## 19.3 The Select statements

### 19.3.1 The Select case statement

The **select case** statement is an alternative syntactic form of the **if-else** statement.

***Syntactical Structure***

**select** "(" *SingleExpression* ")" "{"

 { **case** "(" { *SingleExpression* [","] } ")" *StatementBlock* }+

 [ **case** **else** *StatementBlock* ]

"}"

***Semantic Description***

The **select case** statement is an alternative to using **if** .. **else** **if** .. **else** statements when comparing a value to one or several other values. The statement contains a header part and one or more branches. Never more than one of the branches is executed.

In the header part of the **select case** statement an expression shall be given. Each branch starts with the **case** keyword followed by a list of templateInstance (a list branch, which may also contain a single element) or the **else** keyword (an else branch) and a statement block.

All templateInstance in all list branches shall be of a type compatible with the type of the expression in the header.
A list branch is selected and the statement block of the selected branch is executed only, if any of the templateInstance matches the value of the expression in the header of the statement. On executing the statement block of the selected branch (i.e. not jumping out by a go to statement), execution continues with the statement following the select case statement.

The statement block of an else branch is always executed if no other branch textually preceding the else branch has been selected.

Branches are evaluated in their textual order. If none of the templateInstance-s matches the value of the expression in the header and the statement contains no else branch, execution continues without executing any of the **select case** branches.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

1. The **select** *SingleExpression* and the **case** *SingleExpression*-s shall be type compatible.

***Examples***

 **select** (MyModulePar) // where MyModulePar is of charstring type

 { **case** ("firstValue")
 {
 **log** ("The first branch is selected");
 }
 **case** (MyCharVar, MyCharConst)
 {
 **log** ("The second branch is selected");
 }
 **case** **else**
 {
 **log** ("The value of the module parameter MyModulePar is selected");
 }

 }

 // the above select statement is equivalent to the following nested if-else statement.

 // Note: the following textual replacement of the select-case statement is described in

 // the operational semantics of TTCN-3.

 {

 **var charstring** myTempVar := MyModulePar;

 **if** (**match**(myTempVar, "firstValue")
 {
 **log** ("The first branch is selected");
 }
 **else** **if** (**match**(myTempVar, MyCharVar) **or** **match**(myTempVar, MyCharConst))
 {
 **log** ("The second branch is selected");
 }
 **else**
 {
 **log** ("The value of the module parameter MyModulePar is selected");
 }

 }

### 19.3.2 The Select union statement

To allow easier usage of the select statement for values of union types, a special form of the select statement exists.

***Syntactical Structure***

**select union** "(" *TemplateInstance* ")" "{"

 { **case** "(" { *Identifier* [","] } ")" *StatementBlock* }+

 [ **case** **else** *StatementBlock* ]

"}"

***Semantic Description***

The statement contains a header part and one or more branches. Never more than one of the branches is executed.

In the header part of the **select union** statement a template instance of union type shall be given. Each branch shall start with the **case** keyword followed by one or more identifiers of the alternatives (fields) of the union type (a list branch) or the **else** keyword (an else branch) and a statement block. The StatementBlock of the list branch containing the identifier of the chosen alternative is executed. If no case exists for the chosen alternative, the StatementBlock of the else branch, if it is present, is executed. Otherwise, the **select union** statement has no effect.

***Restrictions***

1. The *TemplateInstance* in the header of the **select union** statement shall be of a **union** type. It shall be at least partially initialized.
2. Every *Identifier* in a **case** of the **select union** statement shall be an identifier of an alternative of the **union** type of the template instance given to the statement's header.
3. No two cases in a **select union** statement shall have the same case *Identifier*.

***Examples***

 **type** **union** Messages {

 MyMessageType1 msg1,

 MyMessageType2 msg2,

 MyMessageType3 msg3,

 MyMessageType4 msg4,

 MyMessageType5 msg5

 }

 **function** f(**in** Messages msg) {

 **select** **union** (msg) {

 **case** (msg1) { **log**(msg.msg1); }

 **case** (msg2) { **log**(msg.msg2); }

 **case** (msg3, msg4) { **log**("either msg3 or msg4"); }

 **case** else { **log**("unhandled variant"); }

 }

#### A.1.6.8.3 Basic statements

BasicStatements ::= [Assignment](#TAssignment) |

 [LogStatement](#TLogStatement) |

 [LoopConstruct](#TLoopConstruct) |

 [ConditionalConstruct](#TConditionalConstruct) |

 [SelectCaseConstruct](#TSelectCaseConstruct) |

 [StatementBlock](#TStatementBlock)

