Appendix A: Grammar in EBNF

A.1 Non-Terminal Symbols

Compilation Units

#1 Compilation Unit
```
compilationUnit :
    prototype | definitionOfModule | IMPLEMENTATION? programModule ;
```

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

#2.1 Prototype Identifier
```
prototypeIdent : Ident ;
```

#2.2 RequiredConformance
```
requiredConformance : prototypeIdent ;
```

#3 Program Module
```
programModule :
    MODULE moduleIdent ";"
    importList* block moduleIdent "." ;
```

#3.1 Module Identifier
```
moduleIdent : Ident ;
```

#4 Definition Of Module
```
definitionOfModule :
    DEFINITION MODULE moduleIdent ( "[" requiredConformance "]" )? ";"
    importList* definition*
    END moduleIdent "." ;
```

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
```
simpleProtoliteral[1] : Ident;
```

[1] 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 )* ) "}" ;
```

#9 Required Binding

```
requiredBinding :
    CONST “[“ constBindableIdent “]” “:” pervasiveType |
    procedureHeader ;
```

#9.1 CONST Bindable Identifier

```
constBindableIdent[2] : Ident ;
```

#9.2 Pervasive Type

```
pervasiveType : Ident ;
```

Import Lists, Blocks, Declarations and Definitions

#10 Import List

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

#11 Block

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

#12 Declaration

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

#13 Definition

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

Constant Declarations

#14 Constant Declaration

```
constantDeclaration :
    Ident “=” constExpression[3] ;
```

#14.1 Constant Expression

```
constExpression : expression ;
```

---

[2] CONST bindable identifiers are TSIG and TEXP.

[3] Constants may not be declared as aliases of type identifiers.

Type Declarations

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

#15.1 Type Identifier
```
typeIdent : qualident ;
```

#16 Range
```
range :
    "[" constExpression ".." constExpression "]" ;
```

#17 Enumeration Type
```
enumerationType :
    "(" ( ( "+" enumTypeIdent ) | Ident )
          ( "," ( ( "+" enumTypeIdent ) | Ident ) )* ")" ;
```

#17.1 Enumeration Type Identifier
```
enumTypeIdent : typeIdent ;
```

#18 Array Type
```
arrayType :
    ( ARRAY componentCount ( "," componentCount )* |
      ASSOCIATIVE ARRAY ) OF typeIdent ;
```

#18.1 Component Count
```
componentCount : constExpression ;
```

#19 Record Type
```
recordType :
    RECORD ( fieldList ( ";" fieldList )* indeterminateField |
    "(" baseType ")" fieldList ( ";" fieldList )* ) END ;
```

#19.1 Field List
```
fieldList : variableDeclaration ;
```

#19.2 Base Type
```
baseType : typeIdent ;
```

#20 Indeterminate Field
```
indeterminateField :
    INDETERMINATE Ident ":" ARRAY discriminantField OF typeIdent ;
```

#20.1 Discriminant Field
```
discriminantField : Ident ;
```

#21 Set Type
```
setType :
    SET OF ( enumTypeIdent | "(" identList ")" ) ;
```

#22 Pointer Type
```
pointerType :
    POINTER TO CONST? typeIdent ;
```

#23 Procedure Type
```
procedureType :
    PROCEDURE
    ( "(" formalTypeList ")" )?
    ( ":" returnedType )? ;
```


#23.1 Returned Type
```
returnedType : typeIdent ;
```

#24 Formal Type List
```
formalTypeList :
    formalType ( “,” formalType )* ;
```

#25 Formal Type
```
formalType :
    attributedFormalType | variadicFormalType ;
```

#26 Attributed Formal Type
```
attributedFormalType :
    ( CONST | VAR )? simpleFormalType ;
```

#27 Simple Formal Type
```
simpleFormalType :
    ( CAST? ARRAY OF )? namedType ;
```

#28 Variadic Formal Type
```
variadicFormalType :
    VARIADIC OF
    ( attributedFormalType |
      “{+” attributedFormalType ( “,” attributedFormalType )* “}+”  ) ;
```

Variable Declarations

#29 Variable Declaration
```
variableDeclaration :
    identList “:” ( range OF )? typeIdent ;
```

Procedure Declarations

#30 Procedure Declaration
```
procedureDeclaration :
    procedureHeader “;” block Ident ;
```

#31 Procedure Header
```
procedureHeader :
    PROCEDURE
    ( “[” bindableEntity “]” )?
    Ident ( “(” formalParamList “)” )? ( “:” returnedType )? ;
```

#32 Bindable Entity
```
bindableEntity :
    DIV | MOD | FOR | DESCENDING |
    “::” | “:=” | “?” | “!” | “~” | “+” | “-” | “*” | “/” | “=” | “<” | “>” |
    bindableIdent;
```

#32.1 Bindable Identifier
```
bindableIdent[4] : Ident;
```

#33 Formal Parameter List
```
formalParamList :
    formalParams ( “;” formalParams )* ;
```

---

[4] PROCEDURE bindable identifiers are ABS, NEG, ODD, COUNT, LENGTH, NEW, DISPOSE, RETAIN, RELEASE, TLIMIT, TMIN, TMAX, SXF and VAL.

#34 Formal Parameters
```
formalParams :
    simpleFormalParams | variadicFormalParams ;
```

#35 Simple Formal Parameters
```
simpleFormalParams :
    ( CONST | VAR )? identList ":" simpleFormalType ;
```

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

#36.1 Variadic Formal Parameters
```
variadicTerminator : constExpression ;
```

Statements

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

#38 Statement Sequence
```
statementSequence :
    statement ( ";" statement )* ;
```

#39 Assignment Or Procedure Call
```
assignmentOrProcedureCall :
    designator ( ":=" expression | "++" | "--" | actualParameters )? ;
```

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

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

#42 Case
```
case :
    caseLabels ( "," caseLabels )* ":" statementSequence ;
```

#43 Case Labels
```
caseLabels :
    constExpression ( ".." constExpression )? ;
```

#44 WHILE Statement
```
whileStatement :
    WHILE expression DO statementSequence END ;
```

#45 REPEAT Statement
```
repeatStatement :
    REPEAT statementSequence UNTIL expression ;
```

#46 LOOP Statement
```
loopStatement :
    LOOP statementSequence END ;
```

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

#47.1 Control Variable
```
controlVariable : Ident ;
```

#48 Designator
```
designator :
    qualident designatorTail? ;
```

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

Expressions

#50 Expression List
```
expressionList :
    expression ( "," expression )* ;
```

#51 Expression
```
expression :
    simpleExpression ( relOp simpleExpression )? ;
```

#51.1 Relational Operator
```
relOp :
    "=" | "#" | "<" | "<=" | ">" | ">=" | IN ;
```

#52 Simple Expression
```
simpleExpression :
    ( "+" | "-" )? term ( addOp term )* ;
```

#52.1 Add Operator
```
addOp :
    "+" | "-" | OR ;
```

#53 Term
```
term :
    factor ( mulOp factor )* ;
```

#53.1 Multiply Operator
```
mulOp :
    "*" | "/" | DIV | MOD | AND ;
```

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

#55 Designator Or Function Call
```
designatorOrFunctionCall :
    designator actualParameters? ;
```


#56 Actual Parameters

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

Value Constructors

#57 Structured Value

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

#58 Value Component

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

Identifiers

#59 Qualified Identifier

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

#60 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 PLACEHOLDER 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 :
    "0" | "1" | "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[5] : { 0uFE, 0uBB, 0uBF } ;
```

