**SLang/E# syntax-validity-semantics manual. Version 0.99.20, December 10th 2023**

1. **SLang keywords:**

|  |  |  |
| --- | --- | --- |
| ***#*** | ***Name*** | ***Brief description*** |
| 1 | **abstract** | Unit/Routine characteristic: Bodyless (‘abstract’) unit routine or objectless (‘abstract’) unit. |
| 2 | **active** | Type/entity/statement: It can be a unit or data attribute or raise statement |
| 3 | **alias** | Unit/Routine characteristic: The alternative name of the unit or routine |
| 4 | **as** | Unit level/Type: Another name in use-as directive or anchor reference |
| 5 | **case** | Statement: start of alternative (may be removed if proper parsing done for ‘:’ after alternative tag ) |
| 6 | **const** | Unit level: Start of constant objects declaration section or constant attribute declaration  Unit-routine level: Import of constant objects of some unit |
| 7 | **do** | Statement: Start of the block |
| 8 | **else** | Statement: Start of else part |
| 9 | **elsif** | Statement: Start of the else if section |
| 10 | **end** | End of block or other syntax construction |
| 11 | **ensure** | Predicate: Routine post-condition clause start |
| 12 | **extend** | Unit level: Used to support inheritance and unit extensions. |
| 13 | **final** | Unit level: The unit cannot have descendants  Unit member characteristic: Member can not be overridden down in the inheritance hierarchy. Also final can be applied to attribute of the unit to state finalization action. And it is possible to give a final name to some routine to use it in descendants |
| 14 | **foreign** | Routine characteristic: The body of the routine is coded in 3rd party language |
| 16 | **if** | Statement: Conditional statement start |
| 17 | **in** | Operator: checks if some expression value belongs to the range of values |
| 18 | **init** | Unit: initialization procedure declaration and its call |
| 19 | **is** | Statement: Definition of the initial value of an attribute. Checks for the value or type of expression |
| 20 | **new** | Statement/Expression: Creation of an object. Maybe skipped |
| 21 | **old** | Expression/Statement: Value of some attribute before the routine started. To be used in post-conditions only. For the routine body, it means a call to the previous version of the overridden routine – precursor call |
| 22 | **override** | Unit member characteristics: States that this member overrides all possible inherited versions. |
| 23 | **pure** | Routine characteristic: Routine is prohibited to write into unit attributes or read them. Must work only with its parameters. No side effects. Can be safely evaluated once. Can be overridden only by pure routines |
| 24 | **raise** | Statement: Raises a new exception with some object as an argument. If no argument is provided then it raises the last exception occurred |
| 25 | **ref** | Type: States that an object will be of the reference nature  Expression: produces the reference version of the expression result |
| 26 | **require** | Predicate: Start of precondition clause of the routine, unit, or loop invariant |
| 27 | **return** | Statement: Stops execution of the routine and returns result in case of function.  Expression/Postcondition: Refers to the function result |
| 28 | **rigid** | Attribute prefix: A deep version of attribute immutability. Deep constant |
| 29 | **rtn** | Type: Has 2 meanings – denotes the routine type after a colon (‘:’) or creates a routine object from some routine in expressions |
| 30 | **safe** | Routine characteristic: Routine is prohibited to write into unit attributes but it can read them |
| 31 | **select** | Unit level: select one version among several versions to resolve ambiguity to support polymorphic assignments |
| 32 | **this** | Expression: Reference to the current object  Visibility: private |
| 33 | **unit** | Unit level: Start of the unit description  Expression: duck typing style type check |
| 34 | **use** | Unit/Routine level: It states that the unit mentioned in the use directive will be used as a module (singleton) at the current unit or routine level. It allows renaming units as well. Unit level: give a new name to the inherited member  System-level: import constants of some unit for the current source |
| 35 | **val** | Type: States that an object will be of value nature. The object itself but not a reference to it. |
| 36 | **var** | Attribute/parameter prefix: States that attribute can be assigned many times. It is a variable attribute of any type including routine one. If it is routine parameters then routines with side–effects can be called upon this parameter, as well as an assignment into it |
| 37 | **when** | Statement: Exception handling condition clause. Part of the block |
| 38 | **while** | Statement: Loop condition clause |

* All entity names (unit attributes constant and variable, routine local attributes constant and variable) and routine names are started with low-case character while all unit names are started with upper-case one.
* SLang supports 2 modes of syntax ‘Pascal-like’ and ‘C-like’ depending on the source file extension (.slang and .clang accordingly)
* Lexical elements are literals:
  + numbers (integer (decimal, hex, binary) and real ones)
  + characters (use single quotes ‘’)
  + strings (use double quotes “”)
* Compiler is the tool which receives set of sources and processes them according to their content. Every source should be a valid [Compilation](#Compilation).
* Notation:
  + [] – optional
  + {} – repetition zero or more times
  + () – grouping
  + | - or
  + [CompilationUnit](#CompilationUnit) - non\_terminal
  + **build** – keyword
  + “,” - symbol, separator
  + Cmod – syntax mode, default is ‘Pascal-like’

1. **SLang syntax and validity: list of all syntax rules decorated with validity rules and runtime semantics**
2. Compilation: {[[Context](#Context)] [CompilationUnit](#CompilationUnit)}

S(build, S([CompilationUnit](#CompilationUnit)))

F(EOF, build, S([CompilationUnit](#CompilationUnit)))

**VAL-COMP**: is valid if and only if all [CompilationUnit](#CompilationUnit)s are valid in the context of the nearest [Context](#Context) provided or if no [Context](#Context) provided then in the default or specified as the compiler argument context which is used for validation then.

1. Context:

**build** [FSname](#FileName) (**from** [FSname](#FileName) {[“,”] [FSname](#FileName)})|(**entry** [Identifier](#Identifier))

/\* build library: **from** list of input paths for the build.

E.g. **build** Kernel **from** “.” “../some path”

Open topic: recursive paths like “../some path/\*\*” - not supported currently

or

build program: **entry** point [Identifier](#Identifier) stands for the unit or routine name

In case of the unit the initialization procedure with the single parameter Array<String> will be used as an entry point. If no such procedure found then the initialization procedure with no parameters be assigned as an entry point. It is CTE if none is found.

\*/

[**target** [Win32|Win64|Lin32|Lin64|Android|iOS|MSIL|JVM|C|ARK|All] // target code kind

[**cluster**

{[FSname](#FileName) // Cluster name or path to look for units

[

[**hide** [Identifier](#Identifier) {[“,”][Identifier](#Identifier)}]

/\* Exclude/hide unit clause. Do not consider some units for the current build \*/

[**use** [Identifier](#Identifier) **as** [Identifier](#Identifier) {[“,”][Identifier](#Identifier) **as** [Identifier](#Identifier)}]

/\* Rename unit clause to ensure this unit will be used in the current build under the new name\*/

[**select** [Identifier](#Identifier) {[“,”][Identifier](#Identifier)}]

/\* Select unit clause works to resolve the case when several clusters have units with the same name. Unit name is strictly attached to the particular cluster unit for the current build\*/

[**end**] // if ‘hide’ or ‘use’ or ‘select’ specified then we need ‘end’

]}

]

[**foreign** {[FSname](#FileName)}] // List of 3rd party modules to be linked in

**end**

S(build)

F(S([CompilationUnit](#CompilationUnit)))

**VAL-BLD**: is valid if and only if

1. All [FSname](#FileName)s are valid OS names of files or folders after environment variable references are unfolded
2. Any list (from, hide, use, select) has unique names
3. TBD
4. FSname: (PathOrFileName| [StringConstant](#StringConstant)) [“\*”] // star means recursive ?