Expression ::= [SingleExpression](#TSingleExpression) | [CompoundExpression](#TCompoundExpression)

CompoundExpression ::= [FieldExpressionList](#TFieldExpressionList) | [ArrayExpression](#TArrayExpression)

/\* STATIC SEMANTICS - Within CompoundExpression the ArrayExpression can be used for Arrays, record, record of and set of types. \*/

FieldExpressionList ::= "{" [FieldExpressionSpec](#TFieldExpressionSpec) {"," [FieldExpressionSpec](#TFieldExpressionSpec)}

 "}"

FieldExpressionSpec ::= [FieldReference](#TFieldReference) [AssignmentChar](#TAssignmentChar) [NotUsedOrExpression](#TNotUsedOrExpression)

ArrayExpression ::= "{" [[ArrayElementExpressionList](#TArrayElementExpressionList)] "}"

ArrayElementExpressionList ::= [NotUsedOrExpression](#TNotUsedOrExpression) {"," [NotUsedOrExpression](#TNotUsedOrExpression)}

NotUsedOrExpression ::= [Expression](#TExpression) | [Minus](#TMinus)

ConstantExpression ::= [SingleExpression](#TSingleExpression) | [CompoundConstExpression](#TCompoundConstExpression)

BooleanExpression ::= [SingleExpression](#TSingleExpression)

/\* STATIC SEMANTICS - BooleanExpression shall resolve to a Value of type Boolean \*/

CompoundConstExpression ::= [FieldConstExpressionList](#TFieldConstExpressionList) | [ArrayConstExpression](#TArrayConstExpression)

/\* STATIC SEMANTICS - Within CompoundConstExpression the ArrayConstExpression can be used for arrays, record, record of and set of types. \*/

FieldConstExpressionList ::= "{" [FieldConstExpressionSpec](#TFieldConstExpressionSpec) {"," [FieldConstExpressionSpec](#TFieldConstExpressionSpec)}

 "}"

FieldConstExpressionSpec ::= [FieldReference](#TFieldReference) [AssignmentChar](#TAssignmentChar) [ConstantExpression](#TConstantExpression)

ArrayConstExpression ::= "{" [[ArrayElementConstExpressionList](#TArrayElementConstExpressionList)] "}"

ArrayElementConstExpressionList ::= [ConstantExpression](#TConstantExpression) {"," [ConstantExpression](#TConstantExpression)}

Assignment ::= [VariableRef](#TVariableRef) [AssignmentChar](#TAssignmentChar) ([Expression](#TExpression) | [TemplateBody](#TTemplateBody))

/\* STATIC SEMANTICS - The Expression on the right hand side of Assignment shall evaluate to an explicit value of a type compatible with the type of the left hand side for value variables and shall evaluate to an explicit value, template (literal or a template instance) or a matching mechanism compatible with the type of the left hand side for template variables. \*/

SingleExpression ::= [XorExpression](#TXorExpression) {"or" [XorExpression](#TXorExpression)}

/\* STATIC SEMANTICS - If more than one XorExpression exists, then the XorExpressions shall evaluate to specific values of compatible types \*/

XorExpression ::= [AndExpression](#TAndExpression) {"xor" [AndExpression](#TAndExpression)}

/\* STATIC SEMANTICS - If more than one AndExpression exists, then the AndExpressions shall evaluate to specific values of compatible types \*/

AndExpression ::= [NotExpression](#TNotExpression) {"and" [NotExpression](#TNotExpression)}

/\* STATIC SEMANTICS - If more than one NotExpression exists, then the NotExpressions shall evaluate to specific values of compatible types \*/

NotExpression ::= ["not"] [EqualExpression](#TEqualExpression)

/\* STATIC SEMANTICS - Operands of the not operator shall be of type boolean or derivatives of type Boolean. \*/

EqualExpression ::= [RelExpression](#TRelExpression) {[EqualOp](#TEqualOp) [RelExpression](#TRelExpression)}

/\* STATIC SEMANTICS - If more than one RelExpression exists, then the RelExpressions shall evaluate to specific values of compatible types. If only one RelExpression exists, it shall not derive to a CompoundExpression. \*/

RelExpression ::= [ShiftExpression](#TShiftExpression) [[RelOp](#TRelOp) [ShiftExpression](#TShiftExpression)] | [CompoundExpression](#TCompoundExpression)

