

## Appendix A: Grammar in EBNF

### A.1 Non-Terminal Symbols

#### Compilation Units

##### #1 Compilation Unit

```
compilationUnit :  
    IMPLEMENTATION? programModule | definitionOfModule | prototype ;
```

##### #2 Program Module

```
programModule :  
    MODULE moduleIdent ";"  
    importList* block moduleIdent "." ;
```

##### #2.1 Module Identifier

```
moduleIdent : Ident ;
```

##### #3 Definition Of Module

```
definitionOfModule :  
    DEFINITION MODULE moduleIdent ( "[" requiredConformance "]" )? ";"  
    importList* definition*  
    END moduleIdent "." ;
```

##### #4 Prototype

```
prototype :  
    PROTOTYPE prototypeIdent "[" requiredConformance "]" ";"  
    ( PLACEHOLDERS identList ";" )?  
    requiredTypeDefinition  
    ( ";" requiredBinding )*  
    END prototypeIdent "." ;
```

##### #4.1 Prototype Identifier

```
prototypeIdent : Ident ;
```

##### #4.2 RequiredConformance

```
requiredConformance : prototypeIdent ;
```

##### #4.3 RequiredBinding

```
requiredBinding : procedureHeader ;
```

#### Prototype Definitions

##### #5 Required Type Definition

```
requiredTypeDefinition :  
    TYPE "=" permittedTypeDefinition ( "|" permittedTypeDefinition )*  
    ( "!=" protoliteral ( "|" protoliteral )* )?
```

##### #6 Permitted Type Definition

```
permittedTypeDef :  
    RECORD | OPAQUE RECORD?
```

##### #7 Proto-Literal

```
protoliteral :  
    simpleProtoliteral | structuredProtoliteral ;
```

##### #7.1 Simple Proto-Literal

```
simpleProtoliteral1 : Ident;
```

---

<sup>1</sup> Simple protoliterals are CHAR, INTEGER and REAL, representing any quoted literals, whole numbers and real numbers.

**#8 Structured Proto-Literal**

```
structuredProtoliteral :
    "{" ( VARIADIC OF simpleProtoliteral ( "," simpleProtoliteral )* |
    structuredProtoliteral ( "," structuredProtoliteral )* ) "}" ;
```

**Import Lists, Blocks, Declarations and Definitions****#9 Import List**

```
importList :
    ( FROM moduleIdent IMPORT ( identList | "*" ) |
    IMPORT moduleIdent "+"? ( "," moduleIdent "+"? )* ) ";" ;
```

**#10 Block**

```
block :
    declaration*
    ( BEGIN statementSequence )? END ;
```

**#11 Declaration**

```
declaration :
    CONST ( constantDeclaration ";" )+ |
    TYPE ( Ident "=" type ";" )+ |
    VAR ( variableDeclaration ";" )+ |
    procedureDeclaration ";" ;
```

**#12 Definition**

```
definition :
    CONST ( ( "[" constBindableIdent "]" )? constantDeclaration ";" )+ |
    TYPE ( Ident "=" ( type | OPAQUE recordType? ) ";" )+ |
    VAR ( variableDeclaration ";" )+ |
    procedureHeader ";" ;
```

**#12.1 CONST Bindable Identifier**

```
constBindableIdent2 : Ident ;
```

**Constant Declarations****#13 Constant Declaration**

```
constantDeclaration :
    Ident "=" constExpression3 ;
```

**#13.1 Constant Expression**

```
constExpression : expression ;
```

**Type Declarations****#14 Type**

```
type :
    ( ( ALIAS | range ) OF )? typeIdent | enumerationType |
    arrayType | recordType | setType | pointerType | procedureType ;
```

**#14.1 Type Identifier**

```
typeIdent : qualident ;
```

**#15 Range**

```
range :
    "[" constExpression ".." constExpression "]" ;
```

---

<sup>2</sup> CONST bindable identifiers are TSIG and TEXP.

<sup>3</sup> Constants may not be declared as aliases of type identifiers.