**VAL-FSNM**: is valid if and only if FSname refers to a OS-valid folder or file name

1. CompilationUnit: { [GlobalAlias](#GlobalAlias) | [UseDirective](#UseDirective) } ([AnonymousRoutine](#AnonymousRoutine)|[StandaloneRoutine](#StandaloneRoutine)|[UnitDeclaration](#UnitDeclaration))

S(alias, use, S([AnonymousRoutine](#AnonymousRoutine)), S([StandaloneRoutine](#StandaloneRoutine)), S([UnitDeclaration](#UnitDeclaration)))

F(build, S([CompilationUnit](#CompilationUnit)))

**VAL-CU**: is valid if and only if

1. all its [UseDirective](#UseDirective)s are valid
2. all its GlobalAliases are valid
3. [AnonymousRoutine](#AnonymousRoutine) or [StandaloneRoutine](#StandaloneRoutine) or [UnitDeclaration](#UnitDeclaration) is valid

**SEM-CU**:

1. all GlobalAliases are accumulated in accordance with the textual order
2. only the latest [UseDirective](#UseDirective) is actual for the current compilation unit
3. UseDirective:

**use const** [UnitTypeName](#UnitTypeName) {“**,**” [UnitTypeName](#UnitTypeName)} [NewLine]

S(use)

F()

**VAL-USDIR**: is valid if and only if

1. every [UnitTypeName](#UnitTypeName) is valid
2. all [UnitTypeName](#UnitTypeName)s are unique across [UseDirective](#UseDirective)

**SEM-USDIR**: all constants declared in the unit called [UnitTypeName](#UnitTypeName) can be used in the current compilation unit with no qualification.

1. GlobalAlias: **alias** [GlobalAliasElement](#GlobalAliasElement) {“**,**” [GlobalAliasElement](#GlobalAliasElement)}

S(alias)

F()

1. GlobalAliasElement: [AttachedType](#AttachedType) **as** [UnitName](#UnitName)

S(S([AttachedType](#AttachedType)))

F()

1. EnclosedUseDirective:

[**use**

[[EnclosedUseEement](#EnclosedUseEement) {“**,**” [EnclosedUseEement](#EnclosedUseEement)}]

[**const** [UnitTypeName](#UnitTypeName) {“**,**” [UnitTypeName](#UnitTypeName)}]

] [NewLine]

S(use)

F()

**VAL-ENCUSDIR**: is valid if and only if TBD

1. EnclosedUseEement: [UnitTypeName](#UnitTypeName) [**as** [UnitName](#UnitName)]

S(<ident>)

F()

**VAL-ENCUE**: is valid if and only if

1. [UnitTypeName](#UnitTypeName) is valid
2. If [UnitName](#UnitName) is provided then
   1. it is valid
   2. and [UnitTypeName](#UnitTypeName) and [UnitName](#UnitName) refer to different names
   3. [UnitName](#UnitName) is to be unique within the surrounding context (unit or routine)
3. AnonymousRoutine: [[UseDirective](#UseDirective)] [StatementsList](#StatementsList)

S(S([UseDirective](#UseDirective)), S([Statement](#Statement)))

F()

**VAL-ARTN**: is valid if and only if is [StatementsList](#StatementsList) valid and if [UseDirective](#UseDirective) is present then it is valid too.

1. StatementsList: {[Statement](#Statement)[“**;**”]}

S([Statement](#Statement))

F()

**VAL-STMLST**: is valid if and only if every [Statement](#Statement) is valid

1. StandaloneRoutine: [**pure**|**safe**] [Identifier](#Identifier) [[FormalGenerics](#FormalGenerics)] [[Parameters](#Parameters)] [[ReturnType](#ReturnType) [Type](#Type)] [[EnclosedUseDirective](#EnclosedUseDirective)] [[RequireBlock](#RequireBlock)] ([InnerBlock](#InnerBlock) [[EnsureBlock](#EnsureBlock)] BlockEnd)|(((“**=>**”[Expression](#Expression))|**foreign**) [[EnsureBlock](#EnsureBlock) BlockEnd])

S(pure, safe, <ident>)

F()

**VAL-STALNRTN**: is valid if and only if TBD

1. InnerBlock:

(“**{**”Cmod|**do**)|**safe**|**pure** [GroupStart [Identifier](#Identifier) {“,” [Identifier](#Identifier)} GroupEnd]

[StatementsList](#StatementsList) [[WhenClause](#WhenClause) {[WhenClause](#WhenClause)} [**else** [StatementsList](#StatementsList)]]

S(“{”|do,safe,pure)

F()

**VAL-INNBLK**: is valid if and only if TBD

**SEM-INNBLK**:

1. “{” [Identifier](#Identifier) {“,” [Identifier](#Identifier)} “}” - do not check invariants for these entities within the [InnerBlock](#InnerBlock)
2. [StatementsList](#StatementsList) is executed, if exception is raised then current exception is matched against all [WhenClause](#WhenClause)s in the order of declaration. If match is detected then exception is handled running the code of the matched [WhenClause](#WhenClause). If no match then [StatementsList](#StatementsList) is to be executed and exception is handled. If no [StatementsList](#StatementsList) provided then exception is propagated.
3. WhenClause: **when** ([[Identifier](#Identifier)**:**][UnitType](#UnitType))| [Expression](#Expression) BlockStart [StatementsList](#StatementsList)

S(when)

F()

**VAL-WHNCLS**: is valid if and only if

1. [StatementsList](#StatementsList) should be valid
2. If [UnitType](#UnitType) is provided then it is valid and if [Identifier](#Identifier) is provided then it is unique within the surrounding routine context
3. Otherwise [Expression](#Expression) is valid

**SEM-WHNCLS**:

1. If [UnitType](#UnitType) is provided and the current exception type conforms to the [UnitType](#UnitType) or
2. the current exception value is equal to the [Expression](#Expression) value
3. then [StatementsList](#StatementsList) is executed and current exception is handled
4. UnitRoutineParameters: “**(**”[[UnitRoutineParameter](#Parameter){”**;**”|”,” [UnitRoutineParameter](#Parameter)}]“**)**”

S(“(”)

F()

**VAL-UNTRTNPARS**: is valid if and only if

1. every [UnitRoutineParameter](#Parameter) is valid
2. all [UnitRoutineParameter](#Parameter) are unique within the context of [UnitRoutineParameters](#UnitRoutineParameters)
3. StandaloneRoutineParameters:

“**(**”[[StandaloneRoutineParameter](#Parameter){”**;**”|”,” [StandaloneRoutineParameter](#Parameter)}]“**)**”

S(“(”)

F()

**VAL-STALNRTNPARS**: is valid if and only if

1. every [StandaloneRoutineParameter](#Parameter) is valid
2. all [StandaloneRoutineParameter](#Parameter) are unique across [StandaloneRoutineParameters](#StandaloneRoutineParameters)
3. UnitRoutineParameter: [StandaloneRoutineParameter](#StandaloneRoutineParameter)|(“**:=**” [[Identifier](#Identifier)]))

S(rigid, <ident>, “:=”)

F()

**VAL-UNTRTNPAR**: is valid if and only if

1. [StandaloneRoutineParameter](#Parameter) is valid if provided
   1. All [Identifier](#Identifier)s used in [StandaloneRoutineParameter](#Parameter) are uniqiue across all members of the surrounding unit
2. [Identifier](#Identifier) is valid
3. [Identifier](#Identifier) is the name of the surrounding unit attribute
4. StandaloneRoutineParameter: ([[**rigid**] [Identifier](#Identifier){“**,**” [**rigid**] [Identifier](#Identifier)} “**:**” [Type](#Type)) | ([Identifier](#Identifier) “**is**” [Expression](#Expression))

S(rigid, <ident>)

F()

**VAL-STALNRTNPAR**: is valid if and only if TBD

1. every [Identifier](#Identifier) is valid
2. every [Type](#Type) is valid if provided
3. [Expression](#Expression) is valid if provided
4. All [Identifier](#Identifier)s are unique across all current routine parameters
5. There are no parameters with default value after every one with the default value
6. TBD
7. RequireBlock : **require** [PredicatesList](#PredicatesList)

S(require)

F()

**VAL-PREBLK**: is valid if and only if [PredicatesList](#PredicatesList) is valid

1. EnsureBlock : **ensure** [PredicatesList](#PredicatesList)

S(ensure)

F()

**VAL-PSTBLK**: is valid if and only if [PredicatesList](#PredicatesList) is valid

1. InvariantBlock: **require** [PredicatesList](#PredicatesList)

S(require)

F()

**VAL-INVBLK**: is valid if and only if [PredicatesList](#PredicatesList) is valid

1. PredicatesList: [[Predicate](#Predicate){[”**;**”] [Predicate](#Predicate)}]

S([Expression](#Expression))

F()

**VAL-PRDLST**: is valid if and only if

1. every [Predicate](#Predicate) is valid
2. all [Predicate](#Predicate)s are unique
3. Predicate: [BooleanExpression](#BooleanExpression) [[DocumentingComment](#DocumentingComment)]

S([Expression](#Expression))

F()

**VAL-PRD**: is valid if and only if TBD

1. [BooleanExpression](#BooleanExpression) is valid
2. [DocumentingComment](#DocumentingComment) is valid if provided

**SEM-PRD**: [BooleanExpression](#BooleanExpression) is evaluated and if it was evaluated to false then an exception is generated. If [DocumentingComment](#DocumentingComment) is provided then it is passed as an argument for exception object creation.

