## 7.1 Operators

### 7.1.0 General

TTCN‑3 supports a number of predefined operators that may be used in the terms of TTCN‑3 expressions. The predefined operators fall into seven categories:

a) arithmetic operators;

b) list operator;

c) relational operators;

d) logical operators;

e) bitwise operators;

f) shift operators;

g) rotate operators.

h) presence checking operators.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5, the following restrictions apply:

a) When an expression is evaluated, the evaluated values used as the operands of operators shall be completely initialized, except for those operands for which it is explicitly allowed to be partially initialized (see clause 11.1).

These operators are listed in table 5.

Table 5: List of TTCN‑3 operators

| Category | Operator | Symbol or Keyword |
| --- | --- | --- |
| **Arithmetic operators** | addition | **+** |
| subtraction | **-** |
| multiplication | **\*** |
| division | **/** |
| modulo | **mod** |
| remainder | **rem** |
| **String operators** | concatenation | **&** |
| **Relational operators** | equal | **==** |
| less than | **<** |
| greater than | **>** |
| not equal | **!=** |
| greater than or equal | **>=** |
| less than or equal | **<=** |
| **Logical operators** | logical not | **not** |
| logical and | **and** |
| logical or | **or** |
| logical xor | **xor** |
| **Bitwise operators** | bitwise not | **not4b** |
| bitwise and | **and4b** |
| bitwise or | **or4b** |
| bitwise xor | **xor4b** |
| **Shift operators** | shift left | **<<** |
| shift right | **>>** |
| **Rotate operators** | rotate left | **<@** |
| rotate right | **@>** |
| **Presence checking operators** | field presence check | **ispresent** |
| chosen alternative check | **ischosen** |
| value check | **isvalue** |
| bound check | **isbound** |

The precedence of these operators is shown in table 6. Within any row in this table, the listed operators have equal precedence. If more than one operator of equal precedence appears in an expression, the operations are evaluated from left to right. Parentheses may be used to group operands in expressions, in which case a parenthesized expression has the highest precedence for evaluation.

Table 6: Precedence of Operators

|  |  |  |
| --- | --- | --- |
| Priority | Operator type | Operator |
| highest  Lowest | Unary  Unary  Binary  Binary  Unary  Binary  Binary  Binary  Binary  Binary  Binary  Unary  Binary  Binary  Binary | **( … )**  **ispresent, ischosen, isvalue, isbound**  **+, -**  **\*, /**, **mod**, **rem**  **+, -, &**  **not4b**  **and4b**  **xor4b**  **or4b**  **<<, >>, <@, @>**  **<, >, <=, >=**  **==, !=**  **not**  **and**  **xor**  **or** |

### 7.1.1 Arithmetic operators

The arithmetic operators represent the operations of addition, subtraction, multiplication, division, modulo and remainder. Operands of these operators shall be of **integer** values (including derivations of **integer**) or floating‑point numbers (including derivations of **float**, containing numeric values only), except for **mod** and **rem** which shall be used with **integer** (including derivations of **integer**) types only.

The usage of the special float values **infinity**, **-infinity** and **not\_a\_number** in arithmetic operators shall follow the rules defined in IEEE 754 [6].

With **integer** types, the result type of arithmetic operations is **integer**. With float types, the result type of arithmetic operations is **float**.

In the case where plus (+) or minus (-) is used as the unary operator the rules for operands apply as well. The result of using the minus operator is the negative value of the operand if it was positive and vice versa. The result of using the plus operator is the value of the operand, i.e. a positive value if the operand value was positive and a negative value if the operand value was negative.

The result of performing the division operation (/) on two:

a) **integer** values gives the whole **integer** part of the value resulting from dividing the first **integer** by the second (i.e. fractions are discarded);

b) numeric **float** values gives the **float** value resulting from dividing the first **float** by the second (i.e. fractions are not discarded).

The operators **rem** and **mod** compute on operands of type **integer** and have a result of type **integer**. The operations x **rem** y and x **mod** y compute the rest that remains from an integer division of x by y. Therefore, they are only defined for non-zero operands y. For positive x and y, both x **rem** y and x **mod** y have the same result but for negative arguments they differ.

Formally, **mod** and **rem** are defined as follows:

x **rem** y = x - y \* (x/y)

x **mod** y = x **rem** |y| when x >= 0

= 0 when x < 0 and x **rem** |y| = 0

= |y| + x **rem** |y| when x < 0 and x **rem** |y| < 0

Table 7 illustrates the difference between the **mod** and **rem** operator.