[5] 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 | pragmaVOLATILE | 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 Volatile Atrribute Pragma
```
pragmaVOLATILE :
    VOLATILE ;
```

#16 Body Of Implementation Defined Pragma
```
implDefinedPragma :
    implDefPragmaName ( "=" inPragmaExpression )? ;
```

#18 In-Pragma Expression
```
inPragmaExpression :
    inPragmaSimpleExpr ( inPragmaRelOp inPragmaSimpleExpr )? ;
```

#18.1 In-Pragma Relational Operator
```
inPragmaRelOp :
    "=" | "#" | "<" | "<=" | ">" | ">=" ;
```

#19 In-Pragma Simple Expression
```
inPragmaSimpleExpr :
    ( "+" | "-" )? inPragmaTerm ( addOp inPragmaTerm )* ;
```

#20 In-Pragma Term
```
inPragmaTerm :
    inPragmaFactor ( inPragmaMulOp inPragmaFactor )* ;
```

#20.1 In-Pragma Multiply Operator
```
inPragmaMulOp :
    "*" | DIV | MOD | AND ;
```

#21 In-Pragma Factor
```
inPragmaFactor :
    wholeNumber | constQualident | "(" inPragmaExpression ")" |
    inPragmaPervasiveCall | NOT inPragmaFactor ;
```

#21.1 Whole Number
```
wholeNumber : NumericLiteral ;
```

#22 In-Pragma Pervasive Call
```
inPragmaPervasiveCall :
    Ident[1] "(" inPragmaExpression ( "," inPragmaExpression )* ")" ;
```

[1] Permissible are ABS, NEG, ODD, ORD, LENGTH, TMIN, TMAX, TSIZE, TLIMIT and macros in module COMPILER.