1. UnitDeclaration: ([**final**] [**ref**|**val**|**active**])|[**abstract**]|[**extend**] **unit** [UnitName](#UnitName) [**alias** [UnitName](#UnitName)] [[FormalGenerics](#FormalGenerics)] [[InheritDirective](#InheritDirective)] [[EnclosedUseDirective](#EnclosedUseDirective)] [[MemberSelection]](#MemberSelection) [[InheritedMemberOverriding]](#InheritedMemberOverriding) [“**{**”**Cmod**] [[ConstObjectsDeclaration](#ConstObjectsDeclaration)] { ( [MemberVisibility](#MemberVisibility) “:” {[MemberDeclaration](#MemberDeclaration)}) | [MemberDeclaration](#MemberDeclaration) } [[InvariantBlock](#InvariantBlock)] BlockEnd

S(final, ref, val, active, abstract, extend, unit)

F(build, S([CompilationUnit](#CompilationUnit)))

**VAL-UNTDCL**: is valid if and only if TBD

1. InheritDirective: **extend** [Parent](#Parent) {“,” [Parent](#Parent)}

S(extend)

F()

**VAL-INHDIR**: is valid if and only if

1. every [Parent](#Parent) is valid
2. all [Parent](#Parent)s are unique within the [InheritDirective](#InheritDirective)
3. Parent: [UnitTypeName](#UnitTypeName) | (“**~**” [UnitTypeName](#UnitTypeName) [“(”[MemberName](#MemberName){“,”[MemberName](#MemberName)}“)”])

S(<ident>, “**~**”)

F()

**VAL-PRNT**: is valid if and only if TBD

1. MemberName: [Identifier](#Identifier)|([RoutineName](#RoutineName) [[Signature](#Signature)])

S(<ident>, S([RoutineName](#RoutineName)))

F()

**VAL-MBRNM**: is valid if and only if TBD

1. FormalGenerics: GenericsStart [FormalGeneric](#FormalGeneric) {“**,**” [FormalGeneric](#FormalGeneric)} GenericsEnd

S()

F()

**VAL-FGLST**: is valid if and only if

1. every [FormalGeneric](#FormalGeneric) is valid
2. all [FormalGeneric](#FormalGeneric)s are unique
3. FormalGeneric: ([UnitName](#UnitName) [**extend** [UnitTypeName](#UnitTypeName)] [**init** [[Signature](#Signature)]])| ([Identifier](#Identifier) “**:**” [UnitType](#UnitType)|[RoutineType](#RoutineType))

S(S([UnitName](#UnitName)), <ident>)

F()

**VAL-FG**: is valid if and only if

1. TBD every is valid
2. MemberSelection: **select** [MemberName](#MemberName) {“**,**” [MemberName](#MemberName)}

S(select)

F()

**VAL-MBR-SEL**: is valid if and only if

1. every [MemberName](#MemberName) is valid and unique
2. InheritedMemberOverriding: **override** [UnitTypeName](#UnitTypeName)”**.**”[MemberName](#MemberName) {“,” [UnitTypeName](#UnitTypeName)”**.**”[MemberName](#MemberName)}

S(override)

F()

**VAL-INH-MBR-OVRD**: is valid if and only if

1. TBD every is valid
2. MemberVisibility: “**{**” [**this**| [UnitName](#UnitName) {“**,**” [UnitName](#UnitName)} ] “**}**”

S(“{“)

F()

**VAL-MBR-VSBL**: is valid if and only if list of [UnitName](#UnitName)s is provided then all [UnitName](#UnitName)s are valid and unique within the list.

1. MemberDeclaration: [[MemberVisibility](#MemberVisibility)] ([**override**] [**final**] [UnitAttribiteDeclaration](#UnitAttributeDeclaration)|[UnitRoutineDeclaration](#UnitRoutineDeclaration)) | [InitDeclaration](#InitDeclaration)

S(S([MemberVisibility](#MemberVisibility)), override, final, S([UnitAttribiteDeclaration](#UnitAttributeDeclaration)), S([UnitRoutineDeclaration](#UnitRoutineDeclaration)), S([InitDeclaration](#InitDeclaration)))

F()

**VAL-MBR-DCL**: is valid if and only if

1. [MemberVisibility](#MemberVisibility) is valid if provided
2. [UnitAttribiteDeclaration](#UnitAttributeDeclaration) or [UnitRoutineDeclaration](#UnitRoutineDeclaration) or [InitDeclaration](#InitDeclaration) is valid
3. InitDeclaration: **init** [[UnitRoutineParameters](#UnitRoutineParameters)] [[EnclosedUseDirective](#EnclosedUseDirective)] [[RequireBlock](#RequireBlock)]

[“**:**” (**init** [[Arguments](#Arguments)]) | ([ParentInitCall](#ParentInitCall) {“**,**” [ParentInitCall](#ParentInitCall)} [“**,**” **init** [[Arguments](#Arguments)]])]

([InnerBlock](#InnerBlock) [[EnsureBlock](#EnsureBlock)] BlockEnd)|(**foreign**|**none** [[EnsureBlock](#EnsureBlock) BlockEnd])

S(new)

F()

**VAL-INIT-DCL**: is valid if and only if

1. TBD every is valid
2. ParentInitCall: [UnitTypeName](#UnitTypeName) [[Arguments](#Arguments)]

S(init, S([UnitTypeName](#UnitTypeName)))

F()

**VAL-INIT-CALL**: is valid if and only if

1. TBD every is valid
2. UnitRoutineDeclaration: [**pure**|**safe**] [RoutineName](#RoutineName) [**final** [Identifier](#Identifier)] [[UnitRoutineParameters](#UnitRoutineParameters)] [[ReturnType](#ReturnType) [Type](#Type)] [[EnclosedUseDirective](#EnclosedUseDirective)][[RequireBlock](#RequireBlock)](([InnerBlock](#InnerBlock)) [[EnsureBlock](#EnsureBlock)] BlockEnd) | ((**abstract**|**foreign**|**none**|**old**| (“**=>**”[Expression](#Expression)))[[EnsureBlock](#EnsureBlock) BlockEnd])

S(pure, safe, <ident>, “()”, “:=”)

F()

**VAL-UNT-RTN-DCL**: is valid if and only if

1. TBD every is valid
2. RoutineName: [Identifier](#Identifier) |“**()**”|“**:=**”|([OperatorName](#OperatorName) [[AliasName](#AliasName)])

S(<ident>, “()”, “:=”, S([OperatorName](#OperatorName)))

F()

**VAL-RTN-NM**: is valid if and only if

1. TBD every is valid
2. AliasName: **alias** ([Identifier](#Identifier)|“*and then*”|“*or else*” )

S(alias)

F()

**VAL-ALS-NM**: is valid if and only if

1. TBD every is valid
2. OperatorName : [OperatorSign](#OperatorSign) [[OperatorSign](#OperatorSign)]

S([OperatorSign](#OperatorSign))

F()

