FORTRAN 77 Semantic Analysis

W. M. Waite

November 3, 2006

Abstract

This document describes the semantic analysis problem for FORTRAN 77. It is a part of an Eli specification from which a compiler for FORTRAN 77 can be generated.

Only the semantic analysis task (determining the properties of all entities appearing in a program and decorating the abstract syntax tree) is covered in this document. Modules specifying the lexical and syntactic analysis tasks are also available, and can be combined with this module to specify a processor that extracts symbol information from FORTRAN 77 programs. These specifications can also be used as the basis for processors that derive metrics characterizing FORTRAN 77 programs, convert FORTRAN 77 programs into programs in other languages, or implement extensions.

Contents

1 Introduction

This document is a formal specification of the symbol table construction task for FORTRAN 77. It encodes ANSI X3.9-1978's natural-language description of the properties of FORTRAN 77 symbolic names and the context conditions governing their use. The encoding describes relationships among attributes of the nodes of a FORTRAN 77 program's abstract syntax tree, and the properties of entities appearing in the program. In order to provide traceability, the structure of the document mirrors that of the standard.

All of the computations described in this document are carried out on an abstract syntax tree having the form defined by the phrase structure definition module for FORTRAN 77. Eli can generate either a FORTRAN 77 or a FORTRAN 90 scanner/parser from that specification, depending upon the setting of a `#define` directive. The directive must be set for FORTRAN 77 if the generated scanner/parser is to be used with the symbol table constructor described here.

File `Semantic/F77Semantics.fw` contains the specification from which this document was generated. An implementation of the symbol table construction task will be produced if `Semantic/F77Lint.specs` is named as one of the specification components. Any arbitrary additional components may be supplied to describe other tasks.

Section 2 summarizes the information defined by this module that is likely to be of interest to other specification modules. Users of this module should therefore concentrate their attention on Section 2.

2 FORTRAN Terms and Concepts

The basic concepts of FORTRAN are reflected in the grammar symbols and their attributes, and the entities with their associated properties. For this specification, only the following concepts are needed:

FORTRAN Terms and Concepts[1] ≡

Syntactic Items[5]
Program Units and Procedures[18]

This macro is invoked in definition 244.

A grammar symbol characterizes a class of tree nodes, while an attribute characterizes a value associated with a specific node in the tree. Each tree node in the class characterized by a specific grammar symbol has the same set of attributes, but the values of these attributes may differ between nodes.

The properties of an entity are values of arbitrary type stored in the definition table and accessible via a `DefTableKey` value representing the entity. Any property may be associated with any entity. When a property is queried, the query operation must provide a "default" value to be used in satisfying the query if the entity does not possess the queried property.

2.1 Sequence

The elements of a list are ordered by a one-to-one correspondence with the numbers 1, 2, through the length of the list. `SeqCount` gives the number of elements in each list or sublist, and `SeqIndex` gives the position of a specific element.

Sequence[2] ≡

```
ATTR SeqIndex, SeqCount: int;
```

This macro is defined in definitions 2 and 4.
This macro is invoked in definition 244.

Sequence the elements[3](◇1) ≡

```
RULE: ◇1List ::= ◇1
COMPUTE
  ◇1List.SeqCount=1;
  ◇1.SeqIndex=1;
END;

RULE: ◇1List ::= ◇1List ',' ◇1
COMPUTE
  ◇1List[1].SeqCount=◇1.SeqIndex;
  ◇1.SeqIndex=ADD(◇1List[2].SeqCount,1);
END;
```

This macro is invoked in definitions 91, 190, 226, and 236.

The statement function ambiguity gives rise to a complex situation on the left-hand side of an assignment. In that case the nature of the list may change partway along, and it may be impossible to group the components of a single element properly. This results in a general list of expressions:

Sequence[4] ≡

```
RULE: xExprList ::= xExpr
COMPUTE
  xExprList.SeqCount=1;
END;

RULE: xExprList ::= ':'
COMPUTE
  xExprList.SeqCount=1;
END;

RULE: xExprList ::= ':' xExpr
COMPUTE
  xExprList.SeqCount=1;
END;

RULE: xExprList ::= xExpr ':'
COMPUTE
  xExprList.SeqCount=1;
END;

RULE: xExprList ::= xExpr ':' xExpr
COMPUTE
  xExprList.SeqCount=1;
END;

RULE: xExprList ::= xExprList ',' xSectionSubscript
COMPUTE
  xExprList[1].SeqCount=xSectionSubscript.SeqIndex;
  xSectionSubscript.SeqIndex=ADD(xExprList[2].SeqCount,1);
END;
```

```
RULE: xExprList ::= xSFDummyArgNameList ',' xExpr
COMPUTE
  xExprList.SeqCount=ADD(xSFDummyArgNameList.SeqCount,1);
END;

RULE: xExprList ::= xSFDummyArgNameList ',' ':'
COMPUTE
  xExprList.SeqCount=ADD(xSFDummyArgNameList.SeqCount,1);
END;

RULE: xExprList ::= xSFDummyArgNameList ',' ':' xExpr
COMPUTE
  xExprList.SeqCount=ADD(xSFDummyArgNameList.SeqCount,1);
END;

RULE: xExprList ::= xSFDummyArgNameList ',' xExpr ':'
COMPUTE
  xExprList.SeqCount=ADD(xSFDummyArgNameList.SeqCount,1);
END;

RULE: xExprList ::= xSFDummyArgNameList ',' xExpr ':' xExpr
COMPUTE
  xExprList.SeqCount=ADD(xSFDummyArgNameList.SeqCount,1);
END;
```

This macro is defined in definitions 2 and 4.
This macro is invoked in definition 244.

2.2 Syntactic Items

The definition of syntactic items in terms of letters, digits and special characters of the FORTRAN character set are the concern of the phrase structure definition module. Here we are concerned with the grammar symbols representing those syntactic items, and the attributes attached to them by this specification:

Syntactic Items[5] ≡

> *Symbolic Name Characteristics*[6]
> *Statement Label Characteristics*[15]

This macro is invoked in definition 1.

2.2.1 Symbolic Name Characteristics

`SymbolicName`, which does not appear in the grammar, embodies the FORTRAN concept of a symbolic name. Computations associated with `SymbolicName` must be carried out for *every* context in which a symbolic name appears.

The following attributes of symbolic names are established by this specification. All of these attributes are attached to each individual occurrence of a symbolic name in the program:

Symbolic Name Characteristics[6] ≡

```
SYMBOL SymbolicName:
  Sym: int,
  Coord: CoordPtr,
  ObjectKey: DefTableKey,
  UnitKey: DefTableKey;
```

This macro is defined in definitions 6 and 14.
This macro is invoked in definition 5.

`Sym` gives the index in the string table of the character sequence representing the symbolic name. The `StringTable` function exported by the Character Storage module of the Eli library returns a pointer to the actual character string. `Sym` is used in constructing reports, and in computations that need access to the characters making up the name.

`Coord` gives the position of this instance of the symbolic name in the original source text of the program. The type `CoordPtr` is exported by the Error module of the Eli library. `Coord` is used to attach reports to a particular instance of a symbolic name.

Symbolic names identify entities like variables, arrays, and functions. A particular identification holds over a region of the program called the *scope* of the identification. The value of the `ObjectKey` attribute of a specific occurrence of a symbolic name represents the entity identified by that occurrence. The type `DefTableKey` is exported by the Definition Table module of the Eli library. It allows access to any of the properties of the entity it represents.

Properties accessible via the ObjectKey attribute[7] ≡

```
Value: int;
Intrinsic: int;
```

This macro is invoked in definition 243.

The `Value` property is associated with each symbolic constant; other entities do not have this property. It is the string table index of a denotation giving the value of the symbolic constant. Queries should be provided with the default value `0`, the string table index of the null string.

The `Intrinsic` property value `1` is associated with entities representing FORTRAN intrinsic functions; other entities do not have this property. Queries should be provided with the default value `0`.

All of the entities identified by a given symbolic name within a program unit share certain properties. The value of the `UnitKey` attribute allows access to these shared properties.

Properties accessible via the UnitKey attribute[8] ≡

```
KindSet: IntSet;
Type: DefTableKey;
Length: int;
StorageUnits: int;
```

This macro is defined in definitions 8, 16, and 19.
This macro is invoked in definition 243.

The `KindSet` property specifies the classification of the symbolic name. It is a set because a particular symbolic name may legally be classified in several different ways within a program unit. The elements of a `KindSet` are members of an enumeration whose elements are roughly the classes of symbolic names given in Section 18 of the standard:

KindSet elements[9] ≡

```
typedef enum {
  CommonBlock,
  ExternalFunction,
  Subroutine,
  MainProgram,
  BlockDataSubprogram,
  Array,
  Variable,
  Constant,
  StatementFunction,
  IntrinsicFunction,
  DummyProcedure,
```

This macro is defined in definitions 9 and 10.
This macro is invoked in definition 248.

Two additional elements are needed to accumulate information during classification:

KindSet elements[10] ≡

```
  InExternalStmt,
  DummyArgument,
} KindSetElement;
```

This macro is defined in definitions 9 and 10.
This macro is invoked in definition 248.

The `KindSet` is implemented using an Eli library module:

Instantiate the integer set property module[11] ≡

```
$/Prop/KindSet.gnrc :inst
```

This macro is invoked in definition 242.

The usual query applied to the `KindSet` property of a symbolic name asks whether that name has a specific classification:

Classified as[12](◇2) ≡

`InIS(◇1,GetKindSet(◇2.UnitKey,NullIS()))` This macro is invoked in definitions 151, 213, 223, 226, 236, 238, 239, and 241.

`NullIS()` is an empty set, and `InIS(e,s)` returns nonzero if and only if element `e` is a member of set `s`.

The `Type` property specifies the FORTRAN data type of the entity represented by the symbolic name. Not all symbolic names have a `Type` property. The following definition table keys are possible values of the `Type` property:

Data Types[13] ≡

```
IntegerType;
RealType;
DoublePrecisionType;
ComplexType;
```

```
    LogicalType;
    CharacterType;
    ErrorType;
```

This macro is invoked in definition 243.

The `Length` property specifies the number of characters in an entity of character type. It is the string table index of a string whose apparent integer value is the number of characters. A string table index of 0 indicates the absence of a `Length`. Only symbolic names having the `Type` property with the `CharacterType` value have a `Length` property.

The `StorageUnits` property specifies the number if storage units occupied by the entity.

Symbolic names appear in a wide variety of contexts, which reflect different views of those names and are therefore represented in the grammar by distinct nonterminals that may have distinct computations. Each of these contexts embodies the FORTRAN concept of a symbolic name, however, so it must inherit any computations performed for all symbolic names:

Symbolic Name Characteristics[14] $\equiv$

```
SYMBOL xBlockDataName           INHERITS SymbolicName END;
SYMBOL xCommonBlockName         INHERITS SymbolicName END;
SYMBOL xDummyArgName            INHERITS SymbolicName END;
SYMBOL xEntryName               INHERITS SymbolicName END;
SYMBOL xExternalName            INHERITS SymbolicName END;
SYMBOL xFunctionName            INHERITS SymbolicName END;
SYMBOL xImpliedDoVariable       INHERITS SymbolicName END;
SYMBOL xIntrinsicProcedureName  INHERITS SymbolicName END;
SYMBOL xName                    INHERITS SymbolicName END;
SYMBOL xNamedConstant           INHERITS SymbolicName END;
SYMBOL xNamedConstantUse        INHERITS SymbolicName END;
SYMBOL xObjectName              INHERITS SymbolicName END;
SYMBOL xProgramName             INHERITS SymbolicName END;
SYMBOL xSFDummyArgName          INHERITS SymbolicName END;
SYMBOL xSFVarName               INHERITS SymbolicName END;
SYMBOL xSubroutineName          INHERITS SymbolicName END;
SYMBOL xSubroutineNameUse       INHERITS SymbolicName END;
SYMBOL xVariableName            INHERITS SymbolicName END;
```

This macro is defined in definitions 6 and 14.
This macro is invoked in definition 5.

Any computation performed for all symbolic names may be overridden in a specific context by attaching a computation of the same value to the grammar symbol representing that context.

2.2.2 Statement Label Characteristics

`Label`, which does not appear in the grammar, embodies the FORTRAN concept of a statement label. Computations associated with `Label` must be carried out for *every* context in which a statement label appears.

Any computation performed for all statement labels may be overridden in a specific context by attaching a computation of the same value to the grammar symbol representing that context.

The following attributes of statement labels are established by this specification. All of these attributes are attached to each individual occurrence of a statement label in the program:

Statement Label Characteristics[15] ≡

```
SYMBOL Label:
  Sym: int,
  UnitKey: DefTableKey;
```

This macro is defined in definitions 15 and 17.
This macro is invoked in definition 5.

`Sym` gives the index in the string table of the character sequence representing the statement label. The `StringTable` function exported by the Character Storage module of the Eli library of the Eli library returns a pointer to the actual character string. `Sym` is used in constructing reports, and in computations that need access to the characters making up the label.

The value of the `UnitKey` attribute of a specific occurrence of a statement label allows access to any of the properties of that label. The type `DefTableKey` is exported by the Definition Table module of the Eli library. A `UnitKey` attribute value of `NoKey` indicates that the label has not been defined.

Properties accessible via the UnitKey attribute[16] ≡

```
Executable: int;
```

This macro is defined in definitions 8, 16, and 19.
This macro is invoked in definition 243.

The `Executable` property value `1` is associated with labels of executable statements; `0` is associated with labels of nonexecutable statements.

Statement labels appear in two contexts, which reflect different views of those labels and are therefore represented in the grammar by distinct nonterminals that may have distinct computations. Each of these contexts embodies the FORTRAN concept of a statement label, however, so it must inherit any computations performed for all statement labels:

Statement Label Characteristics[17] ≡

```
SYMBOL xLblDef INHERITS Label END;
SYMBOL xLblRef INHERITS Label END;
```

This macro is defined in definitions 15 and 17.
This macro is invoked in definition 5.

2.3 Program Units and Procedures

A program unit is a main program, function subprogram, subroutine subprogram or block data subprogram.

Program Units and Procedures[18] ≡

```
SYMBOL xProgramUnit:
  Kind: KindSetElement,
  ClassificationDone: VOID,
  GotAllTypes: VOID,
  GotAllDims: VOID,
  GotAllExecs: VOID;
```

This macro is defined in definitions 18.
This macro is invoked in definition 1.

The `Kind` attribute reflects what kind of program unit the `xProgramUnit` phrase represents. Its value is the classification of the symbolic name of the program unit. (If the program unit has no symbolic name then the value of the `Kind` attribute is `MainProgram`.)

The `ClassificationDone` attribute is an assertion that the `KindSet` property of every entity is valid. Computations that query the `KindSet` property should depend on this attribute unless the `KindSet` property of the specific entity being tested is known to be valid on other grounds.

The `GotAllTypes` attribute is an assertion that the `Type` property of every entity is valid. Computations that query the `Type` property should depend on this attribute unless the `Type` property of the specific entity being tested is known to be valid on other grounds.

The `GotAllDims` attribute is an assertion that the `NumberOfDims` property of every array is valid. Computations that query the `NumberOfDims` property should depend on this attribute unless the `NumberOfDims` property of the specific entity being tested is known to be valid on other grounds.

The `GotAllExecs` attribute is an assertion that the `Executable` property of every label is valid.Computations that query the `Executable` property should depend on this attribute unless the `Executable` property of the specific label being tested is known to be valid on other grounds.

2.4 Array

An array has a fixed number of dimensions:

Properties accessible via the UnitKey attribute[19] ≡

```
NumberOfDims: int;
```

This macro is defined in definitions 8, 16, and 19.
This macro is invoked in definition 243.

2.5 Reference

A variable, array element, or substring reference is the appearance of a variable, array element, or substring name, respectively, in a statement in a context requiring the value of that entity to be used during the execution of the program.

A procedure reference is the appearance of a procedure name in a statement in a context that requires the actions specified by the procedure to be executed during the execution of the executable program.

References are represented in this document by the computational role `Reference`.

Reference[20] ≡

```
CLASS SYMBOL Reference END;
```

This macro is defined in definitions 20.
This macro is invoked in definition 42.

2.6 Storage

A *storage sequence* is a sequence of storage units. A *storage unit* is either a numeric storage unit or a character storage unit.

Storage[21] ≡

```
NumericStorage;
CharacterStorage;
InvalidStorage;
```

This macro is defined in definitions 21, 22, and 23.
This macro is invoked in definition 243.

An integer, real, or logical datum has one numeric storage unit in a storage sequence. A double precision or complex datum has two numeric storage units in a storage sequence. A character datum has one character storage unit in a storage sequence for each character in the datum.

Storage[22] ≡

```
UnitsPerDatum: int;
StorageUnitKind: DefTableKey;

IntegerType         -> UnitsPerDatum={1}, StorageUnitKind={NumericStorage};
RealType            -> UnitsPerDatum={1}, StorageUnitKind={NumericStorage};
DoublePrecisionType -> UnitsPerDatum={2}, StorageUnitKind={NumericStorage};
ComplexType         -> UnitsPerDatum={2}, StorageUnitKind={NumericStorage};
LogicalType         -> UnitsPerDatum={1}, StorageUnitKind={NumericStorage};
CharacterType       -> UnitsPerDatum={1}, StorageUnitKind={CharacterStorage};
ErrorType           -> UnitsPerDatum={1}, StorageUnitKind={InvalidStorage};
```

This macro is defined in definitions 21, 22, and 23.
This macro is invoked in definition 243.

The storage requirements for entities must be computed using safe arithmetic, and therefore must be represented as string table indices. `StorageUnits` is the property used for this purpose.

Storage[23] ≡

```
StorageUnits: int;
```

This macro is defined in definitions 21, 22, and 23.
This macro is invoked in definition 243.

Computation of the number of storage units will always depend ultimately on the number of storage units occupied by the basic type of the object.

Evaluation functions[24] ≡

```
int
#ifdef PROTO_OK
SizeOfType(DefTableKey key)
#else
SizeOfType(key) DefTableKey key;
#endif
{ int units = GetUnitsPerDatum(GetType(key, key), 0);

  return (units == 0 ? 0 : IdnNumb(0, units));
}
```

This macro is defined in definitions 24, 98, 111, 174, 181, 184, and 186.

This macro is invoked in definition 247.

Evaluation function interfaces[25] ≡

```
int SizeOfType ELI_ARG((DefTableKey));
```

This macro is defined in definitions 25, 99, 110, 112, 182, 185, and 187.
This macro is invoked in definition 248.

3 Characters, Lines and Execution Sequence

Section 3 of the standard defines the gross structure of the program. Since this specification is concerned only with the extraction of symbol table information, only the following subsections are relevant:

Characters, Lines and Execution Sequence[26] ≡

Statement Labels[27]
Order of Statements and Lines[36]

This macro is invoked in definition 244.

Most of the remaining subsections of the standard are implemented by the specification module defining the phrase structure.

3.1 Statement Labels

According to Section 3.4 of the standard, only labeled executable statements and FORMAT statements may be referred to by the use of statement labels. This restriction can be enforced by the specifications of individual statements, provided that properties of a label can be established at the point of definition and queried at the point of use.

Here is a consistent renaming specification that associates a unique `DefTableKey`-valued attribute named `UnitKey` with every occurrence of a given label in a program unit. Occurrences of distinct labels, and occurrences of the same label in distinct program units, will be associated with distinct `DefTableKey` values:

Statement Labels[27] ≡

```
SYMBOL xSourceFile INHERITS UnitRootScope END;
SYMBOL xProgramUnit INHERITS UnitRangeScope END;
SYMBOL xLblDef INHERITS UnitIdDefScope END;
SYMBOL xLblRef INHERITS UnitIdUseEnv END;
```

This macro is defined in definitions 27, 29, 31, and 32.
This macro is invoked in definition 26.

The nonterminal `xLblDef` represents the context of a label definition, and the nonterminal `xLblRef` represents the context of a label use.

This consistent renaming specification uses the Eli library module implementing scopes that are the closest-containing range (these are the scope rules of ALGOL 60):

Instantiate the ALGOL 60 scope module[28] ≡

```
$/Name/AlgScope.gnrc +instance=Unit +referto=Unit :inst
```

This macro is defined in definitions 28, 201, and 207.
This macro is invoked in definition 242.

The form of a statement label is a sequence of 1 to 5 digits, one of which must be nonzero. Moreover, blanks and leading zeros in this sequence are not significant in distinguishing between labels. This specification therefore recognizes a label as a decimal integer constant, identifying it as a label only by the context in which it appears:

Statement Labels[29] ≡

```
SYMBOL xLabel: Sym: int;

RULE: xLabel ::= xIcon
COMPUTE
  IF(EQ(xIcon,0),
    message(ERROR,"Zero is not a valid label",0,COORDREF));
  IF(GT(xIcon,99999),
    message(ERROR,"Labels must be 1 to 5 digits",0,COORDREF));
  xLabel.Sym=IdnNumb(0,xIcon);
END;
```

This macro is defined in definitions 27, 29, 31, and 32.
This macro is invoked in definition 26.

The function `IdnNumb` generates a symbol consisting of the string indexed by its first argument and the number given by its second. Since the null string is indexed by 0, this invocation of `IdnNumb` produces a symbol whose string is the sequence of decimal digits (with no leading zeros or embedded blanks) representing the value of the label.

`IdnNumb` is exported by the Eli `MakeName` library module, which must therefore be instantiated:

Instantiate the MakeName library module[30] ≡

```
$/Tech/MakeName.gnrc +instance=xIdent :inst
```

This macro is invoked in definition 242.

Here `xIdent` is the terminal symbol used in the grammar to represent an identifier. When the module creates a symbol, it creates that symbol as though it had been recognized as an instance of the terminal symbol `xIdent`.

Statement Labels[31] ≡

```
RULE: xLblDef ::= xLabel
COMPUTE
  xLblDef.Sym=xLabel.Sym;
END;

RULE: xLblDef ::=
COMPUTE
  xLblDef.Sym=0;
  xLblDef.UnitKey=NoKey;
END;
```

```
RULE: xLblRef ::= xLabel
COMPUTE
  xLblRef.Sym=xLabel.Sym;
END;
```

This macro is defined in definitions 27, 29, 31, and 32.
This macro is invoked in definition 26.

The restriction that the same statement label must not be given to more than one statement in a program unit is typical of a number of such restrictions involving multiple occurrences of symbolic names:

Statement Labels[32] ≡

```
SYMBOL xProgramUnit COMPUTE
  SYNT.GotAllAppearances=
    CONSTITUENTS (xLblDef.GotAppearance,SymbolicName.GotAppearance);
END;
```

Report multiple[33](‘`xLblDef`’,‘`label`’)

```
SYMBOL SymbolicName COMPUTE
  INH.GotAppearance=1;
END;
```

This macro is defined in definitions 27, 29, 31, and 32.
This macro is invoked in definition 26.

Such a restriction is enforced by setting a property of the entity to `1` the first time and to `2` any subsequent time that the entity appears in a given context, and then complaining if the property has the value `2`. The attribute `GotAppearance` asserts that the property has been set at a specific occurrence, while `xProgramUnit.GotAllAppearances` asserts that *all* occurrences in the program unit have been seen. Since not all contexts containing symbolic names are relevant for detecting multiple occurrences, there will be some that have no explicit computation of the `GotAppearance` attribute. The last symbol computation above ensures that `SymbolicName.GotAppearance` will be defined everywhere; this computation is overridden by computations attached to specific nonterminals inheriting from `SymbolicName`, as well as by computations attached to rules.

Here is a macro that defines an overriding computation. The macro's first argument specifies the grammar symbol defining the occurrences to be checked, the second specifies the property to be updated:

Report multiple[33](◇2) ≡

```
SYMBOL ◇1 COMPUTE
  INH.GotAppearance=Set◇2App(THIS.UnitKey,1,2);
  IF(EQ(Get◇2App(THIS.UnitKey,0),2),
    message(ERROR,"Multiply-defined ◇2",0,COORDREF))
  <- INCLUDING xProgramUnit.GotAllAppearances;
END;
```

This macro is invoked in definitions 32, 121, 136, 147, and 149.

`Report multiple` counts every occurrence of the specified symbol. If only the occurrences in certain contexts should be counted, a rule computation must be added to the rules defining those contexts:

Rule multiple[34](◇2) ≡

```
◇1.GotAppearance=Set◇2App(◇1.UnitKey,1,2);
IF(EQ(Get◇2App(◇1.UnitKey,0),2),
  message(ERROR,"Multiply-defined ◇2",0,COORDREF))
<- INCLUDING xProgramUnit.GotAllAppearances;
```

This macro is invoked in definition 49.

