EBNF Notation used in this document

A Notation to Describe the Syntax of Modula-2

The syntax of PIM Modula-2 was formally defined in an extended version of Backus-Naur Formalism, known as Wirth EBNF, in which brackets and braces are used to denote optional and repeating syntactic entities. For the formal definition of the revised Modula-2 language we use a slightly different version of EBNF which employs parentheses and modifier suffixes instead.
Each EBNF rule defines exactly one symbol and is terminated by a semicolon. Names of symbols start with a letter which may be followed by letters, digits, hyphens and low lines. Names may not contain whitespace. Terminal symbols are denoted by names which start with an uppercase letter. Non-terminal symbols are denoted by names which start with a lowercase letter. Literals are enclosed in double or single quotes. By convention, reserved words of the target language are denoted in all-uppercase letters.
EBNF production rules take the following general forms:
Synonym

foo := bar ;
Symbol foo is defined as a synonym for symbol bar.
Sequence

foo := bar baz ;
Symbol foo is defined as a sequence of symbol bar followed by symbol baz.
Alternative

foo := bar | baz ;
Symbol foo is defined as an alternative, either symbol bar or symbol baz, but not both.
Option

foo := bar? ;
Symbol foo is defined by zero or one occurrence of symbol bar.
Repetition

foo := bar+ ;
Symbol foo is defined by one or more occurrences of symbol bar.
Optional Repetition

foo := bar* ;
Symbol foo is defined by zero or more occurrences of symbol bar.
Grouping

Parentheses may be used to group syntactic entities on the right hand side of an EBNF rule. A group may be followed by a ?, + or * modifier which then applies to the group as a whole.
foo := bar (baz | bam) (“,” boo)* ;
Symbol foo is defined by an occurrence of symbol bar followed by an alternative of symbol baz, or symbol bam followed by zero or more

```
occurrences of literal “,” and symbol boo.
EBNF defined in EBNF

syntax :=
    statement* ;

statement :=
    symbol-id “:=” expression “;” ;

expression :=
    term ( “|” term )* ;

term :=
    factor+ ;

factor :=
    ( symbol-id | literal | literal-range | group )

    ( “?” | “+” | “*” )* ;

group : =
    “(“ expression “)” ;

symbol-id :=
    terminal-id | non-terminal-id ;

terminal-id :=

    Uppercase-Letter ( Uppercase-Letter | Digit | “-” | “_” )* ;

non-terminal-id :=

    Lowercase-Letter ( Letter | Digit | “-” | “_” )* ;

Reserved-Word :=
  Uppercase-Letter* ;

literal :=

    ( ‘“‘ ( Character | “‘” )+ ‘“‘ ) |

    ( “‘“ ( Character | ‘“’ )+ “‘“ ) ;

literal-range :=
    literal “..” literal ;

Letter :=
    Uppercase-Letter | Lowercase-Letter ;

Uppercase-Letter :=
    “A” .. “Z” ;

Lowercase-Letter :=
    “a” .. “z” ;
```

```
Digit :=
    “0” .. “9” ;

Character :=

    Letter | Digit |

    “ “ | “!” | “#” | “$” | “%” | “&” | “(“ | “)” | “*” | “+” |

    “,” | “-” | “.” | “/” | “:” | “;” | “<“ | “=” | “>” | “?” |

    “@” | “[“ | “]” | “^” | “_” | “`” | “{“ | “|” | “}” | “~” ;
```

```
/* (C) 2009-2015 by B.Kowarsch & R.Sutcliffe. All rights reserved.
*/

grammar Modula2;

/* M2R10 grammar in ANTLR EBNF notation -- status Aug 31, 2015 */


//
-----------------------------------------------------------------------
-------
// N A M I N G   C O N V E N T I O N
//
-----------------------------------------------------------------------
-------
//
// Non-Terminals:
//  camelCase with first character lowercase
//
// Terminals:
//  CamelCase with first character uppercase
//
// Reserved Words:
//  ALL_UPPERCASE


options {

// *** enforce strict LL(1) ***

    k = 1; backtrack = no;
}


//
-----------------------------------------------------------------------
-------
// T O K E N   S Y M B O L S
```

```
//
-------------------------------------------------------------------
-------
// 50 reserved words, 32 dual-use identifiers, 22 pragma symbols

tokens {

// *** Reserved Words, 50 tokens ***

    ALIAS          = 'ALIAS';
    AND            = 'AND';            /* also a RW within pragmas
*/
    ARGLIST        = 'ARGLIST';
    ARRAY          = 'ARRAY';
    BEGIN          = 'BEGIN';
    BLUEPRINT      = 'BLUEPRINT';
    BY             = 'BY';
    CASE           = 'CASE';
    CONST          = 'CONST';
    COPY           = 'COPY';
    DEFINITION     = 'DEFINITION';
    DIV            = 'DIV';            /* also a RW within pragmas
*/
    DO             = 'DO';
    ELSE           = 'ELSE';           /* also a RW within pragma */
    ELSIF          = 'ELSIF';          /* also a RW within pragma */
    END            = 'END';            /* also a RW within pragma */
    ENUM           = 'ENUM';
    EXIT           = 'EXIT';
    FOR            = 'FOR';
    FROM           = 'FROM';           /* also a RW within pragma */
    GENLIB         = 'GENLIB';
    IF             = 'IF';             /* also a RW within pragma */
    IMPLEMENTATION = 'IMPLEMENTATION';
    IMPORT         = 'IMPORT';
    IN             = 'IN';
    LOOP           = 'LOOP';
    MOD            = 'MOD';            /* also a RW within pragmas
*/
    MODULE         = 'MODULE';
    NEW            = 'NEW';
    NONE           = 'NONE';
    NOT            = 'NOT';            /* also a RW within pragmas
*/
    OF             = 'OF';
    OPAQUE         = 'OPAQUE';
    OR             = 'OR';             /* also a RW within pragmas
*/
    POINTER        = 'POINTER';
    PROCEDURE      = 'PROCEDURE';
    RECORD         = 'RECORD';
    REFERENTIAL    = 'REFERENTIAL';
    RELEASE        = 'RELEASE';
    REPEAT         = 'REPEAT';
```

```
    RETAIN          = 'RETAIN';
    RETURN          = 'RETURN';
    SET             = 'SET';
    THEN            = 'THEN';
    TO              = 'TO';
    TYPE            = 'TYPE';
    UNTIL           = 'UNTIL';
    VAR             = 'VAR';
    WHILE           = 'WHILE';
    YIELD           = 'YIELD';


// *** Dual-Use Identifiers, 32 tokens ***

//  Dual-Use identifiers may be used as RWs depending on context.
