Vol. 22 No. 4 情報処理学会論文誌 July 1981

2パス変換システムによるプリプロセッサ生成機の実現†

海尻 賢二††

本論文では構文マクロを利用した2パスのプリプロセッサ生成機システムについて述べる．構文マクロの利用においては構文を利用するが，入力のすべての構文を解析するのは無駄であるので，変換の必要のない構文を構成する記号列を1つのトークンとみなし，other トークンと呼ぶ．語いレベルで other トークンかマクロ記号かを決定し，以降の解析を行う．すなわち変換の必要のない記号列は other と呼ばれる1つのトークンとみなす．この方法によりマクロの定義は容易になりまた作成されるプリプロセッサの速度も向上する．

マクロの定義はマクロ記号と特別の other トークンを定義する語い定義，構文およびそれに対応する解析木を定義する構文定義，その解析木に対してマクロ本体を定義する意味定義の3つより成る．

システムは各定義を処理してマクロ定義表を作成するプリプロセッサ生成機と，その表によって駆動される核プリプロセッサより成る．プリプロセッサは2パスであり，弱順位文法に基づくパーザと，ユーザの定義した形式の解析木を出力プログラムへ変換するトランスフォーマの2つより成る．

システムはすべて構造化 Fortran RATFOR-R で記述されている．

1. まえがき

言語の新しい概念（データ構造，制御構造等）を簡単に実現するための道具としてプリプロセッサは有用である．しかし毎回新しい概念を考えるごとにプリプロセッサを作ることは大変であり，また無駄が多い．そのためにはプリプロセッサ生成機システム(PGS)が望まれる．PGS はコンパイラ生成機システム (CGS) とは異なり，作成されるプリプロセッサの効率はさほど問題としない（プリプロセス時間は，プリプロセス後のコンパイル時間と同程度であれば十分である）．また機械にも独立であり，言語のみに依存する．その点 CGS よりも作成が容易であるといえる．

PGS の実現にはほとんどの場合汎用のマクロプロセッサが使用される．PGS に利用するマクロプロセッサとしては stage 2[1]，MAX[2] のような行という制限を持ったものは不適当である．行の制限を取りはらい，構文的なイメージを持たせたものに PM[3]，また構文マクロといえるものに SIL[4] がある．両者とも PGS としての使用の実績もある高水準のマクロであるが，マクロという性質上，誤りメッセージが出しにくい，本質的にパターンマッチングで処理を行う，という欠点（ある意味では長所）を持っている．

CGS では YACC, Staple[5] におけるようにまず構文の記述よりパーザを生成し，その解析木に対して意味付け（コード生成）をユーザ自身が手続き的に記述し，それをパーザとリンクすることによりコンパイラを生成するという形態を取る．

そこでこのマクロの考え方（マクロに関係する構文のみ認識し，ほかは単なる記号列として扱う）と，CGS の考え方（構文解析により構文を認識し，解析木に対して意味付けを行う）を併合することを考える．CGS の考え方をそのまま PGS に適用すると不要な構文まで記述し，またそれを認識しなければならなくなり無駄が多い．たとえば Fortran の構造化のためのプリプロセッサを作成する時，While, Repeat 等の新しく付加する命令の構文だけでなく，すべての Fortran の命令の構文を記述しなければならなくなり，その仕事は無駄であり，また困難でもある．

そこで構文マクロ的であり，かつ誤り処理も可能である，そしてマクロに関係する構文のみ構文解析し，関係のない構文（すなわちマクロに対する誤り，いい換えるとプリプロセッサが分担すべき誤りには関与しない構文）については単なる記号列とみなし，変換を行うようなプリプロセッサを作成する PGS を設計し，実現した．

本 PGS で作成されるプリプロセッサは途中に2進木形式の解析木を持つ2パスのプロセッサであり，構文解析は弱順位パーザで行う．PGS の入力記述は，

① 構文および構文に対する解析木の記述（語いの記述を含む）

② 解析木から出力言語への変換規則の記述

の2つである．解析木の形態をユーザが指定するという方法は staple においても採用しているが，生成規則に対して記述する方法よりもより抽象的なレベルに対しての記述となるので簡単となるという利点を持

† Implementation of a Preprocessor Generator with a 2 Pass Transformation System by KENJI KAIJIRI (Department of Information Engineering, Faculty of Engineering, Shinsyu University).