**VAL-OPR-NM**: is valid if and only if

1. TBD every is valid
2. OperatorSign : “**^**” | “**\***” | “**/**” | “**\**” | “**=**” | “**+**” | “**-**“ |”**<**” | ”**>**” | ”**&**” | “**|**”|“#” | “%”| “@”| “!”| “$”| “~”

S(“**^**”, “**\***”, “**/**”, “**\**”, “**=**”, “**+**”, “**-**“, ”**<**”, ”**>**”, ”**&**”, “**|**”, “#”, “%”, “@”, “!”, “$”, “~”)

F()

**VAL-OPR-SGN**: is always valid

1. ConstObjectsDeclaration: **const** “:”[ [ConstObject](#ConstObject) { “**,**” [ConstObject](#ConstObject)} ] BlockEnd

S(const)

F()

**VAL-CNST-OBJS-DCL**: is valid if and only if

1. every [ConstObject](#ConstObject) is valid
2. all [ConstObject](#ConstObject) are unique within the [ConstObjectsDeclaration](#ConstObjectsDeclaration) and surrounding UnitDeclaration
3. ConstObject:

(

( [Constant](#Constant) | ([Idenitifer](#Identifier) [ [Arguments](#Arguments) ]) )

[ [“{”[OperatorName](#OperatorName) [ConstantExpression](#ConstantExpression) “}”] “**..**” ([Constant](#Constant) | ([Idenitifer](#Identifier) [ [Arguments](#Arguments) ])) ]

)

|

(“{” [RegularExpression](#RegularExpression) “}” [IntegerConstant](#IntegerConstant) [“+”])

S(S([Constant](#Constant)), <ident>, “{”)

F()

**VAL-CNST-OBJ-DCL**: is valid if and only if

1. TBD every is valid
2. RegularExpression: [Constant](#Constant)1 ({“**|**”[Constant](#Constant)2}) | (“**|**””**..**” [Constant](#Constant)n)

S(S([Constant](#Constant)))

F()

**VAL-RGL-EXPR**: is valid if and only if

1. every [Constant](#Constant) is valid
2. TBD
3. Statement: [Assignment](#Assignment) | [LocalAttributeDeclaration](#LocalAttributeCreation)| [WritableCall](#WritableCall) | [ObjectCreation](#ObjectCreation) | [Conditional](#Conditional)| [Loop](#Loop) | [Detach](#Detach)|[Return](#Return) |[HyperBlock](#HyperBlock)| [Raise](#Raise) | [UnpackTuple](#UnpackTuple)

S(S([Assignment](#Assignment)), S([LocalAttributeDeclaration](#LocalAttributeCreation)), S([WritableCall](#WritableCall)), S([ObjectCreation](#ObjectCreation)), S([Conditional](#Conditional)), S([Loop](#Loop)), S([Detach](#Detach)), S([Return](#Return)), S([HyperBlock](#HyperBlock)), S([Raise](#Raise)), S([UnpackTuple](#UnpackTuple)))

F()

**VAL-STMT**: is valid if and only if one of the following <[Assignment](#Assignment)> or <[LocalAttributeDeclaration](#LocalAttributeDeclaration)> or <[WritableCall](#WritableCall)> or <[Conditional](#Conditional)> or <[Loop](#Loop)> or <[Detach](#Detach)> or <[ObjectCreation](#ObjectCreation)> or <[Return](#Return)> or <[UnpackTuple](#UnpackTuple)> or <[HyperBlock](#HyperBlock)> or <[Raise](#Raise)> is valid

1. Detach: **?** [Identifier](#Identifier) [NewLine]

S(“?”)

F()

**VAL-DTCH**: is valid if and only if

1. [Identifier](#Identifier) is valid
2. [Identifier](#Identifier) is declared, visible, and writable entity
3. Raise:[**active**] **raise** [[Expression](#Expression)] [NewLine]

S(active, raise)

F()

**VAL-RS**: is valid if and only if

1. [Expression](#Expression) is valid if provided
2. If no [Expression](#Expression) provided then Raise textually belongs to whenClasues or Else section
3. Return: **return** [[Expression](#Expression)] [NewLine]

S(return)

F()

**VAL-RTRN**: is valid if and only if

1. if Expression is provided and then valid and <[Return](#Return)> is in the body of the function and type of the Expression conforms to the type of the function.
2. HyperBlock: [[RequireBlock](#RequireBlock)] [InnerBlock](#InnerBlock) [[EnsureBlock](#EnsureBlock)] BlockEnd

S(S([RequireBlock](#RequireBlock)), S([InnerBlock](#InnerBlock)))

F()

**VAL-HPR-BLK**: is valid if and only if

1. [RequireBlock](#RequireBlock) is valid if provided
2. [InnerBlock](#InnerBlock) is valid
3. [EnsureBlock](#EnsureBlock) is valid if provided
4. Assignment: [Writable](#Writable) “**:=**” [Expression](#Expression) [NewLine]

S(S([Writable](#Writable)))

F()

**VAL-ASGN**: is valid if and only if

1. <[Writable](#Writable)> and <[Expression](#Expression)> are both valid
2. type of <[Expression](#Expression)> conforms to or converts into the type of <[Writable](#Writable)>
3. Writable: [WritableCall](#WritableCall) | (“**(**”[WritableCall](#WritableCall) {“**,**” [WritableCall](#WritableCall) } “**)**”)

S(S([WritableCall](#WritableCall)), “(”)

F()

Examples: (a.x, b(x).y.z, c) := (E1, E2, E3) a := expr a.b.c := expr foo(…).y := expr

**VAL-WRTBL**: is valid if and only if

1. TBD every is valid
2. UnpackTuple: “**(**“ [LocalAttributeNamesList](#LocalAttributeNamesList) “**)**” **is** [Expression](#Expression)

S(“(”)

F()

**VAL-UNPK-TPL**: is valid if and only if

1. [LocalAttributeNamesList](#LocalAttributeNamesList) is valid
2. [Expression](#Expression) is valid
3. LocalAttributeDeclaration:

[LocalAttributeNamesList](#LocalAttributeNamesList) ([“**:**” [Type](#Type)] **is** [Expression](#Expression))|(“**:**” [Type](#Type)) [**final** “**=>**” [Statement](#Statement)]

[NewLine]

S(S([LocalAttributeNamesList](#LocalAttributeNamesList)))

F()

**VAL-LCL-ATTR-DCL**: is valid if and only if

1. TBD every is valid
2. LocalAttributeNamesList: [**var**|**rigid**] [Identifier](#Identifier) {“**,**”[**var**|**rigid**] [Identifier](#Identifier)}

S(var, rigid, <ident>)

F()

**VAL-LCL-ATTRS-LST**: is valid if and only if

1. TBD every is valid
2. UnitAttributeDeclaration: **ПЕРЕДЕЛЫВАТЬ !!!! ЗАЧЕМ ????**

(( [UnitAttributeNamesList](#UnitAttributeNamesList) “:” [Type](#Type)) | ( [Identifier](#Identifier) [“:” [AttachedType](#AttachedType)] **is** [ConstantExpression](#ConstantExpression) [NewLine]) | ( **const**|**rigid** [Identifier](#Identifier) [“:” [AttachedType](#AttachedType)] **is** [ConstantExpression](#ConstantExpression) [NewLine] {“**,**”[Identifier](#Identifier) [“:” [AttachedType](#AttachedType)] **is** [ConstantExpression](#ConstantExpression) [NewLine]} ) | ([Identifier](#Identifier) “:” [Type](#Type) **rtn** “:=” [[[[[UnitRoutineParameters](#UnitRoutineParameters)] [HyperBlock](#HyperBlock)](#TupleExpression)](#OldExpression)]) ) [**final =>** [Statement](#Statement)[NewLine]]

S()

F()

**VAL-UNT-ATTR-DCL**: is valid if and only if

1. TBD every is valid
2. UnitAttributeNamesList: [**const** | **rigid**] [Identifier](#Identifier) {“**,**”[**const** | **rigid**] [Identifier](#Identifier)}

S(const, rigid, <ident>)

F()

