Clause 7

Expressions

The rules applicable to the different forms of expression, and to their evaluation, are given in this ~~section~~ clause[1].

7.1 Expressions

An expression is a formula that defines the computation of a value.

```
expression ::=
        relation { and relation }
      | relation { or relation }
      | relation { xor relation }
      | relation [ nand relation ]
      | relation [ nor relation ]
      | relation { xnor relation }

relation ::=
      shift_expression [ relational_operator shift_expression ]

shift_expression ::=
      simple_expression [ shift_operator simple_expression ]

simple_expression ::=
      [ sign ] term { adding_operator term }

term ::=
      factor { multiplying_operator factor }

factor ::=
        primary [ ** primary ]
      | abs primary
      | not primary

primary ::=
        name
      | literal
      | aggregate
      | function_call
      | qualified_expression
      | type_conversion
      | allocator
      | ( expression )
```

1. To conform to IEEE rules.

Each primary has a value and a type. The only names allowed as primaries are attributes that yield values and names denoting objects or values. In the case of names denoting objects other than objects of file types or protected types[2], the value of the primary is the value of the object. In the case of names denoting either file objects or objects of protected types, the value of the primary is the entity denoted by the name.[3]

The type of an expression depends only upon the types of its operands and on the operators applied; for an overloaded operand or operator, the determination of the operand type, or the identification of the overloaded operator, depends on the context (see 10.5). For each predefined operator, the operand and result types are given in the following clause.

NOTE

—The syntax for an expression involving logical operators allows a sequence of **and**, **or**, **xor**, or **xnor** operators (whether predefined or user-defined), since the corresponding predefined operations are associative. For the operators **nand** and **nor** (whether predefined or user-defined), however, such a sequence is not allowed, since the corresponding predefined operations are not associative.

7.2 Operators

The operators that may be used in expressions are defined below. Each operator belongs to a class of operators, all of which have the same precedence level; the classes of operators are listed in order of increasing precedence.

logical_operator	::=	**and**	|	**or**	|	**nand**	|	**nor**	|	**xor**	|	**xnor**
relational_operator	::=	=	|	/=	|	<	|	<=	|	>	|	>=
shift_operator	::=	**sll**	|	**srl**	|	**sla**	|	**sra**	|	**rol**	|	**ror**
adding_operator	::=	+	|	–	|	&						
sign	::=	+	|	–								
multiplying_operator	::=	*	|	/	|	**mod**	|	**rem**				
miscellaneous_operator	::=	**	|	**abs**	|	**not**						

Operators of higher precedence are associated with their operands before operators of lower precedence. Where the language allows a sequence of operators, operators with the same precedence level are associated with their operands in textual order, from left to right. The precedence of an operator is fixed and ~~may not~~ cannot[4] be changed by the user, but parentheses can be used to control the association of operators and operands.

In general, operands in an expression are evaluated before being associated with operators. For certain operations, however, the right-hand operand is evaluated if and only if the left-hand operand has a certain value. These operations are called *short-circuit* operations. The logical operations **and**, **or**, **nand**, and **nor** defined for operands of types BIT and BOOLEAN are all short-circuit operations; furthermore, these are the only short-circuit operations.

Every predefined operator is a pure function (see 2.1). No predefined operators have named formal parameters; therefore, named association (see 4.3.2.2) ~~may not~~ cannot[5] be used when invoking a predefined operation.

2. Additional P1076a cleanup; noted by Peter Ashenden.
3. Additional P1076a cleanup; noted by Peter Ashenden.
4. IR1000.4.7.
5. IR1000.4.7.

NOTES

1—The predefined operators for the standard types are declared in package STANDARD as shown in 14.2.

2—The operator **not** is classified as a miscellaneous operator for the purposes of defining precedence, but is otherwise classified as a logical operator.

7.2.1 Logical operators

The logical operators **and**, **or**, **nand**, **nor**, **xor**, **xnor**, and **not** are defined for predefined types BIT and BOOLEAN. They are also defined for any one-dimensional array type whose element type is BIT or BOOLEAN. For the binary operators **and**, **or**, **nand**, **nor**, **xor**, and **xnor**, the operands must be of the same base type. Moreover, for the binary operators **and**, **or**, **nand**, **nor**, **xor**, and **xnor** defined on one-dimensional array types, the operands must be arrays of the same length, the operation is performed on matching elements of the arrays, and the result is an array with the same index range as the left operand. For the unary operator **not** defined on one-dimensional array types, the operation is performed on each element of the operand, and the result is an array with the same index range as the operand.

The effects of the logical operators are defined in the following tables. The symbol T represents TRUE for type BOOLEAN, '1' for type BIT; the symbol F represents FALSE for type BOOLEAN, '0' for type BIT.

A	B	A **and** B
T	T	T
T	F	F
F	T	F
F	F	F

A	B	A **or** B
T	T	T
T	F	T
F	T	T
F	F	F

A	B	A **xor** B
T	T	F
T	F	T
F	T	T
F	F	F

A	**not** A
T	F
F	T

A	B	A **nand** B
T	T	F
T	F	T
F	T	T
F	F	T

A	B	A **nor** B
T	T	F
T	F	F
F	T	F
F	F	T

A	B	A **xnor** B
T	T	T
T	F	F
F	T	F
F	F	T

For the short-circuit operations **and**, **or**, **nand**, and **nor** on types BIT and BOOLEAN, the right operand is evaluated only if the value of the left operand is not sufficient to determine the result of the operation. For operations **and** and **nand**, the right operand is evaluated only if the value of the left operand is T; for operations **or** and **nor**, the right operand is evaluated only if the value of the left operand is F.

