑨ Ｃ＿ＡＳＩＧＮ …… 代入文，式に対する処理。

⑩ Ｃ＿ＲＮＳＥＴ …… 整定数，実定数に対する処理。

⑪ Ｃ＿ＣＡＬＬＳ …… ＣＡＬＬ文，ＲＥＴＵＲＮ文，ＩＦ文，ＧＯＴＯ文，ＤＯ文，ＳＴＯＰ文，ＳＵＢＲＯＵＴＩＮＥ文，ＤＯの端末文の処理。

⑫ Ｃ＿ＲＥＢＣＤ …… ＲＥＡＤ文，ＷＲＩＴＥ文に対する処理。

（表２－２）にＳＥＭＡＮＴＩＣＳ記述部の一部を示す。

表２－２　SEMANTICS記述部

```
SEMANTICS DEFINITION;
      DCL  CG_NAMET(1000)TABLE CONTROLLED(CG_NAMEP),                    TBL00010
           * CG_NAY1 CHAR(8),                                           TBL00030
           * CG_NAY2 BIT(36),                                           TBL00040
           * CG_NAY3,                                                   TBL00050
           * CG_NAY4;          /* NAME TABLE */                        TBL00060
      DCL  CG_COMT(1000) TABLE CONTROLLED(CG_COMP),                     TBL00070
           * CG_COMY1 POINTER ;                                         TBL00080
                               /* COMMON POINTER TABLE */               TBL00090
      DCL  CG_CONT(1000) TABLE CONTROLLED(CG_CONP),                     TBL00100
           * CG_CONY1 BIT(36),                                          TBL00110
           * CG_CONY2 NUMERIC;                                          TBL00120
                               /* CONSTANT TABLE  */                    TBL00130
      DCL  CG_KANT(13)  TABLE CONTROLLED(CG_KANP),                      TBL00140
           * CG_KANY1 CHAR(8)                                           TBL00150
      INIT('ABS','IABS','FLOAT','IFIX','SIGN','ISIGN',                  TBL00151
           'EXP','ALOG','SIN' ,'COS','TANH','SQRT','ATAN') ,            TBL00152
           * CG_KANY2 BIT(36)                                          TBL00160
      INIT (8'0110',B'0000',B'0100',B'1010',B'0111',B'0000',            TBL00161
           7*B'1110') ,                                                 TBL00162
           * CG_KANY3 NUMERIC                                          TBL00170
      INIT(4*1,2*2,7*1)      ;                                         TBL00171
                               /* KANSU TABLE */                        TBL00180
      DCL  CG_DOSTC(20)  STACK,                                         TBL00190
           * CG_DOST1  POINTER,                                        TBL00200
           * CG_DOST2  NUMERIC,                                        TBL00210
           * CG_DOST3  NUMERIC,                                        TBL00220
           * CG_DOST4  POINTER,                                        TBL00230
           * CG_DOST5  BIT(35),                                        TBL00240
           * CG_DOST6  POINTER;                                        TBL00250
                               /* DO  STACK  */                         TBL00260
      DCL  CG_STACK(100)  STACK,                                        TBL00270
           * CG_STY1  BIT,                                             TBL00280
           * CG_STY2  POINTER; /*  CG STACK  */                       TBL00290
      DCL  CG_STR     STRUCTURE EXTERNAL,                               TBL00300
           * CG_STRY1 BIT,                                             TBL00310
           * CG_STRY2 POINTER;                                         TBL00320
                               /*  CG STRUCTURE  */                    TBL00330
      DCL  CG_BF  CHAR(256) EXTERNAL ;                                  TBL00340
      DCL  CG_SORT NUMERIC  EXTERNAL ;                                  TBL00350
      DCL  CG_CC   POINTER  EXTERNAL ;                                  TBL00360
      DCL  CG_COND BIT(36)  EXTERNAL ;                                  TBL00370
      DCL  SLIST   BIT(36)  EXTERNAL ;                                  TBL00380
      DCL  FASPCMI STRUCTURE EXTERNAL,                                  TBL00390
           *LABEL CHAR(8) ,                                             TBL00400
           *OPRAT CHAR(8) ,                                             TBL00410
           *TUICA CHAR(4) ,                                             TBL00420
           *LEN   NUMERIC ,                                             TBL00430
           *OFSET NUMERIC ,              /* FASPCMI  */                 TBL00440
     *OPRND CHAR(8), *OPRTT CHAR(112);                                  TBL00441
      DCL  FASPREC  CHAR(80) EXTERNAL  ;                                TBL00450
      DCL  FELNAME  CHAR(8)  EXTERNAL  ;                                TBL00460
      DCL  SYLCOM STRUCTURE  EXTERNAL,                                  TBL00470
           *SYLSTL NUMERIC,                                             TBL00480
     *SYLBF CHAR(12),*SYLBRR CHAR(8),                                   TBL00490
           *SYLSORT NUMERIC,                                            TBL00500
           *SYLCC   NUMERIC,             /* SYLREAD */                  TBL00510
           *SYLWC   NUMERIC,                                            TBL00520
           *SYLEM   NUMERIC,                                            TBL00530
           *SYLWSE(2),                                                  TBL00540
           *SYLBF2(230) CHAR(4),                                        TBL00550
           *EOFAD   LABEL  ,*SYL_DM;                                    TBL00560
      DCL  CG_NONT(100) TABLE CONTROLLED(CG_NP),                        TBL00570
           *CG_NY1 BIT(36),                                             TBL00580
           *CG_NY2  POINTER,                                            TBL00590
           *CG_NY3  NUMERIC;                                            TBL00600
```

```
                                    /* NONT JYOHO TABLE */                TBL00610
        DCL  CG_MAX  EXTERNAL;      /* L.WK PRG MAX     */                TBL00620
        DCL  CG_WK   EXTERNAL;      /* L.WK STM MAX     */                TBL00630
        DCL  CG_IN   LABEL EXTERNAL ;                                     TBL00640
                                    /* NEXT PROG SHORI  */                TBL00650
                                    /* JUMP  BANCHI     */                TBL00660
        DCL  CG_STWK STRUCTURE EXTERNAL,                                  TBL00670
             *CG_STWY1,*CG_STWY2 POINTER;                                 TBL00680
        DCL FASPCMT1  STRUCTURE,                                          TBL00690
            *FASY1 CHAR(28),                                              TBL00700
            *FASY2,                                                       TBL00710
            *FASY3 CHAR(116);                                             TBL00720
        EQU (FASPCMT,FASPCMT1);                                           TBL00730
        DCL C_IF EXTERNAL;                                                TBL00740
                                          /*                  */          TAI00020
                                          /*  C.TAIHI         */          TAI00030
                                          /*                  */          TAI00040
                                          /*          Y.AZUMA */          TAI00050
        DCL  TAID(3) CHAR(40)  ;                                          TAI00070
        DCL  FASWKS  STRUCTURE,                                           TAI00080
             *FY1 CHAR(8),                                                TAI00090
             *FASWK  CHAR(40)  ;                                          TAI00100
        EQU (FASPCMT,FASWKS)   ;                                          TAI00110
        TAID(1) = FASWK ;  RETURN ;                                       TAI00120
C_RESET: ENTRY(I) ;                                                       TAI00130
        FASWK =TAID(1) ;   RETURN ;                                       TAI00140
C_TAIHI2: ENTRY(I);                                                       TAI00150
        TAID(1)=FASWK; TUICA='   C'; LEN=8; OFFSET=0; OPRND='  ';        TAI00160
        RETURN; END;                                                      TAI00170
```

2.4 アセンブラ及びＰＬ/Ｉによるコンパイラの作成

アセンブラ及びＰＬ/Ｉで作成するコンパイラの構造，各処理ルーチンの概略を以下に示す。

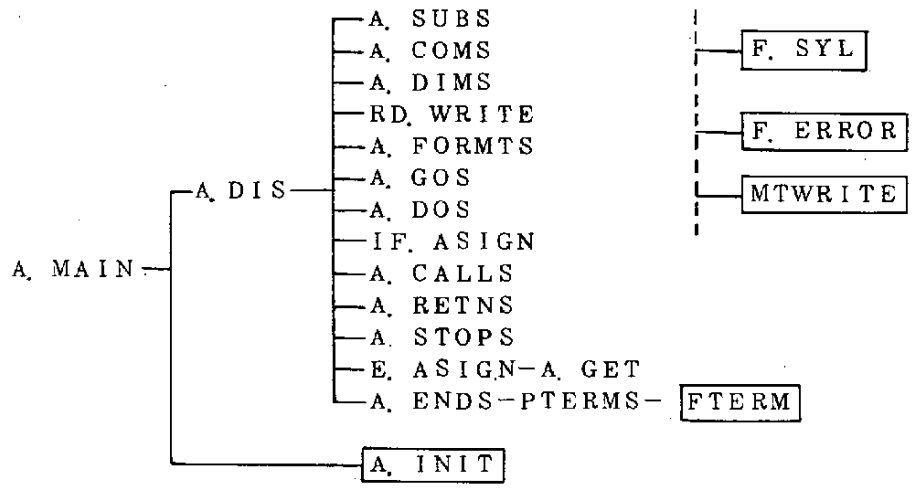
2.4.1 FORTRANコンパイラの構造

作成コンパイラは，１パス１フェースで，メイン・ルーチン（Ａ・ＭＡＩＮ）で各種作業領域及び各ユーティリティー・ルーチンの初期設定を行ない終了後，文識別ルーチン（Ａ・ＤＩＳ）へ制御を渡す。