The
//  resulting ambiguity is resolvable using the Schroedinger's Token
//  technique described in an article published in SP&E:
//
//    Schroedinger’s token
//    John Aycock and Nigel R. Horspool
//    Copyright (c) 2001, Wiley & Sons, Ltd.
//
//  DOI: Softw. Pract. Exper. 2001; 31:803–814 (DOI: 10.1002/spe.
390)
//  URL: http://citeseerx.ist.psu.edu/viewdoc/download?
doi=10.1.1.13.3178

/* Identifiers that are used like RWs within formal parameter lists
*/

    CAST            = 'CAST';           /* -> production #21 */
    OCTET           = 'OCTET';          /* -> production #21 */
    UNSAFE          = 'UNSAFE';         /* -> production #21 */
    ADDRESS         = 'ADDRESS';        /* -> production #21 */

/* Identifiers that are used like RWs within bound constant
declarations */

    TFLAGS          = 'TFLAGS';         /* -> productions #8, #62 */
    TDYN            = 'TDYN';           /* -> productions #8, #62 */
    TREFC           = 'TREFC';          /* -> productions #8, #62 */
    TORDERED        = 'TORDERED';       /* -> productions #8, #62 */
    TSORTED         = 'TSORTED';        /* -> productions #8, #62 */
    TLIMIT          = 'TLIMIT';         /* -> productions #8, #62 */
    TSCALAR         = 'TSCALAR';        /* -> productions #8, #62 */
    TMIN            = 'TMIN';           /* -> productions #8, #62 */
    TMAX            = 'TMAX';           /* -> productions #8, #62 */

/* Identifiers that are used like RWs within bound procedure headers
*/

    COROUTINE       = 'COROUTINE';      /* -> productions #26, #64 */
    ABS             = 'ABS';            /* -> productions #26, #64 */
```

```
    LENGTH          = 'LENGTH';         /* -> productions #26, #64 */
    EXISTS          = 'EXISTS';         /* -> productions #26, #64 */
    SEEK            = 'SEEK';           /* -> productions #26, #64 */
    SUBSET          = 'SUBSET';         /* -> productions #26, #64 */
    READ            = 'READ';           /* -> productions #26, #64 */
    READNEW         = 'READNEW';        /* -> productions #26, #64 */
    WRITE           = 'WRITE';          /* -> productions #26, #64 */
    WRITEF          = 'WRITEF';         /* -> productions #26, #64 */
    SXF             = 'SXF';            /* -> productions #26, #64 */
    VAL             = 'VAL';            /* -> productions #26, #64 */
    COUNT           = 'COUNT';          /* -> productions #26, #64 */
    VALUE           = 'VALUE';          /* -> productions #26, #64 */
    STORE           = 'STORE';          /* -> productions #26, #64 */
    INSERT          = 'INSERT';         /* -> productions #26, #64 */
    REMOVE          = 'REMOVE';         /* -> productions #26, #64 */
    APPEND          = 'APPEND';         /* -> productions #26, #64 */

/* Identifiers that are used like RWs within blueprint syntax */

    TLITERAL        = 'TLITERAL';       /* -> production #57 */


// *** Reserved Words of the Pragma Language, 22 tokens ***

//  Symbols that are reserved words only within pragmas

    MSG             = 'MSG';            /* RW within pragma only */
    INFO            = 'INFO';           /* RW within pragma only */
    WARN            = 'WARN';           /* RW within pragma only */
    ERROR           = 'ERROR';          /* RW within pragma only */
    FATAL           = 'FATAL';          /* RW within pragma only */
    INLINE          = 'INLINE';         /* RW within pragma only */
    NOINLINE        = 'NOINLINE';       /* RW within pragma only */
    NORETURN        = 'NORETURN';       /* RW within pragma only */
    PTW             = 'PTW';            /* RW within pragma only */
    FORWARD         = 'FORWARD';        /* RW within pragma only */
    ENCODING        = 'ENCODING';       /* RW within pragma only */
    ALIGN           = 'ALIGN';          /* RW within pragma only */
    PADBITS         = 'PADBITS';        /* RW within pragma only */
    PURITY          = 'PURITY';         /* RW within pragma only */
    SINGLEASSIGN    = 'SINGLEASSIGN' ;  /* RW within pragma only */
    LOWLATENCY      = 'LOWLATENCY';     /* RW within pragma only */
    VOLATILE        = 'VOLATILE';       /* RW within pragma only */
    DEPRECATED      = 'DEPRECATED';     /* RW within pragma only */
    GENERATED       = 'GENERATED';      /* RW within pragma only */
    ADDR            = 'ADDR';           /* RW within pragma only */
    FFI             = 'FFI';            /* RW within pragma only */
    FFIDENT         = 'FFIDENT';        /* RW within pragma only */


// *** Dual-Use Identifiers for Optional Language Facilities, 2
tokens ***
/*
    ASM             = 'ASM';
```

```
    REG           = 'REG';
*/

// *** Special Characters, 3 tokens ***

    BACKSLASH     = '\\';  /*\*/      /* for readability */
    SINGLE_QUOTE  = '\'' ; /*'*/      /* for readability */
    DOUBLE_QUOTE  = '\"' ; /*"*/      /* for readability */


// *** Ignore Characters, 3 tokens ***

    ASCII_TAB     = '\t';             /* for readability */
    ASCII_LF      = '\n';             /* for readability */
    ASCII_CR      = '\r';             /* for readability */
}


//
-------------------------------------------------------------------
-------
// N O N - T E R M I N A L   S Y M B O L S
//
-------------------------------------------------------------------
-------
// 64 productions

// *** Compilation Units ***

// production #1
compilationUnit :
  definitionModule | implOrPrgmModule | blueprint
  ;

// *** Definition Module Syntax ***

// production #2
definitionModule :
  DEFINITION MODULE moduleIdent
  ( '[' blueprintToObey ']' )? ( FOR typeToExtend )? ';'
  (importList ';')* definition*
  END moduleIdent '.'