Table 7: Effect of mod and rem operator

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| x | -3 | -2 | -1 | 0 | 1 | 2 | 3 |
| x mod 3 | 0 | 1 | 2 | 0 | 1 | 2 | 0 |
| x rem 3 | 0 | -2 | -1 | 0 | 1 | 2 | 0 |

### 7.1.2 List operator

The predefined list operator (&) performs concatenation of values of string types, **record of**, **set of**, or **array** of the same root types. The operation is a simple concatenation from left to right. No form of arithmetic addition is implied. The result type is the root type of the operands.

NOTE 1: In case of the list types, both the outer type (i.e. **record of**, **set of** or **array**) and the iterated inner type need to have the same root type in a recursive manner.

NOTE 2: It is also possible to concatenate two or more value list notation expressions if the result is to be used as a **record of** or **array** of the same root type as the concatenated expressions.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5, the following restrictions apply:

a) When the list concatenation operator is used for record of-s, set of-s and arrays, its operands shall be at least partially initialized.

EXAMPLE:

'1111'B & '0000'B & '1111'B gives '111100001111'B

{1,2} & {3,4} & {5,6} gives the following record of integer {1,2,3,4,5,6}

### 7.1.3 Relational operators

The predefined relational operators are equality (==), less than (<), greater than (>), non‑equality to (!=), greater than or equal to (>=) and less than or equal to (<=). The result type of all these operations is **boolean**.

The relational operators less than (<), greater than (>), greater than or equal to (>=), and less than or equal to (<=) shall have only operands of type **integer** (including derivations of **integer**), **float** (including derivations of **float**), or instances of the same **enumerated** type. It is not allowed to compare instances of different root types.

The **address** type is allowed for the equality (==) and non-equality (!=) operators, independent of its actual type, but when its actual type differs from the types specified above, it can be compared to the literal special value **null** only.

Operands of equality (==) and non-equality (!=) shall be completely initialized values or field references, and with the exception of **enumerated** types, shall be of compatible root types. The values or field references being compared shall obey the following rules. This implies that instances of types not mentioned below shall not be operands of equality and non-equality:

* Two field references are equal if the referenced fields are both **optional** fields and both fields are set to **omit** or if both referenced fields (regardless if they are optional or not) are initialized with values and these values are equal. A field reference is equal to a value if the referenced field is initialized with a value and both values are equal.
* Two integer values are equal if and only if they contain the same value. Otherwise, normal mathematical ordering is applied.
* Enumerated values of the same, or different types can be compared. In the case of different **enumerated** types, expression "b" of type "B" can be compared with expression "a" of type "A" if the two types "A" and "B" can be merged to a consistent larger enumerated type (i.e. where numbers are not associated with different identifiers, see also clause 6.3.2.1). Two enumerated values are equal if and only if both their identifiers and associated integer values (associated either explicitly or implicitly, see clause 6.2.4) are the same.
* Two floating-point numbers are equal if and only if they contain the same value. The values minus zero and plus zero are two distinct values (e.g. they are encoded differently in some standardized languages) and minus zero is less than plus zero, which represents zero. Otherwise, normal mathematical ordering is applied. The special values **‑infinity,** **infinity** and **not\_a\_number** are equal to themselves only. The special value **‑infinity** is less than any other float value. The special value **infinity** is greater than any numerical float values and **-infinity**. The special value **not\_a\_number** is greater than any other float value (including **infinity)**.
* Two charstring or two universal charstring values are equal if and only if they have equal lengths and the characters at all positions are the same.
* For values of bitstring, hexstring or octetstring types, the same equality rule applies as for charstring values with the exception, that fractions which shall equal at all positions are bits, hexadecimal digits or pairs of hexadecimal digits accordingly.
* Two record values, or set values are equal respectively if and only if they are mutually compatible with the type of the other operand (see clause 6.3.2.2), the actual values of all present fields are equal to their corresponding fields and all fields corresponding to omitted fields are also omitted in the peer value.
* Two record of values, set of values or array values, respectively, are equal if and only if they are mutually compatible with the type of the other operand (see clause 6.3.2.3), they both have the same length, and each element of one value is equal to the corresponding element of the other value. Record of values and array values may also be compared, in which case the corresponding record of type of the array is being considered.
* Values of the same union type, and values of different union types in which at least one of the alternatives is compatible with the other type (see clause 6.3.2.4) can be compared (independent if a compatible alternative is the selected one or not). Two values of union types are equal if and only if in both values the name of the selected alternative is identical, they are compatible with the type of the other value, and the actual values of the chosen fields are equal.
* Values of the same or any two anytype types can be compared. For anytype values the same rule apply as to union values, with the addition that names of user-defined types defined with the same name in different modules do not denote the same type name of the selected alternatives.
* Two default, two port, two timer or two component values are equal if and only if they both are initialized with the special value **null** or they both contain a reference to the same entity (i.e. they designate the same default, port, timer or test component, independent of the actual state of the denoted object).
* It is also possible to use compound expressions (field assignment or value list notation) directly as operands of comparison operations of structured types. If there is a compound expression on both sides of the comparison operator, they shall both be value list notation expressions where the elements shall be of the same root type and they shall be compared like record of values with elements of that root type. If only one operand of the comparison operation is a compound expression it shall be compatible with the root type of the other operand and they shall be compared like values of that root type.

