<AFFIRM>PARSERPLUS..13 30-Sep-81 15:30:32

```
(FILECREATED "28-Sep-81 14:57:39" <AFFIRM>PARSERPLUS..13 14103

     changes to:  PARSERPLUSCOMS

     previous date: "26-Sep-81 16:51:50" <AFFIRM>PARSERPLUS..12)


(PRETTYCOMPRINT PARSERPLUSCOMS)

(RPAQQ PARSERPLUSCOMS ((* This file is loaded into the parser itself, and becomes part of its block.
                        Note that DWIMPARSER modifies the parser's BLKAPPLYFNS so only the root and
                        the followoing production names may be called upon externally.)
        [VARS (ParserRoots (QUOTE (denoteSpec distinctSpec expression expressionSeq functionDecl
                                   interfaceList nochangeSpec rangedInterfaceList ruleSeq
                                   variableDecl]
        (FNS * PARSERPLUSFNS)
        (P (CLISPDEC (QUOTE FAST)))
        (DECLARE: DOEVAL@COMPILE
                  (PROP MACRO CopyTopRecord SIFromStack? SIMark SINewStack SINexi SINextNew
                        SINextSaved SISaveLexeme SIToMark SIUnmark UsersNextInput labelFilter))
        (VARS Delimiters KeyWordList ReserveWordList SpecialPrefixOps UpperCaseVars)
        [VARS (PARSERPROMPT (QUOTE ~>))
              (USESLOWERCASE T)
              (PARSERTRACE NIL)
              (COLLECTTOKENS T)
              (UCaseParseAtoms (QUOTE (TRUE FALSE]
        (PROP UCASE * LOWERCASE)
        (P (SETQ PASCAL\READ\TABLE (COPYREADTABLE (QUOTE ORIG)))
           (SETBRK (QUOTE (4 5 6 14 16 17 18 19 20 21 27 28 29 34 30 33 38 40 41 42 43 44 45 46 47 58
                           59 60 61 62 64 91 93 94 123 124 125 126))
                   NIL PASCAL\READ\TABLE))))
[DECLARE: DONTEVAL@LOAD DONTCOPY
```

(* This file is loaded into the parser itself, and becomes part of its block. Note that DWIMPARSER modifies the parser's BLKAPPLYFNS so only the root and the followoing production names may be called upon externally.)]

```
(RPAQQ ParserRoots (denoteSpec distinctSpec expression expressionSeq functionDecl interfaceList
                            nochangeSpec rangedInterfaceList ruleSeq variableDecl))

(RPAQQ PARSERPLUSFNS (CopyTopRecord MatchConstant MatchLexeme PARSEPROGRAM PARSE\ASSERTION
                                    ParserRATOM ReadAtom SIFromStack? SIMark SINewStack SINext
                                    SINextNew SINextSaved SISaveLexeme SIToMark SIUnmark
                                    UsersNextInput identifierFilter infixOpFilter labelFilter
                                    prefixOpFilter simplifyStatement specialPrefixOpFilter
                                    unsignedInteger unsupportedPascalFilter))
(DEFINEQ
```

1

```
(CopyTopRecord
  [LAMBDA (x)
    < ! x>])
```

2

```
(MatchConstant
  [LAMBDA (constants emptyOK)
    (PROGN (if CurrentLexeme MEMB constants
               then ProductionValue←CurrentLexeme
                    (SINext)
                    ProductionValue
           else emptyOK])
```

3

```
(MatchLexeme
  [LAMBDA NIL
    (if CurrentLexeme~=Terminator
        then CurrentRecord:LEXEME←CurrentLexeme
             (SINext)
             T])
```

4

```
(PARSEPROGRAM
  [LAMBDA (FileName)
    (PROG [Value (ERRORTYPELST ('((16 (PROGN BREAKCHK←NIL
                                             PRINTMSG←NIL
                                             (RETFROM ERRORPOS 'STOP T]
          (if FileName=NIL
              then FileName←T)
          (PARSEDTOKENS← <NIL>)
          (if Value←(PARSER 'program <FileName 'STOP > T)=NIL
              then (if FileName=T
                       then (CLEARBUF T)))
          (RETURN (reduceParseTree Value])
```

5

```
(PARSE\ASSERTION
  [LAMBDA NIL
    (PROG (Value)
          (PARSEDTOKENS← <NIL>)
          (if Value←(PARSER 'expression <T '; > NIL)=NIL
              then (CLEARBUF T))
          (RETURN (reduceExpression Value])
```