  ;

// alias #2.1a
moduleIdent : Ident ;

// alias #2.1b
blueprintIdent : Ident ;

// alias #2.1c
typeToExtend : Ident ;

// alias #2.2
```

```
blueprintToObey : blueprintIdent ;


// production #3
importList :
  libGenDirective | importDirective
  ;

// production #4
libGenDirective :
  GENLIB libIdent FROM template FOR templateParamList END
  ;

// fragment #4b
// incorporated into #4 in syntax diagram
templateParamList :
  placeholder '=' replacement ( ';' placeholder '=' replacement )* ;

// alias #4.1a
libIdent : Ident ;

// alias #4.1b
template : Ident ;

// alias #4.1c
placeholder : Ident ;

// fragment #4.2
replacement :
  NumberLiteral | StringLiteral | ChevronText ;


// production #5
importDirective :
  FROM ( moduleIdent | ENUM enumTypeIdent )
    IMPORT ( identifiersToImport | importAll ) |
  IMPORT modulesToImport
  ;

// alias #5.1
enumTypeIdent : typeIdent ;

// alias #5.2
typeIdent : qualident ;

// fragment #5.3
identifiersToImport :
  Ident reExport? ( ',' Ident reExport? )*
  ;

// alias #5.3b
modulesToImport : identifiersToImport ;

// alias #5.4
```

```
reExport : '+' ;

// alias #5.5
importAll : '*' ;


// production #6
qualident :
    Ident ( '.' Ident )*
    ;


// production #7
definition :
  CONST ( constDefinition ';' )+ |
  TYPE ( typeDefinition ';' )+ |
  VAR ( variableDeclaration ';' )+ |
  procedureHeader ';'
  ;


// production #8
constDefinition :
  ( '[' propertyToBindTo ']' | restrictedExport )?
  Ident '=' constExpression
  ;

// alias #8.1
constExpression : expression ;

// alias #8.2
restrictedExport : '*' ;


// production #9
typeDefinition :
  restrictedExport? Ident '=' ( OPAQUE | type )
  ;


// production #10
variableDeclaration :
  identList ':' ( range OF )? typeIdent
  ;


// production #11
identList :
    Ident ( ',' Ident )*
    ;


// production #12
range :
```

```
  '[' greaterThan? constExpression '..' lessThan? constExpression
']'
  ;

// alias #12.1
greaterThan : '>' ;

// alias #12.2
lessThan : '<' ;


// production #13
type :
  typeIdent | derivedSubType | enumType | setType | arrayType |
  recordType | pointerType | coroutineType | procedureType
  ;

// fragment #13.1
derivedSubType :
  ALIAS OF typeIdent |
  range OF ordinalOrScalarType |
  CONST dynamicTypeIdent
  ;

// alias #13.2a
ordinalOrScalarType : typeIdent ;

// alias #13.2b
dynamicTypeIdent : typeIdent ;


// production #14
enumType :
  '(' ( '+' enumTypeToExtend )? identList ')'
  ;

// alias #14.1
enumTypeToExtend : typeIdent ;


// production #15
setType :
  SET OF enumTypeIdent ;


// production #16
arrayType :
  ARRAY componentCount ( ',' componentCount )* OF typeIdent
  ;

// alias #16.1
componentCount : constExpression ;
```

```
// production #17
recordType :
  RECORD
    ( fieldList ( ';' fieldList )* indeterminateField? |
      '(' recTypeToExtend ')' fieldList ( ';' fieldList )* )
  ;

// fragment #17.1
fieldList :
  restrictedExport? variableDeclaration
  ;

// alias #17.2
recTypeToExtend : typeIdent ;

// fragment #17.3
indeterminateField :
  '~' Ident ':' ARRAY discriminantFieldIdent OF typeIdent
  ;

// alias #17.4
discriminantFieldIdent : Ident ;


// production #18
pointerType :
  POINTER TO CONST? typeIdent
  ;


// production #19
coroutineType :
  COROUTINE '(' assocProcType ')'
  ;

// alias #19.1
assocProcType : typeIdent ;



// production #20
procedureType :
  PROCEDURE ( formalType ( ',' formalType )* )? ( ':'
returnedType )?
