Concurrent Clean Syntax and Libraries

This document describes the syntax and libraries of the Clean 1.0 pre-release of march 1995. Although we have put a lot of effort in keeping it correct and up-to-date, there are bound to be some errors and inconsistencies between this description and the actual release on your system. We would appreciate it very much if you could help us removing them by reporting such flaws to us as soon as possible (email: clean@cs.kun.nl or email directly with rinus@cs.kun.nl or marko@cs.kun.nl). Of course, also suggestions for improvement are very welcome.

1 Concurrent Clean syntax

In this section the context-free syntax of Clean 1.0 is given. In Section 1.1 the construction of a Clean program out of definition and implementation modules is given. Hereafter the syntax for, respectively, defining functions (Section 1.2), graphs (Section 1.3,1.4), macro's (Section 1.5) and types (Section 1.6-8) is presented. Overloading is treated in Section 1.8. These sections have some production rules in common which are collected in Section 1.10.
In section 1.11 the lay-out structure is described. This includes the description of a lay-out rule which permits the omission of the semi-colon (';') which ends a definition and of the braces ('{' and '}') which are used to group a list of definitions. The description of the identifiers and literals is also given in that section.
The following notational conventions are used in the context-free syntax descriptions:

[notion]	means that the presence of notion is optional
{notion}	means that notion can occur zero or more times
{notion}+	means that notion occurs at least once
{notion}-*list*	means one or more occurrences of notion separated by comma's
terminals	are printed in **bold** chicago
terminals	that can be left out in lay-out mode are printed in outlined courier
symbols	are printed in *italic* and represent identifiers and literals

1.1 Clean program

```
CleanProgram        =  {Module}+
Module              =  DefModule
                    |  ImplModule
DefModule           =  DefModuleHdrDef {ImportDef} {DefModDef}
```

ImplModule	=	ImplModuleHdrDef {ImportDef} {ImplModDef}
DefModuleHdrDef	=	**definition module** *ModuleSymb* ;
	|	**system module** *ModuleSymb* ;
ImplModuleHdrDef	=	[**implementation**] **module** *ModuleSymb* ;
ImportDef	=	**import** {*ModuleSymb*}-*list* ;
	|	**from** *ModuleSymb* **import** {ImportSymbols}-*list* ;
ImportSymbols	=	*FunctionSymb*
	|	*SelectorVariable*
	|	*MacroSymb*
	|	*TypeSymb*
	|	*ConstructorSymb*
	|	*FieldSymb*
DefModDef	=	TypeDef
	|	AbstractTypeDef
	|	OverloadingDef
	|	FunctionTypeDef
	|	MacroDef
ImplModDef	=	TypeDef
	|	OverloadingDef
	|	FunctionDef
	|	GraphDef
	|	MacroDef

1.2 Function definition

FunctionDef	=	[FunctionTypeDef] DefOfFunction
DefOfFunction	=	{FunctionAltDef}+
FunctionAltDef	=	FunctionSymbol {Pattern}
		{[**|** Guard] **=**[**>**] FunctionBody}+
		[LocalFunctionAltDefs]
Pattern	=	[*Variable* **=:**] BrackPattern
BrackPattern	=	SimplePattern
	|	PatternVariable
	|	BasicValuePattern
	|	ListPattern
	|	TuplePattern
	|	RecordPattern
	|	ArrayPattern
	|	**(**GraphPattern**)**
GraphPattern	=	ConstructorSymbol {Pattern}
	|	GraphPattern *ConstructorSymb* GraphPattern
	|	Pattern
SimplePattern	=	ConstructorSymbol
PatternVariable	=	*Variable*
	|	**_**
BasicValuePattern	=	BasicValue
ListPattern	=	**[**[{LGraphPattern}-*list* [**:** GraphPattern]]**]**
LGraphPattern	=	GraphPattern
	|	*CharsDenot*
TuplePattern	=	**(**GraphPattern**,** {GraphPattern}-*list***)**
RecordPattern	=	**{**[*TypeSymb* **|**] {FieldSymbol [**=** GraphPattern]}-*list***}**
ArrayPattern	=	**{ :**{GraphPattern}-*list* **: }**

```
                          |   {:{ArrayIndex = GraphPattern}-list:}

Guard                     =   BooleanExpr

FunctionBody              =   {StrictLet}+
                              RootExpression ;
                              [LocalFunctionDefs]

StrictLet                 =   let! ⦃ {GraphDef}+ ⦄ in

RootExpression            =   GraphExpr

LocalFunctionDefs         =   [with] ⦃ {LocalDef}+ ⦄
LocalDef                  =   GraphDef
                          |   FunctionDef
LocalFunctionAltDefs      =   [where] ⦃ {LocalDef}+ ⦄
```

1.3 Graph definition

```
GraphDef                  =   Selector =[:]  GraphExpr ;
Selector                  =   BrackPattern
```

1.4 Graph expression

```
Graph                     =   [Process] GraphExpr

GraphExpr                 =   Application
                          |   CaseExpr
                          |   LambdaAbstr

Application               =   {BrackGraph}+
                          |   GraphExpr OperatorSymbol GraphExpr

BrackGraph                =   NodeSymbol
                          |   GraphVariable
                          |   BasicValue
                          |   List
                          |   Tuple
                          |   Record
                          |   RecordSelection
                          |   Array
                          |   ArraySelection
                          |   (GraphExpr)

GraphVariable             =   Variable
                          |   SelectorVariable

BasicValue                =   IntDenot
                          |   RealDenot
                          |   BoolDenot
                          |   CharDenot
                          |   StringDenot

List                      =   [[{LGraphExpr}-list [: GraphExpr]]]
                          |   [GraphExpr [,GraphExpr]..[GraphExpr]]
                          |   [GraphExpr \\ {Qualifier}-list]
LGraphExpr                =   GraphExpr
                          |   CharsDenot
Qualifier                 =   Generators {|  Guard}
Generators                =   {Generator}-list
                          |   Generator {& Generator}
```