†† 信州大学工学部情報工学科

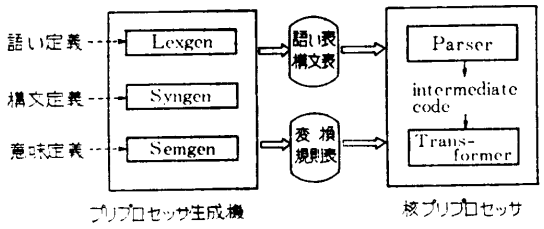

図1 プリプロセッサ生成機システム

Fig. 1 Preprocessor generator system.

つ．本システムではほとんどのマクロにみられるような，特定のマクロ記号でマクロ構文を始めなければならないといった制限はなく，また後で述べる skip および other 記号を利用すれば，マクロ記号であっても他の命令に利用できる．マクロ記号でないトークンは語い解析のレベルで全体として1つのトークンとみなされ，その構造については構文解析では関知しない．

```
token def.->DEFINE TOKEN {token def. statement %}+ DEFEND
token def. statement->DEL( number )= token1|
                      OTHER= token2 token3 |
                      SKIP= token2 token3  |
                      COMMENT= token2 token3
token1->OTHER | symbol symbol | token3
token3-> token2 | EOL
token2-> alphabet{alphabet|digit}* | symbol

syntax def.->DEFINE SYNTAX {syntax def. statement}+ DEFEND
syntax def. statement-> production rule @ tree description @
tree description->tree {; tree}*
tree->ASSIGN name TO node description | node description
node description->TREE( label , node {, node} )|
                  PROG( node ) | NULL( number )|
                  OP( number ) | NIL
label-> alphabet { alphabet | digit }*
name-> alphabet
node-> name | number

semantics def.->DEFINE SEMANTICS {semantics def. statement}+ DEFEND
semantics def. statement-> label:BEGIN semantics des. body END
semantics des. body->{LOCAL namelist;}{assignment;}*code body
namelist-> name {,name}*
assignment-> lefthand = righthand
lefthand-> LH | RH | name
righthand-> HEAD | name{{+|-}number}|@{NUMGEN | LABGEN}
code body-> [code element {,code element}*]
code element->  LH | RH | number{X|T} |
              {NUM | STR}( name ) | " string " | /
```

図2 定義言語

Fig. 2 DEFINITION language.

以上の設計目標に基づいて作成したシステムの概要を**図1**に示す．システムはマクロ定義を処理してマクロ定義表を作成するプリプロセッサ生成機と，その表によって駆動される核プリプロセッサの2つより成る．

以下2章ではマクロ定義について，3章ではシステムについてそれぞれ述べる．なお説明は次のような制御命令を持つ構造化 Fortran の Fortran へのプリプロセッサ作成を例として行う．

If condition Then statement-list {Else statement-list}* Fi
Do control-list statement-list Od
While condition statement-list Endwhile
Repeat statement-list Until condition

2. マクロ定義

従来マクロ定義はマクロ名とマクロ本体という形で行われているが，本システムでは次の3つに分割して行うこととした．

Ⅰ　語い定義：マクロを構成する記号の定義．

Ⅱ　構文定義：マクロ構文とそれに対応する木構造の中間コードの定義．構成的属性を用いて行う．

Ⅲ　意味定義：中間コードをマクロ名とみて，それに対するマクロ本体の定義．相続的属性を用いて行う．

各定義の定義言語の仕様を**図2**に示す．図2において $\{\alpha\}$ は空または α を，$\{\alpha|\beta\}$ は α または β を，$\{\alpha\}^*$ は0回以上の α の繰り返しを，そして $\{\alpha\}^+$ は1回以上の α の繰り返しをそれぞれ表す．

2.1 語い定義

プリプロセッサにおいては入力のすべてを変換するのではなく，ある特定の構文のみ変換する．そのため変換の必要がない構文については一切その構造は問題とせず，単なる記号列として扱うことが望ましい．たとえば算術式などはほとんど変換の必要がない場合が多く，またその出現頻度も高い．これらの構文を単なる記号列として扱うことにより定義が容