**#16 Enumeration Type**

```
enumerationType :
    "(" ( ( "+" enumTypeId ) | Ident )
      ( "," ( ( "+" enumTypeId ) | Ident ) )* ")" ;
```

**#16.1 Enumeration Type Identifier**

```
enumTypeId : typeId ;
```

**#17 Array Type**

```
arrayType :
    ( ARRAY componentCount ( "," componentCount )* |
      ASSOCIATIVE ARRAY ) OF typeId ;
```

**#17.1 Component Count**

```
componentCount : constExpression ;
```

**#18 Record Type**

```
recordType :
    RECORD ( fieldList ( ";" fieldList )* indeterminateField |
      "(" baseType ")" fieldList ( ";" fieldList )* ) END ;
```

**#18.1 Field List**

```
fieldList : variableDeclaration ;
```

**#18.2 Base Type**

```
baseType : typeId ;
```

**#19 Indeterminate Field**

```
indeterminateField :
    INDETERMINATE Ident ":" ARRAY discriminantField OF typeId ;
```

**#19.1 Discriminant Field**

```
discriminantField : Ident ;
```

**#20 Set Type**

```
setType :
    SET OF ( enumTypeId | "(" identList ")" ) ;
```

**#21 Pointer Type**

```
pointerType :
    POINTER TO CONST? typeId ;
```

**#22 Procedure Type**

```
procedureType :
    PROCEDURE
    ( "(" formalTypeList ")" )?
    ( ":" returnType )? ;
```

**#22.1 Returned Type**

```
returnedType : typeId ;
```

**#23 Formal Type List**

```
formalTypeList :
    formalType ( "," formalType )* ;
```

**#24 Formal Type**

```
formalType :
    attributedFormalType | variadicFormalType ;
```

**#25 Attributed Formal Type**

```
attributedFormalType :
    ( CONST | VAR )? simpleFormalType ;
```

**#26 Simple Formal Type**

```
simpleFormalType :
  ( CAST? ARRAY OF )? namedType ;
```

**#27 Variadic Formal Type**

```
variadicFormalType :
  VARIADIC OF
  ( attributedFormalType |
    "{" attributedFormalType ( "," attributedFormalType )* "}" ) ;
```

**Variable Declarations****#28 Variable Declaration**

```
variableDeclaration :
  identList ":" ( range OF )? typeIdent ;
```

**Procedure Declarations****#29 Procedure Declaration**

```
procedureDeclaration :
  procedureHeader ";" block Ident ;
```

**#30 Procedure Header**

```
procedureHeader :
  PROCEDURE
  ( "[" bindableEntity "]" )?
  Ident ( "(" formalParamList ")" )? ( ":" returnedType )? ;
```

**#31 Bindable Entity**

```
bindableEntity :
  DIV | MOD | FOR | DESCENDING |
  ":" | "!=" | "?" | "!" | "~" | "+" | "-" | "*" | "/" | "=" | "<" | ">" |
  bindableIdent;
```

**#31.1 Bindable Identifier**

```
bindableIdent4 : Ident;
```

**#32 Formal Parameter List**

```
formalParamList :
  formalParams ( ";" formalParams )* ;
```

**#33 Formal Parameters**

```
formalParams :
  simpleFormalParams | variadicFormalParams ;
```

**#34 Simple Formal Parameters**

```
simpleFormalParams :
  ( CONST | VAR )? identList ":" simpleFormalType ;
```

**#35 Variadic Formal Parameters**

```
variadicFormalParams :
  VARIADIC ( variadicCounter "[" variadicTerminator "]" )? OF
  ( simpleFormalType |
    "{" simpleFormalParams ( ";" simpleFormalParams )* "}" ) ;
variadicCounter : Ident ;
```

<sup>4</sup> PROCEDURE bindable identifiers are ABS, NEG, ODD, COUNT, LENGTH, NEW, DISPOSE, RETAIN, RELEASE, TLIMIT, TMIN, TMAX, SXF and VAL.