文識別ルーチンでは，文の種類を判別し各ステートメント処理ルーチンに制御を渡す。

そして，ＥＮＤ行の処理終了後，制御は再びメイン・ルーチンに戻る。メイン・ルーチンでは，入力プログラムがまだあれば再度コンパイルを開始し，なければコンパイル作業を終了する。

各処理ルーチンの位置関係は（図２－３）で示される。



備考 □ で囲んであるものは，ユーティリティー・ルーチンである。

図２－３　コンパイラの構造

2.4.2 各処理ルーチン概略

(1) A・MAIN

コンパイラのメイン・ルーチンで，以下の処理を行なう。

① 各種テーブル及び作業領域の初期設定。

② A・INITルーチンをコールすることにより，各ユーティリティーの初期設定を行なう。

③ 初期設定終了後A・DISに制御を渡す。

(2) A・DIS

コンパイラの文識別ルーチンで以下の処理を行なう。

① Syllable Readルーチン F・SYLを用い，現在対象としているステートメントが何かを判断し，各ステートメント処理ルーチンに制御を渡す。

② 文番号の出力処理及びName Tableへの登録処理を行なう。

③ 作業領域PLMAXの値のセット。

④ 最初のステートメントがSUBROUTINE文でないときこのプログラムは，MAINルーチンとし必要な処理を行なう。

(3) A・COMS

COMMON文に対する処理ルーチンで，以下の処理を行なう。

① COMMONで宣言された配列名，変数名をCOMMON Table及びName Tableへ登録する。

(4) A・DIMS

DIMENSION文に対する処理ルーチンで，以下の処理を行なう。

① DIMENSIONで宣言された配列名を，Name Tableへ登録する。

(5) A・SUBS

SUBROUTINE文に対する処理ルーチンで以下の処理を行なう。

① SUBROUTINE文に対するオブジェクトをgenerateする。

(6) RD・WRITE

READ文及びWRITE文に対する処理ルーチンで,以下の処理を行なう。

① E・ASIGNルーチンを使用してREAD文に対するオブジェクトをgenerateする。

② E・ASIGNルーチンを使用してWRITE文に対するオブジェクトをgenerateする。

(7) A・FORMTS

FORMAT文に対するルーチンで以下の処理を行なう。

① FORMAT文に対するオブジェクトをgenerateする。

(8) A・GOS

GO TO文に対する処理ルーチンで以下の処理を行なう。

① GO TO文に対するオブジェクトをgenerateする。

② GO TO文に出現した文番号が未出現ならば，Name Table へ未定義として登録する。

(9) A・DOS

DO文に対する処理ルーチンで，以下の処理を行なう。

① DO文に対するオブジェクトをgenerateする。

② DO endの為の情報をDOスタッカーにスタックする。

(10) IF・ASIGN

IF文に対する処理ルーチンで以下の作業を行なう。

① E・ASIGNルーチンを利用してIF文に対するオブジェクトをgenerateする。

(11) A・CALLS

CALL文に対する処理ルーチンで以下の作業を行なう。

① CALL文に対するオブジェクトをgenerateする。

その際，該当サブルーチン名がName Table上に未登録ならば登録する。

(12) A・RETNS

RETURN文に対する処理ルーチンで，以下の作業を行なう。

① RETURN文に対するオブジェクトをgenerateする。

(13) A・STOPS

STOP文に対する処理ルーチンで，以下の作業を行なう。

① STOP文に対するオブジェクトをgenerate する。

(14) E・ASIGN

このルーチンは入口名がE・ASIGN，E・IF，E・LIST の3つありそれぞれ以下の処理を行なう。

① ASSIGNステートメントに対するオブジェクトをgenerate する。

② IF文のカッコ内の式に対するオブジェクトをgenerate する。

③ 入出力並びの項に対するオブジェクトをgenerate する。

このルーチンは下請けルーチンとしてA・GETを持つ。

(15) A・ENDS

END行に対する処理ルーチンで，以下の処理を行なう。

① END行に対するオブジェクトをgenerate する。

(16) PTERMS

コンパイラ本体の最終処理を行なう。なお，PTERMSの下請けルーチンとしてユーティリティーの最終処理を行なうFTERMがある。

2.5 コンパイラ作成段階における評価

JPACによるコンパイラの作成は，作成コスト節減の点で満足のいくものであった。又コンパイラ作成時において，JPACに関する問題点がいくつか指摘されたので以下に述べる。

2.5.1 JPACにおける問題点

(1) 予約語の設定

予約語を設けなかった場合次のことがいえる。

① 名前（英文字で始まる文字列）を認識したとき，それが関数名か，Key語

か，変数名かを区別することはLexical Analyzerの段階では不可能である。

② 名前が関数名か変数名かの区別をつけることをＢＮＦでは表現できない。

③ Key語をＢＮＦで表現することは可能だが，作成コンパイラの処理速度が極端に遅くなると思われる。

反面，関数名とKey語を予約語としても使用者はさほど不便を感じないであろうという前提のもとで，今回は関数名及びKey語を予約語とし，Lexical Analyzer段階で区別できる形とした。

(2) エラー処理

構文解析の段階でエラーを見つけた場合，とるべき処置は以下のとおりである。

① 不必要な情報を捨て，遷移テーブルのドライブ位置の修正を行なう。

② 適切なエラーメッセージの出力とリカヴァリー可能な場合の回復処理。

①に関してはFORTRANは文の集合と考えられるのでエラー発生の文を読みとばし，コントローラは文に該当するnonterminal記号＜STATEMENT＞を認識した直後の状態にドライブ位置をセットすることとした。

②に関しては最も新しく出現したKey語を保存しておき，それとエラー原因となった記号とによってエラーメッセージを決定する方法とし，またリカヴァリーは一切行なわないこととした。

上記のエラー処理の欠点としては次のものがあげられる。

① エラー発生の場合その文が読みとばされる為，エラー箇所以後の構文チェックがなされない。

② Key語を保存しておくという余計な操作が入る。

以上の欠点にもかかわらず上記のエラー処理方法を採用した理由は以下のとおりである。

① ある文にエラーがあったとき，その文のエラー箇所以降の構文チェックを行なっても労力に比して効果は余りない場合が多い。

② Key語を保存する作業は簡単で実行時の速度にも余り影響はないと考えられる。試作コンパイラの程度のものならばKey語が判っているだけでかなり適切なエラーメッセージを出力できると考えられた。

(3) 入出力文の構文

① （表２－１）の構文によると<DO SPEC>は次のようになっている。

<DO SPEC>::=<VARIABLE>=<IDIN>,<IDIN>,<IDIN>
:C_DOSE11
|<VARIABLE>=<IDIN>,<IDIN>:C_DOSE12;

<DO SPEC>をJIS3000どおりに書くと次のようになる。

<DO SPEC>::=<IDENTIFIER>=<IDIN>,<IDIN>,<IDIN>
:C_DOSE11
|<IDENTIFIER>=<IDIN>,<IDIN>:C_DOSE12;

構文をJISどおりにした場合，<IDENTIFIER>を認識したとき<VARIABLE>に上げるべきか否かでinadequate状態が起こる。このinadequate状態はL(2)R(1)でないと解決できない。文法を（表２－１）で示すように拡張すると上記inadequate状態は回避され処理速度，コアサイズの両面で作成コンパイラの性能は大幅に改善できると考えられる。

② （表２－１）の構文によると<IO LIST>及び<IOGP>は次のようになっている。

<IO LIST>::=<IO ELM>|<IO LISTH><IO ELM>;

<IO LISTH>::=<IO LIST>,;

<IO GP>::=<IO LISTH><DO SPEC>)
|<IO LIST>):C_DOSE2;

最初考えた構文は次のとおりである。

<IO LIST>:=<IO ELM>|<IO LIST>,<IO ELM>;