NOTE

—All of the binary logical operators belong to the class of operators with the lowest precedence. The unary logical operator **not** belongs to the class of operators with the highest precedence.

7.2.2 Relational operators

Relational operators include tests for equality, inequality, and ordering of operands. The operands of each relational operator must be of the same type. The result type of each relational operator is the predefined type BOOLEAN.

Operator	Operation	Operand type	Result type
=	Equality	Any type, other than a file type or a protected type	BOOLEAN
/=	Inequality	Any type, other than a file type or a protected type	BOOLEAN

< <= > >=	Ordering	Any scalar type or discrete array type	BOOLEAN

The equality and inequality operators (= and /=) are defined for all types other than file types and protected types. The equality operator returns the value TRUE if the two operands are equal and returns the value FALSE otherwise. The inequality operator returns the value FALSE if the two operands are equal and returns the value TRUE otherwise.

Two scalar values of the same type are equal if and only if the values are the same. Two composite values of the same type are equal if and only if for each element of the left operand there is a *matching element* of the right operand and vice versa, and the values of matching elements are equal, as given by the predefined equality operator for the element type. In particular, two null arrays of the same type are always equal. Two values of an access type are equal if and only if they both designate the same object or they both are equal to the null value for the access type.

For two record values, matching elements are those that have the same element identifier. For two one-dimensional array values, matching elements are those (if any) whose index values match in the following sense: the left bounds of the index ranges are defined to match; if two elements match, the elements immediately to their right are also defined to match. For two multi-dimensional array values, matching elements are those whose indices match in successive positions.

The ordering operators are defined for any scalar type and for any discrete array type. A *discrete array* is a one-dimensional array whose elements are of a discrete type. Each operator returns TRUE if the corresponding relation is satisfied; otherwise, the operator returns FALSE.

For scalar types, ordering is defined in terms of the relative values. For discrete array types, the relation < (less than) is defined such that the left operand is less than the right operand if and only if

— The left operand is a null array and the right operand is a nonnull array; otherwise,

— Both operands are nonnull arrays, and one of the following conditions is satisfied:

—The leftmost element of the left operand is less than that of the right; or

—The leftmost element of the left operand is equal to that of the right, and the tail of the left operand is less than that of the right (the tail consists of the remaining elements to the right of the leftmost element and can be null).

The relation <= (less than or equal) for discrete array types is defined to be the inclusive disjunction of the results of the < and = operators for the same two operands. The relations > (greater than) and >= (greater than or equal) are defined to be the complements of the <= and < operators respectively for the same two operands.

7.2.3 Shift operators

The shift operators **sll**, **srl**, **sla**, **sra**, **rol**, and **ror** are defined for any one-dimensional array type whose element type is either of the predefined types BIT or BOOLEAN.

Operator	Operation	Left operand type	Right operand type	Result type
sll	Shift left logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left

srl	Shift right logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
sla	Shift left arithmetic	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
sra	Shift right arithmetic	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
rol	Rotate left logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
ror	Rotate right logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left

The index subtypes of the return values of all shift operators are the same as the index subtypes of their left arguments.

The values returned by the shift operators are defined as follows. In the remainder of this ~~section~~ clause[6], the values of their leftmost arguments are referred to as L and the values of their rightmost arguments are referred to as R.

— The **sll** operator returns a value that is L logically shifted left by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic shift operation replaces L with a value that is the result of a concatenation whose left argument is the rightmost (L'Length – 1) elements of L and whose right argument is T'Left, where T is the element type of L. If R is positive, this basic shift operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **srl** –R.

— The **srl** operator returns a value that is L logically shifted right by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic shift operation replaces L with a value that is the result of a concatenation whose right argument is the leftmost (L'Length – 1) elements of L and whose left argument is T'Left, where T is the element type of L. If R is positive, this basic shift operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **sll** –R.

— The **sla** operator returns a value that is L arithmetically shifted left by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic shift operation replaces L with a value that is the result of a concatenation whose left argument is the rightmost (L'Length – 1) elements of L and whose right argument is L(L'Right). If R is positive, this basic shift operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **sra** –R.

— The **sra** operator returns a value that is L arithmetically shifted right by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic shift operation replaces L with a value that is the result of a concatenation whose right argument is the leftmost (L'Length – 1) elements of L and whose left argument is L(L'Left). If R is positive, this basic shift operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **sla** –R.

6. To conform to IEEE rules.

— The **rol** operator returns a value that is L rotated left by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic rotate operation replaces L with a value that is the result of a concatenation whose left argument is the rightmost (L'Length – 1) elements of L and whose right argument is L(L'Left). If R is positive, this basic rotate operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **ror** –R.

— The **ror** operator returns a value that is L rotated right by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic rotate operation replaces L with a value that is the result of a concatenation whose right argument is the leftmost (L'Length – 1) elements of L and whose left argument is L(L'Right). If R is positive, this basic rotate operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **rol** –R.

NOTES

1—The logical operators may be overloaded, for example, to disallow negative integers as the second argument.

2—The subtype of the result of a shift operator is the same as that of the left operand.