**VAL-UNT-ATTRS-LST**: is valid if and only if

1. TBD every is valid
2. BooleanExpression: [Expression](#Expression)

S([Expression](#Expression))

F()

**VAL-BOOL-EXPR**: is valid if and only if

1. [Expression](#Expression) is valid
2. [Expression](#Expression) has type Boolean
3. ConstantExpression: ([Identifier](#Identifier) {“**.**” [Identifier](#Identifier)}) | [Constant](#Constant) [[Operator](#Operator) [ConstantExpression](#ConstantExpression)]

S(<ident>, S([Constant](#Constant)))

F()

**VAL-CNST-EXPR**: is valid if and only if

1. TBD every is valid
2. Expression: [[ForcedType](#ForcedType)] [IfExpression](#IfExpession) |[WritableCall](#WritableCall)| [NewExpression](#NewExpression) | [Expression](#Expression) [Operator](#Operator) [Expression](#Expression) | [Operator](#Operator) [Expression](#Expression) | [Constant | [TypeOfExpression](#TypeOfExpression) | [OldExpression](#OldExpression)](#Constant)| [RangeExpression |](#RangeExpression) [LambdaExpression](#LambdaExpression) | [TupleExpression |](#TupleExpression) [RefExpression](#RefExpression)| “**(**”[Expression](#Expression)“**)**”{[CallChain](#CallChain)}

S()

F()

**VAL-EXPR**: is valid if and only if

1. TBD every is valid
2. [RefExpression:](#TupleExpression) **ref** [Expression](#Expression)

S(ref)

F()

**VAL-REF-EXPR**: is valid if and only if

1. [Expression](#Expression) is valid
2. TBD
3. LambdaExpression: (**rtn** [Identifier](#Identifier) [[Signature](#Signature)])|[InlineLambdaExpression](#InlineLambdaExpression)

S(rtn, S([InlineLambdaExpression](#InlineLambdaExpression)))

F()

**VAL-RTN-EXPR**: is valid if and only if

1. TBD every is valid
2. InlineLambdaExpression: [[[**pure**|**safe**] **rtn** [[StandaloneRoutineParameters](#StandaloneRoutineParameters)] [[ReturnType](#ReturnType) [Type](#Type)] ([[RequireBlock](#RequireBlock)] ([InnerBlock](#InnerBlock) BlockEnd)|(**foreign** [[EnsureBlock](#EnsureBlock)] BlockEnd])|(“**=>**”[Expression](#Expression))](#EnsureBlock)](#OldExpression) S(pure, safe, rtn)

F()

**VAL-LMBD-EXPR**: is valid if and only if

1. TBD every is valid
2. RangeExpression: [Expression](#Expression) [“{” [OperatorName](#OperatorName) [ConstantExpression](#ConstantExpression)“}”] “**..**” [Expression](#Expression)

S(S([Expression](#Expression)))

F()

**VAL-RNG-EXPR**: is valid if and only if

1. TBD every is valid
2. OldExpression: [**old** [Expression](#Expression)](#TupleExpression)

S(old)

F()

**VAL-OLD-EXPR**: is valid if and only if

1. [Expression](#Expression) is valid
2. TBD
3. TupleExpression: “**(**”[[TupleElement](#TupleElement) {“**,**” [TupleElement](#TupleElement)}]“**)**”

S(“(”)

F()

**VAL-TPL-EXPR**: is valid if and only if

1. Every [TupleElement](#TupleElement) is valid
2. TupleElement: [Expression](#Expression)| [Parameter](#Parameter)

S(S([Expression](#Expression)), S([Parameter](#Parameter)))

F()

**VAL-TPL-ELMNT**: is valid if and only if

1. [Expression](#Expression) or [Parameter](#Parameter) is valid
2. TBD
3. TypeOfExpression: [Expression](#Expression) **is** (**“?”**| [UnitType](#UnitType) | [AnonymousUnitType](#AnonymousUnitType))

S(S([Expression](#Expression)))

F()

/\* Duck typing may work only when no preconditions and postconditions are for all routines of the [AnonymousUnitType](#AnonymousUnitType)) \*/

**VAL-IS-EXPR**: is valid if and only if

1. TBD every is valid
2. Operator: [OperatorName](#OperatorName)|**in**

S(S([OperatorName](#OperatorName)), “in”)

F()

**VAL-OPRTR**: is valid if and only if

1. [OperatorName](#OperatorName) is valid if provided
2. Constant: [[UnitTypeName](#UnitTypeName) “.”]( [StringConstant |](#StringConstant) [CharacterConstant |](#CharacterConstant) [IntegerConstant |](#IntegerConstant) [RealConstant |](#RealConstant) [BooleanConstant](#BooleanConstant) | [BitConstant](#BitConstant) | [Identifier](#Identifier) )

S(“, ‘, <digit>, <ident>)

F()

**VAL-CNST**: is valid if and only if

1. TBD every is valid
2. IfExpression:

**if** [Expression](#Expression)1 (**case** [ExpressionAlternatives](#ExpressionAlternatives))|( BlockStart[Expression](#Expression)2)  
{**elsif** [Expression](#Expression)3 (**case** [ExpressionAlternatives](#ExpressionAlternatives))|( BlockStart[Expression](#Expression)4)}  
**else** [Expression](#Expression)5 “}”**Cmod**  
S(if)

F()

**VAL-IF-EXPR**: is valid if and only if

1. TBD every is valid
2. ExpressionAlternatives: [AlternativeTags](#AlternativeTags) [Expression](#Expression) { **case** [AlternativeTags](#AlternativeTags) [Expression](#Expression)}

S(S([AlternativeTags](#AlternativeTags)))

F()

**VAL-IF-ALTS**: is valid if and only if

1. TBD every is valid
2. WritableCall:

((([Identifier](#Identifier)[FactualGenerics])|[UnitTypeName](#UnitTypeName)|**return**|**this**) [“.”([Identifier](#Identifier)|[OperatorName](#OperatorName))])

|(**old** [[ForcedType](#ForcedType)]) //Precursor call – ForcedType resolves Parent to call precursor from

[[Arguments](#Arguments)] {[CallChain](#CallChain)}

S(<ident>, return, this, old)

F()

**VAL-WRTBL-CALL**: is valid if and only if

1. TBD every is valid
2. ObjectCreation: **new** [[ForcedType](#ForcedType)] **return**|[Identifier](#Identifier) [[Arguments](#Arguments)]

S(new)

F()

**VAL-OBJ-CRTN**: is valid if and only if

1. TBD every is valid
2. NewExpression: [**new**] [UnitType](#UnitType) [[Arguments](#Arguments)]

S(new, S([UnitType](#UnitType)))

F()

**VAL-NEW-EXPR**: is valid if and only if

1. [UnitType](#UnitType) is valid
2. [Arguments](#Arguments) are valid if provided
3. If [UnitType](#UnitType) has no initialization procedure or has initialization procedure with no parameters then no or empty [Arguments](#Arguments) are provided
4. If [UnitType](#UnitType) has no initialization procedure with non-empty parameters then TBD
5. TBD
6. CallChain: “**.**”([Identifier](#Identifier)|[OperatorName](#OperatorName)) [[Arguments](#Arguments)]

S(“.”)