/\* STATIC SEMANTICS - If both ShiftExpressions exist, then each ShiftExpression shall evaluate to a specific integer, Enumerated or float Value or derivatives of these types \*/

ShiftExpression ::= [BitOrExpression](#TBitOrExpression) {[ShiftOp](#TShiftOp) [BitOrExpression](#TBitOrExpression)}

/\* STATIC SEMANTICS - Each Result shall resolve to a specific Value. If more than one Result exists the right-hand operand shall be of type integer or derivatives and if the shift op is "<<" or ">>" then the left-hand operand shall resolve to either bitstring, hexstring or octetstring type or derivatives of these types. If the shift op is " \*/

BitOrExpression ::= [BitXorExpression](#TBitXorExpression) {"or4b" [BitXorExpression](#TBitXorExpression)}

/\* STATIC SEMANTICS - If more than one BitXorExpression exists, then the BitXorExpressions shall evaluate to specific values of compatible types \*/

BitXorExpression ::= [BitAndExpression](#TBitAndExpression) {"xor4b" [BitAndExpression](#TBitAndExpression)}

/\* STATIC SEMANTICS - If more than one BitAndExpression exists, then the BitAndExpressions shall evaluate to specific values of compatible types \*/

BitAndExpression ::= [BitNotExpression](#TBitNotExpression) {"and4b" [BitNotExpression](#TBitNotExpression)}

/\* STATIC SEMANTICS - If more than one BitNotExpression exists, then the BitNotExpressions shall evaluate to specific values of compatible types \*/

BitNotExpression ::= ["not4b"] [AddExpression](#TAddExpression)

/\* STATIC SEMANTICS - If the not4b operator exists, the operand shall be of type bitstring, octetstring or hexstring or derivatives of these types. \*/

AddExpression ::= [MulExpression](#TMulExpression) {[AddOp](#TAddOp) [MulExpression](#TMulExpression)}

/\* STATIC SEMANTICS - Each MulExpression shall resolve to a specific Value. If more than one MulExpression exists and the AddOp resolves to StringOp then the MulExpressions shall be valid operands for StringOp. If more than one MulExpression exists and the AddOp does not resolve to StringOp then the MulExpression shall both resolve to type integer or float or derivatives of these types. If only one MulExpression exists, it shall not derive to a CompoundExpression. \*/

MulExpression ::= [UnaryExpression](#TUnaryExpression) {[MultiplyOp](#TMultiplyOp) [UnaryExpression](#TUnaryExpression)} | [CompoundExpression](#TCompoundExpression)

/\* STATIC SEMANTICS - Each UnaryExpression shall resolve to a specific Value. If more than one UnaryExpression exists then the UnaryExpressions shall resolve to type integer or float or derivatives of these types. \*/

UnaryExpression ::= [[UnaryOp](#TUnaryOp)] [Primary](#TPrimary)

/\* STATIC SEMANTICS - Primary shall resolve to a specific Value of type integer or float or derivatives of these types.\*/

Primary ::= [OpCall](#TOpCall) |

 [Value](#TValue) |

 "(" [SingleExpression](#TSingleExpression) ")"

ExtendedFieldReference ::= {([Dot](#TDot) ([Identifier](#TIdentifier) | [PredefinedType](#TPredefinedType))) |

 [ArrayOrBitRef](#TArrayOrBitRef) |

 ("[" [Minus](#TMinus) "]")

 }+

/\* STATIC SEMANTIC - The Identifier refers to a type definition if the type of the VarInstance or ReferencedValue in which the ExtendedFieldReference is used is anytype. ArrayOrBitRef shall be used when referencing elements of values or arrays. The square brackets with dash shall be used when referencing inner types of a record of or set of type. \*/

OpCall ::= [ConfigurationOps](#TConfigurationOps) |

 [GetLocalVerdict](#TGetLocalVerdict) |

 [TimerOps](#TTimerOps) |

 [TestcaseInstance](#TTestcaseInstance) |

 ([FunctionInstance](#TFunctionInstance) [[ExtendedFieldReference](#TExtendedFieldReference)]) |

 ([TemplateOps](#TTemplateOps) [[ExtendedFieldReference](#TExtendedFieldReference)]) |

 [ActivateOp](#TActivateOp)

AddOp ::= "+" |

 "-" |

 [StringOp](#TStringOp)