7.2.4 Adding operators

The adding operators + and – are predefined for any numeric type and have their conventional mathematical meaning. The concatenation operator & is predefined for any one-dimensional array type.

Operator	Operation	Left operand type	Right operand type	Result type
+	Addition	Any numeric type	Same type	Same type
–	Subtraction	Any numeric type	Same type	Same type
&	Concatenation	Any one-dimensional[a] array type	Same array type	Same array type
		Any one-dimensional[b] array type	The element type	Same array type
		The element type	Any one-dimensional[c] array type	Same array type
		The element type	The element type	Any one-dimensional[d] array type

a. Clarification
b. Clarification
c. Clarification
d. Clarification

For concatenation, there are three mutually exclusive cases:

a) If both operands are one-dimensional arrays of the same type, the result of the concatenation is a one-dimensional array of this same type whose length is the sum of the lengths of its operands, and whose elements consist of the elements of the left operand (in left-to-right order) followed by the elements of the right operand (in left-to-right order). ~~The direction of the result is the direction of the left operand, unless the left operand is a null array, in which case the direction of the result is that of the right operand.~~[7]

7. LCS 4.

If both operands are null arrays, then the result of the concatenation is the right operand. Otherwise, the direction and bounds of the result are determined as follows: Let S be the index subtype of the base type of the result. The direction of the result of the concatenation is the direction of S, and the left bound of the result is S'LEFT.

b) If one of the operands is a one-dimensional array and the type of the other operand is the element type of this aforementioned one-dimensional array, the result of the concatenation is given by the rules in case a, using in place of the other operand an implicit array having this operand as its only element. Both the left and right bounds of the index subtype of this implicit array is S'LEFT, and the direction of the index subtype of this implicit array is the direction of S, where S is the index subtype of the base type of the result.[8]

c) If both operands are of the same type and it is the element type of some one-dimensional array type, the type of the result must be known from the context and is this one-dimensional array type. In this case, each operand is treated as the one element of an implicit array, and the result of the concatenation is determined as in case a. The bounds and direction of the index subtypes of the implicit arrays are determined as in the case of the implicit array in case b).[9]

In all cases, it is an error if either bound of the index subtype of the result does not belong to the index subtype of the type of the result, unless the result is a null array. It is also an error if any element of the result does not belong to the element subtype of the type of the result.

Examples:

```
subtype BYTE is BIT_VECTOR (7 downto 0);
type MEMORY is array (Natural range <>) of BYTE;

-- The following concatenation accepts two BIT_VECTORs and returns a BIT_VECTOR
-- (case a):

constant ZERO: BYTE := "0000" & "0000";

-- The next two examples show that the same expression can represent either case a or
-- case c, depending on the context of the expression.

-- The following concatenation accepts two BIT_VECTORS and returns a BIT_VECTOR
-- (case a):

constant C1: BIT_VECTOR := ZERO & ZERO;

-- The following concatenation accepts two BIT_VECTORs and returns a MEMORY
-- (case c):

constant C2: MEMORY := ZERO & ZERO;

-- The following concatenation accepts a BIT_VECTOR and a MEMORY, returning a
-- MEMORY (case b):

constant C3: MEMORY := ZERO & C2;

-- The following concatenation accepts a MEMORY and a BIT_VECTOR, returning a
-- MEMORY (case b):

constant C4: MEMORY := C2 & ZERO;
```

8. LCS 4.
9. LCS 4.

```
-- The following concatenation accepts two MEMORYs and returns a MEMORY (case a):

constant C5: MEMORY := C2 & C3;

type R1 is range[10] 0 to 7;
type R2 is range[11] 7 downto 0;

type T1 is array (R1 range <>) of Bit;
type T2 is array (R2 range <>) of Bit;

subtype S1 is T1(R1);
subtype S2 is T2(R2);

constant K1: S1 := (others => '0');
constant K2: T1 := K1(1 to 3) & K1(3 to 4);      -- K2'Left = 0 and K2'Right = 4
constant K3: T1 := K1(5 to 7) & K1(1 to 2);      -- K3'Left = 0 and K3'Right = 4
constant K4: T1 := K1(2 to 1) & K1(1 to 2);      -- K4'Left = 0 and K4'Right = 1

constant K5: S2 := (others => '0');
constant K6: T2 := K5(3 downto 1) & K5(4 downto 3);   -- K6'Left = 7 and K6'Right = 3
constant K7: T2 := K5(7 downto 5) & K5(2 downto 1);   -- K7'Left = 7 and K7'Right = 3
constant K8: T2 := K5(1 downto 2) & K5(2 downto 1);   -- K8'Left = 7 and K8'Right = 6
```

NOTES

1—For a given concatenation whose operands are of the same type, there may be visible more than one array type that could be the result type according to the rules of case c. The concatenation is ambiguous and therefore an error if, using the overload resolution rules of 2.3 and 10.5, the type of the result is not uniquely determined.

2—Additionally, for a given concatenation, there may be visible array types that allow both case a and case c to apply. The concatenation is again ambiguous and therefore an error if the overload resolution rules cannot be used to determine a result type uniquely.

7.2.5 Sign operators

Signs + and – are predefined for any numeric type and have their conventional mathematical meaning: they respectively represent the identity and negation functions. For each of these unary operators, the operand and the result have the same type.