Generator	=	Selector **<-** ListExpr
	|	Selector **<-:** ArrayExpr
Tuple	=	**(**GraphExpr**,** {GraphExpr}-*list***)**
Record	=	**{**[*TypeSymb* **|**][RecordExpr **&**][{FieldSymbol **=** GraphExpr}-*list*] **}**
RecordSelection	=	RecordExpr **.** [*TypeSymb* **.**]*FieldSymb*
Array	=	**{ :** {GraphExpr}-*list* **: }**
	|	**{ :** ArrayExpr **&** [{ArrayIndex **=** GraphExpr}-*list*] [**** {Qualifier}-*list*] **: }**
	|	**{ :** [ArrayExpr **&**] GraphExpr **** {Qualifier}-*list* **: }**
ArrayIndex	=	**[**{IntegerExpr}-*list***]**
ArraySelection	=	ArrayExpr **.** ArrayIndex
CaseExpr	=	**case** GraphExpr **of** ⦃ {CaseAltDef}+ ⦄
	|	**if** BrackGraph BrackGraph BrackGraph
CaseAltDef	=	[Pattern] {[**|** Guard] **->** FunctionBody}+ [LocalFunctionAltDefs]
LambdaAbstr	=	**** {Pattern}+ **->** GraphExpr
Process	=	**{* I *}**
	|	**{* P** [**at** ProcIdExpr] ***}**

1.5 Macro definition

MacroDef	=	*FunctionSymb* {*Variable*} **:==** GraphExpr ;
	|	**(***FunctionSymb***)** [Fix][Prec] ; **(***FunctionSymb***)** {*Variable*} **:==** GraphExpr ;

1.6 Type definition

TypeDef	=	AlgebraicTypeDef
	|	RecordTypeDef
	|	SynonymTypeDef
AlgebraicTypeDef	=	**::**TypeLhs **=** ConstructorDef {| ConstructorDef} ;
RecordTypeDef	=	**::**TypeLhs **=** **{**{FieldSymbol **::** [Strict]Type}-*list***}** ;
SynonymTypeDef	=	**::**TypeLhs **:==** Type ;
TypeLhs	=	[UnqTypeAttrib] *TypeSymb* {TypeVarDef}
UnqTypeAttrib	=	*****
	|	*UniqueTypeVariable***:**
	|	**.**
TypeVarDef	=	[**E.**] [UnqTypeAttrib] *TypeVariable*
ConstructorDef	=	*ConstructorSymb* {[Strict] BrackType}
	|	**(***ConstructorSymb***)** [Fix][Prec] {[Strict] BrackType}
Fix	=	**infixl**
	|	**infixr**
	|	**infix**
Prec	=	*Digit*
Strict	=	**!**
Type	=	[UnqTypeAttrib] TypeConstructor {BrackType}
	|	BrackType
TypeConstructor	=	*TypeSymb*
	|	**(->)**

```
                              |   []
                              |   ({,}+)
                              |   {::}
BrackType                     =   [UnqTypeAttrib] SimpleType
SimpleType                    =   TypeVariable
                              |   BasicType
                              |   PredefAbstrType
                              |   ListType
                              |   TupleType
                              |   ArrayType
                              |   ArrowType
                              |   (Type)

BasicType                     =   Int
                              |   Real
                              |   Char
                              |   Bool
                              |   String
PredefAbstrType               =   World
                              |   File
                              |   ProcId
                              |   Void
ListType                      =   [Type]
TupleType                     =   ([Strict] Type,{[Strict] Type}-list)
ArrayType                     =   {:[Strict] Type:}
ArrowType                     =   ({BrackType}+ -> Type)
```

1.7 Abstract type definition

```
AbstractTypeDef               =   ::TypeLhs ;
```

1.8 Type specification of function

```
FunctionTypeDef               =   FunctionSymb :: FunctionType ;
                              |   (FunctionSymb) [Fix][Prec] [:: FunctionType ] ;
FunctionType                  =   [{[Strict] BrackType}+ ->] Type [ClassContext]
ClassContext                  =   | ClassSymb TypeVariable {& ClassSymb TypeVariable }
```

1.9 Defining overloaded functions, type classes, constructor classes

```
OverloadingDef                =   OverloadedFunctionTypeDef
                              |   TypeClassDef
                              |   OverloadedFunctionInstanceDef
OverloadedFunctionTypeDef
                              =   overload FunctionSymb TypeVariable ::  FunctionType ;
                              |   overload (FunctionSymb) [Fix][Prec] TypeVariable ::  Function-
Type ;
OverloadedFunctionInstanceDef
                              =   instance FunctionSymb
[BrackType[ClassContext]][=FunctionSymb] ;
                              |   instance FunctionSymb [BrackType[ClassContext]]
                                  DefOfFunction ;
TypeClassDef                  =   class ClassSymb TypeVariable
                                  [where] { {ClassMemberDef}+ }
                                  [instances {[BrackType[ClassContext]]}-list ;
ClassMemberDef                =   FunctionTypeDef [MacroDef]
                              |   FunctionSymb ;
TypeClassInstanceDef
```

```
                        = instance ClassSymb [BrackType[ClassContext]]
                          [where] { {DefOfFunction}+ }
```

1.10 Common definitions

```
NodeSymbol              = FunctionSymbol
                        | ConstructorSymbol
FunctionSymbol          = FunctionSymb
                        | (FunctionSymb)
ConstructorSymbol       = ConstructorSymb
                        | (ConstructorSymb)
OperatorSymbol          = FunctionSymb
                        | ConstructorSymb

ArrayExpr               = GraphExpr
BooleanExpr             = GraphExpr
IntegerExpr             = GraphExpr
ListExpr                = GraphExpr
ProcIdExpr              = GraphExpr
RecordExpr              = GraphExpr
```

1.11 Lexical structure

In this chapter the lexical structure of Clean is explained. It describes the kind of tokens recognized by the scanner/parser (Sections 1.11.1, 1.11.2 and 1.11.3).
In Section 1.11.4 the symbols are summarized which are used in the context-free syntax description in the chapters hereafter (they are written in italic in the syntax description).
As is common in modern functional languages there is a lay-out rule (off-side rule) in Clean which permits the omission of braces and semicolons. This lay-out rule is described in Section 1.11.6. All examples in this manual make use of the lay-out rule.