The arguments of this macro are identical to those of `Report multiple`.

In addition to the operations, a property must be defined for each test:

Properties supporting multiple definition reporting[35] ≡

```
labelApp: int;
```

This macro is defined in definitions 35, 50, 122, 137, 148, and 150.
This macro is invoked in definition 243.

3.2 Order of Statements and Lines

Section 3.5 of the standard defines the required order of statements for a program unit.

The restrictions on PROGRAM, FUNCTION, SUBROUTINE and BLOCK DATA statements are enforced by the grammar. Conditions (1) through (5), as well as the restriction on IMPLICIT statements, are enforced by the specification described below. Proper relative ordering of IMPLICIT and PARAMETER statements is enforced by the specifications of those statements. The restriction on the END statement is enforced by the grammar.

Figure 1 from Section 3.5 of the standard shows that the program can be divided into six segments, with a certain set of statements allowed in each segment. Statements that may appear in more than one of these segments are always restricted to a contiguous subset.

This specification is implemented by defining an integer-valued chain, `Order`, which contains the current segment number at each statement.

Order of Statements and Lines[36] ≡

```
CHAIN Order: int;

SYMBOL xProgramUnit COMPUTE
  CHAINSTART HEAD.Order=0;
END;
```

This macro is defined in definitions 36, 38, 39, and 40.
This macro is invoked in definition 26.

The chain value leaving a statement is the larger of two values: the first segment number in which that statement could appear, and the current segment number. Thus a statement will advance the current segment if that statement cannot appear until a segment following the current one.

A statement is out of order if the last segment in which it could appear is earlier than the current segment:

Seq[37](◇3) ≡

```
RULE: xStmt ::= xLblDef ◇1 xEOS
```

```
COMPUTE
  xStmt.Order=
    LaterOf(◇2,xStmt.Order) <- CONSTITUENTS SymbolicName.GotContext;
  IF(LE(◇3,xStmt.Order),
    message(ERROR,"Position violates the standard, Section 3.5",0,COORDREF));
END;
```

This macro is invoked in definitions 38, 39, and 40.

In addition to verifying the correct statement sequence, `Order` is used to enforce text-order dependence in initial classification. The `GotContext` attribute of `SymbolicName` asserts that the initial classification of the symbol has been updated by this information. Thus, because of the `<-` clause, `Order` asserts that all initial classifications reflect all information available up to this point.

Figure 1 of Section 3.5 of the standard is encoded by specifying, for every statement, the first segment in which that statement may appear and the first segment in which it may *not* appear. (The segment containing IMPLICIT statements is numbered 1; the segment containing the END statement is numbered 5.)

Comment lines are removed during scanning, and hence do not appear in this specification.

Order of Statements and Lines[38] ≡

```
Seq[37](‘’format’ ’(’ xFmtSpec ’)’                ’,‘ 1’,‘ 5’)
Seq[37](‘’entry’ xEntryName xFormalParameterList              ’,‘ 1’,‘ 5’)

Seq[37](‘’parameter’ ’(’ xNamedConstantDefList ’)’ ’,‘ 1’,‘ 3’)

Seq[37](‘’data’ xDatalist                         ’,‘ 3’,‘ 5’)

Seq[37](‘’implicit’ xImplicitSpecList             ’,‘ 1’,‘ 2’)

Seq[37](‘’common’ xComlist                        ’,‘ 2’,‘ 3’)
Seq[37](‘’dimension’ xArrayDeclaratorList         ’,‘ 2’,‘ 3’)
Seq[37](‘’equivalence’ xEquivalenceSetList        ’,‘ 2’,‘ 3’)
Seq[37](‘’external’ xExternalNameList             ’,‘ 2’,‘ 3’)
Seq[37](‘’intrinsic’ xIntrinsicList               ’,‘ 2’,‘ 3’)
Seq[37](‘’save’                                   ’,‘ 2’,‘ 3’)
Seq[37](‘’save’ xSavedEntityList                  ’,‘ 2’,‘ 3’)
Seq[37](‘xTypeSpec xEntityDeclList                ’,‘ 2’,‘ 3’)
```

This macro is defined in definitions 36, 38, 39, and 40.
This macro is invoked in definition 26.

Statement functions are a special case, because what is recognized syntactically is a *possible* statement function definition. The check for an ordering error must be based on the classification of the symbolic name:

Order of Statements and Lines[39] ≡

```
RULE: xStmt ::= xLblDef xName xStmtFunctionRange
COMPUTE
  xStmt.Order=
    LaterOf(
      IF(This is a statement function statement[237],3,4),
      xStmt.Order)
    <- CONSTITUENTS SymbolicName.GotContext;
  IF(LE(
      IF(This is a statement function statement[237],4,5),
      xStmt.Order),
    message(ERROR,"Position violates the standard, Section 3.5",0,COORDREF));
END;
```

```
Seq[37](‘’assign’ xLblRef ’to’ xVariableName             ’,‘ 4’,‘ 5’)
Seq[37](‘’backspace’ ’(’ xIoControlSpecList ’)’          ’,‘ 4’,‘ 5’)
Seq[37](‘’backspace’ xUnitIdentifier                     ’,‘ 4’,‘ 5’)
Seq[37](‘’call’ xSubroutineNameUse ’(’ xArgList ’)’      ’,‘ 4’,‘ 5’)
Seq[37](‘’call’ xSubroutineNameUse                       ’,‘ 4’,‘ 5’)
Seq[37](‘’close’ ’(’ xIoControlSpecList ’)’              ’,‘ 4’,‘ 5’)
Seq[37](‘’continue’                                      ’,‘ 4’,‘ 5’)
Seq[37](‘’do’ xLblRef xLoopControl                       ’,‘ 4’,‘ 5’)
Seq[37](‘’else’                                          ’,‘ 4’,‘ 5’)
Seq[37](‘’elseif’ ’(’ xExpr ’)’ ’then’                   ’,‘ 4’,‘ 5’)
Seq[37](‘’endfile’ ’(’ xIoControlSpecList ’)’            ’,‘ 4’,‘ 5’)
Seq[37](‘’endfile’ xUnitIdentifier                       ’,‘ 4’,‘ 5’)
Seq[37](‘’if’ ’(’ xExpr ’)’ ’then’                       ’,‘ 4’,‘ 5’)
Seq[37](‘’if’ ’(’ xExpr ’)’ xLblRef ’,’ xLblRef ’,’ xLblRef’,‘ 4’,‘ 5’)
```

This macro is defined in definitions 36, 38, 39, and 40.
This macro is invoked in definition 26.

Logical IF statements are also a special case, because they contain two instances of `xStmt` and are not terminated by `xEOS`. Thus they do not quite fit the macro definition:

Order of Statements and Lines[40] ≡

```
RULE: xStmt ::= xLblDef ’if’ ’(’ xExpr ’)’ xStmt
COMPUTE
  xStmt[1].Order=
    LaterOf(4,xStmt[1].Order) <- CONSTITUENTS SymbolicName.GotContext;
  IF(LE(5,xStmt[1].Order),
    message(ERROR,"Position violates the standard, Section 3.5",0,COORDREF));
END;
```

```
Seq[37](‘’inquire’ ’(’ xIoControlSpecList ’)’            ’,‘ 4’,‘ 5’)
Seq[37](‘’open’ ’(’ xIoControlSpecList ’)’               ’,‘ 4’,‘ 5’)
Seq[37](‘’pause’                                         ’,‘ 4’,‘ 5’)
Seq[37](‘’pause’ xIcon                                   ’,‘ 4’,‘ 5’)
Seq[37](‘’pause’ xScon                                   ’,‘ 4’,‘ 5’)
Seq[37](‘’print’ xFormatIdentifier ’,’ xOutputItemList   ’,‘ 4’,‘ 5’)
Seq[37](‘’print’ xFormatIdentifier                       ’,‘ 4’,‘ 5’)
Seq[37](‘’read’ xFormatIdentifier ’,’ xInputItemList     ’,‘ 4’,‘ 5’)
Seq[37](‘’read’ xFormatIdentifier                        ’,‘ 4’,‘ 5’)
Seq[37](‘’read’ xIoControlSpec                           ’,‘ 4’,‘ 5’)
Seq[37](‘’read’ xIoControlSpec xInputItemList            ’,‘ 4’,‘ 5’)
Seq[37](‘’return’                                        ’,‘ 4’,‘ 5’)
Seq[37](‘’return’ xExpr                                  ’,‘ 4’,‘ 5’)
Seq[37](‘’rewind’ ’(’ xIoControlSpecList ’)’             ’,‘ 4’,‘ 5’)
Seq[37](‘’rewind’ xUnitIdentifier                        ’,‘ 4’,‘ 5’)
```

Seq[37](‘`'stop'`’,‘ 4’,‘ 5’)
Seq[37](‘`'stop' xIcon`’,‘ 4’,‘ 5’)
Seq[37](‘`'stop' xScon`’,‘ 4’,‘ 5’)
Seq[37](‘`'write' '(' xIoControlSpecList ')'`’,‘ 4’,‘ 5’)
Seq[37](‘`'write' '(' xIoControlSpecList ')' xOutputItemList`’,‘ 4’,‘ 5’)
Seq[37](‘`GoToKw '(' xLblRefList ')' xExpr`’,‘ 4’,‘ 5’)
Seq[37](‘`GoToKw xLblRef`’,‘ 4’,‘ 5’)
Seq[37](‘`GoToKw xVariableName '(' xLblRefList ')'`’,‘ 4’,‘ 5’)
Seq[37](‘`GoToKw xVariableName`’,‘ 4’,‘ 5’)
Seq[37](‘`xName '(' xExprList ')' '=' xExpr`’,‘ 4’,‘ 5’)
Seq[37](‘`xName '(' xExprList ')' xSubstringRange '=' xExpr`’,‘ 4’,‘ 5’)
Seq[37](‘`xName '=' xExpr`’,‘ 4’,‘ 5’)

This macro is defined in definitions 36, 38, 39, and 40.
This macro is invoked in definition 26.

4 Data Types and Constants

Data Types and Constants[41] ≡

Data Type of a Name[42]
Constants[46]

This macro is invoked in definition 244.

4.1 Data Types

4.1.1 Data Type of a Name

A symbolic name representing a constant, variable, array, or function must have only one type for each program unit. Once a particular name is identified with a particular type in a program unit, that type is implied for any usage in the program unit that requires a type.

Data Type of a Name[42] ≡

```
CLASS SYMBOL TypeName COMPUTE
  IF(EQ(GetType(THIS.UnitKey,NoKey),ErrorType),
    message(ERROR,"Defined with different types",0,COORDREF))
    <- INCLUDING xProgramUnit.GotAllTypes;
END;

SYMBOL xObjectName   INHERITS TypeName END;
SYMBOL xFunctionName INHERITS TypeName END;

CLASS SYMBOL Reference COMPUTE
  SYNT.Type=
    IF(GetIntrinsic(THIS.ObjectKey,0),
      GuaranteeType(THIS.UnitKey,GetType(THIS.ObjectKey,NoKey)),
      GuaranteeType(THIS.UnitKey,DefaultType(THIS.Sym)))
    <- INCLUDING xProgramUnit.GotAllTypeDefs;
  SYNT.Length=
    IF(NE(SYNT.Type,CharacterType),0,
    IF(GetIntrinsic(THIS.ObjectKey,0),
      GuaranteeLength(THIS.UnitKey,MakeName("1")),
      GuaranteeLength(THIS.UnitKey,DefaultLength(THIS.Sym))))
    <- INCLUDING xProgramUnit.GotAllLengths;
END;
```

Reference[20]

```
SYMBOL xVariableName     INHERITS Reference END;
SYMBOL xName             INHERITS Reference END;
SYMBOL xNamedConstantUse INHERITS Reference END;
SYMBOL xSFVarName        INHERITS Reference END;
```

This macro is invoked in definition 41.

The contexts in which a symbolic name is given a type inherit the computational role `TypeName`; those in which usage of the name requires a type inherit the computational role `Reference`.

Properties for type analysis[43] ≡

```
Type: DefTableKey [Guarantee];
Length: int [Guarantee];

TYPE Guarantee(DefTableKey key, TYPE val)
{ if (key == NoKey) return val;
  if (!ACCESS) VALUE = val;
  return VALUE;
}
```

This macro is defined in definitions 43 and 119.
This macro is invoked in definition 243.

4.1.2 Type Rules for Data and Procedure Identifiers

In the absence of an explicit declaration in a type-statement, the type is implied by the first letter of the name.

Return the default type associated with the first letter of a name[44] ≡

```
#define DefaultType(Sym) (TypeFor[*(StringTable(Sym)) - 'a'])
#define DefaultLength(Sym) (LengthOf[*(StringTable(Sym)) - 'a'])
```

This macro is invoked in definition 246.

A first letter of I, J, K, L, M, or N implies type integer and any other letter implies real, unless an IMPLICIT statement is used to change the default implied type.

Set default implicit types[45] ≡

```
{ int i;
  for (i = 0; i <= 'z'-'a'; i++) TypeFor[i] = RealType;
  for (i = 'i'-'a'; i <= 'n'-'a'; i++) TypeFor[i] = IntegerType;
}
```

This macro is invoked in definition 134.

4.2 Constants

The form of the string representing a constant specifies both its value and its data type.

Constants[46] ≡

> *Arithmetic constant type*[47](‘`xIcon`’,‘`IntegerType`’)
> *Arithmetic constant type*[47](‘`xRcon`’,‘`RealType`’)
> *Arithmetic constant type*[47](‘`xDcon`’,‘`DoublePrecisionType`’)
> *Arithmetic constant type*[47](‘`xComplexConst`’,‘`ComplexType`’)

This macro is invoked in definition 41.

Arithmetic constant type[47](◇2) ≡

```
RULE: xUnsignedArithmeticConstant ::= ◇1
COMPUTE
  xUnsignedArithmeticConstant.TypeCode=OilType◇2;
END;
```

This macro is invoked in definition 46.

5 Arrays and Substrings

An array is a nonempty sequence of data.

Arrays and Substrings[48] ≡

> *Array Declarator*[49]
> *Kinds and Occurrences of Array Declarators*[54]
> *Properties of an Array*[57]
> *Array Element Name*[65]
> *Subscript*[67]
> *Character Substring*[74]

This macro is invoked in definition 244.

5.1 Array Declarator

The nonterminal `xArrayDeclarator` represents the context of an array declarator in a DIMENSION or COMMON statement; `xEntityDecl`represents the context of an array declarator in a type statement.

Only one array declarator for an array name is permitted in a program unit:

Array Declarator[49] ≡

```
ATTR UnitKey: DefTableKey;

RULE: xArrayDeclarator ::= xVariableName '(' DimensionDeclarators ')'
COMPUTE
  Rule multiple[34](‘xVariableName’,‘array’)
  DimensionDeclarators.UnitKey=xVariableName.UnitKey;
END;

RULE: xEntityDecl ::= xObjectName '(' DimensionDeclarators ')'
COMPUTE
  Rule multiple[34](‘xObjectName’,‘array’)
  DimensionDeclarators.UnitKey=xObjectName.UnitKey;
END;

RULE: xEntityDecl ::= xObjectName '(' DimensionDeclarators ')' '*' xCharLength
COMPUTE
  Rule multiple[34](‘xObjectName’,‘array’)
  DimensionDeclarators.UnitKey=xObjectName.UnitKey;
END;
```

This macro is defined in definitions 49, 51, 52, and 53.
This macro is invoked in definition 48.

Properties supporting multiple definition reporting[50] ≡

```
arrayApp: int;
```

This macro is defined in definitions 35, 50, 122, 137, 148, and 150.
This macro is invoked in definition 243.

The number of dimensions of the array is the number of dimension declarators in the array declarator. The minimum number of dimensions is one (guaranteed by the grammar) and the maximum is seven.

Array Declarator[51] ≡

```
RULE: DimensionDeclarators ::= xArraySpec
COMPUTE
  IF(GT(xArraySpec.Dims,7),
    message(ERROR,"Maximum number of dimensions is 7",0,COORDREF));
END;
```

This macro is defined in definitions 49, 51, 52, and 53.
This macro is invoked in definition 48.

The nonterminals `xLowerBound` and `xUpperBound` represent the appearance of an expression as a dimension bound. According to the rules of FORTRAN expression evaluation, all of the components of an expression must be of type integer if the value yielded by that expression is of type integer.

Array Declarator[52] ≡

```
RULE: xLowerBound ::= xExpr
COMPUTE
  IF(NE(xExpr.TypeCode,OilTypeIntegerType),
    message(ERROR,"Bound must be an integer",0,COORDREF));
  xLowerBound.Value=xExpr.Value;
END;

RULE: xUpperBound ::= xExpr
COMPUTE
  IF(NE(xExpr.TypeCode,OilTypeIntegerType),
    message(ERROR,"Bound must be an integer",0,COORDREF));
  xUpperBound.Value=xExpr.Value;
END;
```

This macro is defined in definitions 49, 51, 52, and 53.
This macro is invoked in definition 48.

The upper dimension bound of the last dimension may be an asterisk in assumed-size array declarators.

Array Declarator[53] ≡

```
ATTR NotLast: int;

RULE: DimensionDeclarators ::= xArraySpec
COMPUTE
  xArraySpec.NotLast=0;
END;

SYMBOL xArraySpec COMPUTE
  INH.NotLast=1;
  IF(AND(THIS.NotLast,THIS.AssumedSize),
    message(ERROR,"Asterisk only in the last dimension",0,COORDREF));
END;
```

This macro is defined in definitions 49, 51, 52, and 53.
This macro is invoked in definition 48.

5.1.1 Kinds and Occurrences of Array Declarators

Each array declarator is either a constant array declarator, an adjustable array declarator, or an assumed-size array declarator. A constant array declarator is an array declarator in which each of the dimension bound expressions is an integer constant expression. The `Constant` attribute is true for a constant array declarator. An adjustable array declarator is an array declarator that contains one or more variables. The `Constant` and the `AssumedSize` attributes are false for an adjustable array declarator. An assumed-size array declarator is a constant array declarator or an adjustable array declarator, except that the upper bound of the last dimension is an asterisk. The `AssumedSize` attribute is true for an assumed-size array declarator.

Kinds and Occurrences of Array Declarators[54] ≡

```
ATTR Constant, AssumedSize: int;

SYMBOL xArraySpec COMPUTE
  SYNT.Constant=0;
  SYNT.AssumedSize=1;
END;

RULE: xArraySpec ::= xExplicitShapeSpec
COMPUTE
  xArraySpec.Constant=NE(xExplicitShapeSpec.SizeValue,0);
  xArraySpec.AssumedSize=0;
END;

RULE: xArraySpec ::= xArraySpec ',' xExplicitShapeSpec
```

```
COMPUTE
  xArraySpec[1].Constant=
    AND(xArraySpec[2].Constant,NE(xExplicitShapeSpec.SizeValue,0));
  xArraySpec[1].AssumedSize=0;
END;
```

This macro is defined in definitions 54, 55, and 56.
This macro is invoked in definition 48.

An actual array declarator is an array declarator in which the array name is not a dummy argument. Each actual array declarator must be a constant array declarator.

Kinds and Occurrences of Array Declarators[55] ≡

```
ATTR IsDummyArg: int;

RULE: DimensionDeclarators ::= xArraySpec
COMPUTE
  DimensionDeclarators.IsDummyArg=
    InIS(DummyArgument,GetKindSet(DimensionDeclarators.UnitKey,NullIS()))
    <- INCLUDING xProgramUnit.ClassificationDone;
  IF(AND(NOT(DimensionDeclarators.IsDummyArg),NOT(xArraySpec.Constant)),
    message(ERROR,"Must be a constant array declarator",0,COORDREF));
END;
```

This macro is defined in definitions 54, 55, and 56.
This macro is invoked in definition 48.

An actual array declarator is permitted in a DIMENSION statement, type-statement, or COMMON statement. A dummy array declarator is permitted in a DIMENSION statement or a type-statement, but not in a COMMON statement.

Kinds and Occurrences of Array Declarators[56] ≡

```
ATTR InCommonBlock: int;

RULE: xCommonBlockObject ::= xArrayDeclarator
COMPUTE
  xArrayDeclarator.InCommonBlock=1;
END;

SYMBOL xArrayDeclarator COMPUTE
  INH.InCommonBlock=0;
END;

RULE: xArrayDeclarator ::= xVariableName '(' DimensionDeclarators ')'
COMPUTE
  IF(AND(xArrayDeclarator.InCommonBlock,DimensionDeclarators.IsDummyArg),
    message(
      ERROR,
      "Dummy array declarator not allowed in a COMMON statement",
      0,
      COORDREF));
END;
```

This macro is defined in definitions 54, 55, and 56.
This macro is invoked in definition 48.

5.2 Properties of an Array

The number of dimensions of an array is specified by the array declarator, as are the size and bounds of each dimension and therefore the number of array elements.

Properties of an Array[57] ≡

Data Type of an Array and an Array Element[58]
Dimensions of an Array[59]
Size of an Array[62]
Array Storage Sequence[64]

This macro is invoked in definition 48.

5.2.1 Data Type of an Array and an Array Element

Data Type of an Array and an Array Element[58] ≡

This macro is invoked in definition 57.

5.2.2 Dimensions of an Array

`NumberofDims` is a property of the array symbol, and when testing it we need to be certain that it has been set. The void attribute `GotDim` represents the post-condition that the dimension property of a single symbol has been set.

Dimensions of an Array[59] ≡

```
RULE: DimensionDeclarators ::= xArraySpec
COMPUTE
  DimensionDeclarators.GotDim=
    ResetNumberOfDims(DimensionDeclarators.UnitKey,xArraySpec.Dims);
END;

SYMBOL xProgramUnit COMPUTE
  SYNT.GotAllDims=CONSTITUENTS DimensionDeclarators.GotDim;
END;
```

This macro is defined in definitions 59, 60, and 61.
This macro is invoked in definition 57.

The number of dimensions of an array is equal to the number of dimension declarators in the array declarator.

Dimensions of an Array[60] ≡

```
ATTR Dims: int;
```

```
RULE: xArraySpec ::= '*'
COMPUTE
  xArraySpec.Dims=1;
END;

RULE: xArraySpec ::= xLowerBound ':' '*'
COMPUTE
  xArraySpec.Dims=1;
END;

RULE: xArraySpec ::= xExplicitShapeSpec
COMPUTE
  xArraySpec.Dims=1;
END;

RULE: xArraySpec ::= xArraySpec ',' '*'
COMPUTE
  xArraySpec[1].Dims=ADD(xArraySpec[2].Dims,1);
END;

RULE: xArraySpec ::= xArraySpec ',' xLowerBound ':' '*'
COMPUTE
  xArraySpec[1].Dims=ADD(xArraySpec[2].Dims,1);
END;

RULE: xArraySpec ::= xArraySpec ',' xExplicitShapeSpec
COMPUTE
  xArraySpec[1].Dims=ADD(xArraySpec[2].Dims,1);
END;
```

This macro is defined in definitions 59, 60, and 61.
This macro is invoked in definition 57.

The size of a dimension is one larger than the difference between the upper and lower bounds:

Dimensions of an Array[61] ≡

```
ATTR SizeValue: int;

RULE: xExplicitShapeSpec ::= xLowerBound ':' xUpperBound
COMPUTE
  xExplicitShapeSpec.SizeValue=
    MakeName(
      stradd(
        strsub(
          StringTable(xUpperBound.Value),
          StringTable(xLowerBound.Value),
          10),
        "1",
        10));
  IF(AND(
      AND(xUpperBound.Value,xLowerBound.Value),
      NOT(xExplicitShapeSpec.SizeValue)),
    message(ERROR,"Error evaluating array size",0,COORDREF));
END;

RULE: xExplicitShapeSpec ::= xUpperBound
COMPUTE
  xExplicitShapeSpec.SizeValue=xUpperBound.Value;
END;
```

This macro is defined in definitions 59, 60, and 61.
This macro is invoked in definition 57.

If either bound is not a constant expression, the computation of that bound will yield a null string. Thus the `SizeValue` of the `xExplicitShapeSpec` will be a null string if the dimension is adjustable.