Operator	Operation	Operand type	Result type
+	Identity	Any numeric type	Same type
–	Negation	Any numeric type	Same type

NOTE

—Because of the relative precedence of signs + and – in the grammar for expressions, a signed operand must not follow a multiplying operator, the exponentiating operator **, or the operators **abs** and **not**. For example, the syntax does not allow the following expressions:

```
A/+B      -- An illegal expression
A**–B     -- An illegal expression
```

However, these expressions may be rewritten legally as follows:

10. IR1000.1.7.
11. IR1000.1.7.

```
A/(+B)          -- A legal expression
A**(–B)         -- A legal expression
```

7.2.6 Multiplying operators

The operators * and / are predefined for any integer and any floating point type and have their conventional mathematical meaning; the operators **mod** and **rem** are predefined for any integer type. For each of these operators, the operands and the result are of the same type.

Operator	Operation	Left operand type	Right operand type	Result type
*	Multiplication	Any integer type	Same type	Same type
		Any floating-point type	Same type	Same type
/	Division	Any integer type	Same type	Same type
		Any floating-point type	Same type	Same type
mod	Modulus	Any integer type	Same type	Same type
rem	Remainder	Any integer type	Same type	Same type

Integer division and remainder are defined by the following relation:

A = (A/B)*B + (A **rem** B)

where (A **rem** B) has the sign of A and an absolute value less than the absolute value of B. Integer division satisfies the following identity:

(–A)/B = –(A/B) = A/(–B)

The result of the modulus operation is such that (A **mod** B) has the sign of B and an absolute value less than the absolute value of B; in addition, for some integer value N, this result must satisfy the relation:

A = B*N + (A **mod** B)

In addition to the above table, the operators * and / are predefined for any physical type.

Operator	Operation	Left operand type	Right operand type	Result type
*	Multiplication	Any physical type	INTEGER	Same as left
		Any physical type	REAL	Same as left
		INTEGER	Any physical type	Same as right
		REAL	Any physical type	Same as right
/	Division	Any physical type	INTEGER	Same as left
		Any physical type	REAL	Same as left
		Any physical type	The same type	*Universal integer*

Multiplication of a value P of a physical type T_p by a value I of type INTEGER is equivalent to the following computation:

T_p'Val(T_p'Pos(P) * I)

Multiplication of a value P of a physical type T_p by a value F of type REAL is equivalent to the following computation:

T_p'Val(INTEGER(REAL(T_p'Pos(P)) * F))

Division of a value P of a physical type T_p by a value I of type INTEGER is equivalent to the following computation:

T_p'Val(T_p'Pos(P) / I)

Division of a value P of a physical type T_p by a value F of type REAL is equivalent to the following computation:

T_p'Val(INTEGER(REAL(T_p'Pos(P)) / F))

Division of a value P of a physical type T_p by a value P2 of the same physical type is equivalent to the following computation:

T_p'Pos(P) / T_p'Pos(P2)

Examples:

```
  5  rem   3  =  2
  5  mod   3  =  2

(–5) rem   3  = –2
(–5) mod   3  =  1

(–5) rem (–3) = –2
(–5) mod (–3) = –2

  5  rem (–3) =  2
  5  mod (–3) = –1
```

NOTE

—Because of the precedence rules (see 7.2), the expression “–5 **rem** 2” is interpreted as “–(5 **rem** 2)” and not as “(–5) **rem** 2”.

7.2.7 Miscellaneous operators

The unary operator **abs** is predefined for any numeric type.

Operator	Operation	Operand type	Result type
abs	Absolute value	Any numeric type	Same numeric type

The *exponentiating* operator ** is predefined for each integer type and for each floating point type. In either case the right operand, called the exponent, is of the predefined type INTEGER.

Operator	Operation	Left operand type	Right operand type	Result type
**	Exponentiation	Any integer type	INTEGER	Same as left
		Any floating-point type	INTEGER	Same as left

Exponentiation with an integer exponent is equivalent to repeated multiplication of the left operand by itself for a number of times indicated by the absolute value of the exponent and from left to right; if the exponent is negative, then the result is the reciprocal of that obtained with the absolute value of the exponent. Exponentiation with a negative exponent is only allowed for a left operand of a floating point type. Exponentiation by a zero exponent results in the value one. Exponentiation of a value of a floating point type is approximate.

7.3 Operands

The operands in an expression include names (that denote objects, values, or attributes that result in a value), literals, aggregates, function calls, qualified expressions, type conversions, and allocators. In addition, an expression enclosed in parentheses may be an operand in an expression. Names are defined in 6.1; the other kinds of operands are defined in the following subclauses.

7.3.1 Literals

A literal is either a numeric literal, an enumeration literal, a string literal, a bit string literal, or the literal **null**.

```
literal ::=
        numeric_literal
      | enumeration_literal
      | string_literal
      | bit_string_literal
      | null

numeric_literal ::=
        abstract_literal
      | physical_literal
```

Numeric literals include literals of the abstract types *universal_integer* and *universal_real*, as well as literals of physical types. Abstract literals are defined in 13.4; physical literals are defined in 3.1.3.

Enumeration literals are literals of enumeration types. They include both identifiers and character literals. Enumeration literals are defined in 3.1.1.

String and bit string literals are representations of one-dimensional arrays of characters. The type of a string or bit string literal must be determinable solely from the context in which the literal appears, excluding the literal itself but using the fact that the type of the literal must be a one-dimensional array of a character type. The lexical structure of string and bit string literals is defined in ~~Section~~ Clause[12] 13.