1.11.1 Lexical program structure

```
LexProgram              = { Lexeme | {Whitespace}+ }
Lexeme                  = ReservedKeyword           // see Section 3.3
                        | ReservedSymbol            // see Section 3.3
                        | ReservedChar
                        | Literal                   // see Section 3.2
                        | Identifier
Identifier              = LowerCaseId
                        | UpperCaseId
                        | FunnyId
LowerCaseId             = LowerCaseChar~{IdChar}
UpperCaseId             = UpperCaseChar~{IdChar}
FunnyId                 = {SpecialChar}+

LowerCaseChar           = a | b | c | d | e | f | g | h | i | j
                        | k | l | m | n | o | p | q | r | s | t
                        | u | v | w | x | y | z
UpperCaseChar           = A | B | C | D | E | F | G | H | I | J
                        | K | L | M | N | O | P | Q | R | S | T
                        | U | V | W | X | Y | Z
SpecialChar             = ~ | @ | # | $ | % | ^ | ? | !
                        | + | - | * | < | > | \ | / | | | & | =
                        | :
IdChar                  = LowerCaseChar
```

```
                    | UpperCaseChar
                    | Digit
                    | _      | '
Digit               = 0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |  8  |  9
CharDel             = '
StringDel           = "

Whitespace          = space                     //  a space character
                    | tab                       //  a horizontal tab
                    | newline                   //  a newline char
                    | formfeed                  //  a formfeed
                    | verttab                   //  a vertical tab
                    | Comment
Comment             = //  AnythingTillNL newline
                    | /*  AnythingTill/* Comment AnythingTill*/ */
                    | /*  AnythingTill*/ */
AnythingTillNL      = {AnyChar/ ~newline}       //  no newline
AnythingTill/*      = {AnyChar/ ~/* }           //  no "/*"
AnythingTill*/      = {AnyChar/ ~*/ }           //  no "*/"
AnyChar             = IdChar | ReservedChar | Special
ReservedChar        = (  |  )  |  {  |  }  |  [  |  ]  |  ;  |  ,  |  .
Special             = \n  |  \r  |  \f  |  \b   // newline,return,formf,backspace
                    | \t  |  \\  |  \ CharDel   // tab,backslash,character delete
                    | \ StringDel               // string delete
                    | \ {OctDigit}+             // octal number
                    | \x {HexDigit}+            // hexadecimal number
OctDigit            = 0  |  1  |  2  |  3  |  4  |  5  |  6  |  7
HexDigit            = 0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |  8  |  9
                    | A  |  B  |  C  |  D  |  E  |  F
                    | a  |  b  |  c  |  d  |  e  |  f
```

1.11.2 Literals

```
Literal             = IntegerDenot
                    | RealDenot
                    | BoolDenot
                    | CharDenot
                    | CharsDenot
                    | StringDenot
IntegerDenot        = [Sign]~{Digit}+              // decimal
                    | [Sign]~0 ~{OctDigit}+        // octal
                    | [Sign]~0x~{HexDigit}+        // hexadecimal
Sign                = + | - | ~
RealDenot           = [Sign~]{Digit~}+ .  {~Digit}+[~E  [~Sign]{~Digit}+]
BoolDenot           = True | False
CharDenot           = CharDel~AnyChar/ ~CharDel.CharDel
CharsDenot          = CharDel~{AnyChar/ ~CharDel}+.CharDel
StringDenot         = StringDel~{AnyChar/  ~StringDel}~StringDel
```

Example (literals):

Integer (decimal):	0|1|2|…|8|9|10| … |-1|-2| …
Integer (octal):	00|01|02|…|07|010| … |-01|-02| …
Integer (hexadecimal):	0x0|0x1|0x2|…|0x8|0x9|0xA|0xB … |-0x1|-0x2| …
Real:	0.0|1.5|0.314E10| …
Boolean:	True | False
Character:	'a'|'b'|…|'A'|'B'|…
String:	"" | "Rinus"|"Marko"|…
List of characters:	['Rinus']|['Marko']|…

1.11.3 Reserved keywords and symbols

Below the symbols are listed which have a special meaning in the language. Some symbols only have a special meaning in a certain context. Outside this context they are ordinary identifiers. In the comment it is indicated for which context (name space) the symbol is predefined.

```
ReservedKeyword      =                          // in all contexts:

                       /*                       // begin of comment block
                     | */                       // end of comment block
                     | //                       // rest of line is comment
                     | ::                       // begin of a type def. or type synonym
                     | :==                      // in a synonym type or macro definition
                     | =                        // in a function, graph, alg.  type, rec. field
                     | =:                       // in a graph definition
                     | =>                       // in a function definition
                     | ->                       // in a case expression, lamda
abstraction
                     | [                        // begin of a list
                     | :                        // cons node
                     | ]                        // end of a list
                     | \\                       // begin of list or array comprehension
                     | <-                       // in list gen. in list or array compr.
                     | <-:                      // in array gen. in list or array compr.
                     | {                        // begin of a record, begin of a block
                     | }                        // end of a record, end of a block
                     | &                        // an update of a record or array
                     | {:                       // begin of an array
                     | :}                       // end of an array
                     | {*                       // begin of process annotations
                     | *}                       // end of process annotations
                     | case                     // begin of case expression
                     | class                    // begin of type class definition
                     | code                     // begin code block in a syst impl. module
                     | definition               // begin of definition module
                     | from                     // begin of symbol list for imports
                     | if                       // begin of a conditional expression
                     | implementation           // begin of implementation module
                     | import                   // begin of import list
                     | in                       // end of strict let expression
                     | infix                    // infix indication in operator definition
                     | infixl                   // infix left indication in operator definition
                     | infixr                   // infix right indication in operator def.
                     | instance                 // def of instance of overloaded function
                     | instances                // def of known instances
                     | let!                     // begin of strict let expression
                     | module                   // in module header
                     | of                       // in case statement
                     | overload                 // def of type scheme of overl. function
                     | system                   // begin of system module
                     | where                    // begin of local def of a function altern.
                     | with                     // begin of local def in a rule alternative


ReservedSymbol       = //                       in type specifications:

                     | !                        // strict type
                     | .                        // unique type variable
                     | *                        // unique type
                     | :                        // in uniqueness types
                     | ->                       // function type constructor
                     | []                       // list type constructor
```

	|	**(,)**	// tuple type constructor
	|	**{::}**	// array type constructor
	|	**Bool**	// type boolean
	|	**Char**	// type character
	|	**File**	// type file
	|	**Int**	// type integer
	|	**ProcId**	// type process id
	|	**Real**	// type real
	|	**String**	// type string
	|	**Void**	// type void
	|	**World**	// type world
			// in process annotations:
	|	**at**	// followed by processor id
	|	**P**	// a parallel process to normal form
	|	**I**	// an interleaved process to normal form

1.11.4 Symbols, identifiers and name spaces

In the context-free syntax description given in this reference manual the symbols listed below are used. The **symbols** are **identifiers** used to name modules, functions, operators, graphs, constructors, (node) variables, field names, macros, types, type variables, uniqueness types, uniqueness type variables and type classes. The convention used is that variables always start with a lowercase character while constructors and types always start with an uppercase character. The other identifiers may either start with an uppercase or a lowercase character.

It is allowed to use the same identifier for different purposes as long as the symbols belong to different **name spaces**. Function-, operator-, constructor-, graph-, macro-symbols and node variables form one name space. Type variables and uniqueness type variables form a different name space. All other symbols form a name space on their own.