* オプションを示す

易になり，またプリプロセッサの処理速度も早まる．この変換の必要のない構文に対する記号列を1つのトークンとみなし，other と呼び，マクロ名として使われる記号(列)を delimiter と呼ぶ．語い定義では次の4種類の文により delimiter および特別な other トークンの定義を行う．

i) Del(i)=α

内部コードiを持つ delimiter α の定義．特別の α として "other" と "eol" がある．"other" は other トークンを意味し，定義の便宜上 delimiter の定義文中に含めてある．"eol" は行末を示し，delimiter として利用する時にのみ使用する．

ii) Other=$\gamma\ \delta$

γ で始まり δ で終る記号列（ただし1行に限る）を other として定義する．

iii) Skip=$\gamma\ \delta$

other 文と同様であるがトークンとしてγとδを含めない．

iv) Comment=$\gamma\ \delta$

γ で始まり δ で終る記号列を入力プログラムの注釈とみなして無視する．

入力は上記4種類の定義に従い delimiter と other に分類される．ii)からiv)の文で定義された記号列を除いて，delimiter で挟まれた任意の記号列（ただし1行以内）は1つの other トークンである．(定義文上は "other" も1つの delimiter であるが "other" という文字列が delimiter となるわけではない．) Other とすべき記号列中に delimiter として定義した記号列を含めたい時にはii)またはiii)の文を利用する．マクロにおいては特別の接頭辞を必要とするものが多いが，この方法では原則としてそれは不要である．ただし delimiter として定義した記号列はii)またはiii)の文を利用して使用しない限り，delimiter としてしか使えない．

例 1. 構造化 Fortran の語い定義

図3に語い定義の例を示す．この定義に従うならば

"If I=2 Then J=2"

は delimiter If Then を含んでいるが，"で囲まれているので左端および右端の" を含めて1つの other トークンとみなされる．また#で始まる行は注釈となり無視される．**図4**に入力プログラムの例を示すが，図中で下線を引いた記号列が各々1つのトークンである．(例終)

図3では skip 記号として？ EOL を定義しているのでプログラム中のマクロに関係のない行は先頭に？をつけることによっても区別できることになる．たとえば算術 If 文を使いたい時には，

？ If(I) 10, 20, 30

とすればよい．一般のマクロではこのようなものは then (そして else) がないことから識別する．

```
010 DEFINE TOKEN
020  DEL(1)=END %
030  DEL(2)=IF %
040  DEL(3)=THEN %
050  DEL(4)=FI %
060  DEL(5)=ELSE %
070  DEL(6)=WHILE %
080  DEL(7)=ENDWHILE %
090  DEL(8)=DO %
100  DEL(9)=OD %
110  DEL(10)=REPEAT %
120  DEL(11)=UNTIL %
130  DEL(12)= & %
140  DEL(13)= ! %
150  DEL(14)= ^ %
160  DEL(15)= < %
170  DEL(16)= <= %
180  DEL(17)= == %
190  DEL(18)= ^= %
200  DEL(19)= >= %
210  DEL(20)= > %
220  DEL(21)= [ %
230  DEL(22)= ] %
240  DEL(23)=OTHER %
250  DEL(24)=EOL %
260  OTHER= " " %
270  COMMENT= # EOL %
280  SKIP= ? EOL %
290 DEFEND
```

図3 構造化 Fortran の語い定義

Fig. 3 Token definition of a structured Fortran.

```
# PRIME NUMBER
READ,N
I=2
WRITE(6,100) I
100 FORMAT(1H ,I4)
DO I=3,N,2
  K=3
  WHILE K*K<=I & MOD(I,K)^=0
    K=K+2
  ENDWHILE
  IF K*K>I THEN
    WRITE(6,100) I
  FI
OD
STOP
END
```

図4 プログラム例

Fig. 4 Example program.