For a nonnull array value represented by either a string or bit-string literal, the direction and bounds of the array value are determined according to the rules for positional array aggregates, where the number of elements in the aggregate is equal to the length (see 13.6 and 13.7) of the string or bit string literal. For a null array value represented by either a string or bit-string literal, the direction and leftmost bound of the array value are determined as in the non-null case. If the direction is ascending, then the rightmost bound is the predecessor (as given by the 'PRED attribute) of the leftmost bound; otherwise the rightmost bound is the successor (as given by the 'SUCC attribute) of the leftmost bound.

12. To conform to IEEE rules.

The character literals corresponding to the graphic characters contained within a string literal or a bit string literal must be visible at the place of the string literal.

The literal **null** represents the null access value for any access type.

Evaluation of a literal yields the corresponding value.

Examples:

```
3.14159_26536          -- A literal of type universal_real.
5280                   -- A literal of type universal_integer.
10.7 ns                -- A literal of a physical type.
O"4777"                -- A bit-string literal.
"54LS281"              -- A string literal.
""                     -- A string literal representing a null array.
```

7.3.2 Aggregates

An aggregate is a basic operation (see the introduction to ~~Section~~ Clause[13] 3) that combines one or more values into a composite value of a record or array type.

```
aggregate ::=
    ( element_association { , element_association } )

element_association ::=
    [ choices => ] expression

choices ::=  choice { | choice }

choice ::=
        simple_expression
    | discrete_range
    | element_simple_name
    | others
```

Each element association associates an expression with elements (possibly none). An element association is said to be *named* if the elements are specified explicitly by choices; otherwise, it is said to be *positional*. For a positional association, each element is implicitly specified by position in the textual order of the elements in the corresponding type declaration.

Both named and positional associations can be used in the same aggregate, with all positional associations appearing first (in textual order) and all named associations appearing next (in any order, except that ~~no~~ it is an error if any[14] associations ~~may~~[15] follow an **others** association). Aggregates containing a single element association must always be specified using named association in order to distinguish them from parenthesized expressions.

An element association with a choice that is an element simple name is only allowed in a record aggregate. An element association with a choice that is a simple expression or a discrete range is only allowed in an array aggregate: a simple expression specifies the element at the corresponding index value, whereas a discrete range specifies the elements at each of the index values in the range. The discrete range has no significance other than to define the set of choices implied by the discrete range. In particular, the direction specified or implied by the discrete range has no significance. An element association with the choice **others** is allowed in either an array aggregate or a record aggregate if the association appears last and has this single choice; it specifies all remaining elements, if any.

13. To conform to IEEE rules.
14. IR1000.4.7.
15. IR1000.4.7.

Each element of the value defined by an aggregate must be represented once and only once in the aggregate.

The type of an aggregate must be determinable solely from the context in which the aggregate appears, excluding the aggregate itself but using the fact that the type of the aggregate must be a composite type. The type of an aggregate in turn determines the required type for each of its elements.

7.3.2.1 Record aggregates

If the type of an aggregate is a record type, the element names given as choices must denote elements of that record type. If the choice **others** is given as a choice of a record aggregate, it must represent at least one element. An element association with more than one choice, or with the choice **others**, is only allowed if the elements specified are all of the same type. The expression of an element association must have the type of the associated record elements.

A record aggregate is evaluated as follows. The expressions given in the element associations are evaluated in an order (or lack thereof) not defined by the language. The expression of a named association is evaluated once for each associated element. A check is made that the value of each element of the aggregate belongs to the subtype of this element. It is an error if this check fails.

7.3.2.2 Array aggregates

For an aggregate of a one-dimensional array type, each choice must specify values of the index type, and the expression of each element association must be of the element type. An aggregate of an n-dimensional array type, where n is greater than 1, is written as a one-dimensional aggregate in which the index subtype of the aggregate is given by the first index position of the array type, and the expression specified for each element association is an (n–1)-dimensional array or array aggregate, which is called a *subaggregate*. A string or bit string literal is allowed as a subaggregate in the place of any aggregate of a one-dimensional array of a character type.

Apart from a final element association with the single choice **others**, the rest (if any) of the element associations of an array aggregate must be either all positional or all named. A named association of an array aggregate is allowed to have a choice that is not locally static, or likewise a choice that is a null range, only if the aggregate includes a single element association and this element association has a single choice. An **others** choice is locally static if the applicable index constraint is locally static.

The subtype of an array aggregate that has an **others** choice must be determinable from the context. That is, an array aggregate with an **others** choice ~~may only appear~~ must appear only in one of the following contexts:[16]

a) As an actual associated with a formal parameter or formal generic declared to be of a constrained array subtype (or subelement thereof)

b) As the default expression defining the default initial value of a port declared to be of a constrained array subtype

c) As the result expression of a function, where the corresponding function result type is a constrained array subtype

d) As a value expression in an assignment statement, where the target is a declared object, and the subtype of the target is a constrained array subtype (or subelement of such a declared object)

e) As the expression defining the initial value of a constant or variable object, where that object is declared to be of a constrained array subtype

f) As the expression defining the default values of signals in a signal declaration, where the corresponding subtype is a constrained array subtype