  ;

// fragment #20.1
formalType :
  simpleFormalType | attributedFormalType | variadicFormalType
  ;

// alias #20.2
returnedType : typeIdent ;
```

```
// production #21
simpleFormalType :
  ( ARRAY OF )? typeIdent | castingFormalType
  ;

// fragment #21.1
castingFormalType :
  CAST ( ARRAY OF OCTET | addressTypeIdent )
  ;

// fragment #21.2
addressTypeIdent :
  ( UNSAFE '.' )? ADDRESS
  ;


// production #22
attributedFormalType :
  ( CONST | VAR | NEW ) ( simpleFormalType |
simpleVariadicFormalType )
  ;


// production #23
simpleVariadicFormalType :
  ARGLIST reqNumOfArgs? OF simpleFormalType terminator?
  ;

// fragment #23.1
reqNumOfArgs :
  greaterThan? constExpression
  ;

// fragment #23.2
terminator :
  '|' constQualident
  ;

// alias #23.3
constQualident : qualident ;


// production #24
variadicFormalType :
  ARGLIST reqNumOfArgs? OF
    ( '{' nonVariadicFormalType ( ';' nonVariadicFormalType )* '}' |
      simpleFormalType ) terminator?
  ;


// production #25
nonVariadicFormalType :
  ( CONST | VAR | NEW )? simpleFormalType
```

```
  ;


// production #26
procedureHeader :
  PROCEDURE ( '[' ( entityToBindTo | COROUTINE ) ']' |
restrictedExport )?
  procedureSignature
  ;


// production #27
procedureSignature :
  Ident ( '(' formalParams ( ';' formalParams )* ')' )? ( ':'
returnedType )?
  ;


// production #28
formalParams :
  identList ':' ( simpleFormalType | variadicFormalParams ) |
  attributedFormalParams
  ;


// production #29
attributedFormalParams :
  ( CONST | VAR | NEW ) identList ':'
  ( simpleFormalType | simpleVariadicFormalType )
  ;


// production #30
variadicFormalParams :
  ARGLIST reqNumOfArgs? OF
    ( ( '{' nonVariadicFormalParams ( ';' nonVariadicFormalParams )*
'}') |
      simpleFormalType ) terminator?
  ;


// production #31
nonVariadicFormalParams :
  ( CONST | VAR | NEW )? identList ':' simpleFormalType
  ;


// *** Implementation or Program Module Syntax

// production #32
implOrPrgmModule :
  IMPLEMENTATION? MODULE moduleIdent ';'
  ( importList ';' )* block moduleIdent '.'
  ;
```

```
// production #33
block :
  declaration* ( BEGIN statementSequence )? END
  ;


// production #34
declaration :
  CONST ( Ident '=' constExpression ';' )+ |
  TYPE ( Ident '=' type ';' )+ |
  VAR ( variableDeclaration ';' )+ |
  procedureHeader ';' block Ident ';'
  ;


// production #35
statementSequence :
  statement ( ';' statement )*
  ;


// production #36
statement :
  memMgtOperation | updateOrProcCall | ifStatement | caseStatement |
  loopStatement | whileStatement | repeatStatement | forStatement |
  ( RETURN | YIELD ) expression? | EXIT
  ;


// production #37
memMgtOperation :
  NEW designator ( OF initSize | := initValue )? |
  RETAIN designator |
  RELEASE designator
  ;

// alias #37.1a
initSize : expression ;

// alias #37.1b
initValue : expression ;


// production #38
updateOrProcCall :
  designator ( ':=' expression | incOrDecSuffix |
actualParameters )? |
  COPY designator ':=' expression
  ;

// fragment #38.1
incOrDecSuffix :
```

```
  '++' | '--'
  ;


// production #39
ifStatement :
  IF boolExpression THEN statementSequence
  ( ELSIF boolExpression THEN statementSequence )?
  ( ELSE statementSequence )?
  END
  ;

// alias #39.1
boolExpression : expression ;


/* ANTLR has a design flaw in that it cannot handle any rule
identifiers that
   coincide with reserved words of the language it uses to generate
the parser,
   by default Java, thus Java reserved words can't be used as rule
identifiers.

   For this reason we are using "case777" here instead of the proper
"case". */

// production #40
caseStatement :
  CASE expression OF ( '|' case777 )+ ( ELSE statementSequence )?
END
  ;

// fragment #40.1
case777 :
  caseLabels ( ',' caseLabels )* ':' statementSequence
  ;

// fragment #40.2
caseLabels :
  constExpression ( '..' constExpression )?
  ;


// production #41
loopStatement :
  LOOP statementSequence END
  ;


// production #42
whileStatement :
  WHILE boolExpression DO statementSequence END
  ;
```

```
// production #43
repeatStatement :
  REPEAT statementSequence UNTIL boolExpression
  ;


// production #44
forStatement :
  FOR forLoopVariants IN iterableEntity DO statementSequence END
  ;

// fragment #44.1
forLoopVariants :
  accessor ascOrDesc? ( ',' value )? |
  VALUE value ascOrDesc?
  ;

// alias #44.2a
accessor : Ident ;

// alias #44.2b
value : Ident ;

// fragment #44.3
iterableEntity :
  designator | range OF ordinalType
  ;

// alias #44.4
ascOrDesc : incOrDecSuffix ;

// alias #44.5
ordinalType : typeIdent ;


// production #45
designator :
  qualident designatorTail?