2.2 構文定義

構文定義は生成規則の BNF による定義と，中間コードである解析木の部分木を属性値とするような構成的属性の計算規則の2つから成り，次のような形式

の構文定義文の集合である.

production rule @ tree description @

Tree description の部分で示した規則に従って求めた属性値（解析木の部分木）が production rule の左辺の非終端語の構成的属性の値（非終端語に対する解析木）となる．解析木の葉はすべて other トークンであり，1つ以上の部分木が1つの生成規則と対応する(ただし部分木と対応しない生成規則もある).

Tree description は次の2種類の文から成る.

Ⅰ. Assign name To node-description

Node-description に従い木を作成し，それにアルファベット1字から成る名前をつける.

Ⅱ. node-description

それまでに作られた木をもとにして解析木を作る命令で，次の4種類のものがある．(以下で nodeに対応する木とは，ⅰ) node が数字 i の時は生成規則の右辺の i 番目の記号（非終端語に限る）に対する木，ⅱ) node が name の時はそれまでに syntax def. statement 内でその名前をつけられた木，をそれぞれ表す)

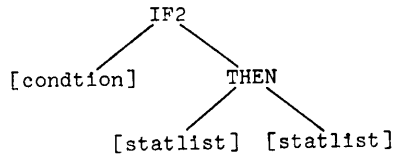

図 5 [Ifs] に対する解析木

Fig. 5 Parse tree for a nonterminal [Ifs].

Ⅱ-1. Tree (label, node {, node})

label をノードラベルとして持ち，node に対応する木を左部分木(および右部分木)として持つ木を作る.

例 2. [Ifs] に対する木

[Ifs]#IF [condtion] THEN [eolsym] [statlist]
ELSE [eolsym] [statlist] FI
@ Assign A to Tree (then, 5, 8); Tree (If 2, 2, A) @

まず then と else の後の statlist に対応する木より "then" をノードラベルとする木をつくりAと名付ける．次にこの木と condtion に対応する木より "If2" をノードラベルとする**図5**のような木をつくり，これを Ifs の構成的属性の値とする．(例終)

Ⅱ-2. Prog (node)

node に対応する木の根を全体の木の根として定める．**図6**の構造化 Fortran の構文定義では subprog に対する木を解析木と定めている.

```
010 DEFINE SYNTAX
020  [PROGRAM]#[SUBPROG]                                    @PROG(1)@
030  [SUBPROG]#[SUBPROG] [STATLIST] END EOL                 @TREE(PROGPARA,1,2)@
040  [SUBPROG]#[STATLIST] END EOL  -                        @TREE(PROG,1)@
050  [STATLIST]#[STATLIST] [STATMENT] EOL                   @TREE(PARA,1,2)@
060  [STATLIST]#[STATMENT] EOL                              @NULL(1)@
070  [STATMENT]#[IFS]                                       @NULL(1)@
080  [STATMENT]#[WHILES]                                    @NULL(1)@
090  [STATMENT]#[REPEATS]                                   @NULL(1)@
100  [STATMENT]#[DOS]                                       @NULL(1)@
110  [STATMENT]#[OTHER]                                     @NULL(1)@
120  [IFS]#IF [CONDTION] THEN [EOLSYM] [STATLIST] FI        @TREE(IF1,2,5)@
130  [IFS]#IF [CONDTION] THEN [EOLSYM] [STATLIST] ELSE [EOLSYM] [STATLIST] FI
140       @ASSIGN A TO TREE(THEN,5,8); TREE(IF2,2,A)@
150  [WHILES]#WHILE [COND2] [EOLSYM] [STATLIST] ENDWHILE    @TREE(WHILE,2,4)@
160  [COND2]#[CONDTION]                                     @NULL(1)@
170  [DOS]#[DOOTHER] [EOLSYM] [STATLIST] OD                 @TREE(DO,1,3)@
180  [DOOTHER]#DO [OTHER]                                   @NULL(2)@
190  [REPEATS]#REPEAT [EOLSYM] [STATLIST] UNTIL [CONDTION]
200       @TREE(REPEAT,3,5)@
210  [EOLSYM]#EOL                                           @NIL@
220  [CONDTION]#[CONDTION] & [CEXP]                         @TREE(AND,1,3)@
230  [CONDTION]#[CONDTION] ! [CEXP]                         @TREE(OR,1,3)@
240  [CONDTION]#^ [CEXP]                                    @TREE(NOT,2)@
250  [CONDTION]#[CEXP]                                      @NULL(1)@
260  [CEXP]#[CEXP] < [CTERM]                                @TREE(LT,1,3)@
270  [CEXP]#[CEXP] <= [CTERM]                               @TREE(LE,1,3)@
280  [CEXP]#[CEXP] == [CTERM]                               @TREE(EQ,1,3)@
290  [CEXP]#[CEXP] ^= [CTERM]                               @TREE(NE,1,3)@
300  [CEXP]#[CEXP] >= [CTERM]                               @TREE(GE,1,3)@
310  [CEXP]#[CEXP] > [CTERM]                                @TREE(GT,1,3)@
320  [CEXP]#[CTERM]                                         @NULL(1)@
330  [CTERM]#[OTHER]                                        @NULL(1)@
340  [CTERM]#[ [CONDTION] ]                                 @NULL(2)@
350  [OTHER]#OTHER                                          @OP(1)@
360 DEFEND
```