16. IR1000.4.7.

g) As the expression defining the value of an attribute in an attribute specification, where that attribute is declared to be of a constrained array subtype

h) As the operand of a qualified expression whose type mark denotes a constrained array subtype

i) As a subaggregate nested within an aggregate, where that aggregate itself appears in one of these contexts

The bounds of an array that does not have an **others** choice are determined as follows. If the aggregate appears in one of the contexts in the preceding list, then the direction of the index subtype of the aggregate is that of the corresponding constrained array subtype; otherwise, the direction of the index subtype of the aggregate is that of the index subtype of the base type of the aggregate. For an aggregate that has named associations, the leftmost and rightmost bounds are determined by the direction of the index subtype of the aggregate and the smallest and largest choices given. For a positional aggregate, the leftmost bound is determined by the applicable index constraint if the aggregate appears in one of the contexts in the preceding list; otherwise, the leftmost bound is given by S'LEFT where S is the index subtype of the base type of the array. In either case, the rightmost bound is determined by the direction of the index subtype and the number of elements.

The evaluation of an array aggregate that is not a subaggregate proceeds in two steps. First, the choices of this aggregate and of its subaggregates, if any, are evaluated in some order (or lack thereof) that is not defined by the language. Second, the expressions of the element associations of the array aggregate are evaluated in some order that is not defined by the language; the expression of a named association is evaluated once for each associated element. The evaluation of a subaggregate consists of this second step (the first step is omitted since the choices have already been evaluated).

For the evaluation of an aggregate that is not a null array, a check is made that the index values defined by choices belong to the corresponding index subtypes, and also that the value of each element of the aggregate belongs to the subtype of this element. For a multidimensional aggregate of dimension n, a check is made that all (n-1)-dimensional subaggregates have the same bounds. It is an error if any one of these checks fails.

7.3.3 Function calls

A function call invokes the execution of a function body. The call specifies the name of the function to be invoked and specifies the actual parameters, if any, to be associated with the formal parameters of the function. Execution of the function body results in a value of the type declared to be the result type in the declaration of the invoked function.

function_call ::=
*function*_name [(actual_parameter_part)]

actual_parameter_part ::= *parameter*_association_list

For each formal parameter of a function, a function call must specify exactly one corresponding actual parameter. This actual parameter is specified either explicitly, by an association element (other than the actual part **open**) in the association list, or in the absence of such an association element, by a default expression (see 4.3.2).

Evaluation of a function call includes evaluation of the actual parameter expressions specified in the call and evaluation of the default expressions associated with formal parameters of the function that do not have actual parameters associated with them. In both cases, the resulting value must belong to the subtype of the associated formal parameter. (If the formal parameter is of an unconstrained array type, then the formal parameter takes on the subtype of the actual parameter.) The function body is executed using the actual parameter values and default expression values as the values of the corresponding formal parameters.

NOTE

—If a name (including one used as a prefix) has an interpretation both as a function call and an indexed name, then the innermost complete context is used to disambiguate the name. If, after applying this rule, there is not exactly one interpretation of the name, then the name is ambiguous. See 10.5.

7.3.4 Qualified expressions

A qualified expression is a basic operation (see the introduction to ~~Section~~ Clause[17] 3) that is used to explicitly state the type, and possibly the subtype, of an operand that is an expression or an aggregate.

```
qualified_expression ::=
        type_mark ' ( expression )
      | type_mark ' aggregate
```

The operand must have the same type as the base type of the type mark. The value of a qualified expression is the value of the operand. The evaluation of a qualified expression evaluates the operand and checks that its value belongs to the subtype denoted by the type mark.

NOTE

—Whenever the type of an enumeration literal or aggregate is not known from the context, a qualified expression can be used to state the type explicitly.

7.3.5 Type conversions

A type conversion provides for explicit conversion between closely related types.

```
type_conversion ::=  type_mark ( expression )
```

The target type of a type conversion is the base type of the type mark. The type of the operand of a type conversion must be determinable independent of the context (in particular, independent of the target type). Furthermore, the operand of a type conversion is not allowed to be the literal **null**, an allocator, an aggregate, or a string literal. An expression enclosed by parentheses is allowed as the operand of a type conversion only if the expression alone is allowed.

If the type mark denotes a subtype, conversion consists of conversion to the target type followed by a check that the result of the conversion belongs to the subtype.

Explicit type conversions are allowed between *closely related types*. In particular, a type is closely related to itself. Other types are closely related only under the following conditions:

a) *Abstract Numeric Types*—Any abstract numeric type is closely related to any other abstract numeric type. In an explicit type conversion where the type mark denotes an abstract numeric type, the operand can be of any integer or floating point type. The value of the operand is converted to the target type, which must also be an integer or floating point type. The conversion of a floating point value to an integer type rounds to the nearest integer; if the value is halfway between two integers, rounding may be up or down.

b) *Array Types*—Two array types are closely related if and only if

—The types have the same dimensionality;

—For each index position, the index types are either the same or are closely related; and

—The element types are the same.

In an explicit type conversion where the type mark denotes an array type, the following rules apply: if the type mark denotes an unconstrained array type and if the operand is not a null array, then, for each index position, the bounds of the result are obtained by converting the bounds of the operand to the corresponding index type of the target type. If the type mark denotes a constrained array subtype, then

17. To conform to IEEE rules.

the bounds of the result are those imposed by the type mark. In either case, the value of each element of the result is that of the matching element of the operand (see 7.2.2).