**#35.1 Variadic Terminator**

```
variadicTerminator : constExpression ;
```

**Statements****#36 Statement**

```
statement :
  ( assignmentOrProcedureCall | ifStatement | caseStatement |
    whileStatement | repeatStatement | loopStatement |
    forStatement | RETURN expression? | EXIT )? ;
```

**#37 Statement Sequence**

```
statementSequence :
  statement ( ";" statement )* ;
```

**#38 Assignment Or Procedure Call**

```
assignmentOrProcedureCall :
  designator ( "!=" expression | "++" | "--" | actualParameters )? ;
```

**#39 IF Statement**

```
ifStatement :
  IF expression THEN statementSequence
  ( ELSIF expression THEN statementSequence )*
  ( ELSE statementSequence )?
  END ;
```

**#40 CASE Statement**

```
caseStatement :
  CASE expression OF case ( "|" case )+ ( ELSE statementSequence )? END ;
```

**#41 Case**

```
case :
  caseLabels ( "," caseLabels )* ":" statementSequence ;
```

**#42 Case Labels**

```
caseLabels :
  constExpression ( ".." constExpression )? ;
```

**#43 WHILE Statement**

```
whileStatement :
  WHILE expression DO statementSequence END ;
```

**#44 REPEAT Statement**

```
repeatStatement :
  REPEAT statementSequence UNTIL expression ;
```

**#45 LOOP Statement**

```
loopStatement :
  LOOP statementSequence END ;
```

**#46 FOR Statement**

```
forStatement :
  FOR DESCENDING? controlVariable
  IN ( designator | range OF typeIdent )
  DO statementSequence END ;
```

**#46.1 Control Variable**

```
controlVariable : Ident ;
```

**#47 Designator**

```
designator :
    qualident designatorTail? ;
```

**#48 Designator Tail**

```
designatorTail :
    ( ( "[" expressionList "]" | "^" ) ( "." Ident )* )+ ;
```

**Expressions****#49 Expression List**

```
expressionList :
    expression ( "," expression )* ;
```

**#50 Expression**

```
expression :
    simpleExpression ( relOp simpleExpression )? ;
```

**#50.1 Relational Operator**

```
relOp :
    "=" | "#" | "<" | "<=" | ">" | ">=" | IN ;
```

**#51 Simple Expression**

```
simpleExpression :
    ( "+" | "-" )? term ( addOp term )* ;
```

**#51.1 Add Operator**

```
addOp :
    "+" | "-" | OR ;
```

**#52 Term**

```
term :
    factor ( mulOp factor )* ;
```

**#52.1 Multiply Operator**

```
mulOp :
    "*" | "/" | DIV | MOD | AND ;
```

**#53 Factor**

```
factor :
    ( NumericLiteral | StringLiteral | structuredValue |
      designatorOrFunctionCall | "(" expression ")" )
    ( "::" namedType )? | NOT factor ;
```

**#54 Designator Or Function Call**

```
designatorOrFunctionCall :
    designator actualParameters? ;
```

**#55 Actual Parameters**

```
actualParameters :
    "(" expressionList? ")" ;
```

## Value Constructors

### #56 Structured Value

```
structuredValue :  
    "{" ( valueComponent ( "," valueComponent )* )? "}" ;
```

### #57 Value Component

```
valueComponent :  
    expression ( ( BY | ".." ) constExpression )? ;
```

## Identifiers

### #58 Qualified Identifier

```
qualident :  
    Ident ( "." Ident )* ;
```

### #59 Identifier List

```
identList :  
    Ident ( "," Ident )* ;
```

## A.2 Terminal Symbols

### #1 Reserved Words

ReservedWord :  
 ALIAS AND ARRAY ASSOCIATIVE BEGIN BY CASE CONST DEFINITION DESCENDING  
 DIV DO ELSE ELSIF END EXIT FOR FROM IF IMPLEMENTATION IMPORT IN  
 INDETERMINATE LOOP MOD MODULE NOT OF OPAQUE OR PLACEHOLDERS POINTER  
 PROCEDURE PROTOTYPE RECORD REPEAT RETURN SET THEN TO TYPE UNTIL VAR  
 VARIADIC WHILE ;

### #2 Identifier

Ident :  
 IdentLeadChar IdentTailChar\* ;

#### #2.1 Identifier Leading Character

IdentLeadChar :  
 " \_ " | "\$" | Letter ;

#### #2.2 Identifier Tail Character

IdentTailChar :  
 IdentLeadChar | Digit ;

### #3 Numeric Literal

NumericLiteral :  
 "0"  
 ( DecimalNumberTail |  
 "b" Base2DigitSeq |  
 "x" Base16DigitSeq |  
 "u" Base16DigitSeq )?  
 | "1" .. "9" DecimalNumberTail? ;