5.2.3 Size of an Array

The size of an array is equal to the number of elements in the array, and is therefore the product of the sizes of the array's dimensions:

Size of an Array[62] ≡

```
RULE: xArraySpec ::= xExplicitShapeSpec
COMPUTE
  xArraySpec.SizeValue=xExplicitShapeSpec.SizeValue;
END;

RULE: xArraySpec ::= xArraySpec ',' xExplicitShapeSpec
COMPUTE
  xArraySpec[1].SizeValue=
    MakeName(
      strmult(
        StringTable(xExplicitShapeSpec.SizeValue),
        StringTable(xArraySpec[2].SizeValue),
        10));
  IF(AND(
      AND(xExplicitShapeSpec.SizeValue,xArraySpec[2].SizeValue),
      NOT(xArraySpec[1].SizeValue)),
    message(ERROR,"Error evaluating array size",0,COORDREF));
END;
```

This macro is defined in definitions 62 and 63.
This macro is invoked in definition 57.

If any dimension is not defined by a constant expression, the corresponding `SizeValue` will be a null string. This null string will propagate through the computation, and therefore the `SizeValue` of an adjustable array declarator will be the null string.

If the upper bound of the last dimension of the array is an asterisk, then neither of the rules above apply. In that case an assumed-size array is being declared, and the `SizeValue` should be a null string:

Size of an Array[63] ≡

```
SYMBOL xArraySpec COMPUTE
  SYNT.SizeValue=0;
```

```
END;
```

This macro is defined in definitions 62 and 63.
This macro is invoked in definition 57.

5.2.4 Array Storage Sequence

The number of storage units in an array is `x*z`, where `x` is the number of elements in the array and `z` is the number of storage units in each array element.

Array Storage Sequence[64] ≡

```
RULE: DimensionDeclarators ::= xArraySpec
COMPUTE
  DimensionDeclarators.GotStorageUnits=
    ResetStorageUnits(
      DimensionDeclarators.UnitKey,
      ApplyStrmath(strmult,
        xArraySpec.SizeValue,
        SizeOfType(DimensionDeclarators.UnitKey)));
END;

SYMBOL xProgramUnit COMPUTE
  SYNT.GotAllStorageUnits=CONSTITUENTS DimensionDeclarators.GotStorageUnits;
END;
```

This macro is invoked in definition 57.

5.3 Array Element Name

The number of subscript expressions must be equal to the number of dimensions in the array declarator for the array name.

Array Element Name[65] ≡

```
RULE: xStmt ::= xLblDef xName ’(’ xExprList ’)’ ’=’ xExpr xEOS
COMPUTE
  Verify dimensionality[66](‘xExprList’)
END;

RULE: xStmt ::= xLblDef xName ’(’ xExprList ’)’ xSubstringRange ’=’ xExpr xEOS
COMPUTE
  Verify dimensionality[66](‘xExprList’)
END;

RULE: xComplexDataRef ::= xName ’(’ xSectionSubscriptList ’)’
COMPUTE
  Verify dimensionality[66](‘xSectionSubscriptList’)
END;
```

This macro is invoked in definition 48.

The syntactic form of an array name is not unique, and therefore it is necessary to verify that `xName` actually refers to an array:

Verify dimensionality[66](◇1) ≡

```
IF(
  AND(
    InIS(Array,GetKindSet(xName.UnitKey,NullIS())),
    NE(GetNumberOfDims(xName.UnitKey,0),◇1.SeqCount)),
  message(ERROR,"Wrong number of dimensions",0,COORDREF))
<- (
  INCLUDING xProgramUnit.ClassificationDone,
  INCLUDING xProgramUnit.GotAllDims);
```

This macro is invoked in definition 65.

5.4 Subscript

Subscript[67] ≡

Subscript Expression[68]
Subscript Value[69]

This macro is invoked in definition 48.

5.4.1 Subscript Expression

A subscript expression is an integer expression.

Within a program unit, the value of each subscript expression must be greater than or equal to the corresponding lower dimension bound in the array declarator for the array. The value of each subscript expression must not exceed the corresponding upper dimension bound declared for the array in the program unit.

Subscript Expression[68] ≡

```
SYMBOL xSubscript INHERITS BoundDeListElem END;
SYMBOL Subscripts INHERITS BoundDeListRoot END;

RULE: xVariable ::= xVariableName '(' Subscripts ')'
COMPUTE
  Subscripts.BoundList=
    GetArrayBounds(xVariableName.UnitKey,NULLBoundList)
    <- INCLUDING xProgramUnit.GotAllBounds;
END;

RULE: xVariable ::= xVariableName '(' Subscripts ')' xSubstringRange
COMPUTE
  Subscripts.BoundList=
    GetArrayBounds(xVariableName.UnitKey,NULLBoundList)
    <- INCLUDING xProgramUnit.GotAllBounds;
END;
```

```
RULE: Subscripts ::= xSubscriptList END;

RULE: xSubscript ::= xExpr
COMPUTE
  IF(NE(xExpr.TypeCode,OilTypeIntegerType),
    message(ERROR,"Subscript must be an integer",0,COORDREF),
  IF(NOT(xExpr.Value),
    message(ERROR,"Must be a constant expression",0,COORDREF),
  IF(
    OR(
      LessThan(xExpr.Value,LowerOf(xSubscript.BoundElem)),
      GreaterThan(xExpr.Value,UpperOf(xSubscript.BoundElem))),
    message(ERROR,"Subscript out of bounds",0,COORDREF))));
END;
```

This macro is invoked in definition 67.

5.4.2 Subscript Value

Table 1 of the standard specifies the value of a subscript. It is based on the values of the upper and lower bounds of each dimension and the product of the lengths of previous dimensions. These values must be obtained from the array declaration.

Subscript Value[69] ≡

```
ATTR Index: int;
CHAIN PartialIndex: int;

SYMBOL Subscripts COMPUTE
  CHAINSTART HEAD.PartialIndex=MakeName("1");
  SYNT.Index=TAIL.PartialIndex;
END;

RULE: xSubscript ::= xExpr
COMPUTE
  xSubscript.PartialIndex=
    ApplyStrmath(stradd,
      xSubscript.PartialIndex,
      ApplyStrmath(strmult,
        ApplyStrmath(strsub,xExpr.Value,LowerOf(xSubscript.BoundElem)),
        StrideOf(xSubscript.BoundElem)));
END;

SYMBOL xArraySpec INHERITS BoundListElem COMPUTE
  THIS._cBoundListPtr=
    RefEndConsBoundList(TAIL._cBoundListPtr,THIS.BoundElem);
END;

SYMBOL DimensionDeclarators INHERITS BoundListRoot COMPUTE
  SYNT.GotBounds=ResetArrayBounds(THIS.UnitKey,THIS.BoundList);
```

```
END;

SYMBOL xProgramUnit COMPUTE
  SYNT.GotAllBounds=CONSTITUENTS DimensionDeclarators.GotBounds;
END;

RULE: xArraySpec ::= '*'
COMPUTE
  xArraySpec.BoundElem=NewBound(MakeName("1"),0,MakeName("1"));
END;

RULE: xArraySpec ::= xLowerBound ':' '*'
COMPUTE
  xArraySpec.BoundElem=NewBound(xLowerBound.Value,0,MakeName("1"));
END;

ATTR LowerValue, UpperValue: int;

RULE: xArraySpec ::= xExplicitShapeSpec
COMPUTE
  xArraySpec.BoundElem=
    NewBound(
      xExplicitShapeSpec.LowerValue,
      xExplicitShapeSpec.UpperValue,
      MakeName("1"));
END;

RULE: xArraySpec ::= xArraySpec ',' '*'
COMPUTE
  xArraySpec[1].BoundElem=NewBound(MakeName("1"),0,xArraySpec[2].SizeValue);
END;

RULE: xArraySpec ::= xArraySpec ',' xLowerBound ':' '*'
COMPUTE
  xArraySpec[1].BoundElem=NewBound(xLowerBound.Value,0,xArraySpec[2].SizeValue);
END;

RULE: xArraySpec ::= xArraySpec ',' xExplicitShapeSpec
COMPUTE
  xArraySpec[1].BoundElem=
    NewBound(
      xExplicitShapeSpec.LowerValue,
      xExplicitShapeSpec.UpperValue,
      xArraySpec[2].SizeValue);
END;

RULE: xExplicitShapeSpec ::= xLowerBound ':' xUpperBound
COMPUTE
  xExplicitShapeSpec.LowerValue=xLowerBound.Value;
  xExplicitShapeSpec.UpperValue=xUpperBound.Value;
END;
```

```
RULE: xExplicitShapeSpec ::= xUpperBound
COMPUTE
  xExplicitShapeSpec.LowerValue=MakeName("1");
  xExplicitShapeSpec.UpperValue=xUpperBound.Value;
END;
```

This macro is defined in definitions 69.
This macro is invoked in definition 67.

Bound list property[70] ≡

```
ArrayBounds: BoundList; "BoundList.h"
```

This macro is invoked in definition 243.

Instantiate a list of bound specifications[71] ≡

```
$/Adt/LidoList.gnrc+instance=Bound +referto=f77semantics :inst
```

This macro is invoked in definition 242.

Bound information access[72] ≡

```
typedef struct { int lower, upper, stride; } Bound;

#define LowerOf(x) ((x).lower)
#define UpperOf(x) ((x).upper)
#define StrideOf(x) ((x).stride)

extern Bound NoBound;
extern Bound NewBound ELI_ARG((int lower, int upper, int stride));
```

This macro is invoked in definition 248.

Bound information[73] ≡

```
Bound NoBound = {0, 0, 0};

Bound
#ifdef PROTO_OK
NewBound(int lower, int upper, int stride)
#else
NewBound(lower, upper, stride) int lower, upper, stride;
#endif
{ Bound temp;
  temp.lower = lower; temp.upper = upper; temp.stride = stride;
  return temp;
}
```

This macro is invoked in definition 247.

5.5 Character Substring

Character Substring[74] ≡

> *Substring Name*[75]
> *Substring Expression*[76]

This macro is invoked in definition 48.

5.5.1 Substring Name

Substring Name[75] ≡

```
RULE: xSubscriptTriplet ::= ':'
COMPUTE
  xSubscriptTriplet.Index=MakeName("1");
END;

RULE: xSubscriptTriplet ::= ':' xExpr
COMPUTE
  xSubscriptTriplet.Index=MakeName("1");
END;

RULE: xSubscriptTriplet ::= xExpr ':'
COMPUTE
  xSubscriptTriplet.Index=xExpr.Value;
END;

RULE: xSubscriptTriplet ::= xExpr ':' xExpr
COMPUTE
  xSubscriptTriplet.Index=xExpr[1].Value;
END;
```

This macro is invoked in definition 74.

5.5.2 Substring Expression

A substring expression may be any integer expression.

Substring Expression[76] ≡

```
RULE: xSubscriptTriplet ::= ':' xExpr
COMPUTE
  IF(NE(xExpr.TypeCode,OilTypeIntegerType),
    message(ERROR,"Must be an integer expression.",0,COORDREF));
END;

RULE: xSubscriptTriplet ::= xExpr ':'
COMPUTE
  IF(NE(xExpr.TypeCode,OilTypeIntegerType),
    message(ERROR,"Must be an integer expression.",0,COORDREF));
END;

RULE: xSubscriptTriplet ::= xExpr ':' xExpr
COMPUTE
  IF(OR(
    NE(xExpr[1].TypeCode, OilTypeIntegerType),
    NE(xExpr[2].TypeCode, OilTypeIntegerType)),
    message(ERROR,"Must both be integer expressions.",0,COORDREF));
END;
```

This macro is invoked in definition 74.

6 Expressions

This section describes the typing and evaluation rules for expressions.

Expressions[77] ≡

Arithmetic Expressions[80]
Character Expressions[83]
Relational Expressions[86]
Logical Expressions[88]
Evaluation of Expressions[91]
Constant Expressions[95]

This macro is defined in definitions 77 and 78.
This macro is invoked in definition 244.

Each of the next four subsections deals with one kind of expression, defining the operators that can be used in the expression, the types for literal constant operands (if applicable), and the rules governing the type consistency of the expressions.

All of the rules governing type consistency are written in *OIL*, the *O*perator *I*dentification *L*anguage. They define an *operator identification* process, in which each operator indication is identified as an instance of a specific operator. Operator identification is carried out by an attribute computation described in the fifth subsection of this section.

The attribute computation is based on encodings of operators and types exported by the operator identification library:

Expressions[78] ≡

```
ATTR TypeCode: tOilType;
ATTR Indication, Operator: tOilOp;
```

This macro is defined in definitions 77 and 78.
This macro is invoked in definition 244.

Operator encodings are associated with the external representations of the operators by computations at the operator nodes of the tree:

Operator indication[79](◇3) ≡

```
RULE: ◇1 ::= ◇2
COMPUTE
  ◇1.Indication=OilOp◇3;
END;
```

This macro is invoked in definitions 80, 83, 86, and 88.

Here the first parameter specifies the class of node (`xUnOp` or `xBinOp`), the second gives the representation of the operator in the source program, and the third is the name used to represent the operator's encoding.

Precedence of operators is not defined here; that is a part of the phrase structure specification.

An expression is a constant expression if a value can be computed, as indicated in the sixth subsection.

6.1 Arithmetic Expressions

There are five arithmetic operators, two of which may be either monadic or dyadic:

Arithmetic Expressions[80] ≡

```
Operator indication[79](‘xBinOp’,‘'**'’,‘Exponentiation’)
Operator indication[79](‘xBinOp’,‘'/'’,‘Division’)
Operator indication[79](‘xBinOp’,‘'*'’,‘Multiplication’)
Operator indication[79](‘xBinOp’,‘'-'’,‘Subtraction’)
Operator indication[79](‘xUnOp’,‘'-'’,‘Negation’)
Operator indication[79](‘xBinOp’,‘'+'’,‘Addition’)
Operator indication[79](‘xUnOp’,‘'+'’,‘Identity’)

RULE: xExpr ::= xIcon
COMPUTE
  xExpr.TypeCode=OilTypeIntegerType;
END;

RULE: xExpr ::= xUnsignedArithmeticConstant
COMPUTE
  xExpr.TypeCode=xUnsignedArithmeticConstant.TypeCode;
END;
```

This macro is invoked in definition 77.

When the operator `+` or `-` operates on a single operand, the data typs of the resulting expression is the same as the dtate type of the operand:

Monadic arithmetic operators[81] ≡

```
INDICATION
  Identity: iIdn, rIdn, dIdn, cIdn;
  Negation: iNeg, rNeg, dNeg, cNeg;

OPER
  iIdn,iNeg(IntegerType): IntegerType;
  rIdn,rNeg(RealType): RealType;
  dIdn,dNeg(DoublePrecisionType): DoublePrecisionType;
```

```
    cIdn,cNeg(ComplexType): ComplexType;
```

This macro is invoked in definition 245.

When an arithmetic operator operates on a pair of operands, the data type of the resulting expression is given in Tables 2 and 3 of the standard:

Table 2, including replications, and Table 3[82] ≡

```
INDICATION
  Addition:       iAdd, rAdd, dAdd, cAdd;
  Subtraction:    iSub, rSub, dSub, cSub;
  Multiplication: iMul, rMul, dMul, cMul;
  Division:       iDiv, rDiv, dDiv, cDiv;
  Exponentiation: iExp, riExp, diExp, ciExp, rExp, dExp, cExp;

OPER
  iAdd,iSub,iMul,iDiv,iExp(IntegerType,IntegerType): IntegerType;
  rAdd,rSub,rMul,rDiv,rExp(RealType,RealType): RealType;
  dAdd,dSub,dMul,dDiv,dExp
    (DoublePrecisionType,DoublePrecisionType): DoublePrecisionType;
  cAdd,cSub,cMul,cDiv,cExp(ComplexType,ComplexType): ComplexType;

  riExp(RealType,IntegerType): RealType;
  diExp(DoublePrecisionType,IntegerType): DoublePrecisionType;
  ciExp(ComplexType,IntegerType): ComplexType;

COERCION
  cREAL(IntegerType): RealType;
  cDBLE(RealType): DoublePrecisionType;
  cCMPLX(RealType): ComplexType COST 2;
```

This macro is invoked in definition 245.

The prohibited combinations in Tables 2 and 3 are prohibited here by the facts that it is impossible to coerce a double to a complex or vice-versa, and that double and complex operations require identical operand data types.

6.2 Character Expressions

There is one character operator, which is dyadic:

Character Expressions[83] ≡

Operator indication[79](‘`xBinOp`’,‘`'//'`’,‘`Concatenation`’)

This macro is defined in definitions 83 and 84.
This macro is invoked in definition 77.

The data type of a string constant is always `CharacterType`:

Character Expressions[84] ≡

```
RULE: xExpr ::= xScon
```

```
COMPUTE
  xExpr.TypeCode=OilTypeCharacterType;
END;

RULE: xFormatIdentifier ::= xExpr xBinOp xExpr
COMPUTE
  xBinOp.Operator=
    OilIdOp2(xBinOp.Indication,xExpr[1].TypeCode,xExpr[2].TypeCode);
END;
```

This macro is defined in definitions 83 and 84.
This macro is invoked in definition 77.

Concatenation takes character data objects and returns the same type:

Concatenation[85] ≡

```
INDICATION Concatenation: Cnc;

OPER Cnc(CharacterType,CharacterType): CharacterType;
```

This macro is invoked in definition 245.

6.3 Relational Expressions

There are six relational operators, all dyadic:

Relational Expressions[86] ≡

Operator indication[79](‘`xBinOp`’,‘`'.lt.'`’,‘`Less`’)
Operator indication[79](‘`xBinOp`’,‘`'.le.'`’,‘`LessOrEqual`’)
Operator indication[79](‘`xBinOp`’,‘`'.eq.'`’,‘`Equal`’)
Operator indication[79](‘`xBinOp`’,‘`'.ne.'`’,‘`NotEqual`’)
Operator indication[79](‘`xBinOp`’,‘`'.gt.'`’,‘`Greater`’)
Operator indication[79](‘`xBinOp`’,‘`'.ge.'`’,‘`GreaterOrEqual`’)

This macro is invoked in definition 77.

There are no relational constants.

All relational operators demand operands of identical type and return a logical value. Relational operators cannot be applied to logical values:

Relational operators[87] ≡

```
INDICATION Less:           iLss, rLss, dLss, cLss, sLss;
INDICATION LessOrEqual:    iLeq, rLeq, dLeq, cLeq, sLeq;
INDICATION Equal:          iEql, rEql, dEql, cEql, sEql;
INDICATION NotEqual:       iNeq, rNeq, dNeq, cNeq, sNeq;
INDICATION Greater:        iGtr, rGtr, dGtr, cGtr, sGtr;
INDICATION GreaterOrEqual: iGeq, rGeq, dGeq, cGeq, sGeq;

OPER iLss,iLeq,iEql,iNeq,iGeq,iGtr
  (IntegerType,IntegerType): LogicalType;
```

```
OPER rLss,rLeq,rEql,rNeq,rGeq,rGtr
  (RealType,RealType): LogicalType;
OPER dLss,dLeq,dEql,dNeq,dGeq,dGtr
  (DoublePrecisionType,DoublePrecisionType): LogicalType;
OPER cLss,cLeq,cEql,cNeq,cGeq,cGtr
  (ComplexType,ComplexType): LogicalType;
OPER sLss,sLeq,sEql,sNeq,sGeq,sGtr
  (CharacterType,CharacterType): LogicalType;
```

This macro is invoked in definition 245.

6.4 Logical Expressions

There are five logical operators, one of which is monadic:

Logical Expressions[88] ≡

Operator indication[79](‘`xUnOp`’,‘`'.not.'`’,‘`LogicalNegation`’)
Operator indication[79](‘`xBinOp`’,‘`'.and.'`’,‘`LogicalConjunction`’)
Operator indication[79](‘`xBinOp`’,‘`'.or.'`’,‘`LogicalInclusiveDisjunction`’)
Operator indication[79](‘`xBinOp`’,‘`'.eqv.'`’,‘`LogicalEquivalence`’)
Operator indication[79](‘`xBinOp`’,‘`'.neqv.'`’,‘`LogicalNonEquivalence`’)

This macro is defined in definitions 88 and 89.
This macro is invoked in definition 77.

The data type of a logical constant is always `LogicalType`:

Logical Expressions[89] ≡

```
RULE: xExpr ::= xLogicalConstant
COMPUTE
  xExpr.TypeCode=OilTypeLogicalType;
END;
```

This macro is defined in definitions 88 and 89.
This macro is invoked in definition 77.

Logical operators demand operands of type logical and produce a result of type logical:

Logical operators[90] ≡

```
INDICATION
  LogicalNegation:             lNeg;
  LogicalConjunction:          lAnd;
  LogicalInclusiveDisjunction: lOr;
  LogicalEquivalence:          lEqv;
  LogicalNonEquivalence:       lNeq;

OPER
  lNeg(LogicalType): LogicalType;
  lAnd,lOr,lEqv,lNeq(LogicalType,LogicalType): LogicalType;
```

This macro is invoked in definition 245.

6.5 Evaluation of Expressions

The type of a FORTRAN expression is determined from the bottom up, starting with the operands. Types of constants are determined by the form of the constant, and the type of a name is obtained from the symbol table information for that name:

Evaluation of Expressions[91] ≡

```
RULE: xExpr ::= xName
COMPUTE
  xExpr.TypeCode=GetOilType(xName.Type,OilErrorType());
END;

RULE: xExpr ::= xSFVarName
COMPUTE
  xExpr.TypeCode=GetOilType(xSFVarName.Type,OilErrorType());
END;

RULE: xExpr ::= xName '(' ')'
COMPUTE
  xExpr.TypeCode=GetOilType(xName.Type,OilErrorType());
END;

RULE: xExpr ::= xComplexDataRef
COMPUTE
  xExpr.TypeCode=
    GetOilType(
      CONSTITUENT xName.Type SHIELD xSectionSubscriptList,
      OilErrorType());
END;
```

Sequence the elements[3](‘`xSectionSubscript`’)

```
RULE: xSectionSubscript ::= xExpr
COMPUTE
  xSectionSubscript.Type=OilTypeName(xExpr.TypeCode);
END;

SYMBOL xSectionSubscript COMPUTE
  SYNT.Type=NoKey;
END;
```

This macro is defined in definitions 91, 92, and 93.
This macro is invoked in definition 77.

The data type of an expression containing one or more operands is determined from the types of the operands, and must be compatible with the operator indication:

Evaluation of Expressions[92] ≡

```
RULE: xExpr ::= xUnOp xExpr
COMPUTE
  xExpr[1].TypeCode=OilGetArgType(xUnOp.Operator,0);
  xUnOp.Operator=OilIdOp1(xUnOp.Indication,xExpr[2].TypeCode);
END;

RULE: xExpr ::= xExpr xBinOp xExpr
COMPUTE
  xExpr[1].TypeCode=OilGetArgType(xBinOp.Operator,0);
  xBinOp.Operator=
    OilIdOp2(xBinOp.Indication,xExpr[2].TypeCode,xExpr[3].TypeCode);
END;
```

This macro is defined in definitions 91, 92, and 93.
This macro is invoked in definition 77.

These computations determine the operator on the basis of the operator indication and the types of the operands, and then uses the result type of the identified operator as the type of the expression. Errors are detected when an operator cannot be identified:

Evaluation of Expressions[93] ≡

```
SYMBOL OpNode COMPUTE
  IF(NOT(OilIsValidOp(THIS.Operator)),
    message(ERROR,"Operator does not agree with its operand(s)",0,COORDREF));
END;

SYMBOL xUnOp INHERITS OpNode END;
SYMBOL xBinOp INHERITS OpNode END;
```

This macro is defined in definitions 91, 92, and 93.
This macro is invoked in definition 77.

Complete the type lattice[94] ≡

```
SET AllTypes = [IntegerType,RealType,DoublePrecisionType,ComplexType,
                CharacterType, LogicalType];
COERCION Bottom(ErrorType): AllTypes;
```

This macro is invoked in definition 245.

6.6 Constant Expressions

A constant expression is an arithmetic constant expression, a character constant expression or a logical constant expression. There are no relational constant expressions.

In each case the primaries of the expression may be constants, symbolic names of constants or constant expressions enclosed in parentheses. Variable, array element, and function references are not allowed.

In this specification, all values are represented by string table indices. The absence of a value is represented by the index 0, which indexes the null string in the table. (According to Section 4.8.1 of the standard, the null string is not a valid character constant. Since only the concatenation operator is permitted in character constant expressions, there is no possibility of ambiguity.) Arithmetic values are represented by strings that obey the rules for FORTRAN real constants, except that they have no embedded spaces. Character values are represented by strings with no delimiting apostrophes, and with embedded apostrophes not doubled. The logical value `.true.` is represented by the string `1` and `.false.` is represented by the string `0`.