<IO GP>::=<IO LIST>,<DO SPEC>)
|<IO LIST>):C_DOSE2;

最初の構文ではlook-back数が2のものが（表2－1）の構文だと1で処理でき，遷移テーブルの大幅な減少（2068語が1274語となった）ができ，そして処理速度の大幅な短縮が期待できる。

(4) Lexical Analyzer

ＪＰＡＣによるコンパイラのLexical Analyzerは，Syllable Readルーチンを下請けとしている。文番号，Key語等の処理もLexical Analyzerで行なう為構造が複雑になり，作成コンパイラの処理速度に少なからず影響を与えていると思われる。

(5) ＬＳＤ言語

セマンティックス・ルーチンをＬＳＤ言語で記述する過程で気づいた点を以下に示す。

① テーブルの最初及び最後に登録された構造体へのポインターを参照したいことが度々生じた。現仕様でには，それらを直接に参照できない。

② テーブルサーチの際，ビット型の項目に対してSEARCHステートメントが有効に使えない場合がある。

③ ＰＯＰステートメント（スタッカーのポップアップを行なう）のポップアップ数は現仕様では定数のみしか書けない。しかし，現実には変数としたい場合が度々ある。

2.5.2 作成コストの比較

評価の第1点は，コンパイラ作成時の労力と計算機使用時間両方の節減である。たとえ労力の節減が達成されたとしても計算機使用時間が極端に増大したのでは，コンパイラ作成時の総コスト（人件費＋計算機使用費）が増大して上記の目的が達成されたとはいえない。以下に示す表によって，ＪＰＡＣが労力はもとより計算機使用時間も節減できることがわかるであろう。

（表2－3）は，3種の方法（アセンブラ，ＰＬ／Ｉ，ＪＰＡＣ）による，コンパイラ作成時に要した労力および計算時間等の比較表である。

表2－3　　比　較　表

比較項目＼使用言語	アセンブラ	PL／I	JPAC
労　　力 （単位　：時間）	447	350	179
計算機時間 （CPU） （単位：HH：MM'SS）	1：47' 30.	7：17 '30.	1：21 '14.
RUN数 （単　位：回）	365	385	199
使用カード枚数 （単　位：枚）	5054	4529	1564

(注)　労力はプログラム設計書（フローチャート設計も含む），コーディング，デバッグに要した時間の総和である。計算機時間及びRUN数は，デバッグ完了（指定したプログラムが全て正常コンパイルできるまで）までに要したCPU時間及び計算機使用回数である。カード枚数は，各手法によって作成されたコンパイラの構成カード枚数でありステップ数とは異なる。

コンパイラ作成労力の比率は，

アセンブラ：PL／I：JPAC＝5：4：2

である。よって作成労力に関し満足できる結果と言ってよいであろう。又，

計算機時間に関して

$$\frac{\text{JPAC}}{\text{アセンブラ}} \times 100 = 75\%$$

カード枚数に関して

$$\frac{\text{JPAC}}{\text{アセンブラ}} \times 100 = 30\%$$

である。それ故，評価の第1点に関し他の方法と比べて格段に優れていると言ってよいであろう。

表で見る限り，PL／Iによるコンパイラの作成は決して有利ではないと

思われるが，かかる結果の原因としては次のような事が考えられる。

① ＰＬ／Ｉコンパイルに時間がかかる。

② ＦＡＣＯＭ２３０－６０のＰＬ／Ｉに％ＩＮＣＬＵＤＥの機能がない為，共通テーブルに関する宣言を全てのルーチンで行なわねばならなかった。

③ ＰＬ／Ｉによるコンパイラ作成者は，言語には精通しているが，他の方法（ＪＰＡＣ，アセンブラ）による作成者に比べてコンパイラ作成の経験がやや乏しかった。

2.6 作成コンパイラの性能評価

前節で述べたように，作成時の総コスト節減という目的は達成できたと言える。しかし，ＪＰＡＣによって作成されたコンパイラが他の手法（ＰＬ／Ｉ，アセンブラ）によって作成されたものに比べて性能が著しく劣るならば，ＪＰＡＣは実用になるとは言いきれないであろう。それ故，３手法（ＪＰＡＣ，ＰＬ／Ｉ，アセンブラ）によって作成されたコンパイラの性能を，コンパイル速度・コンパイラサイズという面で比較検討を行なった。

2.6.1 コンパイル速度の比較

テスト用ＦＯＲＴＲＡＮプログラムとしては次の４種のものを用意した。

① ＳＴＯＰ文とＥＮＤ行のみで構成されるもの。

② 主として算術代入文からなるもの。

③ 主として入出力文からなるもの。

④ 一般的プログラム。

①はＦＯＲＴＲＡＮプログラムを構成する最小単位であり，そのコンパイルに要する時間は各コンパイラの最小コンパイル時間と考えられる。

②及び③はＦＯＲＴＲＡＮステートメントの中で，手書きのコンパイラとの差が大きく出ると思われたステートメントに関するものである。

④はコンパイル時間の平均的相異を調べる為のものである。

（表２－４）は上記４種のＦＯＲＴＲＡＮプログラムをコンパイルした結果を表にしたものである。

表によると，ＪＰＡＣで作成したコンパイラはアセンブラ記述のものに比べて，最大２３倍のコンパイル時間を要し，構文解析のみでも最大7.6倍の時間がかかる。

一般的プログラムに関しては，12.7～16.5倍であるが，コンパイルに必要な最小時間（１の時間）を差し引いたものでは17.7～19.3倍となる。前者はソース・ステートメント数か少ない場合，後者は多い場合の平均コンパイル時間の差と考えられる。

表２－４　コンパイル時間の比較（ＣＰＵ時間）

プログラム番号	アセンブラ（単位：ms）	ＰＬ／Ｉ（単位：ms）	ＪＰＡＣ（単位：ms）	ＪＰＡＣ／アセンブラ	プログラム概要	
					ステートメント数	ステートメント種類
1	588	656	1017	1.7	2	最小のプログラムSTOPとENDのみ
2	17354	37206	315741	18.2	1003	簡単な算術代入文
3	7318	19974	168332（55289）	23.0（7.6）	103	単純変数の加算からなる算術代入文
4	22995	46488	361201（138337）	15.7（6.1）	104	配列要素を含んだ算術代入文
5	18873	43652	359843（105628）	19.0（5.6）	103	関数を含んだ算術代入文
6	22157	38964	434220	19.6	1004	簡単な出力文
7	8931	16720	191162	21.4	104	ＤＯ型並びを含んだ入力文
8	3789	17610	62716	16.5	118	一般的なプログラム
9	1872	5107	23751	12.7	45	一般的なプログラム
10	2207	6338	31163	14.1	67	一般的なプログラム
11	2453	9088	32828	13.4	95	一般的なプログラム

備考　①　測定は全てソース・リストを出力せずに行なった。

②　表中（　）で囲んであるのは構文解析のみの結果である。

③　なおこの時間測定に用いたテスト・プログラムは最後に付録として掲載する。

以下にＪＰＡＣによるコンパイラの速度が遅い原因について検討した結果を示す。

① 構文解析において，nonterminal記号の置き換えのみを行なう生成規則(例えば＜Ａ＞::＝＜Ｂ＞；)の適用はＣＰＵ時間を消費するだけである。(表２－１)の構文に従ってＩ＝Ｊという代入文の構文解析を行なった場合，＜ＡＳＳＩＧＮＭＥＮＴ＞となるまでに(図２－３)で示す過程をたどる。