6

```
(ParserRATOM
  [LAMBDA (filename FileNameStopAtom ReadTable)          (* R.Bates "19-OCT-78 10:08")
    (PROG (temp)
          (RETURN (if filename=NIL
                      then FileNameStopAtom:2
                    elseif (LISTP filename)
                      then temp←FileNameStopAtom:1:1
```

```
                    FileNameStopAtom:1←FileNameStopAtom:1::1
                    temp
               else (RATOM filename ReadTable])
```

7

```
(ReadAtom
  [LAMBDA (FileNameStopAtom)                          (* R.Erickson "24-Sep-81 15:42")
                                                      (* R.Bates "19-SEP-78 01:07")
    (PROG (atomgot upatom filename)
          (filename←FileNameStopAtom:1)
      TOP (if PARSERPROMPT and filename=T
              then (PROMPTCHAR PARSERPROMPT))
          (atomgot←(ParserRATOM filename FileNameStopAtom PASCAL\READ\TABLE))
          (if atomgot='%"
              then (bind (atomstring) first atomstring←"" eachtime atomgot←(ParserRATOM filename
                                                                                FileNameStopAtom
                                                                                PASCAL\READ\TABLE)
                         while atomgot~='%" do atomstring←(CONCAT atomstring " " atomgot)
                         finally atomgot←(MKATOM atomstring))
            elseif atomgot='{
              then (while atomgot~='} do atomgot←(ParserRATOM filename FileNameStopAtom
                                                                PASCAL\READ\TABLE))
                   (GO TOP)
            elseif upatom←(GETPROP atomgot 'UCASE)
              then atomgot←upatom
            elseif upatom←(UCaseToList atomgot UCaseParseAtoms)
              then atomgot←upatom)
          (if COLLECTTOKENS
              then (TCONC PARSEDTOKENS atomgot))
          (if PARSERTRACE
              then (PRIN1 atomgot)
                   (if ~(MEMB atomgot '(: = < > %( %)))
                       then (PRIN1 " "))
                   (if (MEMB atomgot '(; AND &))
                       then (TERPRI)))
          (RETURN (if (MEMB atomgot FileNameStopAtom::1)
                      then FileNameStopAtom
                    else atomgot])
```

8

```
(SIFromStack?
  [LAMBDA NIL OutputStream::1])
```

9

```
(SIMark
  [LAMBDA NIL SIEntries← <OutputStream ! SIEntries>])
```

10

```
(SINewStack
  [LAMBDA NIL
    (PROGN SIEntries←NIL
           OutputStream← <NIL>])
```

11

```
(SINext
  [LAMBDA NIL
    (if (SIFromStack?)
        then (SINextSaved)
      else (SINextNew])
```

12

```
(SINextNew
  [LAMBDA NIL
    (PROGN (if CurrentLexeme~=Terminator
               then CurrentLexeme←(UsersNextInput InputFileHandle))
           (if CurrentLexeme=InputFileHandle
               then CurrentLexeme←Terminator)
           (SISaveLexeme])
```

13

```
(SINextSaved
  [LAMBDA NIL
    (PROGN OutputStream←OutputStream::1
           CurrentLexeme←OutputStream:1])
```

14

```
(SISaveLexeme
  [LAMBDA NIL
    (PROGN OutputStream::1← <CurrentLexeme> OutputStream←OutputStream::1])
```

15

```
(SIToMark
  [LAMBDA NIL
    (PROGN OutputStream←SIEntries:1
           CurrentLexeme←OutputStream:1])
```

16

```
(SIUnmark
  [LAMBDA NIL SIEntries←SIEntries::1])
```

17

```
(UsersNextInput
  [LAMBDA (FileNameStopAtom)
    (ReadAtom FileNameStopAtom])
```

18

```
(identifierFilter
  [LAMBDA (x)                                          (* R.Bates "17-Sep-79 13:51")
    x←x:2
    (if (NUMBERP x) or (MEMB x ReserveWordList) or (MEMB x SpecialPrefixOps)
        or (MEMB x Delimiters) or (MEMB x '(< = > :))
      then NIL
    else x])
```

19

```
(infixOpFilter
  [LAMBDA (x)
    (if (MEMB x:LEXEME KeyWordList) or (MEMB x:LEXEME Delimiters)
      then NIL
    else x:LEXEME])
```

20

```
(labelFilter
  [LAMBDA (X)
    (OR (unsignedInteger X)
        (identifierFilter X])
```