Constant Expressions[95] ≡

```
ATTR Value: int;

RULE: xUnsignedArithmeticConstant ::= xIcon
COMPUTE
  xUnsignedArithmeticConstant.Value=IdnNumb(0,xIcon);
END;

RULE: xUnsignedArithmeticConstant ::= xRcon
COMPUTE
  xUnsignedArithmeticConstant.Value=xRcon;
END;

RULE: xUnsignedArithmeticConstant ::= xDcon
COMPUTE
  xUnsignedArithmeticConstant.Value=xDcon;
END;

RULE: xLogicalConstant ::= '.true.'
COMPUTE
  xLogicalConstant.Value=IdnNumb(0,1);
END;

RULE: xLogicalConstant ::= '.false.'
COMPUTE
  xLogicalConstant.Value=IdnNumb(0,0);
END;

SYMBOL xUnsignedArithmeticConstant COMPUTE
  SYNT.Value=0;
END;
```

This macro is defined in definitions 95 and 96.
This macro is invoked in definition 77.

Complex constant expressions are not allowed in this implementation.

Constant Expressions[96] ≡

```
RULE: xExpr ::= xIcon
COMPUTE
  xExpr.Value=IdnNumb(0,xIcon);
END;

RULE: xExpr ::= xUnsignedArithmeticConstant
COMPUTE
  xExpr.Value=xUnsignedArithmeticConstant.Value;
END;

RULE: xExpr ::= xScon
COMPUTE
    xExpr.Value=xScon;
```

```
END;

RULE: xExpr ::= xName
COMPUTE
  xExpr.Value=GetValue(xName.ObjectKey,0) <- xExpr.SymConstEval;
END;

RULE: xExpr ::= xUnOp xExpr
COMPUTE
  xExpr[1].Value=
    IF(xExpr[2].Value,
      MakeName(
        APPLY(
          GetUnOpExec(OilOpName(xUnOp.Operator),NoUnOpExec),
          StringTable(xExpr[2].Value))),
      0);
END;

RULE: xExpr ::= xExpr xBinOp xExpr
COMPUTE
  xExpr[1].Value=
    IF(AND(xExpr[2].Value,xExpr[3].Value),
      ApplyStrmath(
        GetBinOpExec(OilOpName(xBinOp.Operator),NoBinOpExec),
        xExpr[2].Value,
        xExpr[3].Value),
      0);
END;

SYMBOL xExpr COMPUTE
  SYNT.Value=0;
END;
```

This macro is defined in definitions 95 and 96.
This macro is invoked in definition 77.

Indication/Evaluation function correspondence[97] ≡

```
UnOpExec: ExecUnOpFn;
BinOpExec: ExecBinOpFn;

iNeg -> UnOpExec={strneg};
rNeg -> UnOpExec={strneg};
dNeg -> UnOpExec={strneg};

iAdd -> BinOpExec={stradd};
rAdd -> BinOpExec={stradd};
dAdd -> BinOpExec={stradd};

iSub -> BinOpExec={strsub};
rSub -> BinOpExec={strsub};
dSub -> BinOpExec={strsub};
```

```
iMul -> BinOpExec={strmult};
rMul -> BinOpExec={strmult};
dMul -> BinOpExec={strmult};

iDiv -> BinOpExec={strdivi};
rDiv -> BinOpExec={strdivf};
dDiv -> BinOpExec={strdivf};

iExp -> BinOpExec={strpow};
riExp -> BinOpExec={strpow};
diExp -> BinOpExec={strpow};

lNeg -> UnOpExec={strnot};
lAnd -> BinOpExec={strand};
lOr -> BinOpExec={strior};
lEqv -> BinOpExec={streqv};
lNeq -> BinOpExec={strneq};
```

This macro is invoked in definition 243.

Evaluation functions[98] ≡

```
char *
#ifdef PROTO_OK
NoUnOpExec(char *v)
#else
NoUnOpExec(v) char *v;
#endif
{ return ""; }

char *
#ifdef PROTO_OK
strneg(char *v)
#else
strneg(v) char *v;
#endif
{ return strsub("0", v, 10); }

char *
#ifdef PROTO_OK
strnot(char *v)
#else
strnot(v) char *v;
#endif
{ return (*v == '0' ? "1" : "0"); }

char *
#ifdef PROTO_OK
NoBinOpExec(char *v1, char *v2, int radix)
#else
NoBinOpExec(v1, v2, radix) char *v1, *v2; int radix;
```

```
#endif
{ return ""; }

char *
#ifdef PROTO_OK
strand(char *v1, char *v2, int radix)
#else
strand(v1, v2, radix) char *v1, *v2; int radix;
#endif
{ return (*v1 == '0' ? v1 : v2); }

char *
#ifdef PROTO_OK
strior(char *v1, char *v2, int radix)
#else
strior(v1, v2, radix) char *v1, *v2; int radix;
#endif
{ return (*v1 == '1' ? v1 : v2); }

char *
#ifdef PROTO_OK
streqv(char *v1, char *v2, int radix)
#else
streqv(v1, v2, radix) char *v1, *v2; int radix;
#endif
{ return (*v1 == *v2 ? "1" : "0"); }

char *
#ifdef PROTO_OK
strneq(char *v1, char *v2, int radix)
#else
strneq(v1, v2, radix) char *v1, *v2; int radix;
#endif
{ return (*v1 != *v2 ? "1" : "0"); }

int
#ifdef PROTO_OK
ApplyStrmath(ExecBinOpFn op, int v1, int v2)
#else
ApplyStrmath(op, v1, v2) ExecBinOpFn op; int v1, v2;
#endif
{ return MakeName(op(StringTable(v1), StringTable(v2), 10)); }

int
#ifdef PROTO_OK
LessThan(int v1, int v2)
#else
LessThan(v1, v2) int v1, v2;
#endif
{ char *temp = strsub(StringTable(v1), StringTable(v2), 10);
```

```
  return (temp == 0 ? 1 : *temp == '-' ? 1 : 0);
}

int
#ifdef PROTO_OK
GreaterThan(int v1, int v2)
#else
GreaterThan(v1, v2) int v1, v2;
#endif
{ char *temp = strsub(StringTable(v2), StringTable(v1), 10);

  return (temp == 0 ? 1 : *temp == '-' ? 1 : 0);
}
```

This macro is defined in definitions 24, 98, 111, 174, 181, 184, and 186.
This macro is invoked in definition 247.

Evaluation function interfaces[99] ≡

```
typedef char *(*ExecUnOpFn) ELI_ARG((char *));
typedef char *(*ExecBinOpFn) ELI_ARG((char *, char *, int));
char *NoUnOpExec ELI_ARG((char *));
char *strneg ELI_ARG((char *));
char *strnot ELI_ARG((char *));
char *NoBinOpExec ELI_ARG((char *, char *, int));
char *stradd ELI_ARG((char *, char *, int));
char *strsub ELI_ARG((char *, char *, int));
char *strmult ELI_ARG((char *, char *, int));
char *strdivi ELI_ARG((char *, char *, int));
char *strdivf ELI_ARG((char *, char *, int));
char *strpow ELI_ARG((char *, char *, int));
char *strand ELI_ARG((char *, char *, int));
char *strior ELI_ARG((char *, char *, int));
char *streqv ELI_ARG((char *, char *, int));
char *strneq ELI_ARG((char *, char *, int));
int ApplyStrmath ELI_ARG((ExecBinOpFn op, int v1, int v2));
int LessThan ELI_ARG((int v1, int v2));
int GreaterThan ELI_ARG((int v1, int v2));
```

This macro is defined in definitions 25, 99, 110, 112, 182, 185, and 187.
This macro is invoked in definition 248.

The exponent markers for FORTRAN are `E` and `D`. In the result of a computation, the exponent markers will all be `E`; if a number must be output with a type-dependent exponent marker, then that must be arranged at the time of output by invoking `strnorm`.

Initialization[100] ≡

```
strmath(STRM_EXP_SYMBOLS, "EeDd");
```

This macro is defined in definitions 100 and 188.
This macro is invoked in definition 249.

7 Executable and Nonexecutable Statement Classification

Each statement is classified as executable or nonexecutable. Nonexecutable statements may be labeled, but such labels must not be used to control the execution sequence. To enforce this restriction, we define the `Executable` attribute of each `xLblDef` node and use that attribute to set the `Executable` property of the label.

Executable and Nonexecutable Statement Classification[101] ≡

```
SYMBOL xLblDef: Executable: int;

SYMBOL xLblDef COMPUTE
  INH.Executable=0;
  SYNT.GotExec=ResetExecutable(THIS.UnitKey,THIS.Executable);
END;

SYMBOL xProgramUnit COMPUTE
  SYNT.GotAllExecs=CONSTITUENTS xLblDef.GotExec;
END;
```

This macro is defined in definitions 101, 104, and 105.
This macro is invoked in definition 244.

The computation of `Executable` above will appear in all contexts where the statement is *not* executable; it will be overridden in the context of executable statements by the rule computations given below.

Most executable statements have one of the following two forms:

Executable[102](◇1) ≡

```
RULE: xStmt ::= xLblDef ◇1 xEOS
COMPUTE
  xLblDef.Executable=1;
END;
```

This macro is invoked in definition 104.

End statement[103](◇1) ≡

```
RULE: ◇1 ::= xLblDef 'end' xEOS
COMPUTE
  xLblDef.Executable=1;
END;
```

This macro is invoked in definition 104.

The only exceptions are the logical IF, which has no `xEOS`, and the terminator for a block IF, which is like and `End statement` but uses the keyword `endif`.

The following classification follows the list given in Section 7.1 of the standard:

Executable and Nonexecutable Statement Classification[104] ≡

Executable[102](‘`xName '=' xExpr`’)
Executable[102](‘`xName '(' xExprList ')' '=' xExpr`’)

Executable[102](‘`xName '(' xExprList ')' xSubstringRange '=' xExpr`’)
Executable[102](‘`'assign' xLblRef 'to' xVariableName`’)

Executable[102](‘`GoToKw '(' xLblRefList ')' xExpr`’)
Executable[102](‘`GoToKw xLblRef`’)
Executable[102](‘`GoToKw xVariableName '(' xLblRefList ')'`’)
Executable[102](‘`GoToKw xVariableName`’)

Executable[102](‘`'if' '(' xExpr ')' xLblRef ',' xLblRef ',' xLblRef`’)

```
RULE: xStmt ::= xLblDef 'if' '(' xExpr ')' xStmt
COMPUTE
  xLblDef.Executable=1;
END;
```

Executable[102](‘`'if' '(' xExpr ')' 'then'`’)

```
RULE: xEndIfStmt ::= xLblDef 'endif' xEOS
COMPUTE
  xLblDef.Executable=1;
END;
```

Executable[102](‘`'continue'`’)

Executable[102](‘`'stop'`’)
Executable[102](‘`'stop' xIcon`’)
Executable[102](‘`'stop' xScon`’)
Executable[102](‘`'pause'`’)
Executable[102](‘`'pause' xIcon`’)
Executable[102](‘`'pause' xScon`’)

Executable[102](‘`'do' xLblRef xLoopControl`’)

Executable[102](‘`'read' xFormatIdentifier ',' xInputItemList`’)
Executable[102](‘`'read' xFormatIdentifier`’)
Executable[102](‘`'read' xIoControlSpec`’)
Executable[102](‘`'read' xIoControlSpec xInputItemList`’)
Executable[102](‘`'write' '(' xIoControlSpecList ')'`’)
Executable[102](‘`'write' '(' xIoControlSpecList ')' xOutputItemList`’)
Executable[102](‘`'print' xFormatIdentifier ',' xOutputItemList`’)
Executable[102](‘`'print' xFormatIdentifier`’)

Executable[102](‘`'rewind' '(' xIoControlSpecList ')'`’)
Executable[102](‘`'rewind' xUnitIdentifier`’)
Executable[102](‘`'backspace' '(' xIoControlSpecList ')'`’)
Executable[102](‘`'backspace' xUnitIdentifier`’)
Executable[102](‘`'endfile' '(' xIoControlSpecList ')'`’)
Executable[102](‘`'endfile' xUnitIdentifier`’)
Executable[102](‘`'open' '(' xIoControlSpecList ')'`’)
Executable[102](‘`'close' '(' xIoControlSpecList ')'`’)
Executable[102](‘`'inquire' '(' xIoControlSpecList ')'`’)

Executable[102](‘`'call' xSubroutineNameUse '(' xArgList ')'`’)

Executable[102](‘`'call' xSubroutineNameUse`’)
Executable[102](‘`'return'`’)
Executable[102](‘`'return' xExpr`’)

End statement[103](‘`xEndBlockDataStmt`’)
End statement[103](‘`xEndSubroutineStmt`’)
End statement[103](‘`xEndFunctionStmt`’)
End statement[103](‘`xEndProgramStmt`’)

This macro is defined in definitions 101, 104, and 105.
This macro is invoked in definition 244.

Although the ELSE IF and ELSE statements are listed in Section 7.1 as being executable, Sections 11.7.2 and 11.8.2 state that their statement labels, if any, must not be referenced by any statement. Thus they are omitted from the list above.

A statement that has the form of a statement function must be checked semantically:

Executable and Nonexecutable Statement Classification[105] ≡

```
RULE: xStmt ::= xLblDef xName xStmtFunctionRange
COMPUTE
  xLblDef.Executable=IF(xStmtFunctionRange.InStmtFunc,0,1);
END;
```

This macro is defined in definitions 101, 104, and 105.
This macro is invoked in definition 244.

8 Specification Statements

Section 8 of the standard describes the properties of the FORTRAN specification statements.

Specification Statements[106] ≡

DIMENSION Statement[107]
EQUIVALENCE Statement[108]
COMMON Statement[114]
Type-Statements[118]
IMPLICIT Statement[130]
PARAMETER Statement[136]
EXTERNAL Statement[146]
INTRINSIC Statement[149]

This macro is invoked in definition 244.

8.1 DIMENSION Statement

DIMENSION Statement[107] ≡

```
RULE: xStmt ::= xLblDef 'dimension' xArrayDeclaratorList xEOS END;
RULE: xArrayDeclaratorList LISTOF xArrayDeclarator END;
```

This macro is invoked in definition 106.

No specific analysis of a DIMENSION Statement is required.

8.2 EQUIVALENCE Statement

EQUIVALENCE Statement[108] ≡

```
RULE: xStmt ::= xLblDef 'equivalence' xEquivalenceSetList xEOS END;
RULE: xEquivalenceSetList LISTOF xEquivalenceSet END;

ATTR Point: EquivPoint;

RULE: xEquivalenceSet LISTOF xEquivalenceObject
COMPUTE
  xEquivalenceSet.Point=
    CONSTITUENTS xEquivalenceObject.Point
      WITH (EquivPoint,MakeEquiv,IDENTICAL,NoEquivPoint);
END;

RULE: xEquivalenceObject ::= xVariable
COMPUTE
  xEquivalenceObject.Point=
    NewEquivPoint(CONSTITUENT xVariableName.UnitKey,xVariable.Index,COORDREF)
    <- INCLUDING xProgramUnit.GotAllCommon;
END;

RULE: xVariable ::= xVariableName '(' Subscripts ')'
COMPUTE
  xVariable.Index=Subscripts.Index;
END;

RULE: xVariable ::= xVariableName '(' Subscripts ')' xSubstringRange
COMPUTE
  xVariable.Index=
    ApplyStrmath(stradd,
      Subscripts.Index,
      ApplyStrmath(strsub,
        xSubstringRange CONSTITUENT xSubscriptTriplet.Index
          SHIELD xSubscriptTriplet,
        MakeName("1")));
END;

SYMBOL xVariable COMPUTE
  SYNT.Index=MakeName("1");
END;
```

This macro is invoked in definition 106.

Properties for equivalence analysis[109] ≡

```
Parent: DefTableKey;
Offset: int;
```

This macro is invoked in definition 243.

Evaluation function interfaces[110] ≡

```
typedef struct { DefTableKey var; int index; CoordPtr loc; } EquivPoint;
#define NoEquivPoint() NewEquivPoint(NoKey,0,NoPosition)
```

This macro is defined in definitions 25, 99, 110, 112, 182, 185, and 187.
This macro is invoked in definition 248.

Evaluation functions[111] ≡

```
EquivPoint
#ifdef PROTO_OK
NewEquivPoint(DefTableKey var, int index, CoordPtr loc)
#else
NewEquivPoint(var, index, loc) DefTableKey var; int index; CoordPtr loc;
#endif
{ EquivPoint temp;

  temp.var = var; temp.index = index; temp.loc = loc;

  if (var != NoKey && GetBlockIn(var,NoKey) == NoKey) {
    DefTableKey leader = NewKey();
    ResetStorageUnits(leader,GetStorageUnits(var,0));
    ResetBlockIn(var,leader);
  }

  return temp;
}

static void
#ifdef PROTO_OK
CompressEquivPath(DefTableKey var)
#else
CompressEquivPath(var) DefTableKey var;
#endif
{ DefTableKey temp, leader;
  int offset = MakeName("0");

  for (temp=GetBlockIn(var,NoKey); temp!=NoKey; temp=GetParent(temp,NoKey)) {
    EquivStackPush(var); var = temp;
  }

  leader = var;
  while (!EquivStackEmpty) {
    var = EquivStackPop;
    if (!EquivStackEmpty) ResetParent(var,leader);
    else ResetBlockIn(var,leader);
    offset = ApplyStrmath(stradd,offset,GetOffset(var,MakeName("0")));
    ResetOffset(var,offset);
  }
}
```

```
EquivPoint
#ifdef PROTO_OK
MakeEquiv(EquivPoint p1, EquivPoint p2)
#else
MakeEquiv(p1, p2) EquivPoint p1, p2;
#endif
{ int offset1, offset2, diff, p1size, p2size;
  DefTableKey temp, lead1, lead2;

  if (p1.var == NoKey) return p2;
  if (p2.var == NoKey) return p1;

  CompressEquivPath(p1.var);
  lead1 = GetBlockIn(p1.var,NoKey);
  offset1 = ApplyStrmath(stradd,p1.index,GetOffset(p1.var,MakeName("0")));

  CompressEquivPath(p2.var);
  lead2 = GetBlockIn(p2.var,NoKey);
  offset2 = ApplyStrmath(stradd,p2.index,GetOffset(p2.var,MakeName("0")));

  if (lead1 == lead2 && offset1 != offset2) {
    message(ERROR,"Inconsistent equivalence",0,p2.loc);
    return p1;
  }

  if (InIS(CommonBlock,GetKindSet(lead1,NullIS()))) {
    if (InIS(CommonBlock,GetKindSet(lead2,NullIS()))) {
      message(ERROR,"Equivalences different common blocks",0,p2.loc);
      return p1;
    }
    diff = ApplyStrmath(strsub,offset1,offset2);
    if ((StringTable(diff))[0] == '-') {
      message(ERROR,"Equivalence changes common origin",0,p2.loc);
      return p1;
    }
  } else if (InIS(CommonBlock,GetKindSet(lead2,NullIS()))) {
    DefTableKey td; int ti;
    diff = ApplyStrmath(strsub,offset2,offset1);
    if ((StringTable(diff))[0] == '-') {
      message(ERROR,"Equivalence changes common origin",0,p2.loc);
      return p1;
    }
    td = lead1; lead1 = lead2; lead2 = td;
    ti = offset1; offset1 = offset2; offset2 = ti;
  } else {
    DefTableKey td; int ti;
    diff = ApplyStrmath(strsub,offset1,offset2);
    if ((StringTable(diff))[0] == '-') {
      diff = MakeName(StringTable(diff)+1);
      td = lead1; lead1 = lead2; lead2 = td;
      ti = offset1; offset1 = offset2; offset2 = ti;
    }
  }

  ResetParent(lead2,lead1); ResetOffset(lead2,diff);

  p1size = GetStorageUnits(lead1,0);
  p2size = ApplyStrmath(stradd,diff,GetStorageUnits(lead2,0));
  if (LessThan(p1size,p2size)) ResetStorageUnits(lead1,p2size);

  return p1;
}
```

This macro is defined in definitions 24, 98, 111, 174, 181, 184, and 186.
This macro is invoked in definition 247.

Evaluation function interfaces[112] ≡

```
extern EquivPoint NewEquivPoint ELI_ARG((DefTableKey, int, CoordPtr));
extern EquivPoint MakeEquiv ELI_ARG((EquivPoint, EquivPoint));
```

This macro is defined in definitions 25, 99, 110, 112, 182, 185, and 187.
This macro is invoked in definition 248.

Instantiate the equivalence stack module[113] ≡

```
$/Adt/Stack.gnrc +instance=Equiv +referto=DefTableKey :inst
```

This macro is invoked in definition 242.

8.3 COMMON Statement

The COMMON statement provides a means of associating entities in different program units. Because we can never guarantee that we have access to *all* program units, this specification makes no attempt to verify context conditions that must hold between program units.

COMMON Statement[114] ≡

Form of a COMMON Statement[115]
Common Block Storage Sequence[116]

This macro is invoked in definition 106.

8.3.1 Form of a COMMON Statement

Each omitted `xCommonBlockName` specifies the blank common block. If the first `xCommonBlockName` is omitted, the first two slashes are optional.

In each COMMON statement, the `xCommonBlockObjects` following a `xCommonBlockName` are declared to be in that common block. All `xCommonBlockObjects` preceding the first `xCommonBlockName` are declared to be in blank common.

Any `xCommonBlockName` or omitted `xCommonBlockName` for blank common may occur more than once in one or more COMMON statements in a program unit. The `xCommonBlockObjects` following each successive appearance of the same `xCommonBlockName` are treated as a continuation of the list for that common block name.

The chain `CommonKey` embodies these conditions.

Form of a COMMON Statement[115] ≡

```
SYMBOL xProgramUnit: BlankCommonKey: DefTableKey;
CHAIN CommonKey: DefTableKey;

SYMBOL xProgramUnit COMPUTE
  SYNT.BlankCommonKey=NewKey();
  CHAINSTART HEAD.CommonKey=SYNT.BlankCommonKey;
  SYNT.GotAllCommon=TAIL.CommonKey;
END;

RULE: xStmt ::= xLblDef 'common' xComlist xEOS
COMPUTE
  xComlist.CommonKey=
    INCLUDING xProgramUnit.BlankCommonKey <- xStmt.CommonKey;
END;

RULE: xComlist LISTOF xComblock | xCommonBlockObject END;

RULE: xComblock ::= '/' '/'
COMPUTE
  xComblock.CommonKey=
    INCLUDING xProgramUnit.BlankCommonKey <- xComblock.CommonKey;
END;

RULE: xComblock ::= '/' xCommonBlockName '/'
COMPUTE
  xComblock.CommonKey=xCommonBlockName.UnitKey <- xComblock.CommonKey;
END;
```

This macro is invoked in definition 114.

8.3.2 Common Block Storage Sequence

A storage sequence is formed consisting of the storage sequences of all `xCommonBlockObjects` for the common block. The order of the storage sequence is the same as the order of appearance of the `xCommonBlockObjects` in the program unit.

Common Block Storage Sequence[116] ≡

```
ATTR PreviousSize: int;

SYMBOL xCommonBlockObject COMPUTE
  SYNT.PreviousSize=GetBlockSize(THIS.CommonKey,MakeName("0"));
  THIS.CommonKey=
    ORDER(
      ResetBlockIn(CONSTITUENT xVariableName.UnitKey,THIS.CommonKey),
      ResetOffset(CONSTITUENT xVariableName.UnitKey,SYNT.PreviousSize),
      ResetBlockSize(
        THIS.CommonKey,
        ApplyStrmath(stradd,
          SYNT.PreviousSize,
          GetStorageUnits(
            CONSTITUENT xVariableName.UnitKey,
            SizeOfType(CONSTITUENT xVariableName.Type))
            <- INCLUDING xProgramUnit.GotAllStorageUnits)),
    THIS.CommonKey);
END;
```

This macro is invoked in definition 114.

Properties for Common Block Storage Sequence[117] ≡

```
BlockIn: DefTableKey;
BlockSize: int;
```

This macro is invoked in definition 243.

8.4 Type-Statements

The nonterminal `xObjectName` represents the appearance of a symbolic name in a type statement. This appearance specifies the data type of that name for all appearances in the program unit, and may confirm implicit typing (e.g. for a constant defined earlier in the program).

Type-Statements[118] ≡

```
ATTR Type: DefTableKey;

RULE: xStmt ::= xLblDef xTypeSpec xEntityDeclList xEOS
COMPUTE
  xEntityDeclList.Type=xTypeSpec.Type;
END;

RULE: xEntityDeclList LISTOF xEntityDecl END;
```

Type specified in type-statement as[138](‘`INCLUDING xEntityDeclList.Type`’)

Arithmetic and LOGICAL Type Statements[125]
CHARACTER Type Statements[126]

This macro is defined in definitions 118, 121, 123, and 124.
This macro is invoked in definition 106.