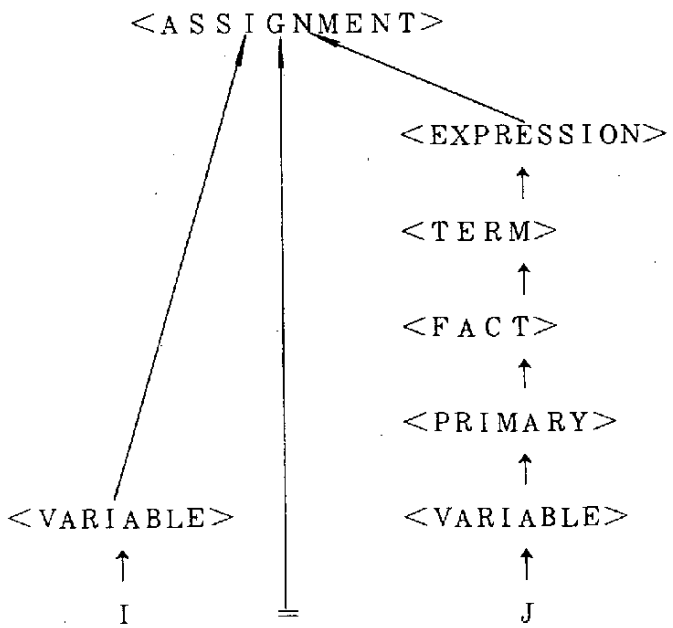


図２－３　Ｉ＝Ｊに対する構文Tree

図からも判るように＜ＡＳＳＩＮＧＮＭＥＮＴ＞となるまでに７個の生成規則が適用されるが，そのうちセマンティックスルーチンに制御が渡されるのは，

Ｉ → ＜ＶＡＲＩＡＢＬＥ＞

Ｊ → ＜ＶＡＲＩＡＢＬＥ＞

＜ＶＡＲＩＡＢＬＥ＞＝＜ＥＸＰＲＥＳＳＩＯＮ＞→＜ＡＳＳＩＧＮＭＥＮＴ＞

の３個だけで，残り４個はnonterminal記号の置き換えのみを行なっている。

以上述べたようなことは手書きコンパイラでは起こらない。

② ＪＰＡＣで作成されたコンパイラは，構文解析部分とセマンティックス処理

部分が完全に分離している為，手書きのものに比べて実行時にルーチン間制御の受け渡しが多いと考えられる。ルーチン間で制御が渡される場合，各ルーチン本来の目的とする処理以外に，レジスター退避・復帰等の作業を行なわねばならない。

ゆえに，手書きコンパイラに比べてルーチン間で制御が移動する回数が多い分だけ実行時の処理が遅くなると言える。

③ ＪＰＡＣで作成されたコンパイラのセマンティックス処理時間（コンパイル時間から構文解析の時間を除いたもの）が多いということは，ＬＳＤ言語で記述したセマンティックス・ルーチンの実行速度が遅いということになる。そこでセマンティックス・ルーチンのオブジェクト等を詳細に検討した結果次のようなことが判った。

セマンティックス・ルーチンのオブジェクトは，文字処理に関するものがほとんどであり，処理時間の大多数を文字処理用ランタイムルーチンが使っている。

よって処理速度改善の為に，ＬＳＤ言語のオブジェクト及びランチタイムルーチンのうち文字処理関係を改善する必要がある。

④ 遷移決定（arrow選択）の際，記号の先読み（look-ahead）あるいはＣＧ＿ＳＴＡＣＫの中味参照（look-back）を行なわなければならないことがある。

そのとき現在のＪＰＡＣの作り出す遷移テーブルでは遷移決定に多くの時間を必要とするので，遷移決定が容易にできるよう遷移テーブルの構成を変更する必要がある。

⑤ ＪＰＡＣで試作したコンパイラでは，Lexical Analyzer は Syllable Read ルーチンを下請けルーチンとして用いている。

その為，専用のLexical Analyzer を作った場合に比べて実行速度が遅くなっている。

上記の原因のうち，①はＪＰＡＣシステムに手を加えることによって改善することは不可能であり，ＪＰＡＣを使ってコンパイラを作成するユーザが効率のよい構文を定義する以外に方法はない。②～④はＪＰＡＣシステムを改善する余地

があることを示す。⑤は試作コンパイラのみに関係する特殊事情である。

2.6.2　コンパイラのコアサイズ比較

（表２－５）はアセンブラ，ＪＰＡＣ，ＰＬ／Ｉによって作成されたコンパイラのコアサイズ比較表であり，（図２－４）は，それを棒グラフに書き改めたものである。

表によると作成コンパイラのサイズ比は

アセンブラ：ＪＰＡＣ：ＰＬ／Ｉ＝１：1.6：2.7

であり，ＪＰＡＣで作成されたコンパイラはサイズの面で，ＰＬ／Ｉにより優れ，アセンブラより劣っている。但し，前にも述べたことだがＰＬ／Ｉによるコンパイラの作成者はコンパイラ作成の経験に乏しいため，作成コンパイラに多少の冗長性があると思われるので，ＰＬ／Ｉによるコンパイラのサイズは割り引いて考える必要がある。

表２－５　コアサイズ比較表（単位：WORD）

分類項 ＼ 手法	アセンブラ	ＪＰＡＣ	ＰＬ／Ｉ
ランタイム・ルーチン	0	1657	10038
ユーティリティ	6840	6840	6840
テーブル	6060	9989	8120
処理ルーチン	6076	12330	26708
合　計	18976	30816	51706

（注）① ユーティリティはＦＴＥＲＭ，ＭＴＷＲＩＴＥ，Ｆ・ＳＹＬ及びＦ・ＥＲＲＯＲのサイズを合計したものである。

② テーブルはＰＬ／Ｉ，アセンブラで作成したコンパイラではname-table，constant-table，do-stackを示す。ＪＰＡＣで作成したものは，さらに遷移テーブル等構文解析用領域が加算されている。

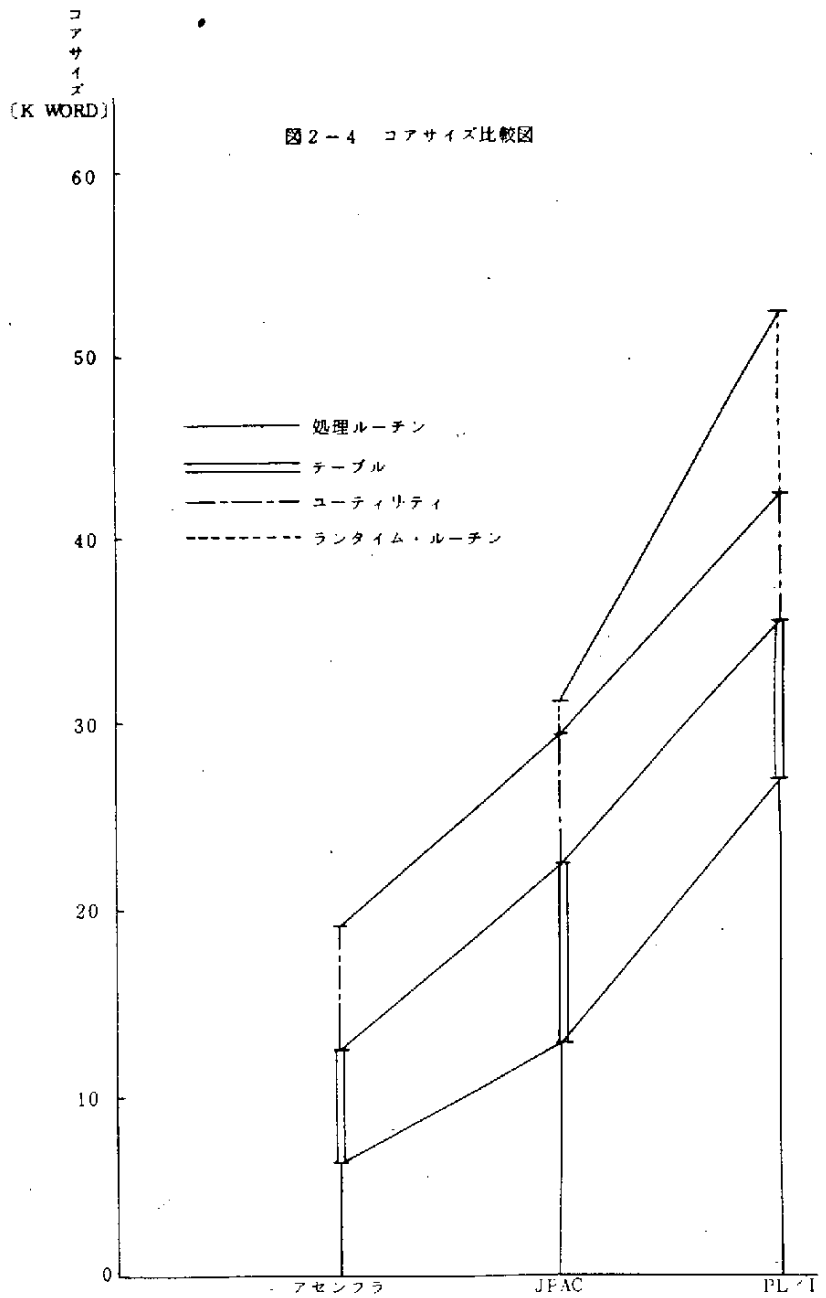


図２－４　　コアサイズ比較図

ＪＰＡＣで作成されたコンパイラはアセンブラ記述のものと比べてコアサイズが大きくなっているが，その理由としては次の２点が考えられる。

① 構文解析を自動的に行なう為に必要である遷移テーブル，コントローラ等のコアサイズが加算される。

② ＬＳＤコンパイラによって作り出されたオブジェクト（ＪＰＡＣのセマンティックス・ルーチンはＬＳＤ言語で記述される）がアセンブラ記述のものより効率が劣る。

よって以下のコアサイズの検討は上記２点を中心として行なった。

(1) 構文解析用領域

ＪＰＡＣで作成されたコンパイラにおける，遷移テーブル等の構文解析用テーブル及び処理ルーチンのコアサイズを（表２－６）に示す。

表２－６　構文解析用領域サイズ
（構文解析用テーブル，処理ルーチンサイズ）

	テーブル 処理ルーチン	サイズ（単位：WORD）	備　　考
テーブル	CG_SENI	1274	遷移テーブルである。構文の書き方により増減する。
	CG_STACK	204	
	CG_NONT	312	nontorminal 記号情報セット用領域として定義した，大きさ１００
	その他	72	CG_STR及びCG_LEXとの情報受け渡し用領域
処理ルーチン	CG_MAIN	264	コントローラ
	CG_SEM	216	セマンティックス・ルーチン数によって増減する。
	トータル・サイズ	2342	