Under certain conditions it is allowed to use the same name for different functions and operators (overloading).

Notice that for the identifiers names can be used consisting of a combination of lower and/or uppercase characters but one can also define identifiers constructed from special characters like `+`, `<`, etc. These two kind of identifiers cannot be mixed. This makes it possible to leave out white space in expressions like `a+1` (same as `a + 1`).

ModuleSymb	=	LowerCaseId	|	UpperCaseId	|	FunnyId
FunctionSymb	=	LowerCaseId	|	UpperCaseId	|	FunnyId
ConstructorSymb	=	UpperCaseId	|	FunnyId		
SelectorVariable	=	LowerCaseId				
Variable	=	LowerCaseId				
MacroSymb	=	LowerCaseId	|	UpperCaseId	|	FunnyId
FieldSymb	=	LowerCaseId				
TypeSymb	=	UpperCaseId	|	FunnyId		
TypeVariable	=	LowerCaseId				
UniqueTypeVariable	=	LowerCaseId				

1.11.5 Scope of definitions

The **scope** is the program region in which an introduced definition (e.g. function definition, type definition) and corresponding names (e.g. function name, variable name, type name, type variable name) has a meaning. Scopes can be **nested**: within a scope new scopes (**program regions**) can be defined. Within such a nested scope new definitions can be given, new names can be introduced. As usual it is allowed in a nested scope to re-define definitions or re-use names given in a surrounding scope. A definition given or a name introduced in a (nested) scope has no meaning in surrounding scopes. It has a meaning for all scopes nested within it (unless they are redefined within such a nested scope).

- Within a scope different objects of the same kind (i.e. belonging to the same name space) must have different names.

Scope of definitions given in a definition module

The definitions of a definition module have the widest scope. Firstly, all symbols that are defined in a definition module are also automatically defined in the corresponding implementation module (not yet implemented). Secondly, all symbols that are defined in a definition module are also automatically visible (exported) to *all* other modules. In the latter case imports are required to effectuate the actual scope of a symbol to the other module.

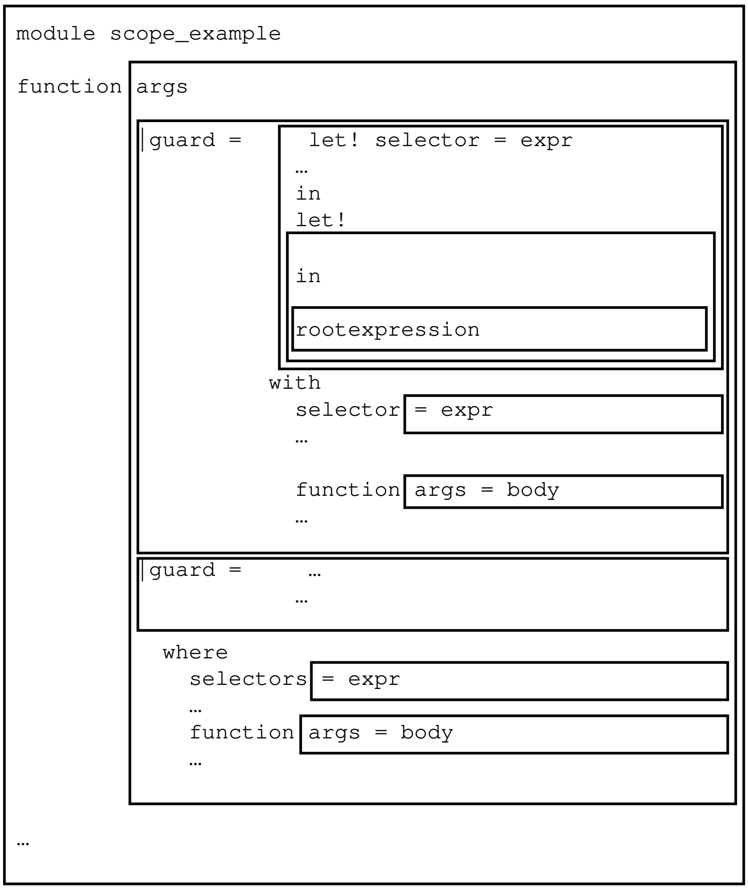
- Within one module a symbol can be defined only once within the same scope and within the same name space.

Scope of global definitions given in an implementation module

Definitions on the **global** (= outermost) level have in principle as scope the module they are defined in, unless they are exported.
Type variables introduced on the left-hand side of a (algebraic, record, synonym, overload, class, instance, function, graph) type definition have the right-hand side of the type definition as scope.

Scope within functions and graphs

More complex are the scope rules within function and graph definitions. In the picture below the general appearance of a function definition is given and the scope that is introduced by the different kind of definitions that can appear.
One can deduce from the picture that a new scope is introduced for each function alternative. Within such a function alternative new functions and graphs can be defined locally within a where block. Each of the guarded rule alternatives also introduce a new scope. Within such a rule alternative new functions and graphs can be defined locally within a with block. Each strict let clause introduces a new nested scope.



New scopes are also introduced by list and array comprehensions and by case expressions. In a list and in an array comprehension new variables can be introduced when generators are specified. These variables have the whole comprehension structure as scope. Each case alternative definition invokes a new scope which is identical to the scope rules of an ordinary function definition as indicated in the picture above.

1.11.6 Lay-out rule

As is common in modern functional languages, there is a lay-out rule in Clean. When the definition of the module header of a module is not ended by a semicolon a Clean program has become lay-out sensitive. The **lay-out rule** assumes the omission of the semi-colon ('`;`') that ends a definition and of the braces ('`{`' and '`}`') that are used to group a list of definitions. These symbols are automatically added according to the following rules:

In lay-out sensitive mode the indentation of the first lexeme after the keywords `let!`, `of`, `where`, or `with` determines the indentation the group of definitions following the keyword has to obey. Depending on the indentation of the first lexeme on a subsequent line the following happens. A new definition is assumed if the lexeme starts on the same indentation (and a semicolon is inserted). A previous definition is assumed to be continued if the lexeme is indented more. The group of definitions ends (and a close brace is inserted) if the lexeme is indented less. Global definitions are assumed to start in column 0.

For reasons of portability it is assumed that a tab space is set to 4 white spaces and that a non-proportional font is used.