21

```
(prefixOpFilter
  [LAMBDA (x)                                          (* R.Bates "17-Sep-79 13:46")
    (if (MEMB x:2 KeyWordList) or (MEMB x:2 Delimiters) or (MEMB x:LEXEME '(< = : >))
      then NIL
    else x:2])
```

22

```
(simplifyStatement
  [LAMBDA (X)
    (PROG (value)
          (if value←X:assignmentStatement
            then (RETURN value)
```

```
        elseif value←X:labelStatement
          then value:simpleStatement←value:simpleStatement:ALTERNATIVESUBNODE
               (RETURN value)
        elseif value←X:simpleStatement
          then (RETURN value:ALTERNATIVESUBNODE)
        else (RETURN X])
```

23

```
(specialPrefixOpFilter
  [LAMBDA (x)
    (if (MEMB x:LEXEME SpecialPrefixOps)
        then x:LEXEME
      else NIL])
```

24

```
(unsignedInteger
  [LAMBDA (X)
    (NUMBERP X:LEXEME])
```

25

```
(unsupportedPascalFilter
  [LAMBDA (x)                                        (* R.Erickson "24-Aug-81 18:53")

          (* * x is a production record. We don't support that subset of Pascal, and tell the user so.)


    (AffirmError <"Affirm's subset of Pascal does not include the production" x:1
                 "For help in using the proper notation, send a message to Affirm@Isif."
                 >)
    NIL])
)
(CLISPDEC (QUOTE FAST))
(DECLARE: DOEVAL@COMPILE

(PUTPROPS CopyTopRecord MACRO ((x)
                               (APPEND x)))

(PUTPROPS SIFromStack? MACRO (NIL (CDR OutputStream)))

(PUTPROPS SIMark MACRO (NIL (SETQ SIEntries (CONS OutputStream SIEntries))))

(PUTPROPS SINewStack MACRO [NIL (PROGN (SETQ SIEntries NIL)
                                       (SETQ OutputStream (LIST NIL])

(PUTPROPS SINext MACRO [NIL (COND
                             ((SIFromStack?)
                               (SINextSaved))
                             (T (SINextNew])

(PUTPROPS SINextNew MACRO (NIL (PROGN [COND
                                       ((NEQ CurrentLexeme Terminator)
                                         (SETQ CurrentLexeme (UsersNextInput InputFileHandle]
                                      (COND
                                        ((EQ CurrentLexeme InputFileHandle)
                                          (SETQ CurrentLexeme Terminator)))
                                      (SISaveLexeme))))

(PUTPROPS SINextSaved MACRO [NIL (PROGN (SETQ OutputStream (CDR OutputStream))
                                        (SETQ CurrentLexeme (CAR OutputStream])

(PUTPROPS SISaveLexeme MACRO [NIL (PROGN (RPLACD OutputStream (LIST CurrentLexeme))
                                         (SETQ OutputStream (CDR OutputStream])

(PUTPROPS SIToMark MACRO [NIL (PROGN (SETQ OutputStream (CAR SIEntries))
                                     (SETQ CurrentLexeme (CAR OutputStream])

(PUTPROPS SIUnmark MACRO (NIL (SETQ SIEntries (CDR SIEntries))))

(PUTPROPS UsersNextInput MACRO ((FileNameStopAtom)
                                (ReadAtom FileNameStopAtom)))

(PUTPROPS labelFilter MACRO [LAMBDA (X)
                             (OR (unsignedInteger X)
                                 (identifierFilter X])

)
```

```
(RPAQQ Delimiters (%( %) , %. : ; @ %[ %] ↑ %|))

(RPAQQ KeyWordList (ALL ALTERS ARRAY ASSERT ASSERTING ASSUME BEGIN BY CASE CONST DO DOWNTO ELSE END
                        ENTRY EXISTS EXIT FILE FOR FORALL FUNCTION GO GOTO IF IMPORTS INLINE LABEL