No other types are closely related.

In the case of conversions between numeric types, it is an error if the result of the conversion fails to satisfy a constraint imposed by the type mark.

In the case of conversions between array types, a check is made that any constraint on the element subtype is the same for the operand array type as for the target array type. If the type mark denotes an unconstrained array type, then, for each index position, a check is made that the bounds of the result belong to the corresponding index subtype of the target type. If the type mark denotes a constrained array subtype, a check is made that for each element of the operand there is a matching element of the target subtype, and vice versa. It is an error if any of these checks fail.

In certain cases, an implicit type conversion will be performed. An implicit conversion of an operand of type *universal_integer* to another integer type, or of an operand of type *universal_real* to another floating point type, can only be applied if the operand is either a numeric literal or an attribute, or if the operand is an expression consisting of the division of a value of a physical type by a value of the same type; such an operand is called a *convertible* universal operand. An implicit conversion of a convertible universal operand is applied if and only if the innermost complete context determines a unique (numeric) target type for the implicit conversion, and there is no legal interpretation of this context without this conversion.

NOTE

—Two array types may be closely related even if corresponding index positions have different directions.

7.3.6 Allocators

The evaluation of an allocator creates an object and yields an access value that designates the object.

allocator ::=
    new subtype_indication
  | **new** qualified_expression

The type of the object created by an allocator is the base type of the type mark given in either the subtype indication or the qualified expression. For an allocator with a subtype indication, the initial value of the created object is the same as the default initial value for an explicitly declared variable of the designated subtype. For an allocator with a qualified expression, this expression defines the initial value of the created object.

The type of the access value returned by an allocator must be determinable solely from the context, but using the fact that the value returned is of an access type having the named designated type.

The only allowed form of constraint in the subtype indication of an allocator is an index constraint. If an allocator includes a subtype indication and if the type of the object created is an array type, then the subtype indication must either denote a constrained subtype or include an explicit index constraint. A subtype indication that is part of an allocator must not include a resolution function.

If the type of the created object is an array type, then the created object is always constrained. If the allocator includes a subtype indication, the created object is constrained by the subtype. If the allocator includes a qualified expression, the created object is constrained by the bounds of the initial value defined by that expression. For other types, the subtype of the created object is the subtype defined by the subtype of the access type definition.

For the evaluation of an allocator, the elaboration of the subtype indication or the evaluation of the qualified expression is first performed. The new object is then created, and the object is then assigned its initial value. Finally, an access value that designates the created object is returned.

In the absence of explicit deallocation, an implementation must guarantee that any object created by the evaluation of an allocator remains allocated for as long as this object or one of its subelements is accessible directly or indirectly; that is, as long as it can be denoted by some name.

NOTES

1—Procedure Deallocate is implicitly declared for each access type. This procedure provides a mechanism for explicitly deallocating the storage occupied by an object created by an allocator.

2—An implementation may (but need not) deallocate the storage occupied by an object created by an allocator, once this object has become inaccessible.

Examples:

```
new NODE                                    -- Takes on default initial value.
new NODE'(15 ns, null)                      -- Initial value is specified.
new NODE'(Delay => 5  ns, \Next\ => Stack)  -- Initial value is specified.
new BIT_VECTOR'("00110110")                 -- Constrained by initial value.
new STRING (1 to 10)                        -- Constrained by index constraint.
new STRING                                  -- Illegal: must be constrained.
```

7.4 Static expressions

Certain expressions are said to be *static*. Similarly, certain discrete ranges are said to be static, and the type marks of certain subtypes are said to denote static subtypes.

There are two categories of static expression. Certain forms of expression can be evaluated during the analysis of the design unit in which they appear; such an expression is said to be *locally static*. Certain forms of expression can be evaluated as soon as the design hierarchy in which they appear is elaborated; such an expression is said to be *globally static*.

7.4.1 Locally static primaries

An expression is said to be locally static if and only if every operator in the expression denotes an implicitly defined operator whose operands and result are scalar and if every primary in the expression is a *locally static primary*, where a locally static primary is defined to be one of the following:

a) A literal of any type other than type TIME

b) A constant (other than a deferred constant) explicitly declared by a constant declaration and initialized with a locally static expression

c) An alias whose aliased name (given in the corresponding alias declaration) is a locally static primary

d) A function call whose function name denotes an implicitly defined operator, and whose actual parameters are each locally static expressions

e) A predefined attribute that is a value, other than the predefined attributes 'INSTANCE_NAME and[18] 'PATH_NAME, and whose prefix is either a locally static subtype or is an object name that is of a locally static subtype

f) A predefined attribute that is a function, other than the predefined attributes 'EVENT, 'ACTIVE, 'LAST_EVENT, 'LAST_ACTIVE, 'LAST_VALUE, 'DRIVING, and 'DRIVING_VALUE, whose prefix is either a locally static subtype or is an object that is of a locally static subtype, and whose actual parameter (if any) is a locally static expression

18. LCS 6.

g) A user-defined attribute whose value is defined by a locally static expression

h) A qualified expression whose operand is a locally static expression

i) A type conversion whose expression is a locally static expression

j) A locally static expression enclosed in parentheses

A locally static range is either a range of the second form (see 3.1) whose bounds are locally static expressions, or a range of the first form whose prefix denotes either a locally static subtype or an object that is of a locally static subtype. A locally static range constraint is a range constraint whose range is locally static. A locally static scalar subtype is either a scalar base type or a scalar subtype formed by imposing on a locally static subtype a locally static range constraint. A locally static discrete range is either a locally static subtype or a locally static range.