  ;

// fragment #45.1
designatorTail :
  ( ( '[' exprListOrSlice ']' | '^' ) ( '.' Ident )* )+
  ;

// fragment #45.2
exprListOrSlice :
  expression ( ( ',' expression )* | '..' expression? )
  ;


// production #46
expression :
```

```
  simpleExpression ( operL1 simpleExpression )?
  ;

// fragment #46.1
operL1 :
  '=' | '#' | '<' | '<=' | '>' | '>=' | IN | concatOp | identityOp
  ;

// alias #46.2
concatOp : '&' ;

// alias #46.3
identityOp : '==' ;


// production #47
simpleExpression :
  ( '+' | '-' )? term ( operL2 term )*
  ;

// fragment #47.1
operL2 :
  '+' | '-' | OR
  ;


// production #48
term :
  factorOrNegation ( operL3 factorOrNegation )*
  ;

// fragment #48.1
operL3 :
  '*' | '/' | DIV | MOD | AND | setDiffOp | dotProductOp
  ;

// alias #48.2
setDiffOp : '\' ;

// alias #48.3
dotProductOp : '*.' ;


// production #49
factorOrNegation :
  NOT? factorOrTypeConv
  ;


// production #50
factorOrTypeConv :
  factor ( '::' typeIdent )?
  ;
```

```
// production #51
factor :
  NumberLiteral | StringLiteral | structuredValue |
  '(' expression ')' | designator actualParameters?
  ;


// production #52
actualParameters :
  '(' expressionList ')'
  ;


// production #53
expressionList :
  expression ( ',' expression )*
  ;


// production #54
structuredValue :
  '{' valueComponent ( ',' valueComponent )* '}'
  ;

// fragment #54.1
valueComponent :
  constExpression (( BY | '..' )? constExpression )? |
runtimeExpression
  ;

// alias #54.2
runtimeExpression : expression ;


// *** Blueprint Syntax

// production #55
blueprint :
  BLUEPRINT blueprintIdent ( '[' blueprintToRefine ']' )?
  ( FOR blueprintForTypeToExtend )? ';' ( REFERENTIAL identList
';' )?
  MODULE TYPE '=' ( typeClassification ( ';' literalCompatibility)?
| NONE ) ';'
  ( constraint ';' )* ( requirement ';' )* END blueprintIdent '.'
  ;

// alias #55.1
blueprintIdent : Ident ;

// alias #55.2a
blueprintToRefine : blueprintIdent ;

// alias #55.2b
```

```
blueprintForTypeToExtend : blueprintIdent ;


// production #56
typeClassification :
  '{' determinedClassification ( ';' refinableClassification )?
( ';' '*' )? '}'
  | '*'
  ;

// fragment #56.1
determinedClassification :
  classificationIdent ( ',' classificationIdent )*
  ;

// fragment #56.2
refinableClassification :
  '~' classificationIdent ( ',' '~' classificationIdent )*
  ;

// alias #56.3
classificationIdent : Ident ;


// production #57
literalCompatibility :
  TLITERAL '=' protoLiteral ( '|' protoLiteral )*
  ;

// fragment #57.1
protoLiteral :
  protoLiteralIdent | structuredProtoLiteral
  ;

// alias #57.2
protoLiteralIdent : Ident ;


// production #58
structuredProtoLiteral :
  '{'
    ( ARGLIST reqValueCount? OF
        ( '{' builtinOrReferential ( ',' builtinOrReferential )* '}'
|
          builtinOrReferential ) ) |
    builtinOrReferential '}'
  ;

// fragment #58.1
reqValueCount :
  greaterThan? wholeNumber
  ;

// alias #58.2
```

```
greaterThan : '>' ;

// alias #58.2
wholeNumber : NumberLiteral ;

// alias #58.3
builtinOrReferential : Ident ;


// production #59
constraint :
  constraintTerm ( oneWayDependency | mutualDependencyOrExclusion )
  ;

// fragment #59.1
constraintTerm :
  '(' classificationOrFlagIdent ')' |
  '[' bindableEntityOrProperty ']'
  ;

// fragment #59.2
bindableEntityOrProperty :
  entityToBindTo | propertyToBindTo
  ;

// fragment #59.3
oneWayDependency :
  '->' termList ( '|' termList )*
  ;

// fragment #59.4
mutualDependencyOrExclusion :
  ( '<>' | '><' ) termList
  ;

// fragment #59.5
termList :
  constraintTerm ( ',' constraintTerm )*
  ;

// alias #59.6
classificationOrFlagIdent : Ident ;


// production #60
requirement :
  condition '->' ( typeRequirement | constRequirement |
procRequirement )
  ;

// fragment #60.1
condition :
  NOT? boolConstIdent
  ;
```

```
// alias #60.2
boolConstIdent : Ident ;

// fragment #60.3
typeRequirement :
  TYPE typeDefinition
  ;


// production #61
constRequirement :
  CONST
    ( '[' propertyToBindTo ']' ( simpleConstRequirement | '=' NONE )
|
      restrictedExport? simpleConstRequirement )
  ;

// fragment #61.1
simpleConstRequirement :
  Ident ( '=' constExpression | ':' builtinTypeIdent )
  ;

// alias #61.2
constExpression : expression ;

// alias #61.3
builtinTypeIdent : Ident ;

// alias #61.4
restrictedExport : '*' ;


// production #62
propertyToBindTo :
  memMgtProperty | collectionProperty | scalarProperty | TFLAGS
  ;

// fragment #62.1
memMgtProperty :
  TDYN | TREFC
  ;

// fragment #62.2
collectionProperty :
  TORDERED | TSORTED | TLIMIT
  ;

// fragment #62.3
scalarProperty :
  TSCALAR | TMAX | TMIN
  ;
```

```
// production #63
procedureRequirement :
  PROCEDURE
    ( '[' ( entityToBindTo | COROUTINE ) ']' ( procedureSignature |
'=' NONE ) |
      restrictedExport? procedureSignature )
  ;


// production #64
entityToBindTo :
  bindableResWord | bindableOperator | bindableMacro
  ;

// fragment #64.1
bindableResWord :
  NEW | RETAIN | RELEASE | COPY | bindableFor
  ;

// fragment #64.2
bindableFor :
  FOR forBindingDifferentiator?