（表２－５）と（表２－６）によると

$$\frac{\text{遷移テーブルのサイズ}}{\text{コンパイラのサイズ}} \times 100 = \frac{1274}{30816} \times 100 = 4.1\%$$

$$\frac{\text{構文解析用領域サイズ}}{\text{コンパイラのサイズ}} \times 100 = \frac{2342}{30816} \times 100 = 7.6\%$$

であり，構文解析用領域がコンパイラサイズに占める割合いは，さほど大きくない。

一方，手書きのコンパイラでは構文解析用領域は必要としないかわりに，メイン・ルーチン，文識別ルーチンが必要であり，さらに各文処理ルーチンにおいて構文解析用領域の機能に代る処理*を行なわねばならず，その為に要するステップ数，作業領域はかなりの量に及ぶと考えられる。

以上のことから構文解析を自動化することによって生じたコンパイラサイズの実増加は極めて少ないと考えられる。

(2) ＬＳＤコンパイラのオブジェクト

ＪＰＡＣによって作成されたコンパイラのサイズの内，構文解析用領域（全体の7.6％）を除いた部分はＬＳＤ言語で記述されたセマンティックス・ルーチンのオブジェクトよりなっている。

よってＬＳＤコンパイラのオブジェクトはＰＬ／Ｉコンパイラのそれよりも優れていると言えるが，アセンブラ記述のものよりは効率が劣る。

以下に作成コンパイラの①テーブルサイズ，②処理ルーチンサイズの吟味を通じてＬＳＤコンパイラのオブジェクト効率を検討した結果を述べる。

① テーブルサイズ

ＪＰＡＣで作成されたコンパイラのテーブルサイズ（９９８９WORD）から構文解析用領域のテーブル部分のサイズ（１８６２WORD)を差し引くと８１２７WORDとなり，ＰＬ／Ｉによるもののテーブルサイズとほとんど変らないが，アセンブラ記述のものに比べると３５％増である。よってＬＳＤコンパイラのオブジェクトは領域確保に関しては，アセンブラ記述ほど効率の良いオブジェクトは望めず，ＰＬ／Ｉとほとんど変らないか若干劣る。

領域確保の機能に関して，ＬＳＤコンパイラのオブジェクトがアセンブラ記述に対して劣る理由は"ＬＳＤコンパイラは１変数に対して最低1WORD

＊主な処理は構文解析であるが"式"を処理する場合に"逆ポーランド記法"を用いるならば，作業領域としてオペレータ・スタック，オペランド・スタックが必要であり，処理としてオペレータの順位比較及び前記のスタックの操作を必要とする。ＪＰＡＣでコンパイラを作成する場合には上記の領域及び処理は必要としない。

を割り付ける為，領域確保に無駄が出る"ことである。

なおＰＬ／Ｉではストリングデータに関して領域をパックして使用することが可能であるが，今回作成したコンパイラではメリットを生かせなかった。但し，領域をパックして使用した場合，実行速度は大幅に遅くなると思われる。

② 処理ルーチンサイズ

（表２－５）によると処理ルーチンのサイズ比は，

アセンブラ：ＪＰＡＣ：ＰＬ／Ｉ＝１：２：４

となり，ＬＳＤコンパイラの実行文オブジェクトはＰＬ／Ｉのそれよりも優れているが，アセンブラ記述のものに比べて効率がよくない。*

ＬＳＤコンパイラの作り出す実行文オブジェクトがアセンブラ記述のものに劣る理由を検討したところ，主な原因は"文字処理に関するオブジェクトが極端に悪い"ということであった。試みに文字処理を主とするテストプログラムを同一フローチャートに従ってアセンブラ，ＬＳＤ言語，ＰＬ／Ｉで記述してみると（表２－７）の結果となった。

表２－７　文字処理テストプログラムのサイズ

言　　語	アセンブラ	ＬＳＤ	ＰＬ／Ｉ
サイズ(WORD)	３４６	１１７２	１３３２

表によるとＰＬ／Ｉのオブジェクトもアセンブラ記述に対して極端に劣っていることがわかる。この原因はＬＳＤコンパイラを作成した経験から述べると"ＦＡＣＯＭ２３０－６０はＷＯＲＤマシンであるため，コンパイラ設計上，文字処理に関する最適なオブジェクトを設計するのが困難である"ということである。しかしながら，コンパイラ作成を目的とするならば文字処理機能は重要であり，よってＬＳＤコンパイラの文字処理関係のオブジェクトの最

＊ＰＬ／Ｉ及びアセンブラで記述した処理ルーチンとＬＳＤ言語で記述したものでは機能及び構成が若干異なる為，厳密な意味では比較できないが一応の目安にはなる。

適化を計る必要がある。

(3) ま と め

コアサイズの検討結果をまとめてみると次のようになる。

① 遷移テーブル等構文解析用領域のコンパイラサイズへの影響は極めて小である。

② ＬＳＤコンパイラの文字処理関係のオブジェクト効率が極めて悪く改善を要する。

①に関しては初めの予想（構文解析用領域はコンパイラサイズに大きな割合を占める）とは異なったが，遷移テーブルは構文の書き方１つで大幅に変動するので注意が必要である。

②に関しては作成コンパイラのコンパイル速度にも大きく影響しているので，早急に改善したいと思うが，ワード・マシンのため，抜本的な改善は望めない。

今回の評価ではSyllable Readルーチン，ＦＡＳＰコード・ジェネレートルーチン等はユーティリティとして，同一のものを各作成コンパイラに組み込んだ為，評価の対象から除いた。しかし（表２－５）で示されるとおり，コンパイラのサイズのかなり大きな部分を占めているので，本来ならば各手法で別々に作って比較検討すべきであった。

2.7 総　　評

前節までで述べられた評価結果をまとめ，ＪＰＡＣの不備な点に関して改善方法を述べる。さらにＪＰＡＣのより一層の充実をはかる為，残された課題について述べる。

又，コンパイラ及びコンパイラ・ジェネレータに関しての専門家の意見を本節の終りに掲載する。

A・評価結果のまとめ

コンパイラ作成にＪＰＡＣを用いた場合の利点及び欠点をまとめると次のよう

になる。

(1) 利　　点

① コンパイラの生産コストは手書きに比べて大幅に削減できる。

② 構文の変更はシンタックス記述部を変えるだけでよく，言語仕様の変更に対する柔軟性が高い。

③ コンパイラ作成者が"逆ポーランド記法"等の技法を知らなくともコンパイラを作成することが可能である。さらにＬＳＤ言語にはテーブル操作・スタック操作が容易にできるステートメントが用意されている。よって作成コンパイラの品質は作成者のレベルにそれほど影響されない。

④ 作成コンパイラの言語仕様書を作る際，シンタックス記述部は言語の構文を表わす資料として使える。

(2) 欠　　点

① 作成コンパイラのコアサイズが手書き（アセンブラ記述）に比べて大きくなる。

② 作成コンパイラの実行速度（コンパイル速度）が手書きに比べて著しく遅い。

③ ＬＳＤ言語の機能に幾つかの不備な点がある。

指摘された欠点のうち，①②の主原因はＬＳＤコンパイラの文字処理関係のオブジェクト効率が非常に悪いことである。

Ｂ．　ＪＰＡＣシステムの改善

前記の欠点を改善するために下記の方策を講じる予定である。

(1) ＬＳＤ言語

下記の機能を追加する。

① テーブルの最初及び最後に登録された構造体へのポインターを直接に参照できるようにする。

② ＳＥＡＲＣＨステートメントの仕様をビット型の項目に対しても有効に使えるように変更する。

③　ＰＯＰステートメントの仕様を，ポップアップ数として定数以外に変数も書けるように変更する。

(2)　ＬＳＤコンパイラ

下記の改善を行なう。

①　文字処理関係のオブジェクトの効率化を計る。

②　ランタイム・ルーチンの高速化を計る。

③　Procedure 間で制御が移動する場合，レジスター退避は index 7 だけとする。

(3)　遷移テーブル

look-back , look-ahead が迅速にできるよう，遷移テーブルの構成を変更する。

Ｃ．今後の課題

ＪＰＡＣの今後に残された課題について述べる。

(1)　Lexical Analyzer 専用言語の開発

現仕様のＪＰＡＣでは own-coding の Lexical Analyzer を作成する場合，

①　標準ライブラリを利用してＬＳＤ言語で記述する。

②　アセンブラで記述して実行速度の高速化，サイズの小型化を計る。

が考えられる。

①の方法では，作成は容易であるが，ＬＳＤコンパイラの作り出す文字処理関係のオブジェクト効率が悪いという事実は，Lexical Analyzer の機能を考えると致命的である。