図 6 構造化 Fortran の構文定義

Fig. 6 Syntax definition of a structured Fortran.

```
010 DEFINE SEMANTICS
020  PROGPARA:BEGIN
030    [LH,
040     RH,
050     7T,"END",/
060    ]
070   END
080  PROG:BEGIN
090    [LH,/,
100     7T,"END",/
110    ]
120   END
130  PARA:BEGIN
140    [LH,/,RH]
150   END
160  IF1:BEGIN
170    LOCAL LAB;
180    LAB=@NUMGEN;
190    [7T,"IF(.NOT.(",LH,")) GO TO ",NUM(LAB),/,
200     RH,/,
210     1X,NUM(LAB),7T,"CONTINUE",/
220    ]
230   END
240  IF2:BEGIN
250    LOCAL LAB;
260    LAB=@NUMGEN;
270    RH=LAB;
280    [7T,"IF(.NOT.(",LH,")) GO TO ",NUM(LAB),/,
290     RH,/
300    ]
310   END
320  THEN:BEGIN
330    LOCAL LAB,LAB1;
340    LAB=@NUMGEN; LAB1=HEAD;
350    [LH,/,
360     7T,"GO TO ",NUM(LAB),/,
370     1X,NUM(LAB1),7T,"CONTINUE",/,
380     RH,/,
390     1X,NUM(LAB)," CONTINUE",/
400    ]
410   END
420  WHILE:BEGIN
430    LOCAL LAB,LAB2;
440    LAB=@NUMGEN; LAB2=@NUMGEN;
450    [1X,NUM(LAB),7T,"CONTINUE",/,
460     7T,"IF(.NOT.(",LH,")) GO TO ",NUM(LAB2),/,
470     RH,/,
480     7T,"GO TO ",NUM(LAB),/,
490     1X,NUM(LAB2),7T,"CONTINUE",/
500    ]
510   END
520  DO:BEGIN
530    LOCAL LAB;
540    LAB=@NUMGEN;
550    [7T,"DO ",NUM(LAB),1X,LH,/,
560     RH,/,
570     1X,NUM(LAB),7T,"CONTINUE",/
580    ]
590   END
600  REPEAT:BEGIN
610    LOCAL LAB;
620    LAB=@NUMGEN;
630    [1X,NUM(LAB),7T,"CONTINUE",/,
640     LH,/,
650     7T,"IF(.NOT.(",RH,")) GO TO ",NUM(LAB),/
660    ]
670   END
680  AND:BEGIN
690    [LH,".AND.",RH]
700   END
710  OR:BEGIN
720    [LH,".OR.",RH]
730   END
740  NOT:BEGIN
750    [".NOT.",LH]
760   END
770  LT:BEGIN
780    [LH,".LT.",RH]
790   END
800  LE:BEGIN
810    [LH,".LE.",RH]
820   END
830  EQ:BEGIN
840    [LH,".EQ.",RH]
850   END
860  NE:BEGIN
870    [LH,".NE.",RH]
880   END
890  GE:BEGIN
900    [LH,".GE.",RH]
910   END
920  GT:BEGIN
930    [LH,".GT.",RH]
940   END
950 DEFEND
```

図 7 構造化 Fortran の意味の定義

Fig. 7 Semantics definition of a structured Fortran.

```
      IF(.NOT.( condtion ) GO TO lab
statlist
      GO TO labl
lab   CONTINUE
statlist'
labl CONTINUE
```

図 8 If 文の変換例

Fig. 8 Transformation example of an If statement.