`IsType` is a library access function that will change the `Type` property of `THIS.UnitKey` if it differs from `INCLUDING xEntityDeclList.Type`. If no type property has been set for `THIS.UnitKey`, then `IsType` will set the value of that property to `INCLUDING xEntityDeclList.Type`.

Properties for type analysis[119] ≡

```
Type: DefTableKey [Is];
```

This macro is defined in definitions 43 and 119.
This macro is invoked in definition 243.

Set xTypeSpec.Type[120](◇2) ≡

```
RULE: xTypeSpec ::= '◇1'
COMPUTE
  xTypeSpec.Type=◇2;
  xTypeSpec.Length=0;
END;
```

This macro is invoked in definition 125.

Within a program unit, a name must not have its type explicitly specified more than once.

Type-Statements[121] ≡

Report multiple[33](‘`xObjectName`’,‘`type`’)

This macro is defined in definitions 118, 121, 123, and 124.
This macro is invoked in definition 106.

Properties supporting multiple definition reporting[122] ≡

```
typeApp: int;
```

This macro is defined in definitions 35, 50, 122, 137, 148, and 150.
This macro is invoked in definition 243.

Only a constant, variable, array, external function or statement function may appear in a type statement. (The name of a main program, subroutine, or block data subprogram is disallowed.)

Type-Statements[123] ≡

```
SYMBOL xObjectName COMPUTE
  IF(
    DisjIS(
      GetKindSet(THIS.UnitKey,SingleIS(Variable)),
      ConsIS(Constant,
        ConsIS(Variable,
        ConsIS(Array,
        ConsIS(ExternalFunction,
        ConsIS(IntrinsicFunction,
        ConsIS(StatementFunction,NullIS()))))))),
    message(ERROR,"This class of symbolic name cannot be typed",0,COORDREF))
  <- INCLUDING xProgramUnit.ClassificationDone;
END;
```

This macro is defined in definitions 118, 121, 123, and 124.
This macro is invoked in definition 106.

A type statement may confirm the type of an intrinsic function, but not change it:

Type-Statements[124] ≡

```
SYMBOL xObjectName COMPUTE
```

```
    IF(
      AND(
        GetIntrinsic(THIS.ObjectKey,0),
        NE(GetType(THIS.UnitKey,NoKey),GetType(THIS.ObjectKey,NoKey))),
      message(ERROR,"Cannot change the type of an intrinsic function",0,COORDREF))
    <- INCLUDING xProgramUnit.GotAllTypes;
  END;
```

This macro is defined in definitions 118, 121, 123, and 124.
This macro is invoked in definition 106.

8.4.1 Arithmetic and LOGICAL Type Statements

Arithmetic and LOGICAL Type Statements[125] ≡

> *Set xTypeSpec.Type*[120](‘`integer`’,‘`IntegerType`’)
> *Set xTypeSpec.Type*[120](‘`real`’,‘`RealType`’)
> *Set xTypeSpec.Type*[120](‘`doubleprecision`’,‘`DoublePrecisionType`’)
> *Set xTypeSpec.Type*[120](‘`complex`’,‘`ComplexType`’)
> *Set xTypeSpec.Type*[120](‘`logical`’,‘`LogicalType`’)

This macro is defined in definitions 125.
This macro is invoked in definition 118.

8.4.2 CHARACTER Type Statements

A character type statement may carry a length specification for each entity in the statement not having its own length specification. If a length is not specified for an entity, its length is 1.

CHARACTER Type Statements[126] ≡

```
ATTR Length: int;

RULE: xStmt ::= xLblDef xTypeSpec xEntityDeclList xEOS
COMPUTE
  xEntityDeclList.Length=xTypeSpec.Length;
END;

RULE: xTypeSpec ::= ’character’
COMPUTE
  xTypeSpec.Type=CharacterType;
  xTypeSpec.Length=MakeName("1");
END;

RULE: xTypeSpec ::= ’character’ xCharSelector
COMPUTE
  xTypeSpec.Type=CharacterType;
  xTypeSpec.Length=xCharSelector.Length;
END;

RULE: xCharSelector ::= ’*’ xCharLength
```

```
COMPUTE
  xCharSelector.Length=xCharLength.Length;
END;
```

This macro is defined in definitions 126, 127, 128, and 129.
This macro is invoked in definition 118.

A `xCharLength` must be an unsigned, nonzero integer constant, an integer constant expression with a positive value, or and asterisk. Since a specific value must be positive, we use the value 0 to indicate an asterisk.

CHARACTER Type Statements[127] ≡

```
RULE: xCharLength ::= xIcon
COMPUTE
  xCharLength.Length=IdnNumb(0,xIcon);
  IF(EQ(xIcon,0),
    message(ERROR,"Positive length expected",0,COORDREF));
END;

RULE: xCharLength ::= '(' xTypeParamValue ')'
COMPUTE
  xCharLength.Length=xTypeParamValue.Value;
  IF(AND(xTypeParamValue.Value,LessThan(xCharLength.Length,MakeName("1"))),
    message(ERROR, "Positive length expected", 0, COORDREF));
END;

RULE: xTypeParamValue ::= xExpr
COMPUTE
  xTypeParamValue.Value=xExpr.Value;
  IF(NE(xExpr.TypeCode,OilTypeIntegerType),
    message(ERROR,"Length must be an integer",0,COORDREF));
  IF(NOT(xExpr.Value),
    message(ERROR,"Length must be constant",0,COORDREF));
END;

RULE: xTypeParamValue ::= '*'
COMPUTE
  xTypeParamValue.Value=0;
END;
```

This macro is defined in definitions 126, 127, 128, and 129.
This macro is invoked in definition 118.

A length immediately following the word CHARACTER is the length specification for each entity in the statement not having its own length specification. A length specification immediately following an entity is the length specification for only that entity.

CHARACTER Type Statements[128] ≡

```
RULE: xEntityDecl ::= xObjectName
COMPUTE
  xEntityDecl.Length=INCLUDING xEntityDeclList.Length;
  xEntityDecl.GotLength=ResetLength(xObjectName.UnitKey,xEntityDecl.Length);
END;
```

```
RULE: xEntityDecl ::= xObjectName '(' DimensionDeclarators ')'
COMPUTE
  xEntityDecl.Length=INCLUDING xEntityDeclList.Length;
  xEntityDecl.GotLength=ResetLength(xObjectName.UnitKey,xEntityDecl.Length);
END;

RULE: xEntityDecl ::= xObjectName '*' xCharLength
COMPUTE
  xEntityDecl.Length=xCharLength.Length;
  xEntityDecl.GotLength=ResetLength(xObjectName.UnitKey,xEntityDecl.Length);
END;

RULE: xEntityDecl ::= xObjectName '(' DimensionDeclarators ')' '*' xCharLength
COMPUTE
  xEntityDecl.Length=xCharLength.Length;
  xEntityDecl.GotLength=ResetLength(xObjectName.UnitKey,xEntityDecl.Length);
END;

SYMBOL xProgramUnit COMPUTE
  SYNT.GotAllLengths=
    CONSTITUENTS (xEntityDecl.GotLength,xImplicitSpec.GotLength);
END;
```

This macro is defined in definitions 126, 127, 128, and 129.
This macro is invoked in definition 118.

An entity declared in a CHARACTER statement must have a nonzero length unless that entity is an external function, a dummy argument of an external procedure, or a character constant that has a symbolic name.

CHARACTER Type Statements[129] ≡

```
ATTR Class: IntSet;

RULE: xEntityDecl ::= xObjectName
COMPUTE
  xEntityDecl.Class=
    GetKindSet(xObjectName.UnitKey,NullIS()) <- xEntityDecl.Order;
END;

RULE: xEntityDecl ::= xObjectName '(' DimensionDeclarators ')'
COMPUTE
  xEntityDecl.Class=
    GetKindSet(xObjectName.UnitKey,NullIS()) <- xEntityDecl.Order;
END;

RULE: xEntityDecl ::= xObjectName '*' xCharLength
COMPUTE
  xEntityDecl.Class=
    GetKindSet(xObjectName.UnitKey,NullIS()) <- xEntityDecl.Order;
END;

RULE: xEntityDecl ::= xObjectName '(' DimensionDeclarators ')' '*' xCharLength
```

```
COMPUTE
  xEntityDecl.Class=
    GetKindSet(xObjectName.UnitKey,NullIS()) <- xEntityDecl.Order;
END;

SYMBOL xEntityDecl COMPUTE
  IF(
    AND(
      AND(EQ(INCLUDING xEntityDeclList.Type,CharacterType),EQ(THIS.Length,0)),
      EmptyIS(
        InterIS(
          THIS.Class,
          ConsIS(ExternalFunction,
            ConsIS(DummyArgument,
            ConsIS(Constant,
            NullIS())))))),
    message(ERROR,"A length must be specified",0,COORDREF));
END;
```

This macro is defined in definitions 126, 127, 128, and 129.
This macro is invoked in definition 118.

8.5 IMPLICIT Statement

The nonterminal `xImplicitSpec` represents the association of a type with a set of letters. A symbolic name that is not explicitly typed, beginning with one of the letters in the set, is implicitly given the associated type.

IMPLICIT Statement[130] ≡

```
RULE: xImplicitSpec ::= xTypeSpec xImpl
COMPUTE
  xImplicitSpec.ImplicitTyping=
    ImplicitType(xTypeSpec.Type,xImpl,COORDREF)
    <- xImplicitSpec.ImplicitTyping;
  xImplicitSpec.GotLength=ImplicitLength(xTypeSpec.Length,xImpl);
END;
```

This macro is defined in definitions 130 and 135.
This macro is invoked in definition 106.

`xImpl` is a sequence of single letters or letter ranges, separated by commas, with the entire sequence enclosed in parentheses. The scanner extracts the parenthesized specification, removes the parentheses and any internal spaces, converts all letters to upper case, and stores the resulting string in the string table. Its index is then made the value of `xImpl`. This string must be broken down, and the key `xTypeSpec.Type` associated with each letter:

Process individual letters of an implicit specification[131] ≡

```
void
#ifdef PROTO_OK
ImplicitType(DefTableKey Type, int Initial, CoordPtr coord)
```

```
#else
ImplicitType(Type, Initial, coord)
DefTableKey Type; int Initial; CoordPtr coord;
#endif
{ char *p = StringTable(Initial);

  do {
    int first = *p++ - 'a', last;
    if (*p != '-') last = first; else { last = p[1] - 'a'; p += 2; }
    if (last < first) message(ERROR, "Improper character range", 0, coord);
    while (first <= last) DeclareImplicitType(first++, Type, coord);
    if (*p == ',') p++;
  } while (*p);
}

void
#ifdef PROTO_OK
ImplicitLength(int Length, int Initial)
#else
ImplicitLength(Length, Initial) int Length, Initial;
#endif
{ char *p = StringTable(Initial);

  do {
    int first = *p++ - 'a', last;
    if (*p != '-') last = first; else { last = p[1] - 'a'; p += 2; }
    while (first <= last) LengthOf[first++] = Length;
    if (*p == ',') p++;
  } while (*p);
}
```

This macro is invoked in definition 247.

The invariant of the loop is that `p` addresses either an individual character or the first character of a range. An individual character is converted to a range, leaving only a single case to be dealt with. If the range is descending, then an error is reported; otherwise `DeclareImplicitType` is used to modify the state of the relationship between a single initial letter and the associated type. This relationship is stored in the array `TypeFor`:

Relate initial letters to types[132] ≡

```
DefTableKey TypeFor[26];
int LengthOf[26];
static int IsDone[26];
static CoordPtr SetAt[26];

static void
#ifdef PROTO_OK
DeclareImplicitType(int index, DefTableKey Type, CoordPtr coord)
#else
DeclareImplicitType(index, Type, coord)
int index; DefTableKey Type; CoordPtr coord;
```

```
#endif
{ if (SetAt[index]) {
    obstack_strgrow(Csm_obstk, "Letter ");
    obstack_1grow(Csm_obstk, 'a' + index);
    message(
     ERROR,
     (char *)obstack_strcpy(Csm_obstk, " multiply defined"),
     0,
     coord);
    if (!IsDone[index]) {
     obstack_strgrow(Csm_obstk, "Letter ");
     obstack_1grow(Csm_obstk, 'a' + index);
     message(
      ERROR,
      (char *)obstack_strcpy(Csm_obstk, " multiply defined"),
      0,
      SetAt[index]);
     IsDone[index] = 1;
    }
  } else {
    SetAt[index] = coord;
    if (TypeFor[index] != Type) {
     Report a PARAMETER sequence error if there was one[144]
    }
  }
  TypeFor[index] = Type;
}
```

This macro is invoked in definition 247.

The test of `IsDone` guarantees that the same letter does not appear as a single letter, and is not included in a range of letters, more than once in all of the implicit statements in a program unit.

The state of the relationship must be initialized at the beginning of each program unit. This can be done by a single call, with appropriate dependence on `ImplicitTyping`, the chain controlling use of the state:

Reset the relationship between initial letters and types[133] ≡

```
DefaultImplicitTypes()
```

This macro is invoked in definition 135.

`DefaultImplicitTypes` simply clears the arrays involved in error reporting and restores the default types:

State initialization[134] ≡

```
void
DefaultImplicitTypes()
{ int i;
  for (i = 0; i < 26; i++)
    { IsDone[i] = 0; SetAt[i] = (CoordPtr)0; LengthOf[i] = 0; }
  Clear the test for PARAMETER sequence errors[142]
  Set default implicit types[45]
}
```

This macro is invoked in definition 247.

This mechanism for implementing IMPLICIT works via side effects on a data structure. The data structure must be re-initialized for each program unit, and all information necessary for one program unit must be extracted before the data structure is re-initialized for the next.

IMPLICIT Statement[135] ≡

```
CHAIN ImplicitTyping: VOID;

SYMBOL xSourceFile COMPUTE
  CHAINSTART HEAD.ImplicitTyping=1;
END;

SYMBOL xProgramUnit COMPUTE
  HEAD.ImplicitTyping=
    Reset the relationship between initial letters and types[133]
    <- THIS.ImplicitTyping;
  SYNT.GotAllTypeDefs=
    CONSTITUENTS xSubprogramRange.GotType <- TAIL.ImplicitTyping;
  SYNT.GotAllTypes=1
    <- (CONSTITUENTS Reference.Type, CONSTITUENTS Reference.Length);
  THIS.ImplicitTyping=SYNT.GotAllTypes;
END;
```

This macro is defined in definitions 130 and 135.
This macro is invoked in definition 106.

8.6 PARAMETER Statement

The nonterminal `xNamedConstant` represents the context of a symbolic name in a PARAMETER statement. A symbolic name of a constant must not become defined more than once in a program unit:

PARAMETER Statement[136] ≡

```
RULE: xStmt ::= xLblDef 'parameter' '(' xNamedConstantDefList ')' xEOS END;
RULE: xNamedConstantDefList LISTOF xNamedConstantDef END;
Report multiple[33]('xNamedConstant','constant')
```

This macro is defined in definitions 136, 139, and 145.
This macro is invoked in definition 106.

Properties supporting multiple definition reporting[137] ≡

```
constantApp: int;
```

This macro is defined in definitions 35, 50, 122, 137, 148, and 150.
This macro is invoked in definition 243.

If a symbolic name of a constant is not of default implied type, its type must be specified by a type-statement or IMPLICIT statement prior to its first appearance in a PARAMETER statement.

Type specified in type-statement as[138](◇1) ≡

```
SYMBOL xObjectName COMPUTE
  THIS.ImplicitTyping=
    IsType(THIS.UnitKey,◇1,ErrorType) <- THIS.ImplicitTyping;
END;
```

This macro is invoked in definition 118.

PARAMETER Statement[139] ≡

```
SYMBOL xNamedConstant COMPUTE
  THIS.ImplicitTyping=
    IF(EQ(GetType(THIS.UnitKey,NoKey),NoKey),
      ResetType(THIS.UnitKey,ParameterType(THIS.Sym,COORDREF)))
    <- THIS.ImplicitTyping;
END;
```

This macro is defined in definitions 136, 139, and 145.
This macro is invoked in definition 106.

`ParameterType` is invoked only if the symbolic name is implicitly typed. It needs to check whether the default implicit type is being used and, if so, arrange for an error report if the default implicit type is changed. An array `ParamCoord` supports this behavior:

Array used for reporting PARAMETER/IMPLICIT sequence errors[140] ≡

```
#include "CoordPtrList.h"

static CoordPtrList ParamCoord[26];
```

This macro is invoked in definition 247.

The array elements are linear lists of coordinate pointers, which are manipulated by operations exported by an instantiation of the generic Eli list module:

Instantiate a module implementing a linear list of CoordPtr[141] ≡

```
$/Adt/List.gnrc +instance=CoordPtr +referto=err :inst
```

This macro is invoked in definition 242.

`CoordPtr` is a type exported by the Eli error module, whose interface is the file `err.h`; this is specified to the instantiation by the parameter `+referto=err`.

The array is initialized to empty list pointers, and any list storage used is reclaimed:

Clear the test for PARAMETER sequence errors[142] ≡

```
{ int i;
  for (i = 0; i < 26; i++) ParamCoord[i] = NULLCoordPtrList;
  FinlCoordPtrList();
}
```

This macro is invoked in definition 134.

Upon typing a `xNamedConstant` whose first letter's default type has not been changed, `ParameterType` marks the default type as having been changed and saves the coordinates of the `xNamedConstant`:

Obtain the type of a named constant[143] ≡

```
DefTableKey
#ifdef PROTO_OK
ParameterType(int Sym, CoordPtr coord)
#else
ParameterType(Sym,  coord)
int Sym; CoordPtr coord;
#endif
{ int index = *(StringTable(Sym)) - 'a';
  if (!IsDone[index]) {
    ParamCoord[index] = ConsCoordPtrList(coord, ParamCoord[index]);
  }
  return TypeFor[index];
}
```

This macro is invoked in definition 247.

The check is made when an IMPLICIT statement is being processed, and `IsDone` is true for the letter in question.

Report a PARAMETER sequence error if there was one[144] ≡

```
while (ParamCoord[index] != NULLCoordPtrList) {
  message(
    ERROR,
    "IMPLICIT setting this type follows",
    0,
    HeadCoordPtrList(ParamCoord[index]));
  ParamCoord[index] = TailCoordPtrList(ParamCoord[index]);
}
```

This macro is invoked in definition 132.

The nonterminal `xNamedConstantDef` represents the context of an assignment to a symbolic name in a PARAMETER statement. Any symbolic name of a constant that appears in an expression must have been defined previously in the same or a different PARAMETER statement in the same program unit. This constraint is enforced by the `SymConstEval` chain.

PARAMETER Statement[145] ≡

```
CHAIN SymConstEval: VOID;

SYMBOL xProgramUnit COMPUTE
  CHAINSTART HEAD.SymConstEval=0;
END;

RULE: xNamedConstantDef ::= xNamedConstant '=' xExpr
COMPUTE
  xNamedConstantDef.SymConstEval=
    ResetValue(xNamedConstant.ObjectKey,xExpr.Value)
    <- xExpr.SymConstEval;
  IF(NOT(xExpr.Value),
    message(ERROR,"Cannot evaluate this expression",0,COORDREF));
END;
```

This macro is defined in definitions 136, 139, and 145.
This macro is invoked in definition 106.

8.7 EXTERNAL Statement

The nonterminal `xExternalName` represents the context of a symbolic name in an EXTERNAL statement. A statement function name must not appear in an EXTERNAL statement. If an intrinsic function name appears in an EXTERNAL statement in a program unit, that name becomes the name of an external procedure and an intrinsic function of the same name is not available for reference in the program unit. If the same name appears in an INTRINSIC statement, then there is an ambiguity that should be reported:

EXTERNAL Statement[146] ≡

```
SYMBOL xExternalName COMPUTE
  Check Ambiguity[213]('StatementFunction','THIS');
  Check Ambiguity[213]('IntrinsicFunction','THIS');
END;
```

This macro is defined in definitions 146 and 147.
This macro is invoked in definition 106.

Only one appearance of a symbolic name in all of the EXTERNAL statements of a program unit is permitted:

EXTERNAL Statement[147] ≡

```
  Report multiple[33]('xExternalName','external')
```

This macro is defined in definitions 146 and 147.
This macro is invoked in definition 106.

Properties supporting multiple definition reporting[148] ≡

```
externalApp: int;
```

This macro is defined in definitions 35, 50, 122, 137, 148, and 150.
This macro is invoked in definition 243.

8.8 INTRINSIC Statement

The nonterminal `xIntrinsicProcedureName` represents the context of a symbolic name in an INTRINSIC statement. Only one appearance of a symbolic name in all of the INTRINSIC statements of a program unit is permitted:

INTRINSIC Statement[149] ≡

```
  Report multiple[33]('xIntrinsicProcedureName','intrinsic')
```

This macro is invoked in definition 106.

Properties supporting multiple definition reporting[150] ≡

```
intrinsicApp: int;
```

This macro is defined in definitions 35, 50, 122, 137, 148, and 150.
This macro is invoked in definition 243.

9 DATA Statement

The nonterminal `xNamedConstantUse` represents the context of a symbolic name in the `clist` of a DATA statement. Only constants may appear in this context:

DATA Statement[151] ≡

```
SYMBOL xNamedConstantUse COMPUTE
  IF(NOT(Classified as[12](‘Constant’,‘THIS’)),
    message(ERROR,"Only a constant allowed here",0,COORDREF))
  <- INCLUDING xProgramUnit.ClassificationDone;
END;
```

This macro is invoked in definition 244.

10 Assignment Statements

Assignment Statements[152] ≡

```
RULE: xStmt ::= xLblDef xName '=' xExpr xEOS
COMPUTE
  Identify the assignment operator[153](‘xName.Type’)
END;

RULE: xStmt ::= xLblDef xName '(' xExprList ')' '=' xExpr xEOS
COMPUTE
  Identify the assignment operator[153](‘xName.Type’)
END;

SYMBOL xStmtFunctionRange: demands: DefTableKey;

RULE: xStmt ::= xLblDef xName xStmtFunctionRange
COMPUTE
  xStmtFunctionRange.demands=xName.Type;
END;

RULE: xStmtFunctionRange ::= '(' ')' '=' xExpr xEOS
COMPUTE
  Identify the assignment operator[153](‘xStmtFunctionRange.demands’)
END;

RULE: xStmtFunctionRange ::= '(' xSFDummyArgNameList ')' '=' xExpr xEOS
COMPUTE
  Identify the assignment operator[153](‘xStmtFunctionRange.demands’)
END;

RULE: xStmt ::= xLblDef xName '(' xExprList ')' xSubstringRange '=' xExpr xEOS
END;
```

This macro is invoked in definition 244.

Identify the assignment operator[153](◇1) ≡

```
.Operator=
  OilIdOp2(
    OilOpAssign,
    GetOilType(◇1,OilErrorType()),
    xExpr.TypeCode);
IF(NOT(OilIsValidOp(.Operator)),
  message(ERROR,"Invalid assignment",0,COORDREF));
```

This macro is invoked in definition 152.

10.1 Arithmetic Assignment Statement

Arithmetic Assignment Statement[154] ≡

```
INDICATION
  Assign:
    iiAsgn, irAsgn, idAsgn, icAsgn,
            rrAsgn, rdAsgn, rcAsgn,
                    ddAsgn, dcAsgn,
                    cdAsgn, ccAsgn;

OPER
 iiAsgn(IntegerType,IntegerType): VoidType;
 irAsgn(IntegerType,RealType): VoidType;
 idAsgn(IntegerType,DoublePrecisionType): VoidType;
 icAsgn(IntegerType,ComplexType): VoidType;

 rrAsgn(RealType,RealType): VoidType;
 rdAsgn(RealType,DoublePrecisionType): VoidType;
 rcAsgn(RealType,ComplexType): VoidType;

 ddAsgn(DoublePrecisionType,DoublePrecisionType): VoidType;
 dcAsgn(DoublePrecisionType,ComplexType): VoidType;

 cdAsgn(ComplexType,DoublePrecisionType): VoidType;
 ccAsgn(ComplexType,ComplexType): VoidType;
```

This macro is invoked in definition 245.

10.2 Logical Assignment Statement

Logical Assignment Statement[155] ≡

```
INDICATION
  Assign: llAsgn;

OPER
 llAsgn(LogicalType,LogicalType): VoidType;
```

This macro is invoked in definition 245.