#### #3.1 Decimal Number Tail

DecimalNumberTail :  
 DigitSep? DigitSeq  
 ( "." DigitSeq ( "e" ( "+" | "-" )? DigitSeq )? )? ;

#### #3.2 Digit Sequence

DigitSeq :  
 Digit+ ( DigitSep Digit+ )\* ;

#### #3.3 Base-2 Digit Sequence

Base2DigitSeq :  
 Base2Digit+ ( DigitSep Base2Digit+ )\* ;

#### #3.4 Base-16 Digit Sequence

Base16DigitSeq :  
 Base16Digit+ ( DigitSep Base16Digit+ )\* ;

#### #3.5 Digit Separator

DigitSep : " " ;

### #4 String Literal

StringLiteral :  
 SingleQuotedString | DoubleQuotedString ;

#### #4.1 Single Quoted String

''' ( QuotableCharacter | ''' )\* ''' ;

#### #4.2 Double Quoted String

"" ( QuotableCharacter | "" )\* "" ;



**#4.3 Quotable Character**

QuotableCharacter :

Digit | Letter | Space | QuotableGraphicChar | EscapedCharacter ;

**#4.4 Digit**

Digit :

Base2Digit | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

**#4.5 Base-2 Digit**

Base2Digit :

"0" | "1" ;

**#4.6 Base-16 Digit**

Base16Digit :

Digit | "A" | "B" | "C" | "D" | "E" | "F" ;

**#4.7 Letter**

Letter :

"A" .. "Z" | "a" .. "z" ;

**#4.8 Space**

Space : " " ;

**#4.9 Quotable Graphic Character**

QuotableGraphicChar :

"!"	"#"	"\$"	"%"	"&"	"("	")"	"*"	"+"	","	
"_"	"."	"/"	":"	";"	"<"	"="	">"	"?"	"@"	
"["	"]"	"^"	"_"	"`"	"{"	" "	"}"	"~"	;	

**#4.10 Escaped Character**

EscapedCharacter :

"\" ( "0" | "n" | "t" | "\" | "'" | " " ) ;

**A.3 Ignore Symbols****#1 Whitespace**

Whitespace :

Space | ASCII\_TAB ;

**#2 Single-Line Comment**

SingleLineComment :

"//" ~( EndOfLine ) \* EndOfLine ;

**#3 Multi-Line Comment**

MultiLineComment :

"(\*" ( ~( "(" | ")" ) \* ( MultiLineComment | EndOfLine ) ? ) \* ")" ;

**#4 End Of Line Marker**

EndOfLine :

ASCII\_LF ASCII\_CR? | ASCII\_CR ASCII\_LF? ;

**A.4 Control Codes****#1 Horizontal Tab**

ASCII\_TAB : CHR(8) ;

**#2 Line Feed**

ASCII\_LF : CHR(10) ;

**#3 Carriage Return**

ASCII\_CR : CHR(13) ;

**#4 UTF8 BOM**UTF8\_BOM<sup>5</sup> : { 0xFE, 0xBB, 0xBF } ;

---

<sup>5</sup> BOM support is optional. If supported, a BOM may only occur at the very beginning of a file.



## A.5 Pragma Grammar

### #1 Pragma

```
pragma :
    "<*" ( pragmaMSG | pragmaIF | pragmaENCODING | pragmaGENLIB | pragmaFFI |
    pragmaINLINE | pragmaALIGN | pragmaPADBITS | pragmaADDR | pragmaREG |
    pragmaPURITY | varAttrPragma | pragmaFORWARD | implDefinedPragma ) ">" ;
```

### #2 Body Of Compile Time Message Pragma

```
pragmaMSG :
    MSG "=" ( INFO | WARN | ERROR | FATAL ) ":"
    compileTimeMsgComponent ( "," compileTimeMsgComponent )* ;
```

### #3 Compile Time Message Component

```
compileTimeMsgComponent :
    StringLiteral | ConstQualident |
    "?" ( ALIGN | ENCODING | implDefPragmaName ) ;
```