Example (use of lay-out rule: same example with and without using the lay-out sensitive mode)

```
primes :: [Int]
primes  =  sieve [2..]
where
    sieve :: [Int] -> [Int]
    sieve [pr:r]  =  [pr:sieve (filter pr r)]

    filter :: Int [Int] -> [Int]
    filter pr [n:r]
        | n Mod pr == 0   =  filter pr r
                          =  [n:filter pr r]


primes :: [Int];
primes  =  sieve [2..];
where
{   sieve :: [Int] -> [Int];
    sieve [pr:r]  =  [pr:sieve (filter pr r)];

        filter :: Int [Int] -> [Int];
        filter pr [n:r] | n Mod pr == 0  =  filter pr r; =      [n:filter pr r];
}
```

2 Standard Environment

2.1 General library entrance (StdEnv)

```
definition module StdEnv

import
   StdClass,   // imports also all functions on basic types

   StdList,
   StdTuple,
   StdArray,
   StdCharList,
   StdFunc,
   StdMisc,
   StdFile
```

2.2 Predefined classes (StdClass)

```
definition module StdClass

import StdOverloaded
import StdBool, StdInt, StdReal, StdChar, StdString //  operations on basic types

//    Remark: derived class members are not implemented yet!
//    For the time-being, macro definitions are used for this purpose
//    This may cause misleading error messages in case of type errors

class PlusMin a     where   +; -; zero

class MultDiv a     where   *; /; one

class Arith a       where   +; -; zero; *; /; one; abs; sign; ~

class Enum a        where   <; inc; +; -; zero

class Eq a          where
 ==

 (<>) infix  2 :: !a   !a    ->    Bool | Eq a
 (<>) x y :== not (x == y)

class Ord a         where
 <

 (>) infix  2 :: !a !a   ->    Bool | Ord a
 (>) x y  :== y < x

 (<=) infix  2 :: !a !a  ->    Bool | Ord a
 (<=) x y :== not (y<x)

 (>=) infix  2 :: !a !a  ->    Bool | Ord a
 (>=) x y :== not (x<y)

 min::!a !a ->   a | Ord a
 min x y  :== if (x<y) x y

 max::!a !a ->   a | Ord a
 max x y  :== if (x<y) y x

class ToInt         a  where toInt
class ToChar        a  where toChar
class ToBool        a  where toBool
class ToReal        a  where toReal
```

```
class ToString      a  where toString

class FromInt       a  where fromInt
class FromChar      a  where fromChar
class FromBool      a  where fromBool
class FromReal      a  where fromReal
class FromString    a  where fromString
```

2.3 Predefined overloaded operations (StdOverloaded)

```
definition module StdOverloaded

overload (+)  infixl 6   a  :: !a   !a   ->   a    // Add arg1 to arg2
overload (-)  infixl 6   a  :: !a   !a   ->   a    // Subtract arg2 from arg1
overload zero            a  :: a              // Zero (unit element for addition)

overload (*)  infixl 7   a  :: !a   !a   ->   a    // Multiply arg1 with arg2
overload (/)  infix  7   a  :: !a   !a   ->   a    // Divide arg1 by arg2
overload one             a  :: a              // One (unit element for multiplication)

overload (^)  infixr 8   a  :: !a   !a   ->   a    // arg1 to the power of arg2
overload inc             a  :: !a        ->   a    // Increment with 1
overload dec             a  :: !a        ->   a    // Decrement with 1
overload abs             a  :: !a        ->   a    // Absolute value
overload sign            a  :: !a        ->   Int  // 1 (pos value) 0 (if zero)
                                                   // -1 (if neg value)
overload ~               a  :: !a        ->   a    // -a1

overload (==) infix  2   a  :: !a   !a   ->   Bool // True if arg1 is equal to arg2

overload (<)  infix  2   a  :: !a   !a   ->   Bool // True if arg1 is less than arg2

overload toInt           a  :: !a      ->   Int       //   Convert into Int
overload toChar          a  :: !a      ->   Char      //   Convert into Char
overload toBool          a  :: !a      ->   Bool      //   Convert into Bool
overload toReal          a  :: !a      ->   Real      //   Convert into Real
overload toString        a  :: !a      ->   String    //   Convert into String

overload fromInt         a  :: !Int    -> a           //   Convert from Int
overload fromChar        a  :: !Char   -> a           //   Convert from Char
overload fromBool        a  :: !Bool   -> a           //   Convert from Bool
overload fromReal        a  :: !Real   -> a           //   Convert from Real
overload fromString      a  :: !String -> a           //   Convert from String

overload  #              a  :: !a            -> Int   //   Number of elemenents in arg
overload  ##             a  :: !a            -> Int   //   Upperbound of arg
overload  (%)  infixl 9  a  :: !a !(!Int,!Int)  -> a  //   Slice a part from arg1

overload (+++) infixr 0  a  :: !a   !a         -> a   //   Append args
```

2.4 Operations on Booleans (StdBool)

```
system module StdBool

import  StdOverloaded

instance ==           Bool

instance toBool       Bool
instance toString     Bool

instance fromBool     Bool
instance fromBool     String

//    Additional Logical Operators:

not              :: !Bool      ->    Bool  //    Not arg1
(||)  infixr 0   :: !Bool Bool ->    Bool  //    Conditional or  of arg1 and arg2
(&&)  infixr 1   :: !Bool Bool ->    Bool  //    Conditional and of arg1 and arg2
```

```
//    Miscellaneous:

otherwise :== True                       //    To be used in guards
```

2.5 Operations on Characters (StdChar)

```
system module StdChar

import   StdOverloaded

instance ==            Char

instance <             Char

instance toInt         Char
instance toChar        Char
instance toString      Char

instance fromChar      Int
instance fromChar      Char
instance fromChar      String

//    Additional convertions:

digtoInt      :: !Char -> Int          //    Convert Digit into Int
toUpper       :: !Char -> Char         //    Convert Char into an uppercase Char
toLower       :: !Char -> Char         //    Convert Char into a  lowercase Char

//    Tests on Characters:

isAscii          :: !Char -> Bool         //    True if arg1 is an ASCII character
isControl        :: !Char -> Bool         //    True if arg1 is a  control character
isPrint          :: !Char -> Bool         //    True if arg1 is a  printable character
isSpace          :: !Char -> Bool         //    True if arg1 is a  space, tab etc
isUpper          :: !Char -> Bool         //    True if arg1 is an uppercase character
isLower          :: !Char -> Bool         //    True if arg1 is a  lowercase character
isAlpha          :: !Char -> Bool         //    True if arg1 is a  letter
isDigit          :: !Char -> Bool         //    True if arg1 is a  digit
isAlphanum       :: !Char -> Bool         //    True if arg1 is an alphanumerical character
```