  ;

// fragment #64.3
forBindingDifferentiator :
  '|' ( '++' | '--' )
  ;

// fragment #64.4
bindableOperator :
  '+' | '-' | '*' | '/' | '\' | '=' | '<' | '>' | '*.' | '::'
  IN | DIV | MOD | unaryMinus
  ;

// fragment #64.5
unaryMinus :
  '+' '/' '-'
  ;

// fragment #64.6
bindableMacro :
  ABS | LENGTH | EXISTS | SEEK | SUBSET | READ | READNEW | WRITE |
WRITEF |
  SXF | VAL | multiBindableMacro1 | multiBindableMacro2 |
multiBindableMacro3
  ;

// fragment #64.7
multiBindableMacro1 :
  ( COUNT | VALUE ) bindingDifferentiator1?
  ;

// fragment #64.8
```

```
bindingDifferentiator1 :
  '|' '#'
  ;

// fragment #64.9
multiBindableMacro2 :
  ( STORE | INSERT | REMOVE ) bindingDifferentiator2?
  ;

// fragment #64.10
bindingDifferentiator2 :
  '|' ( ',' | '#' | '*' )
  ;

// fragment #64.11
multiBindableMacro3 :
  APPEND bindingDifferentiator3?
  ;

// fragment #64.12
bindingDifferentiator3 :
  '|' ( ',' | '*' )
  ;


//
-------------------------------------------------------------------
-------
// O P T I O N A L   L A N G U A G E   F A C I L I T I E S
//
-------------------------------------------------------------------
-------
/*
// *** Architecture Specific Implementation Module Selection ***

// replacement for implOrPrgmModule
langExtn_implOrPrgmModule :
    IMPLEMENTATION? MODULE moduleIdent ( '(' archSelector ')' )? ';'
    importList* block moduleIdent '.'
    ;

// alias: Architecture Selector
langExtn_archSelector : Ident ;

// *** Register Mapping ***

// replacement for simpleFormalType
langExtn_simpleFormalType :
    typeIdent regAttribute? |
    ARRAY OF typeIdent |
    castingFormalType
    ;

// replacement for castingFormalType
```

```
langExtn_castingFormalType :
    CAST ( ARRAY OF OCTET | addressTypeIdent regAttribute? )
    ;

// Register Mapping Attribute
regAttribute :
    IN REG ( registerNumber | registerMnemonic )
    ;

// alias: Register Number
registerNumber : constExpression ;

// alias: Register Mnemonic
registerMnemonic : qualident ;

// *** Symbolic Assembly Inlining ***

// replacement for statement
langExtn_statement :
    memMgtOperation | updateOrProcCall | ifStatement | caseStatement
|
    loopStatement | whileStatement | repeatStatement | forStatement
|
    assemblyBlock | ( RETURN | YIELD ) expression? | EXIT
    ;

// Assembly Block
assemblyBlock :
    ASM assemblySourceCode END
    ;

// Assembly Source Code
assemblySourceCode :
    <implementation defined syntax>
    ;
*/

//
-------------------------------------------------------------------
-------
// P R A G M A   G R A M M A R
//
-------------------------------------------------------------------
-------
// 23 productions

// *** Pragmas ***

// production #1
pragma :
  '<*' pragmaBody '*>'
  ;

// fragment #1.1
```

```
pragmaBody :
  pragmaMSG | pragmaIF | procDeclAttrPragma | pragmaPTW |
pragmaFORWARD |
  pragmaENCODING | pragmaALIGN | pragmaPADBITS | pragmaPURITY |
  varDeclAttrPragma | pragmaDEPRECATED | pragmaGENERATED |
  pragmaADDR | pragmaFFI | pragmaFFIDENT | implDefinedPragma
 ;

// production #2
pragmaMSG :
  MSG '=' ctMsgMode ':' ctMsgComponentList
  ;

// fragment #2.1
ctMsgMode :
  ( INFO | WARN | ERROR | FATAL {})
  ;

// fragment #2.2
ctMsgComponentList :
  ctMsgComponent ( ',' ctMsgComponent )*
  ;

// fragment #2.3
compileTimeMsgComponent :
    StringLiteral | constQualident | '?' valuePragma
    ;

// alias #2.4
constQualident : qualident ; /* no type and no variable identifiers
*/

// fragment #2.5
valuePragma :
    ALIGN | ENCODING | valuePragmaSymbol
    ;

// alias #2.6
valuePragmaSymbol : PragmaSymbol ;

// fragment #2.7
PragmaSymbol :
  Letter+
  ;

// production #3
pragmaIF :
  ( IF | ELSIF {}) inPragmaExpression | ELSE | ENDIF
  ;

// production #4
procDeclAttrPragma :
  INLINE | NOINLINE | NORETURN
  {} /* make ANTLRworks display separate branches */
```

```
  ;

// production #5
pragmaPTW :
  PTW
  ;

// production #6
pragmaFORWARD :
  FORWARD ( TYPE identList | procedureHeader )
  ;

/* multi-pass compilers ignore and skip any token after FORWARD */

// production #7
pragmaENCODING :
  ENCODING '=' StringLiteral /* "ASCII" or "UTF8" */
  ( ':' codePointSampleList )?