②の方法は①に比べると作成された Lexical Analyzer の品質に関し，より優れたものを期待できる。しかしコンパイラ自動作成を目的とする立場からすると，いかにも能が無い。

目的を Lexical Analyzer 作成だけに限定すると，必要な機能はかなり限られる。よって記述が容易で，効率がアセンブラ記述に劣らない Lexical Analyzer 専用言語を開発することは可能だと考えられる。又，かかる言語を用意

しておくことはＪＰＡＣ作成の目的と一致する。

(2) エラー処理

ＪＰＡＣで作成されたコンパイラでは，構文解析中にエラーを見つけたとき，Basic Statement の指定があれば，

step1　エラー記号の存在場所をパラメータとしてエラー処理ルーチンへ制御を渡す。

step2　エラー処理ルーチンではエラー・メッセージの出力及び必要ならばソース・プログラムの読み飛ばしを行ないリターンする。

step3　コントローラはBasic Statement と指定されたnonterminal記号の直後の記号までＣＧ＿ＳＴＡＣＫをpop upして，構文解析を続行する。

の手続きを行なう。かかる方式の弊害としては，

① 読み飛ばされたソース・プログラムの構文チェックは行なわれない。

② エラー・メッセージ出力に際して，エラー記号だけから適切なエラー・メッセージを出力することは難しい。

③ エラー処理ルーチンにおいて，ソース・プログラムの読み飛ばしを行なう際，エラー記号だけから読み飛ばし範囲を決定するのが困難な場合もある。

④ 構文上のエラーに対しては一切リカヴァリーできない。

等があげられる。

今回試作した言語クラスのコンパイラーでは，上記の弊害は余り目立たないが，作成言語が複雑化すればする程，影響は大きくなってくる。かかる弊害に対する対応策として＂エラー処理の自動化推進＂が考えられ，機能としては，

① リカヴァリー可能なものの自動修正。

② リカヴァリー不能なものに関しては，エラー状況をできるだけ詳しく解析し，エラー処理ルーチンに通知してやる。

等があげられる。しかし，かかる機能をＪＰＡＣに持たせる為には以下の問題点を解決しなければならない。

① エラー処理の形式化及びＪＰＡＣへの入力方法。

② エラー処理が加わった為に起こると考えられる，作成コンパイラのサイズ増加及び実行速度の低下をできるだけ押さえる方法。

よってＪＰＡＣにエラー処理の自動化機能を付け加えるまでには，まだ時間を有するであろう。

(3) セマンティックス記述

ＪＰＡＣのセマンティックス・ルーチン（ＬＳＤ言語で記述）はセマンティックスそのものを記述したものではない。コンパイラの構成をシンタックス処理部とセマンティックス処理部に分けた場合，作成労力は後者の方がより多く要し，従ってコンパイラ自動化の価値は後者の自動化の方が大きい。後者の自動化を達成するには，シンタックス記述に対するＢＮＦのように，セマンティックスの形式的記述方法の確立が必要である。しかしながら現在までのところ，それの実用的表現方法は確立されておらず，当分見込みもない。よってJPACのような実用性を目的としたコンパイラ・ジェネレータのセマンティックス処理部は，現時点ではシステム記述言語を用いて書くのが最良であると考えられる。しかし，コンパイラ・ジェネレータの普及の為には，セマンティックスそのものの形式的記述方法の確立が是非とも必要である。

D・第三者の意見

言語及びコンパイラに関する専門家あるいは学識経験者の意見を

① コンパイラ言語に関して

② コンパイラの作成に関して

③ コンパイラ・ジェネレータに関して

の3点についてまとめたものを以下に示す。

(1) コンパイラ言語に関して

現在のコンパイラ言語は種々の矛盾を内包している。それは人間側の自由度に乏しい（使いにくい）こと，あるいは現在のソフトウェア言語体系はハードウェア体系とかけ離れている為その変換操作（コンパイル）が難しいとともに

ハードウェアの機能を十分に生かしきれない等である。このような矛盾はソフトウェアの巨大化に伴ないますます増大しつつある。

過去の歴史をふり返ってみると言語体系はマシン語からアセンブラ，アセンブラからコンパイラと変遷してきた。それは個々の言語体系の持つ矛盾が増大し繕うことができなくなり，その解決法として新しい言語体系が出現してきたといえる。よって現在のコンパイラの姿と異なった思想の言語体系がおそかれはやかれ出現すると考えられる。

次世代の言語体系は人間工学的要素を取り入れるとともにソフトとハードの体系を区別せず一体化したものであると思われる。それ故，言語の持つべき基本的要素を抽出・形式化し，それを具現化する方向でハードウェアを改変することが必要であろう。その意味でファームウェアは有力なアプローチ方法の一つである。

(2) コンパイラの作成に関して

コンパイラの作成を①アセンブラによる，②システム記述言語による，③コンパイラ・ジェネレータによるものと分けると当面その主流は②になると考えられる。理由としては下記のことがあげられる。

①は製品の品質（コアサイズ，実行速度）は優れたものを作ることは可能だが，作成上の労力は他の手法に比べて多大であり，現在のようなソフトウェア大量生産時代には不向きである。

②は①に比べて作成労力はかなり軽減でき，製品に関してはさほど劣らないものを作ることも可能である。

③は②に比べて作成労力は軽減できるが，作成の際に最も労力の要するセマンティックス記述の形式化が未解決である為，さほどメリットはない。又，作成される製品の品質は極めて劣りこの欠点は作成時のメリットでカヴァーしきれない。

(3) コンパイラ・ジェネレータに関して

セマンティックス記述の形式化に関して過去数年来目立った進歩は見られず，

今後も急速な進歩は望めない。よってＪＰＡＣのようなシステムの活用法としては次のようなものが考えられる。

① 製品の実行速度，サイズ等があまり問題とならないシステム。

② シンタックスに比べてセマンティックスが簡単なシステム。

①に対応するものとしてはシステム記述言語のコンパイラ，実験コンパイラが考えられる。②に対応するものとしてシンタックス・チェッカー等が考えられるが，コンパイラ作成という範囲だけに限定しなければ活用範囲はかなりあるはずである。

上記の活用方法を考慮してみるとＪＰＡＣのオブジェクトはアセンブラよりも汎用言語の方が良かったのではないかと考えられる。

参 考 文 献

1. F・L・Deremer
   Practical Translators for LR(K) Languages
   MAC－TR－65
2. F・L・Deremer
   Simple LR(K) Grammars
   C・ACM Vol.14 №7 1971 PP.453～458
3. D・E・Knuth
   On the translations of languages from left to right
   Inf.Contr.Vol.8 Oct.1965 PP.607～639
4. W・M・Mckeeman, et.al.
   A Compiler Generator
   Prentice-Hall Inc.
5. （財）日本情報処理開発センター
   コンパイラ・ジェネレータの開発（47－S002）
   昭和48年3月

付　　録

付 録　FORTRANプログラム例

コンパイル時間の比較に用いたテスト・プログラム

プログラム1

```
1        STOP
2        END
```

プログラム2

1000回繰り返し→

```
1        COMMON Y,Z
2        X=Y+Z
3        STOP
4        END
```

プログラム3

100回繰り返し→

```
1        COMMON A,B,C,D,E,F,G,H,O,P
2        X=(A+(B+(C+(D+(E+(F+(G+(H+(O+P))))))))
3        STOP
4        END
```

プログラム4

100回繰り返し→

```
1        DIMENSION A(10),I(10)
2        COMMON A,I
3        COMMON J1,J2,J3,J4,J5,J6,J7,J8,J9,J10
4          X=(((((((((A(I(J1))*A(I(J2)))*A(I(J3)))*A(I(J4)))*A(I(J5)))*
        *    A(I(J6)))*A(I(J7)))*A(I(J8)))*A(I(J9)))*A(I(J10)))
5        STOP
6        END
```