/\* STATIC SEMANTICS - Operands of the "+" or "-" operators shall be of type integer or float or derivations of integer or float (i.e. subrange) \*/

MultiplyOp ::= "\*" | "/" | "mod" | "rem"

/\* STATIC SEMANTICS - Operands of the "\*", "/", rem or mod operators shall be of type integer or float or derivations of integer or float (i.e. subrange) \*/

UnaryOp ::= "+" | "-"

/\* STATIC SEMANTICS - Operands of the "+" or "-" operators shall be of type integer or float or derivations of integer or float (i.e. subrange) \*/

RelOp ::= "<" | ">" | ">=" | "<="

/\* STATIC SEMANTICS - the precedence of the operators is defined in Table 6 \*/

EqualOp ::= "==" | "!="

StringOp ::= "&"

/\* STATIC SEMANTICS - Operands of the list operator shall be bitstring, hexstring, octetstring, (universal) character string, record of, set of, or array types, or derivates of these types \*/

ShiftOp ::= "<<" | ">>" | "<@" | "@>"

LogStatement ::= [LogKeyword](#TLogKeyword) "(" [LogItem](#TLogItem) {"," [LogItem](#TLogItem)} ")"

LogKeyword ::= "log"

LogItem ::= [FreeText](#TFreeText) | [InLineTemplate](#TInLineTemplate)

LoopConstruct ::= [ForStatement](#TForStatement) |

 [WhileStatement](#TWhileStatement) |

 [DoWhileStatement](#TDoWhileStatement)

ForStatement ::= [ForKeyword](#TForKeyword) "(" [Initial](#TInitial) [SemiColon](#TSemiColon) [BooleanExpression](#TBooleanExpression)

 [SemiColon](#TSemiColon) [Assignment](#TAssignment) ")" [StatementBlock](#TStatementBlock)

ForKeyword ::= "for"

Initial ::= [VarInstance](#TVarInstance) | [Assignment](#TAssignment)

WhileStatement ::= [WhileKeyword](#TWhileKeyword) "(" [BooleanExpression](#TBooleanExpression) ")" [StatementBlock](#TStatementBlock)

WhileKeyword ::= "while"

DoWhileStatement ::= [DoKeyword](#TDoKeyword) [StatementBlock](#TStatementBlock) [WhileKeyword](#TWhileKeyword) "(" [BooleanExpression](#TBooleanExpression)

 ")"

DoKeyword ::= "do"

ConditionalConstruct ::= [IfKeyword](#TIfKeyword) "(" [BooleanExpression](#TBooleanExpression) ")" [StatementBlock](#TStatementBlock)

 {[ElseIfClause](#TElseIfClause)} [[ElseClause](#TElseClause)]

IfKeyword ::= "if"

ElseIfClause ::= [ElseKeyword](#TElseKeyword) [IfKeyword](#TIfKeyword) "(" [BooleanExpression](#TBooleanExpression) ")" [StatementBlock](#TStatementBlock)

ElseKeyword ::= "else"

ElseClause ::= [ElseKeyword](#TElseKeyword) [StatementBlock](#TStatementBlock)

SelectCaseConstruct ::= [SelectKeyword](#TSelectKeyword) [[UnionKeyword](#TUnionKeyword)] "(" [SingleExpression](#TSingleExpression) ")" [SelectCaseBody](#TSelectCaseBody)

SelectKeyword ::= "select"

SelectCaseBody ::= "{" {[SelectCase](#TSelectCase)}+ [CaseElse] "}"

SelectCase ::= [CaseKeyword](#TCaseKeyword) "(" [InLineTemplate](#TInLineTemplate) {"," [InLineTemplate](#TInLineTemplate)}")" [StatementBlock](#TStatementBlock)

CaseElse ::= CaseKeyword ElseKeyword StatementBlock

/\*\* STATIC SEMANTICS InLineTemplate-s shall be Identifier-s if the UnionKeyword is present in the surrounding SelectCaseConstruct (see clause 19.3.2)\*/

CaseKeyword ::= "case"

ExtendedIdentifier ::= [[Identifier](#TIdentifier) [Dot](#TDot)] [Identifier](#TIdentifier)

IdentifierList ::= [Identifier](#TIdentifier) {"," [Identifier](#TIdentifier)}

QualifiedIdentifierList ::= [QualifiedIdentifier](#TQualifiedIdentifier) {"," [QualifiedIdentifier](#TQualifiedIdentifier)}