  ;

// fragments #7.1a and #7.1b are combined into syntax diagram #7.1

// fragment #7.1a
codePointSampleList :
  codePointSample ( ',' codePointSample )*
  ;

// fragment #7.1b
codePointSample :
  quotedCharacterLiteral '=' characterCodeLiteral
  ;

// alias #7.2
quotedCharacterLiteral : StringLiteral ; /* single character only */

// alias #7.3
characterCodeLiteral : NumberLiteral ; /* unicode code points only
*/

// production #8
pragmaALIGN :
  ALIGN '=' inPragmaExpression
  ;

// production #9
pragmaPADBITS :
  PADBITS '=' inPragmaExpression
  ;

// production #10
pragmaPURITY :
  PURITY '=' inPragmaExpression /* values 0 .. 3 */
  ;
```

```
// production #11
varDeclAttrPragma :
  SINGLEASSIGN | LOWLATENCY | VOLATILE
  ;

// production #12
pragmaDEPRECATED :
  DEPRECATED
  ;

// production #13
pragmaGENERATED :
  GENERATED template ',' datestamp ',' timestamp
  ;

// fragment #13.1
datestamp :
  year '-' month '-' day
  ;

// fragment #13.2
timestamp :
  hours ':' minutes ':' seconds '+' timezone
  ;

// alias #13.3a
year : wholeNumber ;

// alias #13.3b
month : wholeNumber ;

// alias #13.3c
day : wholeNumber ;

// alias #13.3d
hours : wholeNumber ;

// alias #13.3e
minutes : wholeNumber ;

// alias #13.3f
seconds : wholeNumber ;

// alias #13.4g
timezone : wholeNumber ;

// production #14
pragmaADDR :
  ADDR '=' inPragmaExpression
  ;

// production #15
pragmaFFI :
  FFI '=' StringLiteral /* "C", "Fortran", "CLR" or "JVM" */
```

```
  ;

// production #16
pragmaFFIDENT :
  FFIDENT '=' StringLiteral /* foreign library identifier */
  ;

// production #17
implDefinedPragma :
  ( implPrefix '.' )? PragmaSymbol ( '=' inPragmaExpression )? '|'
ctMsgMode
  ;

// fragment #17.1
implPrefix :
  Letter LetterOrDigit*
  ;

// production #18
inPragmaExpression :
/* represents operator precedence level 1 */
  inPragmaSimpleExpression ( inPragmaRelOp
inPragmaSimpleExpression )?
  ;

// fragment #18.1
inPragmaRelOp :
  '=' | '#' | '<' | '<=' | '>' | '>='
  {} /* make ANTLRworks display separate branches */
  ;

// production #19
inPragmaSimpleExpression :
/* represents operator precedence level 2 */
  ( '+' | '-' {})? inPragmaTerm ( addOp inPragmaTerm )*
  ;

// fragment #19.1
addOp :
  '+' | '-' | OR
  {} /* make ANTLRworks display separate branches */
  ;

// production #20
inPragmaTerm :
/* represents operator precedence level 3 */
  inPragmaFactor ( mulOp inPragmaFactor )*
  ;

// fragment #20.1
mulOp :
  '*' | DIV | MOD | AND
  {} /* make ANTLRworks display separate branches */
  ;
```

```
// production #21
inPragmaFactorOrNegation :
/* represents operator precedence level 4 */
  NOT? inPragmaFactor
  ;

// production #22
inPragmaFactor :
  WholeNumber |
  /* constQualident is covered by inPragmaCompileTimeFunctionCall */
  '(' inPragmaExpression ')' | inPragmaCompileTimeFunctionCall
  ;

// production #23
inPragmaCompileTimeFunctionCall :
  qualident ( '(' inPragmaExpression ( ',' inPragmaExpression )*
')' )?
  ;


//
-------------------------------------------------------------------
-------
// T E R M I N A L   S Y M B O L S
//
-------------------------------------------------------------------
-------
// 7 productions

// production #1
ReservedWord :
    ALIAS | AND | ARGLIST | ARRAY | BEGIN | BLUEPRINT | BY | CASE |
CONST |
    COPY | DEFINITION | DIV | DO | ELSE | ELSIF | END | ENUM | EXIT
| FOR |
    FROM | GENLIB | IF | IMPLEMENTATION | IMPORT | IN | LOOP | MOD |
MODULE |
    NEW | NONE | NOT | OF | OPAQUE | OR | POINTER | PROCEDURE |
RECORD |
    REFERENTIAL | RELEASE | REPEAT | RETAIN | RETURN | SET | THEN |
TO |
    TYPE | UNTIL | VAR | WHILE | YIELD
    ;

// production #2
DualUseIdent : /* Ident */
    ABS | ADDRESS | APPEND | CAST | COUNT | COROUTINE | EXISTS |
INSERT |
    LENGTH | OCTET | READ | READNEW | REMOVE | SEEK | STORE | SUBSET
| SXF |
    TDYN | TFLAGS | TLIMIT | TLITERAL | TMAX | TMIN | TORDERED |
TREFC |
    TSCALAR | TSORTED | UNSAFE | VAL | VALUE | WRITE | WRITEF
```

```
    ;

// production #3
SpecialSymbol :
    '.' | ',' | ':' | ';' | '|' | '^' | '~' | '..' | ':=' | '++' |
'--' |
    '::' | '+' | '-' | '*' | '*.' | '/' | BACKSLASH | '=' | '#' |
'>' |
    '>=' | '<' | '<=' | '==' | '&' | '->' | '<>' | '><' | '?' |
    '(' | ')' | '[' | ']' | '{' | '}' | '<*' | '*>' |
    SINGLE_QUOTE | DOUBLE_QUOTE | '<<' | '>>' | '!' | '(*' | '*)' |
    ;

// NB: SINGLE_QUOTE, DOUBLE_QUOTE, //, (*, *), << and >> are NOT
tokens.