プログラム5

100 回繰り返し→

```
      COMMON A,I,C,D,E,F,G,H,O,P,Q
      X=(ABS(A)*(FLOAT(I)*(SIGN(C,D)*(EXP(E)*(ALOG(F)*(SIN(G)*
     *  (COS(H)*(TANH(O)*(SQRT(P)*ATAN(Q))))))))))
      STOP
      END
```

プログラム6

1,000 回繰り返し→

```
      COMMON I
      WRITE(6,10) I
   10 FORMAT(1H0,I5)
      STOP
      END
```

プログラム7

100 回繰り返し→

```
      DIMENSION A(5,3),C(3)
      READ(5,100) A,((A(I,J),I=1,5),C(J),J=1,3)
  100 FORMAT(3F12.0)
      STOP
      END
```

プログラム8

```
      SUBROUTINE WIND
      DIMENSION  X(21),   Z(22),   DX(21),  DZ(21),  DDZ(21),
     1           US(21),  V(21),   T(21),   DU(21),  DV(21),
     2           H(21),   C(21),   DC(21),  CK2(21), CK3(21),
     3           UX(21,15),ZZ(10),  S(100,100),ABC(15), ZU(10),
     4           ZV(10),  ZT(10),  ZA(20),
     5           P(21),
     6           U(21,21), W(21,21), IDTBL(21,21),
     7           IK(2,21), DP(21,21)
      DIMENSION  JZ(10),Q(21)
      COMMON     X,Z,DX,DZ,DDZ,US,V,T,DU,DV,H,C,DC,CK2,CK3,UX,ZZ,S,
     1           ABC,ZU,ZV,ZT,ZA,JC,MC,KC,F,
     2           P,
     3           ELAMDA,KI,G,
     4           IDEGCD,U,W,IDTBL,IKNUM,IK,DP
      COMMON     XX,YY,TH,TT,ZZZ,CC
      COMMON     I,K
      COMMON     UT,WT,DPT
      COMMON     UTSHR,WTSHR
          DO 302 IY=1,10
  302 JZ(IY)=0
          LQ=6
            JJ=0
          JZMAX=0
          DO 30 J = 1,6
          IF (ZZ(J)-Z(MC)) 29,29,30
   29 JJ=JJ+1
          DO 31 I = 1,MC
            JZ(J)=I
          IF(ZZ(J)-Z(1)) 39,39,28
   28 IF(Z(I)-ZZ(J)) 27,27,31
   27 IF(Z(I+1)-ZZ(J)) 31,31,39
   31 CONTINUE
   39     N=JZ(J)
          IF(JZ(J)-JZMAX) 24,30,26
   26 IF(JZ(J)-JZMAX) 23,23,24
   24 JZMAX=JZ(J)
   23 US(N)=ZU(J)
          V(N) = ZV(J)
          T(N) = ZT(J)+273.0
          P(N) = V(N)*F
          Q(N)=-US(N)*F
   30 CONTINUE
            MC=JZMAX
            WRITE(6,555) MC
  555 FORMAT(1H1,3HMC=,I5)
   32 NI = 2
          WRITE(6,63) JZ,JJ
          ELAMDA=0.00027*SQRT(US(MC)**2+V(MC)**2)/F
          DO 25 M = 1,MC
          ZMEA = 0.4*0.5*(Z(M)+Z(M+1))*100.0
          ZA(M+1) = (ELAMDA*ZMEA/(ZMEA+ELAMDA))**2
   25 CONTINUE
   63 FORMAT(1H ,10X,10HAAZZAAZZAA,I8)
          DO 33 N = 1,JJ
          NF = JZ(N)-1
          D = Z(NF+1)-Z(NI-1)
           LI=NI-1
           LF=NF+1
          IF (NF-NI) 33,22,22
   22 DO 34 J=NI,NF
```

```
      DA = (Z(J)-Z(NI-1))/D
      DB = (Z(NF+1)-Z(J))/D
      US(J)=DA*US(LF)+DB*US(LI)
      V(J) = DA*V(LF)+DB*V(LI)
      P(J) = DA*P(LF)+DB*P(LI)
      T(J) = DA*T(LF)+DB*T(LI)
      @(J) = DA*@(LF)+DB*@(LI)
   34 CONTINUE
   33 NI = NF+2
      MB = MC-1
      DO 35 M = 2,MC
         WRITE(6,357)DZ(M)
  857 FORMAT(1H ,5X,4HTTST,E18.5)
      DU(M)=(US(M)-US(M-1))/DZ(M)
      DV(M) = (V(M)-V(M-1))/DZ(M)
      H(M) = (T(M)-T(M-1))/DZ(M)+0.0001
      XX=DU(M)
      YY=DV(M)
      TH=H(M)
      TT=T(M)
      ZZZ=ZA(M)
      CALL VISC
      C(M)=CC
      DC(M) = C(M)/DZ(M)
C
   35 CONTINUE
      DO 36 I = 1,JC
      ERROR = 0.0
      DO 37 M = 2,MB
         DO 145 ITT=1,5
      DCC = DC(M+1)+DC(M)
      UU=(DC(M+1)*US(M+1)+DC(M)*US(M-1)+(F *V(M)-P(M))*DDZ(M))/DCC
      T(M) = (DC(M+1)*T(M+1)+DC(M)*T(M-1))/DCC
      V(M)=(DC(M+1)*V(M+1)+DC(M)*V(M-1)-(F*US(M)+@(M))*DDZ(M))/DCC
      ERROR = ERROR+ABS(UU-US(M))
      US(M)=UU
      DU(M)=(US(M)-US(M-1))/DZ(M)
      DV(M) = (V(M)-V(M-1))/DZ(M)
      H(M) = (T(M)-T(M-1))/DZ(M)+0.0001
      XX=DU(M)
      YY=DV(M)
      TH=H(M)
      TT=T(M)
      ZZZ=ZA(M)
      CALL VISC
      E=CC
      C(M) = SQRT(E*C(M))
      DC(M) = C(M)/DZ(M)
      DU(M+1) = (US(M+1)-US(M))/DZ(M+1)
      DV(M+1) = (V(M+1)-V(M))/DZ(M+1)
      H(M+1) = (T(M+1)-T(M))/DZ(M+1)+0.0001
      XX=DU(M+1)
      YY=DV(M+1)
      TH=H(N+1)
      TT=T(M+1)
      ZZZ=ZA(M+1)
      CALL VISC
      E=CC
      C(M+1) = SQRT(E*C(M+1))
      DC(M+1) = C(M+1)/DZ(M+1)
  145 CONTINUE
   37 CONTINUE
      IF(ERROR-0.001) 36,38,21
   21 WRITE(10,987) I,(US(III),V(III),T(III),C(III),III=1,MC)
  987 FORMAT(1H ,2HI=,I5///,(1H ,4(E15.7,5X)))
   36 CONTINUE
   38 CONTINUE
      RETURN
      END
```

プログラム 9

```
      C     EX3-2
      C     TRAVERSE COKURYO
      1     DIMENSION AL(5),I1(5),I2(5),I3(5),I4(5),J1(5),J2(5),J3(5),J4(5),
     1           BL(5),CL(5),EL(5),ED(5),BBL(5),CCL(5),AJ(5)
      2     READ(5,100)(AL(I),I=1,5),(J1(I),J2(I),I3(I),I=1,5),
     2           J1(1),J2(1),J3(1)
    100 FORMAT(5F6.2/(3I3))
      4     DO 10 I=1,5
      5   10 I4(I)=(I1(I)*60+I2(I))*60+I3(I)
      6     J4(1)=(J1(1)*60+J2(1))*60+J3(1)
      7     DO 11 I=2,5
      8   11 J4(I)=J4(I-1)+180*60*60-I4(I)
      9     K=360*60*60
     10     DO 12 I=1,5
     11     IF(J4(I))13,14,14
     12   14 IF(J4(I)-K)12,12,15
     13   13 J4(I)=J4(I)+K
     14     GO TO 12
     15   15 J4(I)=J4(I)-K
     16   12 CONTINUE
     17     DO 20 I=1,5
     18     AJ(I)=J4(I)
     19     AJ(I)=AJ(I)/3600.0
     20     ARAD=AL(I)*3.14159/180.0
     21     BL(I)=AL(I)*COS(ARAD)
     22     CL(I)=AL(I)*SIN(ARAD)
     23   20 CONTINUE
     24     E1=0
     25     E2=0
     26     AA=0
     27     DO 30 I=1,5
     28     E1=E1+BL(I)
     29     E2=E2+CL(I)
     30     AA=AA+AL(I)
     31   30 CONTINUE
     32     EL1=-E1/AA
     33     ED1=-E2/AA
     34     DO 40 I=1,5
     35     EL(I)=AL(I)*EL1
     36     ED(I)=AL(I)*ED1
     37     BBL(I)=BL(I)+EL(I)
     38     CCL(I)=CL(I)+ED(I)
     39   40 CONTINUE
     40     WRITE(6,200)
     41  200 FORMAT(1H1,36X,9HKOUSEIRYO,3X,10HCHYOSEICHI/
     3 1H0,3X,7HSOKUSEN,6X,4HIKYO,4X,6HKEIKYO,6X,4HIKYO,
     4 4X,6HKEIKYO,6X,4HIKYO,4X,6HKEIKYO)
     42     WRITE(6,201)(AL(I),BL(I),CL(I),EL(I),BBL(I),
     5 CCL(I),I=1,5)
     43  201 FORMAT(1H0,7F10.3)
     44     STOP
     45     END
```