2.6 Operations on Integers (StdInt)

```
system module StdInt

import   StdOverloaded

instance +          Int
instance -          Int
instance zero       Int

instance /          Int
instance *          Int
instance one        Int

instance ^          Int
instance inc        Int
instance dec        Int
instance abs        Int
instance sign       Int
instance ~          Int

instance ==         Int

instance <          Int

instance toInt         Int
instance toChar        Int
```

```
instance toReal       Int
instance toString     Int

instance fromInt      Int
instance fromInt      Char
instance fromInt      Real
instance fromInt      String

// Additional functions for integer arithmetic:

(mod)    infix  7   :: !Int !Int  ->   Int     //   arg1 modulo arg2
(rem)    infix  7   :: !Int !Int  ->   Int     //   remainder after division
gcd                 :: !Int !Int  ->   Int     //   Greatest common divider
lcm                 ::  Int !Int  ->   Int     //   Least common multiple

//    Test on Integers:

isEven           :: !Int       ->   Bool  //   True if arg1 is an even number
isOdd            :: !Int       ->   Bool  //   True if arg1 is an odd  number

//    Operators on Bits:

(bitor)  infix  0   :: !Int !Int  ->   Int   //   Bitwise Or of arg1 and arg2
(bitand) infix 1    :: !Int !Int  ->   Int   //   Bitwise And of arg1 and arg2
(bitxor) infix 0    :: !Int !Int  ->   Int   //   Exclusive-Or arg1 with mask arg2
(<<)  infix  7      :: !Int !Int  ->   Int   //   Shift arg1 to the left arg2 bit places
(>>)  infix  7      :: !Int !Int  ->   Int   //   Shift arg1 to the right arg2 bit places
bitnot              :: !Int       ->   Int   //   One's complement of arg1
```

2.7 Operations on Reals (StdReal)

```
system module StdReal

import   StdOverloaded

instance +          Real
instance -          Real
instance zero       Real

instance *          Real
instance /          Real
instance one        Real

instance ^          Real
instance inc        Real
instance dec        Real
instance abs        Real
instance sign       Real
instance ~          Real

instance ==         Real

instance <          Real

instance toInt        Real
instance toReal       Real
instance toString     Real

instance fromReal     Int
instance fromReal     Real
instance fromReal     String

//    Logarithmical Functions:

ln               :: !Real      ->   Real  //   Logarithm base e
log10            :: !Real      ->   Real  //   Logarithm base 10
exp              :: !Real      ->   Real  //   e to to the power
sqrt             :: !Real      ->   Real  //   Square root

//    Trigonometrical Functions:
```

```
sin             :: !Real     ->   Real  //   Sinus
cos             :: !Real     ->   Real  //   Cosinus
tan             :: !Real     ->   Real  //   Tangens
asin            :: !Real     ->   Real  //   Arc Sinus
acos            :: !Real     ->   Real  //   Arc Cosinus
atan            :: !Real     ->   Real  //   Arc Tangus

//   Additional conversion:

entier          :: !Real     ->   Int     //   Convert Real into Int by taking entier
```

2.8 Operations on Strings (StdString)

```
system module StdString

import   StdOverloaded

instance ==             String

instance <              String

instance toString       String
instance toInt          String
instance toReal         String

instance fromString     String

instance #              String
instance ##             String
instance %              String

instance +++            String

//   additional operator

(!!) infixl 9 :: !String !Int  -> Char            //   select i-th element (0 <= i <= ##arg1)
(:=) infixl 9 :: !String !(!Int,!Char) -> String  //   update i-th element
```

2.9 Operations on Arrays (StdArray)

```
system module StdArray

instance #    {:a:}
instance ##   {:a:}
instance %    {:a:}

createArray ::   !Int a           -> *{:a:}  //   {: [i] = arg2   \\ i <- [0..##arg1] :}
mkArray     ::   !Int (Int -> a)  -> *{:a:}  //   {: [i] = arg2 i \\ i <- [0..##arg1] :}
```

2.10 Operations on Lists of Characters (StdCharList)

```
definition module StdCharList
import StdClass

instance toString [x] | ToChar x               // Convert [Char] into String

//   Functions for outlining

cjustify        :: !.Int ![.Char] -> .[Char]   // Center [Char] in field with width arg1
ljustify        :: !.Int ![.Char] -> .[Char]   // Left justify [Char] in field with width arg1
rjustify        :: !.Int ![.Char] -> [Char]    // Right justify [Char] in field with width arg1

flatlines       :: ![[u:Char]] -> [u:Char]     // Concatenate by adding newlines
mklines         :: ![Char] -> [[Char]]         // Split in lines removing newlines

spaces          :: !.Int -> .[Char]            // Make [Char] containing n space characters
```