Ⅱ-3. Op(i)

生成規則の右辺の i 番目の要素に対して葉ノードを作る．これは終端語 other に対してのみ用いる．

Ⅱ-4. Null(i)

生成規則の右辺の i 番目の要素に対する木をこの規則に対する木とする．図6の34行目では condtion に対する木を cterm に対する木と定めている．

Ⅱ-5. Nil

解析木に関する演算は行わないことを指示する．

構文解析は上昇型で行うので，生成規則の認識ごとに対応する tree description に従って解析木を作っていく．意味定義はこの解析木に対して行うので，tree description ではそのことを考慮して，意味定義の行いやすい構造を定義する必要がある．

2.3 意味定義

本システムで作成されるプリプロセッサは2パスであり，第1パスでユーザの指定した解析木へ変換され，第2パスではその解析木に対してマクロ展開を行う．そこで意味定義ではこの解析木の部分木（ラベル付けされている）ごとに相続的属性を使ってマクロ本

体を定義する．構造化 Fortran の意味定義を**図7**に示す．

定義は各ノードラベルごとに行われ，次のような形をしている．

```
label : Begin
        Local……;
        assignment;
        [マクロ本体]
        End
```

Local…… は局所的変数の宣言でオプションである．Assignment 文は局所的変数および属性値の計算を行うもので，そのために次の2種類の関数が用意されている．

① @ Numgen : 4桁の自然数を発生する

② @ Labgen : 長さ4の文字列を発生する

局所的変数および属性は型を持たず，代入文の型およびマクロ本体で指定される型に従う．属性は各ノードごとに1つずつあり，それぞれ head, lh, rh で表す．

例 3.
```
250 Local lab;
260 lab=@ Numgen;
270 rh=lab;
```
局所的変数 lab に4桁の自然数を代入し，かつその値を右部分木の属性値とする．(例終)

マクロ本体の定義には次の6種類の code element が使われる．

① lh, rh : マクロのパラメータを示す．本システムではパラメータは左部分木と右部分木を示すものしかなく，各々それを示す．

② nX, nT : それぞれ n 個の空白の出力，出力位置を n 桁目に設定を意味する．

③ Num (name) : 局所的変数 name の値を4桁の自然数として出力する．

④ Str (name) : 局所的変数 name の値を長さ4の文字列として出力する．

⑤ "string" : 記号列 string をそのまま出力する．

⑥ ／: その時点での出力，すなわち出力バッファの内容を出力し，初期化することを意味する．

例 4. 図5の If 文に対する解析木のマクロ本体の定義は図7の 240～400 であり，**図8**のような変換を意味する．(例終)

```
      READ,N
      I=2
      WRITE(6,100) I
  100 FORMAT(1H ,I4)
          DO  9451 I=3,N,2
      K=3
        9452CONTINUE
          IF(.NOT.(K*K.LE.I .AND.MOD(I,K).NE.0)) GO TO  9453
      K=K+2
          GO TO  9452
        9453CONTINUE
          IF(.NOT.(K*K.GT.I )) GO TO  9454
      WRITE(6,100) I
        9454CONTINUE
        9451CONTINUE
      STOP
          END
```

図9 出力例
Fig. 9 Output example

3. プリプロセッサ生成機システム

システムはマクロ定義を処理してマクロ定義表を作成するプリプロセッサ生成機と，この表により駆動される核プリプロセッサの2つより成る．構文定義は弱順位行列と，生成規則の右辺の記号により分類したリスト構造の生成規則表に変換される．意味定義は内部コード化される．

核プリプロセッサは入力プログラムの解析を行い，指示された形の解析木へ変換する parser と，解析木を意味表に従い出力プログラムへ変換する transformer の2つより成る．構文解析は弱順位パーザで行うが，生成規則の認識は分類された生成規則リストを辿ることにより行うので効率がよい．