#### #3.1 Constant Qualified Identifier

```
constQualident : qualident ;
```

#### #3.2 Implementation Defined Pragma Name

```
implDefPragmaName : Ident ;
```

### #4 Body Of Conditional Compilation Pragma

```
pragmaIF :
    ( IF | ELSIF ) inPragmaExpression | ELSE | ENDIF ;
```

### #5 Body Of Character Encoding Pragma

```
pragmaENCODING :
    ENCODING "=" ( "ASCII" | "UTF8" ) ( ":" codePointSampleList )? ">" ;
```

### #6 Code Point Sample List

```
codePointSampleList :
    quotedChar "=" characterCode ( "," quotedChar "=" characterCode )* ;
```

#### #6.1 Quoted Character Literal

```
quotedChar : StringLiteral ;
```

#### #6.2 Character Code Literal

```
characterCode : NumericLiteral ;
```

### #7 Library Template Expansion Pragma

```
pragmaGENLIB :
    GENLIB moduleIdent FROM template ":" templateParamList ;
```

#### #7.1 Template Identifier

```
template : Ident ;
```

#### #8 Template Parameter List

```
templateParamList :
    placeholder "=" replacement ( "," placeholder "=" replacement )*
```

#### #8.1 Placeholder

```
placeholder : Ident ;
```

#### #8.2 Replacement

```
replacement : StringLiteral ;
```

### #9 Body Of Foreign Function Interface Pragma

```
pragmaFFI :
    FFI "=" ( "C" | "Fortran" ) ;
```

### #10 Body Of Function Inlining Pragma

```
pragmaINLINE :
    INLINE | NOINLINE ;
```

### #11 Body Of Memory Alignment Pragma

```
pragmaALIGN :
    ALIGN "=" inPragmaExpression ;
```

**#12 Body Of Bit Padding Pragma**

```
pragmaPADBITS :
    PADBITS "=" inPragmaExpression ;
```

**#13 Body Of Memory Mapping Pragma**

```
pragmaADDR :
    ADDR "=" inPragmaExpression ;
```

**#14 Body Of Register Mapping Pragma**

```
pragmaREG :
    REG "=" inPragmaExpression ;
```

**#15 Body Of Purity Attribute Pragma**

```
pragmaPURITY :
    PURITY "=" inPragmaExpression ;
```

**#16 Body Of Variable Attribute Pragma**

```
varAttrPragma :
    SINGLEASSIGN | VOLATILE ;
```

**#17 Body Of Forward Declaration Pragma**

```
pragmaFORWARD :
    FORWARD ( TYPE identList | procedureHeader ) ;
```

**#18 Body Of Implementation Defined Pragma**

```
implDefinedPragma :
    implDefPragmaName ( "=" inPragmaExpression )? ;
```

**#19 In-Pragma Expression**

```
inPragmaExpression :
    inPragmaSimpleExpr ( inPragmaRelOp inPragmaSimpleExpr )? ;
```

**#19.1 In-Pragma Relational Operator**

```
inPragmaRelOp :
    "=" | "#" | "<" | "<=" | ">" | ">=" ;
```

**#20 In-Pragma Simple Expression**

```
inPragmaSimpleExpr :
    ( "+" | "-" )? inPragmaTerm ( addOp inPragmaTerm )* ;
```

**#21 In-Pragma Term**

```
inPragmaTerm :
    inPragmaFactor ( inPragmaMulOp inPragmaFactor )* ;
```

**#21.1 In-Pragma Multiply Operator**

```
inPragmaMulOp :
    "*" | DIV | MOD | AND ;
```

**#22 In-Pragma Factor**

```
inPragmaFactor :
    wholeNumber | constQualident | "(" inPragmaExpression ")" |
    inPragmaCompileTimeFunctionCall | NOT inPragmaFactor ;
```

**#22.1 Whole Number**

```
wholeNumber : NumericLiteral ;
```

**#23 In-Pragma Compile-Time Function Call**

```
inPragmaCompileTimeFunctionCall :
    Ident1 "(" inPragmaExpression ( "," inPragmaExpression )* ")" ;
```

---

<sup>1</sup> Permissible are ABS, NEG, ODD, ORD, LENGTH, TMIN, TMAX, TSIZE, TLIMIT and macros in module COMPILER.