2.11 Operations on Files (StdFile)

```
system module StdFile

//    File modes synonyms

FReadText     :== 0   //    Read from a text file
FWriteText    :== 1   //    Write to a text file
FAppendText   :== 2   //    Append to an existing text file
FReadData     :== 3   //    Read from a data file
FWriteData    :== 4   //    Write to a data file
FAppendData   :== 5   //    Append to an existing data file

//    Seek modes synonyms

FSeekSet   :== 0   //    New position is the seek offset
FSeekCur   :== 1   //    New position is the current position plus the seek offset
FSeekEnd   :== 2   //    New position is the size of the file plus the seek offset

::    *Files

//    Opening and Closing a File from the FileSystem:

openfiles  :: !*World -> (!*Files,!*World)

closefiles :: !*Files !*World -> *World

fopen      :: !String !Int !*Files -> (!Bool,!*File,!*Files)
/*    Opens a file for the first time in a certain mode (read, write or append, text or data).
      The boolean output parameter reports success or failure. */

fclose     :: !*File !*Files -> (!Bool,!*Files)

freopen    :: !*File !Int -> (!Bool,!*File)
/*    Re-opens an open file in a possibly different mode.
      The boolean indicates whether the file was successfully closed before reopening. */

//    Reading from a File:

freadc     :: !*File -> (!Bool,!Char,!*File)
/*    Reads a character from a text file or a byte from a datafile.
      The boolean indicates succes or failure */

freadi     :: !*File -> (!Bool,!Int,!*File)
/*    Reads an Integer from a textfile by skipping spaces, tabs and newlines and
      then reading digits, which may be preceeded by a plus or minus sign.
      From a datafile freadi will just read four bytes (a Clean Int). */

freadr     :: !*File -> (!Bool,!Real,!*File)
/*    Reads a Real from a textfile by skipping spaces, tabs and newlines and then
      reading a character representation of a Real number.
      From a datafile freadr will just read eight bytes (a Clean Real). */

freads     :: ! *File !Int -> (!String,!*File)
/*    Reads n characters from a text or data file, which are returned as a String.
      If the file doesn't contain n characters the file will be read to the end
      of the file. An empty String is returned if no characters can be read. */

freadline  :: !*File -> (!String,!*File)
/*    Reads a line from a textfile. (including a newline character, except for the last
      line) freadline cannot be used on data files. */

//    Writing to a File:

fwritec    :: !Char !*File -> *File
/*    Writes a character to a textfile.
      To a datafile fwritec writes one byte (a Clean Char). */

fwritei    :: !Int !*File -> *File
/*    Writes an Integer (its textual representation) to a text file.
      To a datafile fwritei writes four bytes (a Clean Int). */
```

```
fwriter    :: !Real !*File -> *File
/*    Writes a Real (its textual representation) to a text file.
      To a datafile fwritecr writes eight bytes (a Clean Real). */

fwrites    :: !String !*File -> *File
/*    Writes a String to a text or data file. */

//    Testing:

fend     :: !*File -> (!Bool,!*File)
/*    Tests for end-of-file. */

ferror     :: !*File -> (!Bool,!*File)
/*    Has an error occurred during previous file I/O operations? */

fposition  :: !*File -> (!Int,!*File)
/*    returns the current position of the file poInter as an Integer.
      This position can be used later on for the fseek function. */

fseek      :: !*File !Int !Int -> (!Bool,!*File)
/*    Move to a different position in the file, the first Integer argument is the offset,
      the second argument is a seek mode. (see above). True is returned if successful. */

//    Predefined files.

stdIO      :: !*Files -> (!*File,!*Files)
/*    Open the 'Console' for reading and writing. */

stderr     :: *File
/*    Open the 'Errors' file for writing only. May be opened more than once. */

//    Opening and reading Shared Files:

sfopen     :: !String !Int !*Files -> (!Bool,!File,!*Files)
/*    With sfopen a file can be opened for reading more than once.
      On a file opened by sfopen only the operations beginning with sf can be used.
      The sf... operations work just like the corresponding f... operations.
      They can't be used for files opened with fopen or freopen. */

sfreadc    :: !File -> (!Bool,!Char,!File)
sfreadi    :: !File -> (!Bool,!Int,!File)
sfreadr    :: !File -> (!Bool,!Real,!File)
sfreads    :: !File !Int -> (!String,!File)
sfreadline :: !File -> (!String,!File)
sfseek     :: !File !Int !Int -> (!Bool,!File)

sfend      :: !File -> Bool
sfposition :: !File -> Int
/*    The functions sfend and sfposition work like fend and fposition, but don't return a
      new file on which other operations can continue. They can be used for files opened
      with sfopen or after fshare, and in guards for files opened with fopen or freopen. */

//    Convert a *File into:

fshare     :: !*File -> File
/*    Change a file so that from now it can only be used with sf... operations. */
```

2.12 Operations on polymorphic functions (StdFunc)

```
definition module StdFunc

// Some Classical Functions:

I             :: !.a -> .a                         // Identity function
K             :: !.a .b -> .a                      // Konstant function
S             :: .(a -> .(.b -> .(a -> .c))) .b a -> .c // Distribution function

flip          :: .(.a -> .(.b -> .c)) .b .a -> .c       // Flip arguments

(o)      infixr  9  ::  u:(.a -> .b) u:(.c -> .a) -> u:(.c -> .b)// Function composition
```

```
twice        ::            !(.a -> .a) .a   -> .a      // f (f x)
while        :: !(a -> .Bool) (a -> a)  a   ->  a      // while (p x) f (f x)
until        :: !(a -> .Bool) (a -> a)  a   ->  a      // f (f x) until (p x)
iter         :: !Int        (.a -> .a) .a   -> .a      // f (f..(f x)..)

//    Some handy functions for transforming unique states:

::St s a :== s -> (a,s)

seq          :: ![.(.s -> .s)] .s -> .s              // fn-1 (..(f1 (f0 x))..)
seqList      :: ![St .s .a] .s -> ([.a],.s)          // fn-1 (..(f1 (f0 x))..)
                                                     // monadic style:
(`bind`) infix 0    :: u:(St .s .a) u:(.a -> .(St .s .b)) -> u:(St .s .b);
return          :: u:a -> u:(St .s u:a)
```

2.13 Operations on Lists (StdList)

```
definition module StdList

//    Instances of overloaded functions:

import   StdClass

instance ==    [a] | Eq a

instance <     [a] | Ord a

instance #     [a]
instance ##       [a]
instance %     [a]

instance toString      [x]   | ToChar x    // Convert [x]    via [Char] into String
instance fromString    [x]   | FromChar x  // Convert String via [Char] into [x]

//    List Operators:

(!)   infixl 9   :: [.a] Int -> .a              //    Get nth element of the list
(++)  infixr 0   :: ![.a] u:[.a] -> u:[.a]      //    Append args
flatten          :: ![.[a]] -> [a]              //    e0 ++ e1 ++ ... ++ e##
isEmpty          :: ![.a] -> Bool               //    [] ?

//    List breaking or permuting functions:

hd          :: ![.a] -> .a                      //    Head of the list
tl          :: !u:[.a] -> u:[.a]                //    Tail of the list
last        :: ![.a] -> .a                      //    Last element of the list
take        :: !Int ![.a] -> [.a]               //    Take first arg1 elements of the list
takeWhile   :: (a -> .Bool) !.[a] -> .[a]       //    Take elements while pred holds
drop        :: Int !u:[.a] -> u:[.a]            //    Drop first arg1 elements from the list
dropWhile   :: (a -> .Bool) !u:[a] -> u:[a]     //    Drop elements while pred holds
filter      :: (a -> .Bool) !.[a] -> .[a]       //    Drop all elements not satisfying pred
insert      :: (a a -> .Bool) a !u:[a] -> u:[a] //    Insert arg2 when pred arg2 elem holds
remove      :: !Int !u:[.a] -> u:[.a]           //    Remove arg2!arg1 from list
reverse     :: !.[a] -> [a]                     //    Reverse the list
span        :: !(a -> .Bool) !u:[a] -> (.[a],u:[a])   //    (TakeWhile list,DropWhile list)
splitAt      :: !Int u:[.a] -> ([.a],u:[.a])    //    (Take n list,Drop n list)

//    Creating lists:

map         :: (.a -> .b) ![.a] -> [.b]         //    [f e0,f e1,f e2,...
iterate     :: (a -> a) a -> .[a]               //    [a,f a,f (f a),...
indexList   :: !.[a] -> [Int]                   //    [0..##list]
repeatn     :: !.Int a -> .[a]                  //    [e0,e0,...,e0] of length n
repeat      :: a -> [a]                         //    [e0,e0,...
unzip       ::    ![(a,b)]   -> ([a],[b])       //    ([a0,a1,...],[b0,b1,...])
zip2        :: ![.a] [.b]    -> [(.a,.b)]       //    [(a0,b0),(a1,b1),...
zip         :: !(![.a],[.b]) -> [(.a,.b)]       //    [(a0,b0),(a1,b1),...
diag2       :: !.[a] .[b]    -> [.(a,b)]        //    [(a0,b0),(a1,b0),(a0,b1),...
diag3       :: !.[a] .[b] .[c]  -> [.(a,b,c)]   //    [(a0,b0,c0),(a1,b0,c0),...

//    Folding and scanning:
```

```
foldl       :: (.a -> .(.b -> .a)) !.a ![.b] -> .a //    op(...(op (op (op r e0) e1)...e##)
foldr       :: (.a -> .(.b -> .b)) !.b ![.a] -> .b //    op e0 (op e1(...(op r e##)...)
scan        ::  (a -> .(.b -> a)) a ![.b] -> .[a]  //    [r,op r e0,op (op r e0) e1,...

//    On Booleans

and         :: ![.Bool] -> Bool                    //    e0 && e1 ... && e##
or          :: ![.Bool] -> Bool                    //    e0 || e1 ... || e##
any         :: (.a -> .Bool) ![.a] -> Bool         //    True, if ei is True for some i
all         :: (.a -> .Bool) ![.a] -> Bool         //    True, if ei is True for all i
```