F()

**VAL-CALL-CHN**: is valid if and only if

1. TBD every is valid
2. Arguments: “**(**” [[ExpressionList](#ExpressionList)] ”**)**”

S(“(”)

F()

**VAL-ARGS**: is valid if and only if

1. [ExpressionList](#ExpressionList) is valid if provided
2. TBD
3. ForcedType: GroupStart [UnitType](#UnitType) GroupEnd

S(“{”)

F()

**VAL-FRCD-TYP**: is valid if and only if

1. [UnitType](#UnitType) is valid
2. ExpressionList: [Expression](#Expression) {“**,**” [Expression](#Expression)}

S(S([Expression](#Expression)))

F()

* 1. ~~ExpressionList: [[ForcedType](#ForcedType)]~~ [~~Expression~~](#Expression) ~~{“~~**~~,~~**~~” [[ForcedType](#ForcedType)]~~ [~~Expression~~](#Expression)~~}~~

**VAL-EXPR-LST**: is valid if and only if

1. every [Expression](#Expression) is valid
2. Conditional:

**if** [Expression](#Expression) (**case** [Alternatives](#IfBody))|(BlockStart[StatementsList](#StatementsList))

{**elsif** [Expression](#Expression) (**case** [Alternatives](#IfBody))|(BlockStart[StatementsList](#StatementsList))}

[**else** [StatementsList](#StatementsList)]  
BlockEnd

S(if)

F()

**VAL-IF**: is valid if and only if

1. [Expression](#Expression) (one or two) is valid and has type Boolean
2. All [StatementsList](#StatementsList)s are valid
3. TBD
4. Alternatives:

[AlternativeTags](#AlternativeTags) [StatementsList](#StatementsList) {**case** [AlternativeTags](#AlternativeTags) [StatementsList](#StatementsList)}

S(S([AlternativeTags](#AlternativeTags)))

F()

**VAL-IF-ALTS**: is valid if and only if

1. TBD every is valid
2. AlternativeTags: [AlternativeTag](#AlternativeTag) {“**,**” [AlternativeTag](#AlternativeTag)}

S(S([AlternativeTag](#AlternativeTag)))

F()

**VAL-ALT-TGS**: is valid if and only if

1. TBD every is valid
2. AlternativeTag: [Expression](#Expression) [[GroupStart [OperatorName](#OperatorName) [ConstantExpression](#ConstantExpression) GroupEnd] “**..**”[Expression](#Expression)]

S(S([Expression](#Expression)))

F()

**VAL-ALT-TG**: is valid if and only if

1. TBD every is valid
2. MemberDescription: ([**rtn**] [RoutineName](#RoutineName)[[Signature](#Signature)])| ([Idenitifer](#Identifier){“,”[Idenitifer](#Identifier)} ”**:**” [UnitType](#UnitType)) S(rtn, S([RoutineName](#RoutineName)), <ident>)

F()

**VAL-MBR-DSC**: is valid if and only if

1. TBD every is valid
2. Loop:

(**while** [BooleanExpression](#BooleanExpression){“,”[BooleanExpression](#BooleanExpression)} [[RequireBlock](#RequireBlock)] [InnerBlock](#InnerBlock)) | ([[RequireBlock](#RequireBlock)] [InnerBlock](#InnerBlock) **while** [BooleanExpression](#BooleanExpression){“,”[BooleanExpression](#BooleanExpression)})

[[EnsureBlock](#EnsureBlock)] BlockEnd

S(while, S([RequireBlock](#RequireBlock)), S([InnerBlock](#InnerBlock)))

F()

**VAL-LOOP**: is valid if and only if

1. VAL010\_Loop (LV): <[Loop](#Loop)> is valid if and only if it has no while or only one while clause and …
2. Type: [”**?**”] [AttachedType](#AttachedType)

S(“?”, S([AttachedType](#AttachedType)))

F()

**VAL-TYPE**: is valid if and only if

1. TBD every is valid
2. AttachedType: [UnitType](#UnitType)|[AnchorType](#AnchorType)|[UnionType](#UnionType)|[TupleType](#TupleType)|[RangeType](#RangeType)|[RoutineType](#RoutineType)|[AnonymousUnitType](#AnonymousUnitType)

S(S([UnitType](#UnitType)), S([AnchorType](#AnchorType)), S([UnionType](#UnionType)), S([TupleType](#TupleType)), S([RangeType](#RangeType)), S([RoutineType](#RoutineType)), S([AnonymousUnitType](#AnonymousUnitType)))

F()

**VAL-ATCH-TYP**: is valid if and only if [UnitType](#UnitType)| or [AnchorType](#AnchorType) or [UnionType](#UnionType) or [TupleType](#TupleType) or [RangeType](#RangeType) or [RoutineType](#RoutineType) or [AnonymousUnitType](#AnonymousUnitType) is valid

1. AnonymousUnitType: **unit** [MemberDesciption](#memberDescription) {[“;”] [MemberDesciption](#memberDescription)} BlockEnd

S(unit)

F()

**VAL-ANM-TYP**: is valid if and only if

1. every [MemberDesciption](#memberDescription) is valid
2. all [MemberDesciption](#memberDescription)s are unique within the AnonymousUnitType
3. RoutineType: **rtn** [[Signature](#Signature)]

S(rtn)

F()

**VAL-RTN-TYP**: is valid if and only if

1. [Signature](#Signature) is valid if provided
2. Signature: (“**(**”[[Type](#Type) {“**,**” [Type](#Type)}]“**)**”[ [ReturnType](#ReturnType) [Type](#Type)])| ([ReturnType](#ReturnType) [Type](#Type))

S(“(”, S([ReturnType](#ReturnType)))

F()

**VAL-SGN**: is valid if and only if

1. every [Type](#Type) is valid
2. RangeType: [RangeTypeItem](#RangeTypeItem) {“**,**” [RangeTypeItem](#RangeTypeItem)}

**VAL-RNG-TYP**: is valid if and only if

1. every [RangeTypeItem](#RangeTypeItem) is valid
2. TBD
3. RangeTypeItem: [ConstantExpression](#ConstantExpression) [[GroupStart[OperatorName](#OperatorName) [ConstantExpression](#ConstantExpression) GroupEnd] “**..**” [ConstantExpression](#ConstantExpression) ]

S(S([ConstantExpression](#ConstantExpression)))

F()

**VAL-RNG-TYPE-ITM**: is valid if and only if

1. every [ConstantExpression](#ConstantExpression) is valid
2. TBD
3. AnchorType: **as** (**this**|([Identifier](#Identifier) [[Signature](#Signature)]))

S(as)

F()

**VAL-ANCHR-TYP**: is valid if and only if

1. TBD every is valid
2. UnionType: [UnitType](#UnitType) {“**|**”[UnitType](#UnitType)}

S(S([UnitType](#UnitType)))

F()

**VAL-MLT-TYP**: is valid if and only if

1. every [UnitType](#UnitType) is valid
2. all [UnitType](#UnitType)s are unique within the [UnionType](#UnitType)
3. TBD
4. TupleType: “**(**”[[TupleField](#TupleField) {“**,**”|”**;**” [TupleField](#TupleField)}]“**)**”

S(“(”)

F()

**VAL-TPL-TYP**: is valid if and only if

1. every [TupleField](#TupleField) is valid
2. TupleField: [[Identifier](#Identifier) {“**,**” [Identifier](#Identifier)}“**:**”] [UnitType](#UnitType)

S(<ident>, S([UnitType](#UnitType)))

F()

**VAL-TPL-FLD**: is valid if and only if

1. every [Identifier](#Identifier) is valid if provided
2. [UnitType](#UnitType) is valid
3. UnitTypeName: {[Identifier](#Identifier)“**.**”} [UnitName](#UnitName) [GenericsStart([Type](#Type)|[ConstantExpression](#ConstantExpression)) {“**,**” ([Type](#Type)|[ConstantExpression](#ConstantExpression))} GenericsEnd ]

S(<ident>)

F()

**VAL-UNT-TYPE\_NM**: is valid if and only if

1. TBD every is valid
2. UnitType: [**ref**|**val**|**active**] [UnitTypeName](#UnitTypeName)

S(ref, val, active, <ident>)

F()

**VAL-UNT-TYP**: is valid if and only if

1. [UnitTypeName](#UnitTypeName) is valid
2. UnitName: [UpperCaseLetter](#UpperCaseLetter) { [Letter](#Letter) | [Digit](#Digit) | ’\_’ }

S(<ident>)

**VAL-UNT-NM**: is valid if and only if

1. TBD every is valid
2. DocumentingComment: “**///**” { [Character](#Character) }

**VAL-DOC-CMNT**: is valid if and only if

1. TBD every is valid
2. Comment: (“**//**” {[Character](#Character)}) | (”**/\***” { [Character](#Character) } “**\*/**”)