                        MAINTAIN OF OTHERWISE PACKED POST PRE PROCEDURE PROGRAM PROVE PUBLIC RECORD
                        REPEAT RETURN RETURNS SET SOME THEN THUS TO TYPE UNTIL VAR WHILE WITH XPUBLIC)
)

(RPAQQ ReserveWordList (= ALL ALTERS AND ARRAY ASSERT ASSERTING ASSUME BEGIN BY CASE CONST DIFFERENCE
                          DIV DO DOWNTO ELSE END ENTRY EQ EQV EXISTS EXIT EXPT FILE FIRST FOR FORALL
                          FUNCTION GE GO GOTO GT IF IMP IMPORTS INLINE LABEL LAST LE LT MAINTAIN MAX
                          MIN MOD NE NOT OF OR OTHERWISE PACKED PLUS POST PRE PROCEDURE PROGRAM PROVE
                          PUBLIC RECORD REPEAT RETURN RETURNS SET SOME THEN THUS TIMES TO TYPE UNTIL
                          VAR WHILE WITH XPUBLIC))

(RPAQQ SpecialPrefixOps (! + - ~ NOT))

(RPAQQ UpperCaseVars (TRUE FALSE))

(RPAQQ PARSERPROMPT ~>)

(RPAQ USESLOWERCASE T)

(RPAQ PARSERTRACE NIL)

(RPAQ COLLECTTOKENS T)

(RPAQQ UCaseParseAtoms (TRUE FALSE))

(RPAQQ LOWERCASE (all alters and array assert asserting assume begin by case const difference div do
                      downto else end entry eq eqv exists exit expt false file first for forall
                      function ge go goto gt if imp imports inline label last le lt maintain mod ne
                      not of or otherwise packed plus post pre procedure program prove public record
                      repeat return returns set some then thus times to true type until var while
                      with xpublic))

(PUTPROPS all UCASE ALL)

(PUTPROPS alters UCASE ALTERS)

(PUTPROPS and UCASE AND)

(PUTPROPS array UCASE ARRAY)

(PUTPROPS assert UCASE ASSERT)

(PUTPROPS asserting UCASE ASSERTING)

(PUTPROPS assume UCASE ASSUME)

(PUTPROPS begin UCASE BEGIN)

(PUTPROPS by UCASE BY)

(PUTPROPS case UCASE CASE)

(PUTPROPS const UCASE CONST)

(PUTPROPS difference UCASE DIFFERENCE)

(PUTPROPS div UCASE DIV)

(PUTPROPS do UCASE DO)

(PUTPROPS downto UCASE DOWNTO)

(PUTPROPS else UCASE ELSE)

(PUTPROPS end UCASE END)

(PUTPROPS entry UCASE ENTRY)

(PUTPROPS eq UCASE EQ)

(PUTPROPS eqv UCASE EQV)

(PUTPROPS exists UCASE EXISTS)

(PUTPROPS exit UCASE EXIT)
```

```
(PUTPROPS expt UCASE EXPT)
(PUTPROPS false UCASE FALSE)
(PUTPROPS file UCASE FILE)
(PUTPROPS first UCASE FIRST)
(PUTPROPS for UCASE FOR)
(PUTPROPS forall UCASE FORALL)
(PUTPROPS function UCASE FUNCTION)
(PUTPROPS ge UCASE GE)
(PUTPROPS go UCASE GO)
(PUTPROPS goto UCASE GOTO)
(PUTPROPS gt UCASE GT)
(PUTPROPS if UCASE IF)
(PUTPROPS imp UCASE IMP)
(PUTPROPS imports UCASE IMPORTS)
(PUTPROPS inline UCASE INLINE)
(PUTPROPS label UCASE LABEL)
(PUTPROPS last UCASE LAST)
(PUTPROPS le UCASE LE)
(PUTPROPS lt UCASE LT)
(PUTPROPS maintain UCASE MAINTAIN)
(PUTPROPS mod UCASE MOD)
(PUTPROPS ne UCASE NE)
(PUTPROPS not UCASE NOT)
(PUTPROPS of UCASE OF)
(PUTPROPS or UCASE OR)
(PUTPROPS otherwise UCASE OTHERWISE)
(PUTPROPS packed UCASE PACKED)
(PUTPROPS plus UCASE PLUS)
(PUTPROPS post UCASE POST)
(PUTPROPS pre UCASE PRE)
(PUTPROPS procedure UCASE PROCEDURE)
(PUTPROPS program UCASE PROGRAM)
(PUTPROPS prove UCASE PROVE)
(PUTPROPS public UCASE PUBLIC)
(PUTPROPS record UCASE RECORD)
(PUTPROPS repeat UCASE REPEAT)
(PUTPROPS return UCASE RETURN)
(PUTPROPS returns UCASE RETURNS)
(PUTPROPS set UCASE SET)
(PUTPROPS some UCASE SOME)
(PUTPROPS then UCASE THEN)
```

```
(PUTPROPS thus UCASE THUS)

(PUTPROPS times UCASE TIMES)

(PUTPROPS to UCASE TO)

(PUTPROPS true UCASE TRUE)

(PUTPROPS type UCASE TYPE)