10.3 Character Assignment Statement

Character Assignment Statement[156] ≡

```
INDICATION
  Assign: ssAsgn;

OPER
 ssAsgn(CharacterType,CharacterType): VoidType;
```

This macro is invoked in definition 245.

11 Control Statements

Control statements must be checked to verify that their target labels are attached to executable statements and that their control expressions are of the proper type.

Control Statements[157] ≡

```
SYMBOL xLblRef: JumpTarget: int;

SYMBOL xLblRef COMPUTE
  INH.JumpTarget=0;
  IF(AND(THIS.JumpTarget,NOT(GetExecutable(THIS.UnitKey,0))),
    message(ERROR,"Does not label an executable statement",0,COORDREF))
  <- INCLUDING xProgramUnit.GotAllExecs;
END;

RULE: xLblRefList ::= xLblRef
COMPUTE
  xLblRef.JumpTarget=1;
END;

RULE: xLblRefList ::= xLblRefList ',' xLblRef
COMPUTE
  xLblRef.JumpTarget=1;
END;
```

Unconditional GO TO Statement[158]
Computed GO TO Statement[159]
Assigned GO TO Statement[160]
Arithmetic IF Statement[161]
Logical IF Statement[162]
Block IF Statement[164]
ELSE IF Statement[165]
DO Statement[166]

This macro is invoked in definition 244.

11.1 Unconditional GO TO Statement

Unconditional GO TO Statement[158] ≡

```
RULE: xStmt ::= xLblDef GoToKw xLblRef xEOS
COMPUTE
  xLblRef.JumpTarget=1;
END;
```

This macro is invoked in definition 157.

11.2 Computed GO TO Statement

Computed GO TO Statement[159] ≡

```
RULE: xStmt ::= xLblDef GoToKw '(' xLblRefList ')' xExpr xEOS
COMPUTE
  IF(NE(xExpr.TypeCode,OilTypeIntegerType),
    message(ERROR,"Control expression must yield an integer",0,COORDREF));
END;
```

This macro is invoked in definition 157.

11.3 Assigned GO TO Statement

Assigned GO TO Statement[160] ≡

```
RULE: xStmt ::= xLblDef GoToKw xVariableName '(' xLblRefList ')' xEOS
COMPUTE
  IF(NE(GetType(xVariableName.UnitKey,NoKey),IntegerType),
    message(ERROR,"Target must be an integer variable",0,COORDREF));
END;

RULE: xStmt ::= xLblDef GoToKw xVariableName xEOS
COMPUTE
  IF(NE(GetType(xVariableName.UnitKey,NoKey),IntegerType),
    message(ERROR,"Target must be an integer variable",0,COORDREF));
END;
```

This macro is invoked in definition 157.

11.4 Arithmetic IF Statement

Arithmetic IF Statement[161] ≡

```
RULE: xStmt ::= xLblDef 'if' '(' xExpr ')' xLblRef ',' xLblRef ',' xLblRef xEOS
COMPUTE
  xLblRef[1].JumpTarget=1;
```

```
      xLblRef[2].JumpTarget=1;
      xLblRef[3].JumpTarget=1;
      IF(NOT(OilIsValidCS(OilCoerce(xExpr.TypeCode,OilTypeDoublePrecisionType))),
        message(
          ERROR,
          "Control expression must yield integer, real or double precision",
          0,
          COORDREF));
    END;
```

This macro is invoked in definition 157.

11.5 Logical IF Statement

Logical IF Statement[162] ≡

```
RULE: xStmt ::= xLblDef 'if' '(' xExpr ')' xStmt
COMPUTE
  IF(NE(xExpr.TypeCode,OilTypeLogicalType),
    message(ERROR,"Control expression must be logical",0,COORDREF));
END;
```

This macro is defined in definitions 162 and 163.
This macro is invoked in definition 157.

The statement controlled by the logical IF can be any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END or another logical IF statement. All of these cases except for the logical IF are rejected by the concrete syntax.

Logical IF Statement[163] ≡

```
ATTR InLogicalIF: int;

SYMBOL xStmt COMPUTE
  INH.InLogicalIF=0;
END;

RULE: xStmt ::= xLblDef 'if' '(' xExpr ')' xStmt
COMPUTE
  xStmt[2].InLogicalIF=1;
  IF(xStmt[1].InLogicalIF,
    message(ERROR,"Illegal controlled statement",0,COORDREF));
END;
```

This macro is defined in definitions 162 and 163.
This macro is invoked in definition 157.

11.6 Block IF Statement

Block IF Statement[164] ≡

```
RULE: xStmt ::= xLblDef 'if' '(' xExpr ')' 'then' xEOS
COMPUTE
  IF(NE(xExpr.TypeCode,OilTypeLogicalType),
    message(ERROR,"Control expression must be logical",0,COORDREF));
END;
```

This macro is invoked in definition 157.

11.7 ELSE IF Statement

ELSE IF Statement[165] ≡

```
RULE: xStmt ::= xLblDef 'elseif' '(' xExpr ')' 'then' xEOS
COMPUTE
  IF(NE(xExpr.TypeCode,OilTypeLogicalType),
    message(ERROR,"Control expression must be logical",0,COORDREF));
END;
```

This macro is invoked in definition 157.

11.8 DO Statement

The `xLblRef` of a DO statement must label an executable statement, the controlled variable must be of type integer, real or double precision, and the expressions must yield integer, real or double precision values.

DO Statement[166] ≡

```
RULE: xStmt ::= xLblDef 'do' xLblRef xLoopControl xEOS
COMPUTE
  xLblRef.JumpTarget=1;
END;

RULE: xLoopControl ::= xVariableName '=' xExpr ',' xExpr
COMPUTE
  IF(
    NOT(
      OilIsValidCS(
        OilCoerce(GetOilType(xVariableName.Type,OilErrorType()),
        OilTypeDoublePrecisionType))),
    message(
      ERROR,
      "Controlled variable must be integer, real or double precision",
      0,
      COORDREF));
  IF(NOT(OilIsValidCS(OilCoerce(xExpr[1].TypeCode,OilTypeDoublePrecisionType))),
    message(
      ERROR,
      "Initial value must be integer, real or double precision",
      0,
      COORDREF));
  IF(NOT(OilIsValidCS(OilCoerce(xExpr[2].TypeCode,OilTypeDoublePrecisionType))),
    message(
      ERROR,
      "Final value must be integer, real or double precision",
      0,
      COORDREF));
END;

RULE: xLoopControl ::= xVariableName '=' xExpr ',' xExpr ',' xExpr
COMPUTE
  IF(
    NOT(
      OilIsValidCS(
        OilCoerce(GetOilType(xVariableName.Type,OilErrorType()),
        OilTypeDoublePrecisionType))),
    message(
      ERROR,
      "Controlled variable must be integer, real or double precision",
      0,
      COORDREF));
  IF(NOT(OilIsValidCS(OilCoerce(xExpr[1].TypeCode,OilTypeDoublePrecisionType))),
    message(
      ERROR,
      "Initial value must be integer, real or double precision",
      0,
      COORDREF));
  IF(NOT(OilIsValidCS(OilCoerce(xExpr[2].TypeCode,OilTypeDoublePrecisionType))),
    message(
      ERROR,
      "Final value must be integer, real or double precision",
      0,
      COORDREF));
  IF(NOT(OilIsValidCS(OilCoerce(xExpr[3].TypeCode,OilTypeDoublePrecisionType))),
    message(
      ERROR,
      "Step value must be integer, real or double precision",
      0,
      COORDREF));
END;
```

This macro is defined in definitions 166, 167, and 168.
This macro is invoked in definition 157.

The terminal statement of a DO-loop must not be an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement. If the terminal statement of a DO-loop is a logical IF statement, it may contain any of the statements it could otherwise contain.

DO Statement[167] ≡

```
RULE: xStmt ::= xLblDef GoToKw xLblRef xEOS
COMPUTE
  IF(NE(xStmt.dolist,NULLDefTableKeyList),
    message(ERROR,"Illegal DO-loop terminal statement",0,COORDREF));
```

```
END;

RULE: xStmt ::= xLblDef GoToKw xVariableName '(' xLblRefList ')' xEOS
COMPUTE
  IF(NE(xStmt.dolist,NULLDefTableKeyList),
    message(ERROR,"Illegal DO-loop terminal statement",0,COORDREF));
END;

RULE: xStmt ::= xLblDef GoToKw xVariableName xEOS
COMPUTE
  IF(NE(xStmt.dolist,NULLDefTableKeyList),
    message(ERROR,"Illegal DO-loop terminal statement",0,COORDREF));
END;

RULE: xStmt ::= xLblDef 'if' '(' xExpr ')' xLblRef ',' xLblRef ',' xLblRef xEOS
COMPUTE
  IF(NE(xStmt.dolist,NULLDefTableKeyList),
    message(ERROR,"Illegal DO-loop terminal statement",0,COORDREF));
END;

RULE: xStmt ::= xLblDef 'if' '(' xExpr ')' 'then' xEOS
COMPUTE
  IF(NE(xStmt.dolist,NULLDefTableKeyList),
    message(ERROR,"Illegal DO-loop terminal statement",0,COORDREF));
END;

RULE: xStmt ::= xLblDef 'return' xEOS
COMPUTE
  IF(NE(xStmt.dolist,NULLDefTableKeyList),
    message(ERROR,"Illegal DO-loop terminal statement",0,COORDREF));
END;

RULE: xStmt ::= xLblDef 'return' xExpr xEOS
COMPUTE
  IF(NE(xStmt.dolist,NULLDefTableKeyList),
    message(ERROR,"Illegal DO-loop terminal statement",0,COORDREF));
END;

RULE: xStmt ::= xLblDef 'stop' xEOS
COMPUTE
  IF(NE(xStmt.dolist,NULLDefTableKeyList),
    message(ERROR,"Illegal DO-loop terminal statement",0,COORDREF));
END;

RULE: xStmt ::= xLblDef 'stop' xIcon xEOS
COMPUTE
  IF(NE(xStmt.dolist,NULLDefTableKeyList),
    message(ERROR,"Illegal DO-loop terminal statement",0,COORDREF));
END;

RULE: xStmt ::= xLblDef 'stop' xScon xEOS
```

```
COMPUTE
  IF(NE(xStmt.dolist,NULLDefTableKeyList),
    message(ERROR,"Illegal DO-loop terminal statement",0,COORDREF));
END;

RULE: xEndSubroutineStmt ::= xLblDef 'end' xEOS
COMPUTE
  IF(
    NE(
      GetDoList(xLblDef.UnitKey,NULLDefTableKeyList) <- xLblDef.DoSeq,
      NULLDefTableKeyList),
    message(ERROR,"Illegal DO-loop terminal statement",0,COORDREF));
END;

RULE: xEndFunctionStmt ::= xLblDef 'end' xEOS
COMPUTE
  IF(
    NE(
      GetDoList(xLblDef.UnitKey,NULLDefTableKeyList) <- xLblDef.DoSeq,
      NULLDefTableKeyList),
    message(ERROR,"Illegal DO-loop terminal statement",0,COORDREF));
END;

RULE: xEndProgramStmt ::= xLblDef 'end' xEOS
COMPUTE
  IF(
    NE(
      GetDoList(xLblDef.UnitKey,NULLDefTableKeyList) <- xLblDef.DoSeq,
      NULLDefTableKeyList),
    message(ERROR,"Illegal DO-loop terminal statement",0,COORDREF));
END;
```

This macro is defined in definitions 166, 167, and 168.
This macro is invoked in definition 157.

The range of a DO-loop consists of all of the executable statements that appear following the DO statement that specifies the DO-loop, up to and including the terminal statement of the DO-loop.

If a DO statement appears within the range of a DO-loop, the range of the DO-loop specified by that DO statement must be contained entirely within the range of the outer DO-loop. More than one DO-loop may have the same terminal statement.

DO Statement[168] ≡

```
CHAIN DoSeq: VOID;
ATTR dokey: DefTableKey;
ATTR dolist: DefTableKeyList;

SYMBOL xSourceFile COMPUTE
  CHAINSTART HEAD.DoSeq=1;
END;

SYMBOL xProgramUnit COMPUTE
```

```
  THIS.DoSeq=FlushDoSeq() <- TAIL.DoSeq;
END;

RULE: xStmt ::= xLblDef 'do' xLblRef xLoopControl xEOS
COMPUTE
  .dokey=NewKey();
  xStmt.DoSeq=
    ORDER(
      ResetLocation(.dokey,COORDREF),
      DoSeqStackPush(.dokey),
      NestDoList(xLblRef.UnitKey,.dokey))
    <- xStmt.DoSeq;
  IF(NE(xStmt.dolist,NULLDefTableKeyList),
    message(ERROR,"DO terminator may not be a DO",0,COORDREF));
END;

SYMBOL xStmt COMPUTE
  SYNT.dolist=
    GetDoList(CONSTITUENT xLblDef.UnitKey,NULLDefTableKeyList)
    <- THIS.DoSeq;
  THIS.DoSeq=
    IF(NE(SYNT.dolist,NULLDefTableKeyList),
      CheckNesting(SYNT.dolist),
      THIS.DoSeq);
END;
```

This macro is defined in definitions 166, 167, and 168.
This macro is invoked in definition 157.

Property for DO sequence testing[169] ≡

```
DoList: DefTableKeyList [Nest]; "DefTableKeyList.h"
Location: CoordPtr;

void Nest(DefTableKey key, DefTableKey d)
{ if (key == NoKey) return;
  if (!ACCESS) VALUE = NULLDefTableKeyList;
  VALUE = ConsDefTableKeyList(d, VALUE);
}
```

This macro is invoked in definition 243.

Instantiate modules for DO sequence testing[170] ≡

```
$/Adt/PtrList.gnrc +instance=DefTableKey +referto=deftbl :inst
$/Adt/Stack.gnrc +instance=DoSeq +referto=DefTableKey :inst
```

This macro is invoked in definition 242.

DO nesting checker interface[171] ≡

```
#include "DefTableKeyList.h"
#include "deftbl.h"
```

```
extern void CheckNesting ELI_ARG((DefTableKeyList));
extern void FlushDoSeq ELI_ARG((void));
```

This macro is invoked in definition 248.

Verify legal DO nesting[172] ≡

```
#include "DoSeqStack.h"

static int
#ifdef PROTO_OK
EqDefTableKey(DefTableKey k1, DefTableKey k2)
#else
EqDefTableKey(k1, k2) DefTableKey k1, k2;
#endif
{ return k1 == k2; }

void
#ifdef PROTO_OK
CheckNesting(DefTableKeyList l)
#else
CheckNesting(l) DefTableKeyList l;
#endif
{ if (DoSeqStackEmpty) return;
  while (l != NULLDefTableKeyList) {
    DefTableKey hd = HeadDefTableKeyList(l);
    if (hd == DoSeqStackTop) {
      l = TailDefTableKeyList(l);
      (void)DoSeqStackPop;
    } else {
      DefTableKey *item; int found = 0;
      ForEachDoSeqStackElementDown(item) if (*item == hd) {found = 1; break; }
      if (found) {
        message(
          ERROR,
          "Improperly nested DO",
          0,
          GetLocation(DoSeqStackPop,NoPosition));
      } else l = TailDefTableKeyList(l);
    }
  }
}

void
#ifdef PROTO_OK
FlushDoSeq(void)
#else
FlushDoSeq()
#endif
{ while (!DoSeqStackEmpty)
    message(
      ERROR,
      "Improperly nested DO",
      0,
      GetLocation(DoSeqStackPop,NoPosition));
}
```

This macro is invoked in definition 247.

The restrictions on nesting of DO-loops and block IF constructs are enforced by the grammar.

12 Format Specification

A format specification may be given in a FORMAT statement or as the value of a character array, character variable or other character expression. Because it is possible to construct format specifications at run time, FORTRAN compilers do not generally translate those found in FORMAT statements. Instead, they are simply stored as character constants.

The FORMAT statement can be considered as a character constant expression of a special form, which must be evaluated by the compiler.

Format Specification[173] ≡

```
CHAIN FmtSpecDone: VOID;
ATTR Fmt: CharPtr;

SYMBOL xSourceFile COMPUTE
    CHAINSTART HEAD.FmtSpecDone = 1;
END;

RULE: xStmt ::= xLblDef 'format' '(' xFmtSpec ')' xEOS
COMPUTE
  xFmtSpec.FmtSpecDone=obstack_1grow(Fmt_obstk,'(')
    <- xStmt.FmtSpecDone;
  .Fmt=
    CAST(CharPtr,obstack_strcpy(Fmt_obstk,")")) <- xFmtSpec.FmtSpecDone;
  xStmt.FmtSpecDone = .Fmt;
END;
```

This macro is defined in definitions 173, 175, 177, 179, 180, and 183.
This macro is invoked in definition 244.

Format specifications are stored in a separate area, `Fmt_obstk`, to avoid any interdependence between format specification evaluation and other operations on character strings.

Evaluation functions[174] ≡

```
ObstackP Fmt_obstk = (ObstackP)0;
```

This macro is defined in definitions 24, 98, 111, 174, 181, 184, and 186.
This macro is invoked in definition 247.

A format specification is a list whose items are edit descriptors. The commas separating the elements of the list must appear in the format specification, so they are added to the string being built at the appropriate points:

Format Specification[175] ≡

```
RULE: xFmtSpec ::= xFmtSpec ',' xFmtSpec
COMPUTE
  xFmtSpec[3].FmtSpecDone=
    obstack_1grow(Fmt_obstk,',') <- xFmtSpec[2].FmtSpecDone;
END;

RULE: xFmtSpec ::= xFmtSpec ',' xFmtSpec xFmtSpec
COMPUTE
  xFmtSpec[3].FmtSpecDone=
    obstack_1grow(Fmt_obstk,',') <- xFmtSpec[2].FmtSpecDone;
END;
```

This macro is defined in definitions 173, 175, 177, 179, 180, and 183.
This macro is invoked in definition 244.

Most of the edit descriptors fall into one of two categories, depending upon whether they are represented by literal or nonliteral terminal symbols in the grammar. Literal terminal symbols have no specified values, so strings for them must be constructed as the format specification is being built. Since each edit descriptor that appears as a literal terminal symbol is a single character, that character is simply added to the format specification:

Literal format specification[176](◇1) ≡

```
RULE: xFmtSpec ::= ◇1
COMPUTE
  xFmtSpec.FmtSpecDone=
    obstack_1grow(Fmt_obstk,◇1) <- xFmtSpec.FmtSpecDone;
END;
```

This macro is invoked in definition 177.

Format Specification[177] ≡

Literal format specification[176](`'/'`)
Literal format specification[176](`':'`)

This macro is defined in definitions 173, 175, 177, 179, 180, and 183.
This macro is invoked in definition 244.

Nonliteral terminal symbols have specified values, so these values are available to be added to the string being built. For most edit descriptors of this form, the values are given as string table indices:

Nonliteral format specification[178](◇1) ≡

```
RULE: xFmtSpec ::= ◇1
COMPUTE
  xFmtSpec.FmtSpecDone=
    obstack_strgrow(Fmt_obstk,StringTable(◇1)) <- xFmtSpec.FmtSpecDone;
END;
```

This macro is invoked in definition 179.

Format Specification[179] ≡

Nonliteral format specification[178](‘`xXcon`’)
Nonliteral format specification[178](‘`xPcon`’)
Nonliteral format specification[178](‘`xFcon`’)
Nonliteral format specification[178](‘`xIdent`’)

```
RULE: xFmtSpec ::= xPcon xFmtSpec
COMPUTE
  xFmtSpec[2].FmtSpecDone=
    obstack_strgrow(Fmt_obstk,StringTable(xPcon))
    <- xFmtSpec[1].FmtSpecDone;
END;
```

This macro is defined in definitions 173, 175, 177, 179, 180, and 183.
This macro is invoked in definition 244.

The value of the nonliteral terminal symbol `xIcon` is stored as an integer rather than a string. Thus a routine is required to convert that value into a string and add it to the format specification.

Format Specification[180] ≡

```
RULE: xFmtSpec ::= xIcon xFmtSpec
COMPUTE
  xFmtSpec[2].FmtSpecDone=FmtInt(xIcon) <- xFmtSpec[1].FmtSpecDone;
END;

RULE: xFmtSpec ::= xPcon xIcon xFmtSpec
COMPUTE
  xFmtSpec[2].FmtSpecDone=
    ORDER(obstack_strgrow(Fmt_obstk,StringTable(xPcon)),FmtInt(xIcon))
    <- xFmtSpec[1].FmtSpecDone;
END;
```

This macro is defined in definitions 173, 175, 177, 179, 180, and 183.
This macro is invoked in definition 244.

Evaluation functions[181] ≡

```
void
#ifdef PROTO_OK
FmtInt(int v)
#else
FmtInt(v) int v;
#endif
{ int rest = v / 10, digit = v % 10 + '0';

  if (rest) FmtInt(rest);
  obstack_1grow(Fmt_obstk, digit);
}
```

This macro is defined in definitions 24, 98, 111, 174, 181, 184, and 186.
This macro is invoked in definition 247.

Evaluation function interfaces[182] ≡

```
extern ObstackP Fmt_obstk;
extern void FmtInt ELI_ARG((int));
```

This macro is defined in definitions 25, 99, 110, 112, 182, 185, and 187.
This macro is invoked in definition 248.

String constants are most easily produced by apostrophe editing. This means that the string must be enclosed in apostrophes, and every internal apostrophe must be doubled. Although this involves some effort, the string need only be scanned once. If H editing were to be used, the string would have to be scanned once to determine its length and then again to copy it into the format specification. Moreover, the length would have to be converted to a digit string.

Format Specification[183] ≡

```
RULE: xFmtSpec ::= xScon
COMPUTE
  xFmtSpec.FmtSpecDone=
    FmtString(StringTable(xScon)) <- xFmtSpec.FmtSpecDone;
END;
```

This macro is defined in definitions 173, 175, 177, 179, 180, and 183.
This macro is invoked in definition 244.

Evaluation functions[184] ≡

```
void
#ifdef PROTO_OK
FmtString(char *v)
#else
FmtString(v) char *v;
#endif
{ obstack_1grow(Fmt_obstk, '\'');
  while (*v) {
    if (*v == '\'') obstack_1grow(Fmt_obstk, '\'');
    obstack_1grow(Fmt_obstk, *v);
    v++;
  }
  obstack_1grow(Fmt_obstk, '\'');
}
```

This macro is defined in definitions 24, 98, 111, 174, 181, 184, and 186.
This macro is invoked in definition 247.

Evaluation function interfaces[185] ≡

```
extern void FmtString ELI_ARG((char *));
```

This macro is defined in definitions 25, 99, 110, 112, 182, 185, and 187.
This macro is invoked in definition 248.

`Fmt_obstk` must be defined and then intialized at the beginning of execution:

Evaluation functions[186] ≡

```
static Obstack Fmt_obstk_data;
static char *beginning;

void
#ifdef PROTO_OK
FmtInit(void)
#else
FmtInit()
#endif
{ if (Fmt_obstk) obstack_free(Fmt_obstk, beginning);
  else {
    obstack_init(&Fmt_obstk_data);
    Fmt_obstk = &Fmt_obstk_data;
    beginning = (char *)obstack_alloc(Fmt_obstk, 0);
  }
}
```

This macro is defined in definitions 24, 98, 111, 174, 181, 184, and 186.
This macro is invoked in definition 247.

Evaluation function interfaces[187] ≡

```
extern void FmtInit ELI_ARG((void));
```

This macro is defined in definitions 25, 99, 110, 112, 182, 185, and 187.
This macro is invoked in definition 248.

Initialization[188] ≡

```
FmtInit();
```

This macro is defined in definitions 100 and 188.
This macro is invoked in definition 249.

13 Functions and Subroutines

13.1 Function Subprogram and FUNCTION Statement

The restriction that a FUNCTION statement appear only as the first statement of a function subprogram is enforced by the grammar. The restriction that the symbolic name of an external function not be the same as any local name other than a variable name in the function subprogram also appears in Section 14.2.2, and is enforced as noted there.

Function Subprogram and FUNCTION Statement[189] ≡

```
RULE: xFunctionPrefix ::= xTypeSpec ’function’
COMPUTE
  xFunctionPrefix.Type=xTypeSpec.Type;
END;

RULE: xFunctionPrefix ::= ’function’
COMPUTE
```

```
  xFunctionPrefix.Type=NoKey;
END;

RULE: xProgramUnit ::= xLblDef xFunctionPrefix xFunctionName xSubprogramRange
COMPUTE
  xSubprogramRange.GotType=
    IF(NE(xFunctionPrefix.Type,NoKey),
      IsType(xFunctionName.UnitKey,xFunctionPrefix.Type,ErrorType));
END;

SYMBOL xSubprogramRange COMPUTE
  THIS.GotType=1;
END;
```

This macro is invoked in definition 244.