A locally static index constraint is an index constraint for which each index subtype of the corresponding array type is locally static and in which each discrete range is locally static. A locally static array subtype is a constrained array subtype formed by imposing on an unconstrained array type a locally static index constraint. A locally static record subtype is a record type whose fields are all of locally static subtypes. A locally static access subtype is a subtype denoting an access type. A locally static file subtype is a subtype denoting a file type.

A locally static subtype is either a locally static scalar subtype, a locally static array subtype, a locally static record subtype, a locally static access subtype, or a locally static file subtype.

7.4.2 Globally static primaries

An expression is said to be globally static if and only if every operator in the expression denotes a pure function and every primary in the expression is a *globally static primary*, where a globally static primary is a primary that, if it denotes an object or a function, does not denote a dynamically elaborated named entity (see 12.5) and is one of the following:

a) A literal of type TIME

b) A locally static primary

c) A generic constant

d) A generate parameter

e) A constant (including a deferred constant)

f) An alias whose aliased name (given in the corresponding alias declaration) is a globally static primary

g) An array aggregate, if and only if

   1) All expressions in its element associations are globally static expressions, and

   2) All ranges in its element associations are globally static ranges

h) A record aggregate, if and only if all expressions in its element associations are globally static expressions

i) A function call whose function name denotes a pure function and whose actual parameters are each globally static expressions

j) A predefined attribute that is a value and whose prefix is either a globally static subtype or is an object or function call that is of a globally static subtype

k) A predefined attribute that is a function, other than the predefined attributes 'EVENT, 'ACTIVE, 'LAST_EVENT, 'LAST_ACTIVE, 'LAST_VALUE, 'DRIVING, and 'DRIVING_VALUE, whose prefix is either a globally static subtype or is an object or function call that is of a globally static subtype, and whose actual parameter (if any) is a globally static expression

l) A user-defined attribute whose value is defined by a globally static expression

m) A qualified expression whose operand is a globally static expression

n) A type conversion whose expression is a globally static expression

o) An allocator of the first form (see 7.3.6) whose subtype indication denotes a globally static subtype

p) An allocator of the second form whose qualified expression is a globally static expression

q) A globally static expression enclosed in parentheses

r) A subelement or a slice of a globally static primary, provided that any index expressions are globally static expressions and any discrete ranges used in slice names are globally static discrete ranges

A globally static range is either a range of the second form (see 3.1) whose bounds are globally static expressions, or a range of the first form whose prefix denotes either a globally static subtype or an object that is of a globally static subtype. A globally static range constraint is a range constraint whose range is globally static. A globally static scalar subtype is either a scalar base type or a scalar subtype formed by imposing on a globally static subtype a globally static range constraint. A globally static discrete range is either a globally static subtype or a globally static range.

A globally static index constraint is an index constraint for which each index subtype of the corresponding array type is globally static and in which each discrete range is globally static. A globally static array subtype is a constrained array subtype formed by imposing on an unconstrained array type a globally static index constraint. A globally static record subtype is a record type whose fields are all of globally static subtypes. A globally static access subtype is a subtype denoting an access type. A globally static file subtype is a subtype denoting a file type.

A globally static subtype is either a globally static scalar subtype, a globally static array subtype, a globally static record subtype, a globally static access subtype, or a globally static file subtype.

NOTES

1—An expression that is required to be a static expression ~~may~~ must[19] either be a locally static expression or a globally static expression. Similarly, a range, a range constraint, a scalar subtype, a discrete range, an index constraint, or an array subtype that is required to be static ~~may~~ must[20] either be locally static or globally static.2—The rules for locally and globally static expressions imply that a declared constant or a generic may be initialized with an expression that is neither globally nor locally static; for example, with a call to an impure function. The resulting constant value may be globally or locally static, even though its subtype or its initial value expression is neither. Only interface constant, variable, and signal declarations require that their initial value expressions be static expressions.

7.5 Universal expressions

A *universal_expression* is either an expression that delivers a result of type *universal_integer* or one that delivers a result of type *universal_real*.

19. IR1000.4.7.
20. IR1000.4.7.

The same operations are predefined for the type universal_integer as for any integer type. The same operations are predefined for the type universal_real as for any floating-point type. In addition, these operations include the following multiplication and division operators:

Operator	Operation	Left operand type	Right operand type	Result type
*	Multiplication	*Universal real*	*Universal integer*	*Universal real*
		Universal integer	*Universal real*	*Universal real*
/	Division	*Universal real*	*Universal integer*	*Universal real*

The accuracy of the evaluation of a universal expression of type *universal_real* is at least as good as the accuracy of evaluation of expressions of the most precise predefined floating-point type supported by the implementation, apart from *universal_real* itself.

For the evaluation of an operation of a universal expression, the following rules apply. If the result is of type *universal_integer*, then the values of the operands and the result must lie within the range of the integer type with the widest range provided by the implementation, excluding type *universal_integer* itself. If the result is of type *universal_real*, then the values of the operands and the result must lie within the range of the floating-point type with the widest range provided by the implementation, excluding type *universal_real* itself.

NOTE

—The predefined operators for the universal types are declared in package STANDARD as shown in 14.2.