(PUTPROPS until UCASE UNTIL)

(PUTPROPS var UCASE VAR)

(PUTPROPS while UCASE WHILE)

(PUTPROPS with UCASE WITH)

(PUTPROPS xpublic UCASE XPUBLIC)
(SETQ PASCAL\READ\TABLE (COPYREADTABLE (QUOTE ORIG)))
(SETBRK (QUOTE (4 5 6 14 16 17 18 19 20 21 27 28 29 34 30 33 38 40 41 42 43 44 45 46 47 58 59 60 61
                  62 64 91 93 94 123 124 125 126))
        NIL PASCAL\READ\TABLE)
(DECLARE: DONTCOPY
  (FILEMAP (NIL (2263 8243 (CopyTopRecord 2275 . 2322) (MatchConstant 2326 . 2545) (MatchLexeme 2549 .
2696) (PARSEPROGRAM 2700 . 3135) (PARSE\ASSERTION 3139 . 3364) (ParserRATOM 3368 . 3757) (ReadAtom
3761 . 5326) (SIFromStack? 5330 . 5380) (SIMark 5384 . 5450) (SINewStack 5454 . 5537) (SINext 5541 .
5661) (SINextNew 5665 . 5922) (SINextSaved 5926 . 6034) (SISaveLexeme 6038 . 6146) (SIToMark 6150 .
6251) (SIUnmark 6255 . 6308) (UsersNextInput 6312 . 6399) (identifierFilter 6403 . 6713) (
infixOpFilter 6717 . 6867) (labelFilter 6871 . 6963) (prefixOpFilter 6967 . 7217) (simplifyStatement
7221 . 7616) (specialPrefixOpFilter 7620 . 7749) (unsignedInteger 7753 . 7813) (
unsupportedPascalFilter 7817 . 8240)))))
STOP
```

```
program := ( procedureOrFunctionDeclaration | block ) { '; } { '. }  ;

all := 'ALL;
bracketExprList := '[ { expression ↑ ', } '] ;
coord := number | '- number# | all | lastOne | firstOne ;
denoteSpec := denotePair ↑ ', ;
denotePair := expression 'BY identifier ;
expression :=  primary { infixOp expression } ;
expressionSeq := expression ↑ ',  ;
firstOne := 'FIRST;
functionDecl := expression ↑ ', ': expression#  ;
greaterThanEqual := '> '= ;
identifier := LEXEME |> identifierFilter  ;
identifierSeq := identifier ↑ ', ;
ifExpr := 'IF expression 'THEN expression# { 'ELSE expression## } ;
infixOp :=  notEqual | lessThanEqual | greaterThanEqual | normalInfixOp || ;
lastOne := 'LAST;
lessThanEqual := '< '= ;
machinePair := { identifierSeq } { 'BY identifierSeq# } ;
machineSpec := machinePair ↑ ', ;
normalInfixOp := LEXEME |> infixOpFilter ;
notEqual := '~ '= | '! '= ;
number := LEXEME |> unsignedInteger ;
op := expression | infixOp | specialPrefixOp | prefixOp || ;
parenExpr := '( expression ') ;
prefixExpr := prefixOp '( { expression ↑ ', } ')
        { 'IMPORTS '( identifier ↑ '; ') }  ;
prefixOp := LEXEME |> prefixOpFilter ;
primary := prefixExpr | variable | number | specialPrefixExpr
                | parenExpr | bracketExprList
                | ifExpr | quantifiedExpression || ;
quantifiedExpression := quantifier identifier ↑ ', '( expression ') ;
quantifier := 'ALL | 'FORALL | 'SOME | 'EXISTS ;
range := coord { ': coord# } | '@ ;
rangedInterfaceList := rangedOp ↑ ', ;
rangedOp := { op } { rangeSpec } ;
rangeSpec := '| range ↑ ', '| ;
rule := expression { ( '= '= | '< '= '> | '< '- '> | ': ': ) expression# } ;
ruleSeq := rule ↑ ', ;
specialPrefixExpr := specialPrefixOp primary ;
specialPrefixOp := LEXEME |> specialPrefixOpFilter ;
variable := identifier ;
variableDecl := identifier ↑ ', ': expression ;



interfaceList := op ↑ ', ;
arrayType := 'ARRAY '[ simpleType ↑ ', '] 'OF type ;
assertion := expression ;
assertStatement := 'ASSERT assertion ;
assignmentStatement := variable ': '= expression ;
assumeStatement := 'ASSUME assertion ;
block := { ( 'ENTRY | 'PRE ) assertion '; }
        { ( 'EXIT | 'POST ) assertion# '; }
        { declareopt ↑ '; '; }
        { compoundStatement } ;
caseElementList := caseLabel ↑ ', ': { statement } ;