13.2 ENTRY Statement

The grammar guarantees that an ENTRY statement appears after the FUNCTION or SUBROUTINE statement of a subprogram, but semantic checks are required to verify that it is not in a main program or block data subprogram.

ENTRY Statement[190] ≡

```
RULE: xStmt ::= xLblDef 'entry' xEntryName xFormalParameterList xEOS
COMPUTE
  IF(
    OR(
      EQ(INCLUDING xProgramUnit.Kind,MainProgram),
      EQ(INCLUDING xProgramUnit.Kind,BlockDataSubprogram)),
    message(ERROR,
      "ENTRY statement allowed only in function or subroutine",0,COORDREF));
END;
```

Sequence the elements[3](‘`xFormalParameter`’)

```
RULE: xFormalParameterList ::=
COMPUTE
  xFormalParameterList.SeqCount=0;
END;
```

This macro is invoked in definition 244.

13.3 RETURN Statement

RETURN Statement[191] ≡

```
RULE: xStmt ::= xLblDef 'return' xEOS
COMPUTE
  IF(
    OR(
      EQ(INCLUDING xProgramUnit.Kind,MainProgram),
      EQ(INCLUDING xProgramUnit.Kind,BlockDataSubprogram)),
    message(ERROR,
      "RETURN statement allowed only in function or subroutine",0,COORDREF));
END;

RULE: xStmt ::= xLblDef 'return' xExpr xEOS
COMPUTE
  IF(
    OR(
      EQ(INCLUDING xProgramUnit.Kind,MainProgram),
      EQ(INCLUDING xProgramUnit.Kind,BlockDataSubprogram)),
    message(ERROR,
      "RETURN statement allowed only in function or subroutine",0,COORDREF));
END;
```

This macro is invoked in definition 244.

13.4 Table of Intrinsic Functions

Intrinsic functions are listed in Section 15.10 (Table 5) of the standard. Their symbolic names are predefined:

Table of Intrinsic Functions[192] ≡

```
PreDef("int",    INTKey,    IntegerType)
PreDef("ifix",   IFIXKey,   IntegerType)
PreDef("idint",  IDINTKey,  IntegerType)
PreDef("real",   REALKey,   RealType)
PreDef("float",  FLOATKey,  RealType)
PreDef("sngl",   SNGLKey,   RealType)
PreDef("dble",   DBLEKey,   DoublePrecisionType)
PreDef("cmplx",  CMPLXKey,  ComplexType)
PreDef("ichar",  ICHARKey,  IntegerType)
PreDef("char",   CHARKey,   CharacterType)
PreDef("aint",   AINTKey,   RealType)
PreDef("dint",   DINTKey,   DoublePrecisionType)
PreDef("anint",  ANINTKey,  RealType)
PreDef("dnint",  DNINTKey,  DoublePrecisionType)
PreDef("nint",   NINTKey,   IntegerType)
PreDef("idnint", IDNINTKey, IntegerType)
PreDef("abs",    ABSKey,    RealType)
PreDef("iabs",   IABSKey,   IntegerType)
PreDef("dabs",   DABSKey,   DoublePrecisionType)
PreDef("cabs",   CABSKey,   RealType)
PreDef("mod",    MODKey,    IntegerType)
PreDef("amod",   AMODKey,   RealType)
PreDef("dmod",   DMODKey,   DoublePrecisionType)
PreDef("sign",   SIGNKey,   RealType)
PreDef("isign",  ISIGNKey,  IntegerType)
PreDef("dsign",  DSIGNKey,  DoublePrecisionType)
PreDef("dim",    DIMKey,    RealType)
PreDef("idim",   IDIMKey,   IntegerType)
PreDef("ddim",   DDIMKey,   DoublePrecisionType)
PreDef("dprod",  DPRODKey,  DoublePrecisionType)
PreDef("max",    MAXKey,    IntegerType)
PreDef("max0",   MAX0Key,   IntegerType)
PreDef("amax1",  AMAX1Key,  RealType)
PreDef("dmax1",  DMAX1Key,  DoublePrecisionType)
PreDef("amax0",  AMAX0Key,  RealType)
PreDef("max1",   MAX1Key,   IntegerType)
PreDef("min",    MINKey,    IntegerType)
PreDef("min0",   MIN0Key,   IntegerType)
PreDef("amin1",  AMIN1Key,  RealType)
PreDef("dmin1",  DMIN1Key,  DoublePrecisionType)
PreDef("amin0",  AMIN0Key,  RealType)
PreDef("min1",   MIN1Key,   IntegerType)
PreDef("len",    LENKey,    IntegerType)
PreDef("index",  INDEXKey,  IntegerType)
PreDef("aimag",  AIMAGKey,  RealType)
PreDef("conjg",  CONJGKey,  ComplexType)
PreDef("sqrt",   SQRTKey,   RealType)
PreDef("dsqrt",  DSQRTKey,  DoublePrecisionType)
PreDef("csqrt",  CSQRTKey,  ComplexType)
PreDef("exp",    EXPKey,    RealType)
PreDef("dexp",   DEXPKey,   DoublePrecisionType)
PreDef("cexp",   CEXPKey,   ComplexType)
PreDef("log",    LOGKey,    RealType)
PreDef("alog",   ALOGKey,   RealType)
PreDef("dlog",   DLOGKey,   DoublePrecisionType)
PreDef("clog",   CLOGKey,   ComplexType)
PreDef("log10",  LOG10Key,  RealType)
PreDef("alog10", ALOG10Key, RealType)
PreDef("dlog10", DLOG10Key, DoublePrecisionType)
PreDef("sin",    SINKey,    RealType)
PreDef("dsin",   DSINKey,   DoublePrecisionType)
PreDef("csin",   CSINKey,   ComplexType)
PreDef("cos",    COSKey,    RealType)
PreDef("dcos",   DCOSKey,   DoublePrecisionType)
PreDef("ccos",   CCOSKey,   ComplexType)
PreDef("tan",    TANKey,    RealType)
PreDef("dtan",   DTANKey,   DoublePrecisionType)
PreDef("asin",   ASINKey,   RealType)
PreDef("dasin",  DASINKey,  DoublePrecisionType)
PreDef("acos",   ACOSKey,   RealType)
PreDef("dacos",  DACOSKey,  DoublePrecisionType)
PreDef("atan",   ATANKey,   RealType)
PreDef("datan",  DATANKey,  DoublePrecisionType)
PreDef("atan2",  ATAN2Key,  RealType)
PreDef("datan2", DATAN2Key, DoublePrecisionType)
PreDef("sinh",   SINHKey,   RealType)
PreDef("dsinh",  DSINHKey,  DoublePrecisionType)
PreDef("cosh",   COSHKey,   RealType)
PreDef("dcosh",  DCOSHKey,  DoublePrecisionType)
PreDef("tanh",   TANHKey,   RealType)
PreDef("dtanh",  DTANHKey,  DoublePrecisionType)
PreDef("lge",    LGEKey,    LogicalType)
PreDef("lgt",    LGTKey,    LogicalType)
PreDef("lle",    LLEKey,    LogicalType)
PreDef("llt",    LLTKey,    LogicalType)
```

This macro is invoked in definitions 193, 197, and 240.

Pre-definition of intrinsic function types[193] ≡

```
#define PreDef(sym,key,type) key -> Type={type};
Table of Intrinsic Functions[192]
#undef PreDef
```

This macro is invoked in definition 243.

14 Scope and Classes of Symbolic Names

A symbolic name is represented in the grammar by an `xIdent` symbol, and the sequence of alphanumeric characters making up the name is encoded as the value of that symbol. The scanner requires that a symbolic name consist of one or more alphanumeric characters, the first of which must be a letter; the upper limit of six is enforced by checking the `Sym` attribute value:

Scope and Classes of Symbolic Names[194] ≡

```
SYMBOL SymbolicName COMPUTE
  SYNT.Sym=TERM;
/*
  IF(GT(strlen(StringTable(THIS.Sym)),6),
    message(ERROR,"Symbolic name exceeds six characters",0,COORDREF));
*/
END;
```

This macro is invoked in definition 244.

14.1 Scope of Symbolic Names

The nonterminals `xSourceFile`, `xProgramUnit`, `xStmtFunctionRange` and `xDataImpliedDo` represent the four possible scopes of symbolic names.

Scope of Symbolic Names[195] ≡

```
RULE:  xSourceFile LISTOF xProgramUnit END;

SYMBOL xSourceFile        INHERITS RootScope END;
SYMBOL xProgramUnit       INHERITS RangeScope END;
SYMBOL xStmtFunctionRange INHERITS RangeScope END;
SYMBOL xDataImpliedDo     INHERITS RangeScope END;
```

This macro is defined in definitions 195, 196, 199, 200, 202, and 203.
This macro is invoked in definition 244.

`xStmtFunctionRange` constitutes a new scope only if the symbolic name actually identifies a statement function. If not, then the phrase belongs to the enclosing scope:

Scope of Symbolic Names[196] ≡

```
RULE: xStmt ::= xLblDef xName xStmtFunctionRange
COMPUTE
  xStmtFunctionRange.Env=
    IF(This is a statement function statement[237],
      NewScope(INCLUDING AnyScope.Env),
      INCLUDING AnyScope.Env);
END;
```

This macro is defined in definitions 195, 196, 199, 200, 202, and 203.
This macro is invoked in definition 244.

It is convenient to treat the names of the intrinsic functions as though they were defined in the outermost scope, and then let defining occurrences of symbolic names in inner scopes re-define them. A pre-defined symbol is created by instantiating an Eli library module with the symbol as a parameter:

Place pre-defined symbols into the outermost environment[197] ≡

```
#define PreDef(sym,key,type) PreDefKey(sym, key)
Table of Intrinsic Functions[192]
```

This macro is invoked in definition 250.

This will associate a `DefTableKey`-valued value named `symKey` with the symbol `sym`. The `Key` attribute of each occurrence of the symbol `sym` will have the value `symKey` in any scope where the symbol `sym` is defined.

Instantiate the predefined identifier module[198] ≡

```
$/Name/PreDefine.gnrc +referto=xIdent           :inst
$/Name/PreDefId.gnrc  +referto=(f77semantics.d) :inst
```

This macro is invoked in definition 242.

The following nonterminals represent contexts in which a symbolic name supersedes the same name in an outer scope:

Scope of Symbolic Names[199] ≡

```
SYMBOL xProgramName       INHERITS IdDefScope END;
SYMBOL xBlockDataName     INHERITS IdDefScope END;
SYMBOL xFunctionName      INHERITS IdDefScope END;
SYMBOL xSubroutineName    INHERITS IdDefScope END;
SYMBOL xCommonBlockName   INHERITS IdUseEnv END;
SYMBOL xEntryName         INHERITS IdDefScope END;
SYMBOL xDummyArgName      INHERITS IdDefScope END;
SYMBOL xExternalName      INHERITS IdDefScope END;
SYMBOL xNamedConstant     INHERITS IdDefScope END;
SYMBOL xSFDummyArgName    INHERITS IdDefScope END;
SYMBOL xImpliedDoVariable INHERITS IdDefScope END;
```

This macro is defined in definitions 195, 196, 199, 200, 202, and 203.
This macro is invoked in definition 244.

The following nonterminals represent contexts in which a symbolic name may be a use of a symbolic name valid in the current scope or an outer scope:

Scope of Symbolic Names[200] ≡

```
SYMBOL xIntrinsicProcedureName  INHERITS IdUseEnv END;
SYMBOL xName                    INHERITS IdUseEnv END;
SYMBOL xNamedConstantUse        INHERITS IdUseEnv END;
SYMBOL xObjectName              INHERITS IdUseEnv END;
SYMBOL xSFVarName               INHERITS IdUseEnv END;
SYMBOL xSubroutineNameUse       INHERITS IdUseEnv END;
SYMBOL xVariableName            INHERITS IdUseEnv END;
```

This macro is defined in definitions 195, 196, 199, 200, 202, and 203.
This macro is invoked in definition 244.

This description of the scope rules constitutes a specification of a consistent renaming process that will be carried out by the Eli module implementing scopes that are the closest-containing range (these are the scope rules of ALGOL 60):

Instantiate the ALGOL 60 scope module[201] ≡

```
$/Name/AlgScope.gnrc :inst
```

This macro is defined in definitions 28, 201, and 207.
This macro is invoked in definition 242.

The effect of this module is to associate a unique `DefTableKey`-valued attribute named `Key` with every occurrence of a symbolic name that inherits computations from `IdDefScope`. Such an occurrence is called a *defining* occurrence of the symbolic name, while one that inherits computations from `IdUseEnv` is called an *applied* occurrence of the symbolic name. The `Key` attribute of an applied occurrence will have a value of `NoKey` whenever there is no defining occurrence of that symbol in the current scope or any outer scope.

Symbolic names with different scopes may have properties in common, and occurrences of the same symbol in different scopes may affect one another. Thus it is necessary to consider all instances of a single symbol in a program unit to to be instances of the same entity for classification purposes. Thus a `SymbolicName`, like an `xLabel`, should have a `UnitKey` attribute. The scope rules for this consistent renaming are identical to those for labels:

Scope of Symbolic Names[202] ≡

```
SYMBOL SymbolicName INHERITS UnitIdDefScope END;
```

This macro is defined in definitions 195, 196, 199, 200, 202, and 203.
This macro is invoked in definition 244.

Symbolic names in FORTRAN need not have any occurrence classified as an `IdDefScope`. In this case, the value of the `Key` attribute will be `NoKey`, and no properties can be stored under that key. Any such object has only one incarnation, and therefore the `UnitKey` can be used to access *all* of its properties. The `ObjectKey` attribute is the value used to access properties of a particular incarnation of the object:

Scope of Symbolic Names[203] ≡

```
SYMBOL SymbolicName COMPUTE
  SYNT.ObjectKey=IF(THIS.Key,THIS.Key,THIS.UnitKey);
END;
```

This macro is defined in definitions 195, 196, 199, 200, 202, and 203.
This macro is invoked in definition 244.

14.1.1 Global Entities

A symbolic name in any of the following classes is a global entity in an executable program:

Global Entities[204] ≡

```
CommonBlock
ExternalFunction
Subroutine
MainProgram
BlockDataSubprogram
```

This macro is NEVER invoked.

A symbolic name that identifies a global entity must not be used to identify any other global entity in the same executable program. In general, this condition cannot be verified until link time. Within a program unit, two cases must be considered: common block names and other global entities.

The nonterminal `xComblock` represents the context of a common block name anywhere in a program unit. A specific symbolic name may appear any number of times as a `xComblock` phrase, and every appearance names the same global entity. If a symbolic name appears as a `xComblock` phrase, however, it may *not* appear as the name of any other class of global entity:

Verify that a name identifies no more than one global entity[205] ≡

```
SYMBOL xCommonBlockName COMPUTE
  Check Ambiguity[213](‘ExternalFunction’,‘THIS’);
  Check Ambiguity[213](‘Subroutine’,‘THIS’);
  Check Ambiguity[213](‘MainProgram’,‘THIS’);
  Check Ambiguity[213](‘BlockDataSubprogram’,‘THIS’);
END;
```

This macro is defined in definitions 205, 206, and 208.
This macro is invoked in definition 244.

The nonterminal `xFunctionName` represents the context of a symbolic name immediately following the word FUNCTION in a FUNCTION statement, `xSubroutineName` represents the context of a symbolic name immediately following the word SUBROUTINE in a SUBROUTINE statement, `xProgramName` represents the context of a symbolic name immediately following the word PROGRAM in a PROGRAM statement, and `xBlockDataName` represents the context of a symbolic name immediately following the word BLOCKDATA in a block data statement. In each case, the symbolic name may appear no more than once in such a context, and it may not also appear as a `xComblock` phrase.

`ProgramUnitName` embodies the computations used to verify these restrictions:

Verify that a name identifies no more than one global entity[206] ≡

```
SYMBOL xFunctionName INHERITS ProgramUnitName END;
SYMBOL xSubroutineName INHERITS ProgramUnitName END;
SYMBOL xProgramName INHERITS ProgramUnitName END;
SYMBOL xBlockDataName INHERITS ProgramUnitName END;

SYMBOL xProgramUnit INHERITS RangeUnique END;
SYMBOL ProgramUnitName INHERITS Unique COMPUTE
  Check Ambiguity[213](‘CommonBlock’,‘THIS’);
  IF(NOT(THIS.Unique),
    message(ERROR,"Symbol represents more than one global entity",0,COORDREF));
END;
```

This macro is defined in definitions 205, 206, and 208.
This macro is invoked in definition 244.

The `Unique` module from the Eli library is used to verify that a symbolic name is used no more than once as a global entity other than a common block:

Instantiate the ALGOL 60 scope module[207] ≡

```
$/Prop/Unique.gnrc +referto=Unit :inst
```

This macro is defined in definitions 28, 201, and 207.
This macro is invoked in definition 242.

(Instantiating the module with `+referto=Unit` implies that the `UnitKey` attribute will be used in testing uniqueness.)

An entry point name inherits the classification of the name of the program unit in which it occurs, so it also has the properties of a program name:

Verify that a name identifies no more than one global entity[208] ≡

```
SYMBOL xEntryName INHERITS ProgramUnitName END;
```

This macro is defined in definitions 205, 206, and 208.
This macro is invoked in definition 244.

The nonterminal `xEntryName` represents the context of a symbolic name immediately following the word ENTRY in an ENTRY statement.

14.1.2 Local Entities

A symbolic name in any of the following classes is a local entity in a program unit:

Local Entities[209] ≡

```
  Array
  Variable
  Constant
  StatementFunction
  IntrinsicFunction
  DummyProcedure
```

This macro is NEVER invoked.

Within a program unit, a symbolic name that is in one class of entities local to the program unit must not also be in another class of entities local to the program unit. Violations of this restriction are specified below with each classification specification, because the classification methods make certain ambiguities impossible.

A symbolic name that identifies a global entity in a program unit must not be used to identify a local entity in that program unit, except for a common block name and an external function name:

Verify that a global name does not identify a local entity[210] ≡

```
SYMBOL xSubroutineName INHERITS NotCommonOrExtFn END;
SYMBOL xProgramName    INHERITS NotCommonOrExtFn END;
SYMBOL xBlockDataName  INHERITS NotCommonOrExtFn END;

SYMBOL NotCommonOrExtFn COMPUTE
  Check Ambiguity[213](‘Array’,‘THIS’);
  Check Ambiguity[213](‘Variable’,‘THIS’);
  Check Ambiguity[213](‘Constant’,‘THIS’);
  Check Ambiguity[213](‘StatementFunction’,‘THIS’);
  Check Ambiguity[213](‘IntrinsicFunction’,‘THIS’);
  Check Ambiguity[213](‘DummyProcedure’,‘THIS’);
END;
```

This macro is invoked in definition 244.

Error reports attached to the local entities violating this rule are specified below with each classification specification, because the contexts are relatively complex:

Verify this does not identify a global entity[211](◇2) ≡

```
  Check Ambiguity[213](‘Subroutine’,‘◇1’)◇2
  Check Ambiguity[213](‘MainProgram’,‘◇1’)◇2
  Check Ambiguity[213](‘BlockDataSubprogram’,‘◇1’)
```

This macro is invoked in definitions 231, 232, 234, 235, 238, 239, and 241.

Note that no check is made for an `ExternalFunction`. The reason is that an external function name must be used within that external function as a variable name. In some contexts, therefore, and external function name is legal even though names of other global entities are not. In contexts where an external function name is not allowed, an additional ambiguity check must be inserted.

14.2 Classes of Symbolic Names

The classification of a symbolic name depends on how it is used in the program unit. If a symbol appears in certain contexts, its class is fixed. The standard devotes one subsection to each entity class, describing the conditions under which a symbolic name is determined to be an entity of that class. This section is organized in the same way, using subsections equivalent to those of the standard and occurring in the same order. The specification in each subsection defines the classification in the corresponding subsection of the standard.

Classes of Symbolic Names[212] ≡

```
Common Block[216]
External Function[218]
```

Subroutine[224]
Main Program[227]
Block Data Subprogram[229]
Array[231]
Variable[233]
Constant[235]
Statement Function[236]
Intrinsic Function[239]
Dummy Procedure[241]

This macro is defined in definitions 212 and 215.
This macro is invoked in definition 244.

A `KindSet` property is associated with every symbol, in order to summarize the contexts in which that symbol appears. An ambiguity check is made by asking whether the specified class is a member of the symbol's `KindSet` property. By deferring the test until all usages have been seen, the specification guarantees that an error report is attached to each symbol involved in an ambiguity:

Check Ambiguity[213](◇2) ≡

```
IF(Classified as[12](‘◇1’,‘◇2’),
  message(ERROR,"Also classified as \"◇1\"",0,◇2.Coord))
<- INCLUDING xProgramUnit.ClassificationDone
```

This macro is invoked in definitions 146, 205, 206, 210, 211, 217, 226, 231, 232, 234, 235, 238, 239, and 241.

This operation is only invoked at instances of symbolic names where a particular classification is being set. Uses of a symbolic name that are improper for its classification are reported elsewhere.

The coordinates for the message must be obtained via a computation carried out in the lower context of the node representing the name:

Obtain the coordinates of a symbolic name[214] ≡

```
SYMBOL SymbolicName COMPUTE
  SYNT.Coord=COORDREF;
END;
```

This macro is invoked in definition 244.

The root attribute `GotAllContexts` is an assertion that all contexts in which a symbolic name appears have been summarized in the `KindSet` property. Once this information is available, final classification is possible.

Classes of Symbolic Names[215] ≡

```
SYMBOL xProgramUnit COMPUTE
  SYNT.GotAllContexts=CONSTITUENTS SymbolicName.GotContext;
  SYNT.ClassificationDone=CONSTITUENTS VariableAppearance.Classified;
END;
```

This macro is defined in definitions 212 and 215.
This macro is invoked in definition 244.

14.2.1 Common Block

The nonterminal `xComblock` represents the context of a common block name in a COMMON statement or SAVE statement:

Common Block[216] ≡

```
RULE: xComblock ::= '/' xCommonBlockName '/'
COMPUTE
  xCommonBlockName.GotContext=
    InsertKindSet(xCommonBlockName.UnitKey,CommonBlock);
END;
```

This macro is defined in definitions 216 and 217.
This macro is invoked in definition 212.

A common block name can also be used to name any local entity except a constant or intrinsic function:

Common Block[217] ≡

```
SYMBOL xCommonBlockName COMPUTE
  Check Ambiguity[213](‘Constant’,‘THIS’);
  Check Ambiguity[213](‘IntrinsicFunction’,‘THIS’);
END;
```

This macro is defined in definitions 216 and 217.
This macro is invoked in definition 212.

14.2.2 External Function

The nonterminal `xFunctionName` represents the context of a symbolic name immediately following the word FUNCTION in a FUNCTION statement:

External Function[218] ≡

```
SYMBOL xFunctionName COMPUTE
  INH.GotContext=InsertKindSet(THIS.UnitKey,ExternalFunction);
END;
```

This macro is defined in definitions 218, 219, 221, 222, and 223.
This macro is invoked in definition 212.

The nonterminal `xEntryName` represents the context of a symbolic name immediately following the word ENTRY in an ENTRY statement. That symbol could identify either an external function or a subroutine, depending upon the program unit in which it is embedded:

External Function[219] ≡

```
SYMBOL xEntryName COMPUTE
  INH.GotContext=InsertKindSet(THIS.UnitKey,INCLUDING xProgramUnit.Kind);
END;

RULE: xProgramUnit ::= xLblDef xFunctionPrefix xFunctionName xSubprogramRange
COMPUTE
```

```
    xProgramUnit.Kind=ExternalFunction;
  END;
```

This macro is defined in definitions 218, 219, 221, 222, and 223.
This macro is invoked in definition 212.

The second condition stated in Section 18.2.2 of the standard classifies a symbolic name as an external function if it appears immediately before a left parenthesis in executable code, and is *not* one of a list of local and global entities. Most of these entities can be easily eliminated because they must have been classified earlier and therefore the `KindSet` of the name would reflect that classification:

Set of known classes from condition (2), Section 18.2.2[220] ≡

```
  ConsIS(Array,
  ConsIS(StatementFunction,
  ConsIS(IntrinsicFunction,
  ConsIS(Subroutine,
    NullIS()))))
```

This macro is invoked in definition 223.