構文誤りの処理に関しては Ripley[6] の方法を弱順位パーザ向に修正した次のような方法を採用している．

(弱順位パーザに対する誤り回復アルゴリズム)

① シフト誤り：スタックの内容をすべて消去して記号？を入れる．？はすべての終端語に対してシフト関係にある．

② 還元誤り ($B \gtrdot C$ とする)

ⅰ) B を右辺の右端に持つ生成規則がない時は，シフト誤りと同じ処理を行う．

ⅱ) 途中まで一致している時は，そのような生成規則の内で最も短い規則で還元を行う．

構文解析には弱順位パーザを利用しているが，実際には図6にもあるように uniquely invertible でない規則がしばしば存在する．この点については mixed strategy 法等による解決策も考えられたが，本システムでは効率等の面からそのような方法は採らず，その部分に限り手で parser のプログラムに修正を加える方法を採った．

システムはすべて構造化 Fortran RATFOR-R[7] で作成し，ACOS 600 S の TSS 上で稼動している．図4のプログラムに対する

出力を**図9**に示す．作成した構造化 Fortran のプリプロセッサの処理速度は約 30 行／秒である．

4. むすび

本論文では構文マクロを利用したプリプロセッサ生成機システム (PGS) について述べた．PGS は基本的にはコンパイラ生成機システム (CGS) の考え方を採用しているが，語いレベルで命令を認識する時にはマクロ的に行うという方法を採った．そのため入力の記述は必要最小限，すなわち定義する命令に関与するもののみ，となり，また作成されるプリプロセッサはマクロを利用したものとは異なり，構文的な誤りメッセージも出せ，また誤り回復も行う．PGS によりこれまで，

guarded command から Fortran
構造化アセンブリ言語から ACOS アセンブリ言語
構造化 Fortran から Fortran

の3つのプリプロセッサを作成した．入力の記述はいずれも簡単であり数時間で完了した．特に構造化 Fortran のプリプロセッサは CGS 的アプローチをそのまま採用するならば生成規則の数は数百となり，簡単に記述できるものではない．しかし PGS では "other" という特別のトークンを想定することにより，図6にあるようにわずか 36 行の記述ですんだ（ただし構文のみ）．この方法は既存の言語に新しい命令を追加する時に非常に有効である．マクロを利用する場合でも記述するべき対象は定義する命令だけであるが，その場合くり返しパターン (case 文等) の処理に工夫を要し，また，break, next のような命令の定義は困難である．PGS でも現在のところは break, next のような命令は定義できないが，属性を若干追加することにより可能である．

PGS の記述においての問題点は入力文法を弱順位文法にするということであり，このために文法の変形が必要となってくる．もう少し記述能力の高い文法 (SLR 等) を利用することも考慮中である．

PGS で作成されるプリプロセッサは語いのレベルでマクロパターン (delimeter) か，そのパラメータ (other) かを区別するのでマクロパターンの選択には十分注意する必要がある．しかし余分な接頭辞は原則として必要とせず，また文字レベルではなく単語レベルで識別を行うので，処理速度は早い．

参考文献

1) Waite, W. M.: The Mobile Programming System: STAGE 2, C. ACM, Vol. 13, No. 7, pp. 415-421 (1970).
2) Nudds, D.: The design of the MAX macro-processor, The Computer Journal, Vol. 20, No. 1, pp. 30-36 (1977).
3) Sassa, M.: A Pattern Matching Macro Processor, Software P. & E., Vol. 9, No. 6, pp. 439-456 (1979).
4) 尾内理紀夫，雨宮真人：形手続きによるプログラミング言語 SIL とその処理系，電子通信学会研究会資料，EC 79-30 (1979).
5) Pierce, R. H. and Rowell, J.: A transformation directed compiling system, The Computer Journal, Vol. 20, No. 2, pp. 109-115 (1977).
6) Ripley, G. David: A Simple Recovery Only Procedure For Simple Precedence Parsers, C. ACM, Vol. 21, No. 11, pp. 928-930 (1978).
7) 海尻：構造化 Fortran について，昭和 55 年度電子通信学会総合全国大会，1470.

(昭和55年8月11日受付)
(昭和56年2月19日採録)