**VAL-CMNT**: is valid if and only if

1. TBD every is valid
2. ReturnType: “**:**”|”**->**”
3. BlockEnd: **end**|”**}**”**Cmod**
4. BlockStart: **do**|”**{**”**Cmod**
5. GenericsStart: “**[**“|”**<**”**Cmod**
6. GenericsEnd: “**]**”|”**>**”**Cmod**
7. GroupStart: “**{**”|”**[**“**Cmod**
8. GroupEnd: “**}**”|”**]**“**Cmod**
9. NewLine: “**;**”|”\n”
10. Identifier: [LowerCaseLetter](#LowerCaseLetter) { [Letter](#Letter) | [Digit](#Digit) | ’\_’ }
11. StringConstant: “**”**” { [Character](#Character) } “**”**”
12. CharacterConstant: “**’**” [Character](#Character) “**’**”
13. IntegerConstant: [ “**+**”|”**-**“ ] ([Digit](#Digit) { [Digit](#Digit) } [”**H**”|”**h**”|“**O**”|“**o**”])|(“0” (“**x**”|“**X**”|“**o**”|“**O**”) [Digit](#Digit) { [Digit](#Digit) })
14. BitConstant: [ “**+**”|”**-**“ ] (“0”|”1“ {“0”|”1“} [“**B**”|”**b**”]) | (“0” ( “**b**”|“**B**”) “0”|”1“{“0”|”1“})
15. RealConstant: [ “**+**”|”**-**“ ] [Digit](#Digit) { [Digit](#Digit) } “.”{ [Digit](#Digit) } [“**e**”|”**E**”] [“**+**”|”**-**“] [Digit](#Digit) { [Digit](#Digit) }
16. Character: [Letter](#Letter) | [Digit](#Digit) | [Symbol](#Symbol) | UnicodeSymbol | ControlCharacter
17. Letter : ‘**A**’ | .. ’**Z**’ | ’**a**’ | ..’**z**’
18. UpperCaseLetter: ‘**A**’ | .. ’**Z**’
19. LowerCaseLetter: ’**a**’ | ..’**z**’
20. Digit: ’**0**’ | ..’**9**’ | ’**A**’..’**F**’
21. Symbol: ‘\ASCII symbol code 0..255’
22. UnicodeSymbol: ‘\u’ | ‘\U’ …
23. ControlCharacter: ‘\n’ | ‘\t’
24. **SLang semantics: list of all behavioral patterns**

SEM002\_UnitRoutineDeclaration: **final** [Identifier](#Identifier) allows calling this version from any descendant unit

SEM003\_AnonymousRoutine: Identical to [SEM004\_StatementsList](#SEM004_StatementsList)

SEM004\_StatementsList: { [Statement](#Statement)[“**;**”]} All statements of the list are being executed by the processing element one by one according to [SEM006\_Statement](#SEM006_Statement) unless some may lead to an exception or leave the sequence (return)

SEM005\_InnerBlock: if the list of identifiers “{” [Identifier](#Identifier) {“,” [Identifier](#Identifier)} “}” is provided then for these identifiers calls invariants are not checked within the block. StatemntList is executed according to [SEM004\_StatementsList](#SEM004_StatementsList), if when clauses are provided and execution of StatemntList leads to some exception the check if this exception can be handled by one of when clauses is performed if such intercepting clause is found then when clause body is executed according to [SEM005\_WhenClause](#SEM005_WhenClause) otherwise if the else part is in place it is executed according to [SEM004\_StatementsList](#SEM004_StatementsList) otherwise if no else part present then exception block execution failure exception is raised.

**do** [”**:**”[Label](#Label)][“{”[Identifier](#Identifier) {“,” [Identifier](#Identifier)} “}”]

[StatementsList](#StatementsList)

[ [WhenClause](#WhenClause) {[WhenClause](#WhenClause)}

[**else** [[StatementsList](#StatementsList)]]]

SEM005\_WhenClause: **when** [[Identifier](#Identifier)**:**][UnitType](#UnitType) **do** [StatementsList](#StatementsList) if the type of exception conforms to the type of the when clause ([UnitType](#UnitType)) then do part is being executed according to [SEM004\_StatementsList](#SEM004_StatementsList) and exception is treated as handled. If the identifier is provided then the current exception object is available in the body of when clause handler using the identifier name

Parameters: “**(**”[“**:=**”][Parameter](#Parameter){”**;**” [Parameter](#Parameter)}“**)**”

Parameter: ([[**var**] [Identifier](#Identifier){“**,**” [**var**] [Identifier](#Identifier)} “**:**” [Type](#Type))|([Identifier](#Identifier) “**is**” [Expression](#Expression)|(“**as**” [Identifier](#Identifier)))

SEM031\_RequireBlock: **require** [PredicatesList](#PredicatesList) this clause is evaluated before any routine call according to [SEM034\_PredicatesList](#SEM034_PredicatesList) and if some predicate is evaluated to false exception object will be of type precondition violation

SEM032\_EnsureBlock: **ensure** [PredicatesList](#PredicatesList) this clause is evaluated after any successful routine call according to [SEM034\_PredicatesList](#SEM034_PredicatesList) and if some predicate is evaluated to false exception object will be of type postcondition violation

SEM033\_InvariantBlock: **require** [PredicatesList](#PredicatesList) this clause is evaluated after any successful routine call and then after any successful execution of [SEM032\_EnsureBlock](#SEM032_EnsureBlock) if present according to [SEM034\_PredicatesList](#SEM034_PredicatesList) and if some predicate is evaluated to false exception object will be of type unit invariant violation

SEM034\_PredicatesList: [[Predicate](#Predicate) {[”**;**”] [Predicate](#Predicate)}] each predicate of the list will be evaluated according to [SEM035\_Predicate](#SEM035_Predicate) until the first one which raises an exception. If all predicates were evaluated as true then execution continues

SEM006\_Statement: its execution leads to the execution of one of the particular statements below

[Assignment](#Assignment)| [LocalAttributeDeclaration](#LocalAttributeDeclaration)| [WritableCall](#WritableCall)| [ObjectCreation](#ObjectCreation)| [Conditional](#Conditional)| [Loop](#Loop)| [Break](#Break) | [Detach](#Detach)|[Return](#Return)|[HyperBlock](#HyperBlock)| [Raise](#Raise)

1. **SLang tutorial: examples**

IO.put (“Hello, world!\n”) // That is the simplest program where

// IO is the name of the unit (class or module => compilation unit)

// put is the name of the routine (function) being called

// “Hello, world!\n” is the string passed as an argument to put

/\* All units and types are started with capital letter, while all other identifiers with the lower case one \*/

**use** A **as** B // unit A has got a new, extra name (alias), name A is also valid

B.foo() /\* call foo() from singleton(module) object B which is available across the whole program \*/

b1 **is** B /\* local variable b is declared and initialized with new object of type B \*/

b2 **is** **new** B // one may use keyword **new** for better clarity

b3 **is** **new** B() /\* if B has no initialization procedure (constructor) or one with no parameters – no need to use () \*/

b1 **:=** b2

b2 **:=** b3 /\* **:=** stands for assignment, **is** stands for the declarative initialization \*/

b2 **is** b1 // That is the new b2 potentially with the different type

/\* So, the program may just start as a sequence of statements – in fact that is anonymous routine, which is in fact the program entry point. One may declare standalone routines (functions) which have names as well as units – named types declaration with all fields (attributes) and operations (unit routines). Standalone routines can also be a program entry point or unit initialization procedure can serve as an entry point too. \*/

// Here comes the units

**unit** C

f0: A

f1: B

**end**

**unit** D **{** /\* If one is fond of curly brackets – fine. But files should have different extensions – .slang for **do**-**end** style and .clang for **{}** one. One source file has only one syntax style.\*/

f0: A

f1: B

**}**

functor **is rtn** B.foo // type of functor variable is a routine (function) type

functor /\* is the same as B.foo – call the routine attached to functor variable \*/

/\* Note: anonymous routine statements may be mixed with standalone routines and units declarations \*/