(`ConsIS` and `NullIS` are both constant expressions exported by the Eli integer set module, so the value described above is a constant value of type `IntSet`.)

The condition excludes `DummyArgument` also, but that is omitted from the list above because it must be used to distinguish between an external function and a dummy procedure.

By itself, this set does not guarantee that the symbolic name is not the name of an intrinsic function. Thus an intrinsic function may be classified as external by the operations below. In that case, however, the value of the `ObjectKey` attribute of the name will be the key of the appropriate intrinsic function.

Character variable names appearing immediately before left parentheses are distinguished by the form of the parenthesized phrase: The symbolic name identifies a character variable if and only if the parenthesized phrase (represented by the nonterminal `xSectionSubscriptList`) takes the form of a `xSubscriptTriplet` (see Section 5.7 of the standard).

Two rules relate `xSectionSubscriptList` with `xSubscriptTriplet`:

External Function[221] ≡

```
  ATTR NotCharacterIndex: int;

  RULE: xSectionSubscriptList ::= xSectionSubscript
  COMPUTE
    xSectionSubscriptList.NotCharacterIndex=xSectionSubscript.NotCharacterIndex;
  END;

  RULE: xSectionSubscript ::= xSubscriptTriplet
  COMPUTE
    xSectionSubscript.NotCharacterIndex=0;
  END;
```

This macro is defined in definitions 218, 219, 221, 222, and 223.
This macro is invoked in definition 212.

These two rules set `xSectionSubscriptList.NotCharacterIndex` false when the `xSectionSubscriptList` takes the form of a `xSubscriptTriplet`. In all other cases, this attribute must be set true:

External Function[222] ≡

```
SYMBOL NotSubscriptTriplet COMPUTE
  SYNT.NotCharacterIndex=1;
END;

SYMBOL xSectionSubscriptList INHERITS NotSubscriptTriplet END;
SYMBOL xSectionSubscript INHERITS NotSubscriptTriplet END;
```

This macro is defined in definitions 218, 219, 221, 222, and 223.
This macro is invoked in definition 212.

This computation specifies that the `NotCharacterIndex` attribute of *all* `xSectionSubscriptList` and `xSectionSubscript` nodes should be set true; it is overridden by the rule computations given above. The overall effect is to correctly describe the context represented by a `xSectionSubscriptList`.

The nonterminal `xName` represents the context of a symbolic name appearing in an executable statement.

External Function[223] ≡

```
RULE: xExpr ::= xName '(' ')'
COMPUTE
  xName.GotContext=
    IF(
      EmptyIS(
        InterIS(
          Set of known classes from condition (2), Section 18.2.2[220],
          GetKindSet(xName.UnitKey,NullIS()))),
      IF(Classified as[12](‘DummyArgument’,‘xName’),
        InsertKindSet(xName.UnitKey,DummyProcedure),
        IF(GetIntrinsic(xName.ObjectKey, 0),
          InsertKindSet(xName.UnitKey,IntrinsicFunction),
          InsertKindSet(xName.UnitKey,ExternalFunction))))
    <- xExpr.Order;
END;

RULE: xComplexDataRef ::= xName '(' xSectionSubscriptList ')'
COMPUTE
  xName.GotContext=
    IF(
      EmptyIS(
        InterIS(
          Set of known classes from condition (2), Section 18.2.2[220],
          GetKindSet(xName.UnitKey,NullIS()))),
      IF(xSectionSubscriptList.NotCharacterIndex,
        IF(Classified as[12](‘DummyArgument’,‘xName’),
          InsertKindSet(xName.UnitKey,DummyProcedure),
          IF(GetIntrinsic(xName.ObjectKey, 0),
            InsertKindSet(xName.UnitKey,IntrinsicFunction),
            InsertKindSet(xName.UnitKey,ExternalFunction))),
        InsertKindSet(xName.UnitKey,Variable)))
    <- xComplexDataRef.Order;
END;
```

This macro is defined in definitions 218, 219, 221, 222, and 223.
This macro is invoked in definition 212.

14.2.3 Subroutine

The nonterminal `xSubroutineName` represents the context of a symbolic name immediately following the word SUBROUTINE in a SUBROUTINE statement:

Subroutine[224] ≡

```
SYMBOL xSubroutineName COMPUTE
  INH.GotContext=InsertKindSet(THIS.UnitKey,Subroutine);
END;
```

This macro is defined in definitions 224, 225, and 226.
This macro is invoked in definition 212.

Entry statements are handled as for external functions. The `Kind` attribute of the program unit must indicate that the program unit is a subroutine:

Subroutine[225] ≡

```
RULE: xProgramUnit ::= xLblDef 'subroutine' xSubroutineName xSubprogramRange
COMPUTE
  xProgramUnit.Kind=Subroutine;
END;
```

This macro is defined in definitions 224, 225, and 226.
This macro is invoked in definition 212.

The nonterminal `xSubroutineNameUse` represents the context of a symbolic name immediately following the word CALL in a CALL statement. A symbol appearing in this context is only classed as a subroutine name if it does not also appear in an argument list. The check for appearance in an argument list requires a new `KindSet` element, `DummyArgument`:

Subroutine[226] ≡

```
ATTR junk: VOID;

SYMBOL xSubroutineNameUse COMPUTE
  SYNT.junk = THIS.Order;
  INH.GotContext=
    IF(Classified as[12](‘DummyArgument’,‘THIS’),
      InsertKindSet(THIS.UnitKey,DummyProcedure),
      InsertKindSet(THIS.UnitKey,Subroutine))
    <- THIS.junk;
  IF(Classified as[12](‘Subroutine’,‘THIS’),
    Check Ambiguity[213](‘CommonBlock’,‘THIS’));
END;

Sequence the elements[3](‘xArg’)

RULE: xArgList ::=
COMPUTE
  xArgList.SeqCount=0;
END;
```

This macro is defined in definitions 224, 225, and 226.
This macro is invoked in definition 212.

Section 15.7.4 of the standard guarantees that any dummy argument in an `xSubroutineNameUse` context will have appeared in an argument list that precedes the `xSubroutineNameUse`. Thus dependence on `Order` is sufficient to ensure that the `KindSet` property is properly set.

A dummy procedure name may legitimately be the same as a common block name, so the ambiguity check can only be carried out for symbols known to be subroutine names.

14.2.4 Main Program

The nonterminal `xProgramName` represents the context of a symbolic name immediately following the word PROGRAM in a PROGRAM statement.

Main Program[227] ≡

```
SYMBOL xProgramName COMPUTE
  INH.GotContext=InsertKindSet(THIS.UnitKey,MainProgram);
END;
```

This macro is defined in definitions 227 and 228.
This macro is invoked in definition 212.

A main program cannot legally contain entry statements. The grammar does not prevent this, however, so an appropriate `Kind` attribute value must be set:

Main Program[228] ≡

```
RULE: xProgramUnit ::= xProgramStmt xMainRange
COMPUTE
  xProgramUnit.Kind=MainProgram;
END;

RULE: xProgramUnit ::= xMainRange
COMPUTE
  xProgramUnit.Kind=MainProgram;
END;
```

This macro is defined in definitions 227 and 228.
This macro is invoked in definition 212.

14.2.5 Block Data Subprogram

The nonterminal `xBlockDataName` represents the context of a symbolic name immediately following the word BLOCKDATA in a block data statement.

Block Data Subprogram[229] ≡

```
SYMBOL xBlockDataName COMPUTE
```

```
  INH.GotContext=InsertKindSet(THIS.UnitKey,BlockDataSubprogram);
END;
```

This macro is defined in definitions 229 and 230.
This macro is invoked in definition 212.

A block data subprogram cannot legally contain entry statements. The grammar does not prevent this, however, so an appropriate `Kind` attribute value must be set:

Block Data Subprogram[230] ≡

```
RULE: xProgramUnit ::= xBlockDataStmt xBody xEndBlockDataStmt
COMPUTE
  xProgramUnit.Kind=BlockDataSubprogram;
END;

RULE: xProgramUnit ::= xBlockDataStmt xEndBlockDataStmt
COMPUTE
  xProgramUnit.Kind=BlockDataSubprogram;
END;
```

This macro is defined in definitions 229 and 230.
This macro is invoked in definition 212.

14.2.6 Array

The nonterminal `xArrayDeclarator` represents the context of an array declarator in a DIMENSION or COMMON statement.

Array[231] ≡

```
RULE: xArrayDeclarator ::= xVariableName '(' DimensionDeclarators ')'
COMPUTE
  xVariableName.GotContext=InsertKindSet(xVariableName.UnitKey,Array);
  Verify this does not identify a global entity[211]('xVariableName','; ');
  Check Ambiguity[213]('ExternalFunction','xVariableName');
END;
```

This macro is defined in definitions 231 and 232.
This macro is invoked in definition 212.

The nonterminal `xEntityDecl` represents the context of a declaration within a type statement. If the `xObjectName` is followed by a left parenthesis in this context, the `xEntityDecl` is an array declarator.

Array[232] ≡

```
RULE: xEntityDecl ::= xObjectName '(' DimensionDeclarators ')'
COMPUTE
  xObjectName.GotContext=InsertKindSet(xObjectName.UnitKey,Array);
  Verify this does not identify a global entity[211]('xObjectName','; ');
  Check Ambiguity[213]('ExternalFunction','xObjectName');
END;

RULE: xEntityDecl ::= xObjectName '(' DimensionDeclarators ')' '*' xCharLength
```

```
COMPUTE
  xObjectName.GotContext=InsertKindSet(xObjectName.UnitKey,Array);
  Verify this does not identify a global entity[211](‘xObjectName’,‘;’);
  Check Ambiguity[213](‘ExternalFunction’,‘xObjectName’);
END;
```

This macro is defined in definitions 231 and 232.
This macro is invoked in definition 212.

14.2.7 Variable

Conditions 1, 2 and 4 can be tested by checking the set of context conditions defined by the `KindSet` property of the symbolic name. Condition 3 implies that the check should only be made in certain contexts:

Variable[233] ≡

```
SYMBOL VariableAppearance COMPUTE
  SYNT.Classified=
    IF(
      DisjIS(
        ConsIS(Constant,
          ConsIS(IntrinsicFunction,
          ConsIS(InExternalStmt,
          ConsIS(Array,
          ConsIS(Subroutine,
          ConsIS(MainProgram,
          ConsIS(BlockDataSubprogram,
          ConsIS(ExternalFunction,
          ConsIS(StatementFunction,
          NullIS()))))))))),
        GetKindSet(THIS.UnitKey,NullIS())),
      InsertKindSet(THIS.UnitKey,Variable))
    <- INCLUDING xProgramUnit.GotAllContexts;
END;

SYMBOL xDummyArgName      INHERITS VariableAppearance END;
SYMBOL xImpliedDoVariable INHERITS VariableAppearance END;
SYMBOL xName              INHERITS VariableAppearance END;
SYMBOL xObjectName        INHERITS VariableAppearance END;
SYMBOL xSFDummyArgName    INHERITS VariableAppearance END;
SYMBOL xSFVarName         INHERITS VariableAppearance END;
SYMBOL xVariableName      INHERITS VariableAppearance END;
```

This macro is defined in definitions 233 and 234.
This macro is invoked in definition 212.

According to Section 15.4.1 of the standard, a symbolic name in the dummy argument list of a statement function is a variable name. Thus each symbolic name appearing in that context must be classified as a variable and checked for ambiguous classification.

The nonterminal `xSFDummyArgName` represents the context of a symbolic name in the dummy argument list of a statement function. Because statement functions cannot be distinguished syntactically from array assignments, it is possible that the context represented by the nonterminal `xSFDummyArgName` is actually that of a symbolic name as a subscript of an array appearing on the left-hand side of an assignment. These two contexts must therefore be distinguished:

Variable[234] ≡

```
SYMBOL xSFDummyArgName COMPUTE
  INH.GotContext=
    IF(INCLUDING (xStmtFunctionRange.InStmtFunc,xSourceFile.False),
      InsertKindSet(THIS.UnitKey,Variable));
  IF(INCLUDING (xStmtFunctionRange.InStmtFunc,xSourceFile.False),
    ORDER(
      Verify this does not identify a global entity[211](‘THIS’,‘,’),
      Check Ambiguity[213](‘IntrinsicFunction’,‘THIS’),
      Check Ambiguity[213](‘Array’,‘THIS’),
      Check Ambiguity[213](‘Subroutine’,‘THIS’),
      Check Ambiguity[213](‘MainProgram’,‘THIS’),
      Check Ambiguity[213](‘BlockDataSubprogram’,‘THIS’),
      Check Ambiguity[213](‘StatementFunction’,‘THIS’),
      Check Ambiguity[213](‘DummyProcedure’,‘THIS’)));
END;

ATTR False: int; SYMBOL xSourceFile COMPUTE SYNT.False=0; END;
```

This macro is defined in definitions 233 and 234.
This macro is invoked in definition 212.

All of the ambiguity checks are deduced from conditions 1-4 from Section 18.2.7 of the standard.

14.2.8 Constant

The nonterminal `xNamedConstant` represents the context of a symbolic name in a PARAMETER statement.

Constant[235] ≡

```
SYMBOL xNamedConstant COMPUTE
  INH.GotContext=InsertKindSet(THIS.UnitKey,Constant);
  Verify this does not identify a global entity[211](‘THIS’,‘;’);
  Check Ambiguity[213](‘CommonBlock’,‘THIS’);
  Check Ambiguity[213](‘ExternalFunction’,‘THIS’);
  Check Ambiguity[213](‘Array’,‘THIS’);
  Check Ambiguity[213](‘Variable’,‘THIS’);
  Check Ambiguity[213](‘StatementFunction’,‘THIS’);
  Check Ambiguity[213](‘IntrinsicFunction’,‘THIS’);
  Check Ambiguity[213](‘DummyProcedure’,‘THIS’);
END;
```

This macro is invoked in definition 212.

14.2.9 Statement Function

A symbol is an array name only if it appears as the array name in an array declarator (Section 5.1 of the standard) in a DIMENSION, COMMON or type statement (see Section 18.2.6 of the standard). Since these kinds of statement must precede a statement function statement, `Order` guarantees that the `KindSet` property has been set at the array declarator (if any) before the statement function statement's computation is attempted:

Statement Function[236] ≡

```
SYMBOL xStmtFunctionRange: InStmtFunc: int;

RULE: xStmt ::= xLblDef xName xStmtFunctionRange
COMPUTE
  xStmtFunctionRange.InStmtFunc=
    NOT(Classified as[12](‘Array’,‘xName’))
    <- xStmt.Order;
END;

Sequence the elements[3](‘xSFDummyArgName’)
```

This macro is defined in definitions 236 and 238.
This macro is invoked in definition 212.

`xStmtFunctionRange.InStmtFunc` is true if the statement is a statement function statement, and can be used in any test of this property:

This is a statement function statement[237] ≡
`xStmtFunctionRange.InStmtFunc`This macro is invoked in definitions 39, 196, and 238.

A statement function name may be the same as a common block name, but it cannot be the same as the name of any other global or local entity. It also cannot appear in an EXTERNAL statement:

Statement Function[238] ≡

```
RULE: xStmt ::= xLblDef xName xStmtFunctionRange
COMPUTE
  xName.GotContext=
    IF(This is a statement function statement[237],
      InsertKindSet(xName.UnitKey,StatementFunction));

  Verify this does not identify a global entity[211](‘xName’,‘;’);
  Check Ambiguity[213](‘ExternalFunction’,‘xName’);
  Check Ambiguity[213](‘Variable’,‘xName’);
  Check Ambiguity[213](‘Constant’,‘xName’);
  Check Ambiguity[213](‘IntrinsicFunction’,‘xName’);
  Check Ambiguity[213](‘DummyProcedure’,‘xName’);

  IF(
    AND(
      This is a statement function statement[237],
      Classified as[12](‘InExternalStmt’,‘xName’)),
    message(ERROR,"Also appears in an EXTERNAL statement",0,xName.Coord))
  <- INCLUDING xProgramUnit.ClassificationDone;
END;
```

This macro is defined in definitions 236 and 238.
This macro is invoked in definition 212.

14.2.10 Intrinsic Function

The nonterminal `xIntrinsicProcedureName` represents the context of a symbolic name in an INTRINSIC statement. According to Section 8.8 of the standard, appearance of a name in this context declares that name to be an intrinsic function name. Thus the specification classifies an `xIntrinsicProcedureName` as an intrinsic function, and issues an error report if the name does not appear in Table 5 from Section 15.10 of the standard or has appeared in an EXTERNAL statement:

Intrinsic Function[239] ≡

```
SYMBOL xIntrinsicProcedureName COMPUTE
  INH.GotContext=InsertKindSet(THIS.UnitKey,IntrinsicFunction);
  IF(NOT(GetIntrinsic(THIS.ObjectKey,0)),
    IF(Classified as[12](‘InExternalStmt’,‘THIS’),
      message(ERROR,"Overridden by EXTERNAL statement",0,COORDREF),
      message(ERROR,"Invalid intrinsic function name",0,COORDREF)))
    <- INCLUDING AnyScope.GotKeys;
  Verify this does not identify a global entity[211](‘THIS’,‘;’);
  Check Ambiguity[213](‘CommonBlock’,‘THIS’);
  Check Ambiguity[213](‘Array’,‘THIS’);
  Check Ambiguity[213](‘StatementFunction’,‘THIS’);
  Check Ambiguity[213](‘DummyArgument’,‘THIS’);
  Check Ambiguity[213](‘Constant’,‘THIS’);
END;
```

This macro is invoked in definition 212.

The `Intrinsic` property is pre-defined as `1` for all intrinsic function names:

Pre-definition of the Intrinsic property[240] ≡

```
#define PreDef(sym,key,type) key -> Intrinsic={1};
Table of Intrinsic Functions[192]
#undef PreDef
```

This macro is invoked in definition 243.

14.2.11 Dummy Procedure

In order to be classified as a dummy procedure, a symbol must appear in some dummy argument list *and* satisfy one of three additional conditions, the first of which is that it appears in an EXTERNAL statement. The nonterminal `xDummyArgName` represents the context of a symbolic name in a dummy argument list, and the nonterminal `xExternalName` represents the context of a symbolic name in an EXTERNAL statement:

Dummy Procedure[241] ≡

```
SYMBOL xDummyArgName COMPUTE
  SYNT.junk = THIS.Order;
  INH.GotContext=
    IF(Classified as[12](‘InExternalStmt’,‘THIS’),
      InsertKindSet(THIS.UnitKey,DummyProcedure),
      InsertKindSet(THIS.UnitKey,DummyArgument))
    <- THIS.junk;
  Verify this does not identify a global entity[211](‘THIS’,‘;’);
  Check Ambiguity[213](‘ExternalFunction’,‘THIS’);
  Check Ambiguity[213](‘Constant’,‘THIS’);
END;

SYMBOL xExternalName COMPUTE
  SYNT.junk = THIS.Order;
  INH.GotContext=
    IF(Classified as[12](‘DummyArgument’,‘THIS’),
      InsertKindSet(THIS.UnitKey,DummyProcedure),
      InsertKindSet(THIS.UnitKey,InExternalStmt))
    <- THIS.junk;
END;
```

This macro is defined in definitions 241.
This macro is invoked in definition 212.

These two computations guarantee the correct results regardless of the sequence in which the two appearances occur.

The second additional condition is the appearance of the symbol in a CALL statement. A dummy argument appearing in a CALL statement was classified above (see the Subroutine section).

The third additional condition is almost the same as the condition used to distinguish variables from external procedures. It is also checked above (see the external procedure section).

15 Specification Files for Semantic Analysis

This section briefly describes the specification files that implement the symbol classification process.

15.1 f77semantics.specs

A type-`specs` file provides names of additional objects to be included in the set of specifications. Here the objects are the library instantiations needed to support the classification process.

f77semantics.specs[242] ≡

Instantiate the MakeName library module[30]
Instantiate the predefined identifier module[198]
Instantiate the ALGOL 60 scope module[28]
Instantiate the integer set property module[11]
Instantiate a module implementing a linear list of CoordPtr[141]
Instantiate a list of bound specifications[71]
Instantiate the equivalence stack module[113]
Instantiate modules for DO sequence testing[170]
`$/Tech/strmath.specs`
`$/Tech/Strings.specs`

This macro is attached to a product file.

15.2 f77semantics.pdl

A type-`pdl` file describes the properties that must be available to describe entities.

f77semantics.pdl[243] ≡

`"f77semantics.h"`
Properties accessible via the ObjectKey attribute[7]
Properties accessible via the UnitKey attribute[8]
Storage[21]
Data Types[13]
Pre-definition of the Intrinsic property[240]
Pre-definition of intrinsic function types[193]
Properties for type analysis[43]
Properties supporting multiple definition reporting[35]
Indication/Evaluation function correspondence[97]
Bound list property[70]
Properties for equivalence analysis[109]
Properties for Common Block Storage Sequence[117]
Property for DO sequence testing[169]

This macro is attached to a product file.

15.3 f77semantics.lido

A type-`lido` file describes the relationships that must exist among computations in the tree.

f77semantics.lido[244] ≡

FORTRAN Terms and Concepts[1]
Sequence[2]
Characters, Lines and Execution Sequence[26]
Data Types and Constants[41]
Arrays and Substrings[48]
Expressions[77]
Executable and Nonexecutable Statement Classification[101]
Specification Statements[106]
DATA Statement[151]
Assignment Statements[152]
Control Statements[157]
Format Specification[173]
Function Subprogram and FUNCTION Statement[189]
ENTRY Statement[190]
RETURN Statement[191]
Scope and Classes of Symbolic Names[194]
Scope of Symbolic Names[195]
Verify that a name identifies no more than one global entity[205]
Verify that a global name does not identify a local entity[210]
Obtain the coordinates of a symbolic name[214]
Classes of Symbolic Names[212]

```
SYMBOL SymbolicName COMPUTE
  INH.GotContext=1;
END;
```

This macro is attached to a product file.

15.4 f77semantics.oil

A type-`oil` file defines the compiler's type system.

f77semantics.oil[245] ≡

Complete the type lattice[94]
Table 2, including replications, and Table 3[82]
Monadic arithmetic operators[81]
Concatenation[85]
Relational operators[87]
Logical operators[90]
Arithmetic Assignment Statement[154]
Logical Assignment Statement[155]
Character Assignment Statement[156]

This macro is attached to a product file.

15.5 f77semantics.head

A type-`head` file provides the interfaces for abstract data types used in the tree computations.

f77semantics.head[246] ≡

```
#define LaterOf(a,b) ((a)<(b)?(b):(a))
#include "f77semantics.h"
#include "csm.h"
#include "strmath.h"
#include "DoSeqStack.h"
```
Return the default type associated with the first letter of a name[44]

This macro is attached to a product file.

15.6 f77semantics.c

A type-`c` file provides the implementation of the abstract data types defined by this specification.

f77semantics.c[247] ≡

```
#include "err.h"
#include "csm.h"
#include "pdl_gen.h"
#include "EquivStack.h"
#include "f77semantics.h"
```

```
static char rcsid[] =
  "$Id: F77Semantics.fw,v 1.10 2001/03/09 23:44:29 waite Exp $";
```
Array used for reporting PARAMETER/IMPLICIT sequence errors[140]
Relate initial letters to types[132]
Obtain the type of a named constant[143]
Process individual letters of an implicit specification[131]
State initialization[134]
Evaluation functions[24]
Bound information[73]
Verify legal DO nesting[172]

This macro is attached to a product file.

15.7 f77semantics.h

A type-`h` file provides the interface for the abstract data types defined by this specification.

f77semantics.h[248] ≡

```
#ifndef F77SEMANTIC_H
#define F77SEMANTIC_H

#include "eliproto.h"
#include "deftbl.h"
#include "err.h"
#include "obstack.h"
```

KindSet elements[9]

```
extern DefTableKey TypeFor[];
extern int LengthOf[];

extern DefTableKey ParameterType ELI_ARG((int Sym, CoordPtr coord));
extern void ImplicitType
  ELI_ARG((DefTableKey Type, int Initial, CoordPtr coord));
extern void DefaultImplicitTypes ELI_ARG((void));
```

Evaluation function interfaces[25]
Bound information access[72]
DO nesting checker interface[171]

```
#endif
```

This macro is attached to a product file.

15.8 f77semantics.init

A type-`init` file provides code to be executed before the processor begins reading input text.

f77semantics.init[249] ≡

Initialization[100]

This macro is attached to a product file.

15.9 f77semantics.d

A type-`d` file provides data for the predefined symbol module.

f77semantics.d[250] ≡

Place pre-defined symbols into the outermost environment[197]

This macro is attached to a non-product file.