プログラム10

```
      C     EX1-2-C
      C     BAIRSTOW METHOD
    1       DIMENSION A(20),XR(20),XI(20),B(20),C(20)
    2       READ(5,100) N,EPS,P,Q
    3   100 FORMAT(I2,F10.7,2F5.1)
    4       NN=N+1
    5       READ(5,101)(A(I),I=1,NN)
    6   101 FORMAT(5F5.1)
    7       M=0
    8    10 J=N-2*M
    9       M=M+1
   10       JJ=J+1
   11       IF(J-2)20,25,30
   12    20 XR(N)=-A(2)/A(1)
   13       XI(N)=0
   14       GO TO 60
   15    25 AA=A(1)
   16       BB=A(2)
   17       CC=A(3)
   18       GO TO 51
   19    45 DO 50 I=1,JJ
   20    50 A(I)=B(I)
   21       AA=1.
   22       BB=P
   23       CC=Q
   24    51 D=BB*BB-4.*AA*CC
   25       IF(D)53,52,52
   26    53 X1J=-BB/(2.*AA)
   27       X2J=X1J
   28       X1K=SQRT(-D)/(2.*AA)
   29       X2K=-X1K
   30       GO TO 55
   31    52 DD=SQRT(D)
   32       X1J=(-BB+DD)/(2.*AA)
   33       X2J=(-BB-DD)/(2.*AA)
   34       X1K=0.
   35       X2K=0.
   36    55 XR(2*M-1)=X1J
   37       XR(2*M)=X2J
   38       XI(2*M-1)=X1K
   39       XI(2*M)=X2K
   40       IF(J-2)60,60,10
   41    60 WRITE(6,200)
   42   200 FORMAT(1H ,4HROOT,9X,9HREAL PART,7X,14HIMAGINARY PART)
   43       DO 65 I=1,N
   44    65 WRITE(6,201)I,XR(I),XI(I)
   45   201 FORMAT(1H ,I3,8X,E12.5,7X,E12.5)
   46       STOP
   47    30 NL=1
   48    35 B(1)=A(1)
   49       B(2)=A(2)-P*B(1)
   50       DO 36 I=3,JJ
   51    36 B(I)=A(I)-P*B(I-1)-Q*B(I-2)
   52       C(1)=B(1)
   53       C(2)=B(2)-P*C(1)
   54       DO 37 I=3,J
   55    37 C(I)=B(I)-P*C(I-1)-Q*C(I-2)
   56       CX=C(J)-B(J)
   57       CY=C(J-1)**2-CX*C(J-2)
   58       DP=(B(J)*C(J-1)-B(JJ)*C(J-2))/CY
   59       DQ=(B(JJ)*C(J-1)-B(J)*CX)/CY
   60       P=P+DP
   61       Q=Q+DQ
   62       IF (ABS(DP)-EPS) 41,41,40
   63    41 IF (ABS(DQ)-EPS) 45,40,40
   64    40 IF (NL-100) 42,45,45
   65    42 NL=NL+1
   66       GO TO 35
   67       END
```

プログラム11

```
      C     FX-2-B   ***** NEW CALENDAR *****
    1       DIMENSION ADAY(6,21),BMOON(12),HDAY(31)
    2       DIMENSION IBMOON(12)
    3       DIMENSION IHDAY(31)
    4       READ(5,501) (BMOON(I),I=1,12)
    5       READ(5,501) (HDAY(I),I=1,31)
    6       READ(5,501) XXX
    7       READ(5,501) YYY
    8       READ(5,502) (IBMOON(I),I=1,12)
    9       READ(5,502) (IHDAY(I),I=1,31)
   10     501 FORMAT(A2)
   11     502 FORMAT(I2)
   12       IYEAR=1973
   13       IPURU=1976
   14       IMURU=1972
   15       IX=2
   16       II=0
   17       MMM=1
   18       READ(5,100) NEN
   19     100 FORMAT(I4)
   20       WRITE(6,200) NEN
   21     200 FORMAT(1H1,55X,11H***** YEAR-,I4,6H *****//////)
   22       WRITE(6,201)
   23     201 FORMAT(1H ,55X,26HMADE BY COMPILER GENERATER//)
   24       IF(NEN-IYEAR) 44,22,22
   25      22 IF(IPURU-IYEAR) 5555,33,5555
   26    5555 IF(IYEAR-NEN) 4444,55,4444
   27    4444 IX=IX+1
   28      25 IYEAR=IYEAR+1
   29         IF(IX-7) 22,22,3333
   30    3333 IX=1
   31         GO TO 22
   32      33 IF(IYEAR-NEN) 2222,35,2222
   33    2222 IPURU=IPURU+4
   34         IX=IX+2
   35         IF(IX-9) 25,1111,1111
   36    1111 IX=2
   37         GO TO 25
   38      35 BMOON(2)=YYY
   39      55 DO 10 I=1,21
   40         DO 30 J=1,6
   41         ADAY(J,I)=XXX
   42      30 CONTINUE
   43      10 CONTINUE
   44         ISUN=1
   45         ISAT=7
   46      54 KK=1
   47         JJ=1
   48         II=II+1
   49      53 ADAY(JJ,IX)=HDAY(KK)
   50         IF(IHDAY(KK)-IBMOON(II)) 9999,88,9999
   51    9999 IX=IX+1
   52         KK=KK+1
   53         IF(IX-ISAT) 53,53,8888
   54    8888 JJ=JJ+1
   55         IX=ISUN
   56         GO TO 53
   57      88 IF(ISAT-21) 7777,90,90
   58    7777 ISUN=ISUN+7
   59         ISAT=ISAT+7
   60         IX=IX+8
   61         IF(IX-ISAT) 54,54,6666
   62    6666 IX=ISUN
   63         GO TO 54
   64      90 M=MMM
```

```
      MN=M+1
      MMM=MM+1
      WRITE(6,600)
  600 FORMAT(1H0/1H0)
      WRITE(6,300) M,MN,MMM
  300 FORMAT(1H0,19X,9H*** MOON-,I2,4H ***,23X,9H*** MOON-,I2,4H ***,23X
     -,9H*** MOON-,I2,4H ***)
      WRITE(6,400)
  400 FORMAT(1H0,10X,33HSUN  MON  TUE  WED  THE  FRI  SAT,5X,33HSUN  MON
     -  TUE  WED  THE  FRI  SAT,5X,33HSUN  MON  TUE  WED  THE  FRI  SAT)
      WRITE(6,500) ((ADAY(J,I),I=1,21),J=1,6)
  500 FORMAT(1HO,11X,A2,3X,A2,3X,A2,3X,A2,3X,A2,3X,A2,3X,A2,6X,A2,3X,A2,
     -3X,A2,3X,A2,3X,A2,3X,A2,3X,A2,6X,A2,3X,A2,3X,A2,3X,A2,3X,A2,3X,A2,
     -3X,A2)
      IF(II-12) 4010,999,999
 4010 MMM=MMM+1
      IX=IX-13
      IF(IX-7) 55,55,4020
 4020 IX=1
      GO TO 55
   44 IF(IMURU-IYEAR) 4030,43,4030
 4030 IF(IYEAR-NEN) 4040,55,4040
 4040 IX=IX-1
   45 IYEAR=IYEAR-1
      IF(IX-1) 4050,44,44
 4050 IX=7
      GO TO 44
   43 IF(IYEAR-NEN) 4060,35,4060
 4060 IMURU=IMURU-4
      IX=IX-2
      IF(IX+1) 45,4070,45
 4070 IX=6
      GO TO 45
  999 STOP
      END
```

昭和49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丁目5番8号
機械振興会館内
TEL (434) 8211 (代表)

印刷所　三協印刷株式会社
東京都渋谷区渋谷3－11－11
TEL (407) 7316

48－S003