//     These symbols are NEVER returned as tokens by a Modula-2
lexer.

// production #4
Ident :
    ( Letter | ForeignIdentChar+ LetterOrDigit+ ) IdentTailChar*
    ;

// fragment #4.1
StdLibIdent :
    Letter LetterOrDigit*
    ;

fragment /* #4.2 */
LetterOrDigit :
    Letter | Digit
    ;

fragment /*  #4.3 */
ForeignIdentChar :
    '_' | '$'
    ;

fragment /* #4.4 */
IdentTailChar :
    LetterOrDigit | ForeignIdentChar
    ;

// production #5
NumberLiteral :
    /* number literals starting with digit 0 ... */
    '0' (
        /* without prefix are real numbers */
        RealNumberTail |
        /* with prefix 0b are base-2 numbers */
        'b' Base2DigitSeq |
        /* with prefix 0x are base-16 numbers */
        'x' Base16DigitSeq |
        /* with prefix 0u are unicode code points */
```

```
        'u' Base16DigitSeq
         )?
    /* number literals starting with digits 1 to 9 ... */
    | '1'..'9' DecimalNumberTail? /* are always decimal numbers */
    ;

fragment /* #5.1 */
DecimalNumberTail :
    DigitSep? DigitSeq RealNumberTail? | RealNumberTail
    ;

fragment /* #5.2 */
RealNumberTail :
    '.' DigitSeq ( 'e' ( '+' | '-' {})? DigitSeq )?
    ;

fragment /* #5.3 */
DigitSeq :
    Digit+ ( DigitSep Digit+ )*
    ;

fragment /* #5.4 */
Base16DigitSeq :
    Base16Digit+ ( DigitSep Base16Digit+ )*
    ;

fragment /* #5.5 */
Base2DigitSeq :
    Base2Digit+ ( DigitSep Base2Digit+ )*
    ;

fragment /* #5.6 */
Digit :
    Base2Digit | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    {} /* make ANTLRworks display separate branches */
    ;

fragment /* #5.7 */
Base16Digit :
    Digit | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
    {} /* make ANTLRworks display separate branches */
    ;

fragment /* #5.8 */
Base2Digit :
    '0' | '1'
    {} /* make ANTLRworks display separate branches */
    ;

fragment /* #5.9 */
DigitSep : SINGLE_QUOTE {} /* make ANTLRworks display name, not
literal */ ;

// production #6
```

```
StringLiteral :
    SingleQuotedString | DoubleQuotedString
    ;

fragment /* #6.1 */
SingleQuotedString :
    SINGLE_QUOTE ( QuotableCharacter | DOUBLE_QUOTE )* SINGLE_QUOTE
    ;

fragment /* #6.2 */
DoubleQuotedString :
    DOUBLE_QUOTE ( QuotableCharacter | SINGLE_QUOTE )* DOUBLE_QUOTE
    ;

fragment /* #6.3 */
QuotableCharacter :
    Digit | Letter | Space | NonAlphaNumQuotable | EscapedCharacter
    ;

fragment /* #6.4 */
Letter :
    'A' .. 'Z' | 'a' .. 'z'
    {} /* make ANTLRworks display separate branches */
    ;

fragment /* #6.5 */
Space : ' ' ;

fragment /* #6.6 */
NonAlphaNumQuotable :
    '!' | '#' | '$' | '%' | '&' | '(' | ')' | '*' | '+' | ',' |
    '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' | '?' | '@' |
    '[' | ']' | '^' | '_' | '`' | '{' | '|' | '}' | '~'
    {} /* make ANTLRworks display separate branches */
    ;

fragment /* #6.7 */
EscapedCharacter :
    BACKSLASH ( 'n' | 't' | BACKSLASH {})
    ;

// production #7
ChevronText :
    '<<' ( QuotableCharacter | SINGLE_QUOTE | DOUBLE_QUOTE )* '>>'
    ;


//
-------------------------------------------------------------------
-------
// I G N O R E   S Y M B O L S
//
-------------------------------------------------------------------
-------
```

```
// 5 productions

// *** Whitespace ***

// production #1
Whitespace :
    ({} Space | ASCII_TAB) { $channel = HIDDEN; } /* ignore */
    ;

// *** Comments ***

// pseudo-procudion to make #2 and #3 hidden
Comment :
    BlockComment | LineComment
    { $channel = HIDDEN; } /* ignore */
    ;

// production #2
fragment
BlockComment :
    '(*'
    ( options { greedy=false; }: . )* /* anything other than '(*' or
'*)' */
    BlockComment*
    '*)'
    ;

// production #3
fragment
LineComment :
    '//'
    ( options { greedy=false; }: . )* /* anything other than EOL */
    EndOfLine
    ;

// production #4
fragment
EndOfLine :
    ASCII_LF | ASCII_CR (ASCII_LF{})?
    ;

// END OF FILE
```