**unit** D

f1**:** A

**init** (**:=** f1) **do** // **:=** f1 performs assignment of f1 with the argument

f1 **:= new** A // the body changes f1 into new value

**end**

**end**

**unit** E

f1**:** A

**init** (**:=** f1) **none** /\* **init** has no body, except the assignment into f1 generated by the compiler \*/

method1 **none** // method with an empty body

method2(**:=** f1) **none** // that is in fact setter for f1

**end**

/\* All local variables (attributes) are by default assigned-once while unit variables (attributes) are mutable \*/

local\_attribute **is** 5

local\_attribute **:=** 6 // CTE – it is read-only after initialization

**var** mutable\_local **is** 5

mutable\_local **:=** 6 // OK

**unit** F

unit\_attribute **is** 5

method **do**

unit\_attribute **:=** 6 // OK

readonly\_attribute **:=** 6 /\* CTE – it is read-only after initialization \*/

**end**

**const** readonly\_attribute **is** 5

**end**

// attributes may be of the reference or value nature

ref\_attr **is ref** F

val\_attr **is val** F

/\* both ref\_attr and val\_attr are attached (connected, refer to) to different objects of the same type F, but ref\_attr exhibits reference semantics (copy reference) while the val\_attr has the value (copy the object) one. \*/

// Method call is the same

ref\_attr.method

val\_attr.method

foo1 (p: **ref** F) **do**

**end**

foo2 (p: **val** F) **do**

**end**

foo1(ref\_attr) // foo1 will work with the same object ref\_attr refers to

foo1(val\_attr) /\* foo1 will work with the copy of the object val\_attr refers to \*/

foo1(**ref** val\_attr) // foo1 will work with the same object val\_attr refers to

foo2(ref\_attr) /\* foo2 will work with the copy of the object ref\_attr refers to \*/

foo2(val\_attr) /\* foo2 will work with the copy of the object val\_attr refers to \*/

// More options

**ref unit** RefByDefault // By default all attributes will be of ref nature

**end**

**val unit** ValByDefault // By default all attributes will be of val nature

**end**

**unit** EmbeddedDefaultRef // By default all attributes will be of ref nature

**end**

x1 **is** RefByDefault // ref nature

x2 **is** ValByDefault // val nature

x3 **is** EmbeddedDefaultRef // ref nature

x4 **is** **val** RefByDefault // One may explicitly overrule defaults. Value!

x5 **is** **ref** ValByDefault // One may explicitly overrule defaults. Reference!

// Attributes may be always attached to some object (instance) or be detached. y1**: ?**F // Type of y1 is probably F, it is detached while being declared

y1 **:=** F // y1 is attached to new object of type F

y1.method /\* If compiler can guarantee that y1 is attached then such call is valid, otherwise it is CTE \*/

**if** y1 **is** F **do** /\* safety check (if y1 has type F1 or more precisely if y1 is attached to an object of type which is compatible with F) \*/

y1.method

**?**y1 // We may detach y1

y1.method // CTE as y1 is detached

**end**

**if** y1 **is ? do** // reverse check if y1 is detached

y1 **:=** F // y1 is attached to F object

y1.method // Ok to call

functor **is rtn** y1.method // functor captures y1

functor // and call can be applied later

**?**y1 // Thus functor becomes detached too

functor // CTE

**end**

// What is the inferred type of functor?

functor: **rtn** (**this**: **?**F) **is rtn** y1.method

/\* So, to be 100% type safe one has to check if this parameter is attached to perform the call \*/

**if** functor.**this** **is** F **do**

functor // call it safely

**end**

**unit** G **extend** F **{** // C-style

bar (p1: Integer, p2: Real, p3: String) **{ }**

boo (p1: ?Integer, p2: Real, p3: ?String) **{ }**

**}**

y2 **is** G

functor: **rtn** (**this**: **?**G, **as** G.bar) **is rtn** y2.bar

/\* **rtn** (**this**: **?**G, **as** G.bar) is the type of **rtn** y2.bar So, no need to repeat the signature, or one may fully rely on the type inference \*/

functor **is rtn** y2.bar

functor **is rtn** y2.boo (**?**, 3.1415, **?**) /\* here ‘functor’ has this and second argument captured (p2 parameter)\*/

functor (666,,”a string”) /\* to call ‘functor’ we need to provide only 1st and 3rd arguments \*/

functor (666,”a string”) /\* as here is no ambiguity – based on types of arguments compiler can pass arguments into proper parameters\*/

functor (p1 **:=** 666, p3 **:=** ”a string”) // the most explicit way

functor **is rtn** ?.boo (**?**, ?, **?**) /\* here the ‘functor’ has ‘this’ and all 3 arguments as open – not provided\*/

y2.functor (5, 3.14, “a string”) /\* type of y2 defines the types of expected argument types\*/

functor **is** y2.functor (?, ?, ?) /\*Fix the call target as y2, define new meaning of name ‘functor’ \*/

functor (5, 3.14, “a string”) // Call it with all arguments

stand\_alone\_function (): Integer **{**

**return** 666

**}**

x **is** stand\_alone\_function // that is a call not a routine object!

/\* So, we have used several statements already – call (qualified and non-qualified), declaration (**is**), conditional (**if**), assignment (**:=**), **return**, detach (**?**) and now let’s see more statements \*/

/\* Here comes the block – linear piece of code;

one entry – one exit on success and another one when failure – exception is propagated \*/

**require** // List of optional block preconditions

value > 0

**do**

x **is** value / 0

// Optional exception handling section

**when** ZeroDivision **do** // ZeroDivision is the type !

IO.put (“Zero divide occurred!!!”)

/\* ZeroDivision exception is treated as handled and postconditions will be checked \*/

**when** e: SomeOtherExceptionOccured **do**

// e is a variable, while SomeOtherExceptionOccured is its type

**raise** NewException /\* as new is optional – so, here the new object of type NewException is created \*/

**else**

**raise** // Re-raise the exception

**ensure** // List of optional block postconditions

value = **old** value /\* value after the end of successful block execution should be equal to the value before the block execution started \*/

**end** // do

// Scopes

**do**

variable **is** Type1

variable **is** Type2 /\* That is OK to introduce the new meaning of the same name as the same level of the block \*/

**do**

variable **is** Type3 /\* CTE – just to make sure that then nearest variable of the enclosing block is visible in the embedded one \*/

**end** // do

**end // do**

// variable does not exists here

**do**

variable **is** Type4 // OK - another block at the same level

**end** // do

/\* When some actions are to be performed several times we need a loop – while loop in two forms is available: check the loop continuation condition before the loop body or after \*/

**while** index **in** 1..10

**require** // List of optional loop invariant conditions

index > 0

**do**

IO.put (“Index: ”, index)

**end**

index **is** 1

**require** // List of optional loop invariant conditions

index > 0

**do**

IO.put (“Index: ”, index)

index++

**while** index <= 10 **end**

// If it is necessary to exit the loop use exception technique

**do**

**while** index **in** 1..10 **do**

**if** index = 6 **do raise** -1 **end**

**end**

**when** -1 **do** // loop was exited via raise

**end**

// the same technique can be used to exit any compound statement

/\* there is a more general form of the conditional statement which allows to handle all checks \*/

**if** expression1

**case** value1

// expression1 = value1 case

**case** value2

// expression1 = value2 case

**elsif** expression2

**case** value3

// expression2 = value3 case

**case** value4

// expression2 = value4 case

**else**

/\* expression1 /= value1 and expression1 /= value2

and

expression2 /= value3 and expression2 /= value4

case

\*/

**end**

// So, if-do-else-end loop is just a short form of

**if** boolean\_expression

**case** true

**case** false

**end**

// C-style will look like this

**if** boolean\_expression **{**

// Then part

**else**

// optional else part

**}**

**if** expression **{**

**case** value

// expression = value1 case

**else**

// optional else part

**}**

**while** expression **{**

**{** // a block with exception handling (try scheme)

some\_code

**when** e: T1 **{**

handling\_code1

**}**

**when** e: T2 **{**

handling\_code1

**}**

**}**

**}**