Appendix A: Grammar in EBNF

A.1 Non-Terminal Symbols

Compilation Units

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

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

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

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

#4 Blueprint
```
blueprint :
    BLUEPRINT blueprintIdent "[" requiredConformance "]" ";"
    ( PLACEHOLDERS identList ";" )?
    requiredTypeDeclaration ";"
    ( requiredBinding ";" )*
    END prototypeIdent "." ;
```

#4.1 Blueprint Identifier
```
blueprintIdent : Ident ;
```

#4.2 RequiredConformance
```
requiredConformance : blueprintIdent ;
```

#4.3 RequiredBinding
```
requiredBinding : procedureHeader ;
```

Import Lists, Blocks, Definitions and Declarations

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

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

#7 Definition
```
definition :
    CONST ( publicConstDeclaration ";" )+ |
    TYPE ( publicTypeDeclaration ";" )+ |
    VAR ( variableDeclaration ";" )+ |
    procedureHeader ";" ;
```


#8 Public Constant Declaration
```
publicConstDeclaration :
    ( "[" constBindableIdent "]" )? Ident "=" constExpression ;
```

#8.1 CONST Bindable Identifier
```
constBindableIdent[1] : Ident ;
```

#8.2 Constant Expression
```
constExpression[2] : expression ;
```

#9 Public Type Declaration
```
publicTypeDeclaration :
    Ident "=" ( type | OPAQUE recordType? ) ;
```

#10 Declaration
```
declaration :
    CONST ( Ident "=" constExpression ";" )+ |
    TYPE ( Ident "=" type ";" )+ |
    VAR ( variableDeclaration ";" )+ |
    procedureHeader ";" block Ident ";" ;
```

#11 Required Type Declaration
```
requiredTypeDeclaration :
   TYPE "=" permittedTypeDeclaration ( "|" permittedTypeDeclaration )*
    ( ":=" protoliteral ( "|" protoliteral )* )?
```

#12 Permitted Type Declaration
```
permittedTypeDeclaration :
    RECORD | OPAQUE RECORD?
```

#13 Proto-Literal
```
protoliteral :
    simpleProtoliteral | structuredProtoliteral ;
```

#13.1 Simple Proto-Literal
```
simpleProtoliteral[3] : Ident;
```

#14 Structured Proto-Literal
```
structuredProtoliteral :
    "{" ( VARIADIC OF simpleProtoliteral ( "," simpleProtoliteral )* |
    structuredProtoliteral ( "," structuredProtoliteral )* ) "}" ;
```

Types

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

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

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

---

[1] CONST bindable identifiers are TSIG and TEXP.

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

[3] Simple protoliterals are CHAR, INTEGER and REAL, representing any quoted literals, whole numbers and real numbers.

#17 Enumeration Type
```
enumerationType :
    "(" ( "+" enumBaseType "," )? identList ")" ;
```

#17.1 Enumeration Base Type
```
enumBaseType : 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 discriminantFieldIdent OF typeIdent ;
```

#20.1 Discriminant Field Identifier
```
discriminantFieldIdent : Ident ;
```

#21 Set Type
```
setType :
    SET OF ( enumBaseType | "(" 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 )* "}"  ) ;
```

Variables

#29 Variable Declaration

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

Procedures

#30 Procedure Header

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

#31 Bindable Entity

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

#31.1 Bindable Identifier

```
bindableIdent[4] : 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 ( "[" variadicTerminator "]" )? OF
    ( simpleFormalType |
      "{" simpleFormalParams ( ";" simpleFormalParams )* "}" ) ;
```

#35.1 Variadic Terminator

```
variadicTerminator : constExpression ;
```

---

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

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 :
    factorOrNegation ( mulOp factorOrNegation )* ;
```

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

#53 Factor Or Negation
```
simpleFactor :
    NOT? factor ;
```

#54 Factor
```
factor :
    ( NumericLiteral | StringLiteral | structuredValue |
      designatorOrFunctionCall | "(" expression ")" )
    ( "::" namedType )? ;
```

#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 BLUEPRINT 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 RECORD REPEAT RETURN SET THEN TO TYPE UNTIL VAR VARIADIC WHILE ;
```

#2 Identifier

```
Ident :
    IdentLeadChar IdentTail? ;
```

#2.1 Identifier Lead Character

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

#2.2 Identifier Tail

```
IdentTail :
    ( IdentLeadChar | Digit )+ ;
```

#3 Numeric Literal

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

#3.1 Decimal Number Tail

```
DecimalNumberTail :
    DigitSep? DigitSeq RealNumberTail? | RealNumberTail ;
```

#3.2 Real Number Tail

```
RealNumberTail :
    "." DigitSeq ( "e" ( "+" | "-" )? DigitSeq )? ;
```

#3.3 Digit Sequence

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

#3.4 Base-2 Digit Sequence

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

#3.5 Base-16 Digit Sequence

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

#3.6 Digit

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

#3.7 Base-16 Digit

```
Base16Digit :
    Digit | "A" | "B" | "C" | "D" | "E" | "F" ;
```

#3.8 Base-2 Digit

```
Base2Digit :
    "0" | "1" ;
```

#3.9 Digit Separator

```
DigitSep : "'" ;
```


#4 String Literal
```
StringLiteral :
    SingleQuotedString | DoubleQuotedString ;
```

#4.1 Single Quoted String
```
SingleQuotedString :
    "'" ( QuotableCharacter | '"' )* "'" ;
```

#4.2 Double Quoted String
```
DoubleQuotedString :
    '"' ( QuotableCharacter | "'" )* '"' ;
```

#4.3 Quotable Character
```
QuotableCharacter :
    Digit | Letter | Space | QuotableGraphicChar | EscapedCharacter ;
```

#4.4 Letter
```
Letter :
    "A" .. "Z" | "a" .. "z" ;
```

#4.5 Space
```
Space : " " ;
```

#4.6 Quotable Graphic Character
```
QuotableGraphicChar :
    "!" | "#" | "$" | "%" | "&" | "(" | ")" | "*" | "+" | "," |
    "-" | "." | "/" | ":" | ";" | "<" | "=" | ">" | "?" | "@" |
    "[" | "]" | "^" | "_" | "`" | "{" | "|" | "}" | "~" ;
```

#4.7 Escaped Character
```
EscapedCharacter :
    "\" ( "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_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 | 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 :
    inPragmaFactorOrNegation ( inPragmaMulOp inPragmaFactorOrNegation )* ;
```

#22 In-Pragma Factor Or Negation
```
inPragmaFactorOrNegation :
    NOT? inPragmaFactor ;
```

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

#23 In-Pragma Factor
```
inPragmaFactor :
    wholeNumber | constQualident | "(" inPragmaExpression ")" |
    inPragmaCompileTimeFunctionCall ;
```

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

#24 In-Pragma Compile-Time Function Call
```
inPragmaCompileTimeFunctionCall :
    Ident[1] "(" inPragmaExpression ( "," inPragmaExpression )* ")" ;
```

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