caseLabel := constant ;
```

```
caseStatement := 'CASE expression 'OF caseElementList ↑ ';
        { '; ( 'ELSE | 'OTHERWISE ) statement } { '; } 'END ;
compoundStatement := 'BEGIN statement ↑ '; 'END ;
concurrentAssignmentStatement := variable ↑ ',  ': '= expression ↑ ', ;
constant := LEXEME ;
constDefinition := identifier '= expression ;
declareopt := { 'XPUBLIC | 'PUBLIC } declareType ;
declareType :=   'LABEL label ↑ ',  |
        'CONST constDefinition ↑ '; |
        'TYPE typeDefinition ↑ '; |
        'VAR  varDeclaration ↑ '; |
        procedureOrFunctionDeclaration  ;
direction := 'TO | 'DOWNTO ;
fieldList := { recordSection ↑ '; } { '; variantPart } { '; } ;
fileType := 'FILE 'OF type ;
formalParameterSection := { parameterKind } parameterGroup ;
forStatement := { 'MAINTAIN assertion } 'FOR identifier ': '= expression
        direction expression# 'DO statement { 'THUS assertion# } ;
goToStatement := ( 'GOTO | 'GO 'TO )  label  { 'ASSERTING assertion };
ifStatement := 'IF expression 'THEN statement { 'ELSE statement# } ;
label := LEXEME |> labelFilter ;
labelStatement := label ': { simpleStatement } ;
packed := 'PACKED ;
parameterGroup := identifier ↑ ', { ': type } ;
parameterKind := 'VAR | 'FUNCTION | 'PROCEDURE ;
pointerType := '↑ identifier ;
procedureOrFunctionDeclaration := { 'INLINE } unitKind identifier
        { '( formalParameterSection ↑ '; ') } { 'RETURNS } { identifier# }
        { ': type } { 'IMPORTS '( formalParameterSection# ↑ '; ') }
        { 'ALTERS  identifier## ↑ ', } '; block ;
procedureStatement := identifier { '( expression ↑ ', ') }
         { 'IMPORTS '( identifier# ↑ '; ') }  { 'ALTERS variable ↑ ', } ;
proveStatement := 'PROVE assertion ;
qualifier := '[ expression ↑ ', '] | '. identifier | '↑ ;
recordSection := identifier ↑ ', ': type ;
recordType := 'RECORD fieldList 'END ;
repeatStatement := 'REPEAT statement ↑ '; 'UNTIL expression { 'THUS assertion };
returnStatement := 'RETURN { '( expression ') }
        { 'ASSERTING assertion } ;
scalarType := '( identifier ↑ ', ') ;
setType := 'SET 'OF simpleType ;
simpleStatement := compoundStatement | ifStatement | caseStatement
        | whileStatement  | repeatStatement | forStatement | withStatement
        | goToStatement  | assertStatement | returnStatement | proveStatement
        | assumeStatement | assignmentStatement | concurrentAssignmentStatement
        | procedureStatement || ;
simpleType := scalarType | subrangeType | typeIdentifier || ;
statement := ( assignmentStatement | labelStatement | simpleStatement |
          EMPTY ) |> simplifyStatement ;
structuredType := { packed } unpackedStructuredType ;
subrangeType := ( '* | expression ) '. '. ( '* | expression# ) ;
type := simpleType | structuredType | pointerType || ;
typeDefinition := identifier '= type ;
typeIdentifier := identifier ;
unitKind := 'PROCEDURE | 'FUNCTION | 'PROGRAM ;
unpackedStructuredType := arrayType | recordType | setType | fileType || ;
varDeclaration := varDeclarePart ↑ ', ': type ;
varDeclarePart :=  identifier { '@ expression } { ': '= expression# } ;
variant := { caseLabel ↑ ', ': '( fieldList ') } ;
```

```
variantPart := 'CASE { identifier ': } typeIdentifier 'OF variant ↑ '; ;
whileStatement :=  { 'MAINTAIN assertion } 'WHILE expression 'DO statement
        { 'THUS assertion# } ;
withStatement := 'WITH variable ↑ ', 'DO statement  ;
STOP
```