EXAMPLE 1: Comparing enumerated values

// Given

**type enumerated** EWeekDays {

Mon, Tue, Wed, Thu, Fri, Sat, Sun

};

**type enumerated** EWorkDays {

Mon, Tue, Wed, Thu, Fri

};

**type enumerated** EDesWeekDays {

Tue, Wed, Thu, Fri, Sat, Sun, Mon

};

var EWeekDays v\_myWeekDayMon := Mon

var EWeekDays v\_myWeekDaySun := Sun

var EWorkDays v\_myWorkDayMon := Mon

var EDesWeekDays v\_myDesWeekDayMon := Mon

// Then

v\_myWeekDayMon == v\_myWorkDayMon;

// returns true

v\_myWeekDaySun == v\_myWorkDayMon;

// returns false, because Sun is not a possible value in EworkDays

v\_myDesWeekDayMon == v\_myWeekDayMon;

// returns false: though the identifiers in both enumerated types are the same,

// but the integer values associated with the identifiers are different

EXAMPLE 2: Comparing values of other structured types

// Given

**type** **set** S1 {

**integer** a1 **optional**,

**integer** a2 **optional**,

**integer** a3 **optional**

};

**type** **set** S2 {

**integer** b1 **optional**,

**integer** b2 **optional**,

**integer** b3 **optional**

};

**type** **set** S3 {

**integer** c1 **optional**,

**integer** c2 **optional**,

};

**type** **set** **of** **integer** SI;

**type** **union** U1 {

**integer** d1,

**integer** d2,

};

**type** **union** U2 {

**integer** e1,

**integer** e2,

};

**type** **union** U3 {

**integer** d1,

**integer** d2,

**boolean** d3

};

// And

**const** S1 c\_s1 := { a1 := 0, a2 := **omit**, a3 := 2 };

// Notice that the order of defining values of the fields does not matter

**const** S2 c\_s2a := { b1 := 0, b3 := 2, b2 := **omit** };

**const** S2 c\_s2b := { b2 := 0, b3 := 2, b1 := **omit** };

**const** S3 c\_s3 := { c1 := 0, c2 :=2 };

**var** SI v\_si:= { 0, -, 2 };

**const** SI c\_si := { 0, 2 };

**const** U1 c\_u1 := { d1:= 0 };

**const** U2 c\_u2 := { e1:= 0 };

**const** U3 c\_u3; := { d1:= 0 };

// Then

c\_s1 == c\_s2a;

// returns **true**

c\_s1 == c\_s2b;

// returns **false**, because neither a1 nor a2 are equal to their counterparts

// (the corresponding element is not omitted)

c\_s1 == c\_s3;

// returns **false**, because the effective value structures of s1 and s3 are not compatible

c\_s1 == v\_si;

// causes test case error as v\_si is not completely initialized  
 // (2nd element is left uninitialized)

c\_s1 == c\_si;

// returns **false**, as the counterpart of the omitted a2 is 2,

// but the counterpart of a3 is undefined

c\_s3 == c\_si;

// returns **true**

c\_u1 == c\_u2;

// causes error as U1 and U2 have no common subset of alternatives

c\_u1 == c\_u3;

// returns **true**, as alternatives with the same names are chosen and

// the actual values in the selected alternatives are equal

{ 0, **omit**, 2 } == c\_s1;

// returns true

c\_s2a == { b1 := 0, b2:= omit, b3 := 2 };

// returns true

{ c\_s1, c\_s2b } == { c\_s2a, c\_s1 };

// returns false because c\_s2b != c\_s1

{ c\_s1, c\_s2b, c\_s2a } == { c\_s1 };

// returns false because of different length

c\_s1.a1 == c\_s2a.b1;

// returns true, both fields are initialized with values and the values are equal

c\_s1.a2 == c\_s2a.b2;

// returns true, both fields are omit

c\_s1.a1 == c\_s2a.b2;

// returns false, value vs. omit

c\_s1.a1 == **omit**;

// error, omit is neither a value nor a field reference

c\_s1.a2 == 3;

// false, omit vs. value