2.14 Operations on Lists involving Classes (StdClassList)

```
definition module StdClassList

from StdClass import Eq, Ord, PlusMin, Arith, FromInt

//    When ordering is defined on list elements

maxList     :: !.[a]         -> a        | Ord a    //    Maximum element of list
minList     :: !.[a]         -> a        | Ord a    //    Minimum element of list
sort        :: !u:[a]        -> u:[a] | Ord a       //    Sort the list
merge          :: !.[a] !u:[a]  -> u:[a] | Ord a       //    Merge two sorted lists to a sorted
list

//    When equality is defined on list elements

isMember          ::    a  !.[a]    -> .Bool | Eq a     //    Is element in list
removeMembers :: u:[a] .[a]    -> u:[a] | Eq a     //    Remove arg2s from list arg1
removeDup     :: !.[a]      -> .[a]     | Eq a     //    Remove all duplicates from list

limit             :: !.[a]      -> a        | Eq a     //    [...,a,a]

//    When addition is defined on list elements

sum            :: !.[a] -> a     | PlusMin a           //    sum, sum [] = zero

//    When multiplication and addition is defined on list elements

prod           :: !.[a] -> a     | Arith a           //    product, prod [] = one
avg            :: !.[a] -> a     | Arith a           //    average, avg [] gives error!
```

2.15 Miscellaneous operations (StdMisc)

```
system module StdMisc

abort       :: !String -> .a         // stop reduction, print argument and core dump
undef       :: .a                    // fatal error, stop reduction.
```

2.16 Operations on Tuples (StdTuple)

```
definition module StdTuple

// Standard Functions on Tuples:

import StdClass

instance ==    (a,b)    |  Eq a & Eq b
instance == (a,b,c)     |  Eq a & Eq b & Eq c

instance <  (a,b)    |  Ord a & Ord b
instance <  (a,b,c)  |  Ord a & Ord b & Ord c

fst      :: !(!.a,.b) -> .a                                  // t1 of (t1,t2)
snd      :: !(.a,!.b) -> .b                                  // t2 of (t1,t2)
```

```
fst3  :: !(!.a,.b,.c) -> .a                                 // t1 of (t1,t2,t3)
snd3  :: !(.a,!.b,.c) -> .b                                 // t2 of (t1,t2,t3)
thd3  :: !(.a,.b,!.c) -> .c                                 // t3 of (t1,t2,t3)

app2     :: !(.(.a -> .b),.(.c -> .d)) !(.a,.c) -> (.b,.d)     // f (a,b) = (f a,f b)
app3     :: !(.(.a -> .b),.(.c -> .d),.(.e -> .f)) !(.a,.c,.e) -> (.b,.d,.f)
                                                            // f (a,b,c) = (f a,f b,f c)

curry    :: !.((.a,.b) -> .c) .a .b -> .c                   // f a b   = f (a,b)
uncurry :: !.(.a -> .(.b -> .c)) !(.a,.b) -> .c             // f (a,b) = f a b
```

3 I/O library

This section describes the Clean 0.8 I/O library converted to 1.0 syntax. A new I/O library which allows more locality of state within interactive components is being developed.

3.1 General operations on the IOState (deltaEventIO)

```
definition module deltaEventIO
import deltaIOSystem
```

```
::    * EVENTS
```

The event stream.

```
::    * IOState * s
```

The environment on which all event I/O functions operate.

```
::    InitialIO *s  :==   [s ->  (IOState s) -> (s, IOState s)]
```

The I/O functions that will be evaluated before starting the interaction.

```
OpenEvents    ::    !* WORLD   ->    (!EVENTS, !* WORLD)
```

Retrieves the event stream from the world. Attempts to retrieve the event stream from the world more than once without putting it back result in a run-time error.

```
CloseEvents   ::    !EVENTS  !* WORLD   ->    * WORLD
```

Puts the event stream back into the world.

```
StartIO       ::    !(IOSystem s (IOState s)) !s !(InitialIO s)  !EVENTS  ->    (!s, !EVENTS)
```

Starts a new interaction. The initial state of this interaction is composed of the program state (s) and the initial I/O state, which consists of the devices participating in the interaction (defined in the IOSystem argument) and the event stream (EVENTS). Of each device in the initial IOSystem only the first occurrence is taken into account. The program state must be unique. Before starting the interaction the InitialIO functions are evaluated. At the end of the interaction StartIO returns the final program state and the resulting event stream, from which the events that have been handled during the interaction have been removed.

```
NestIO        ::    !(IOSystem s (IOState s)) !s !(InitialIO s) !(IOState t) -> (!s, !IOState t)
```

Starts a nested interaction. It replaces the current interaction (as specified by the IOState argument) with a completely new one (as specified by the IOSystem argument). It hides the devices of the current interaction (if any) and fills a new IOState with the devices that are specified in the IOSystem and with the event stream of the old IOState. The program state argument (s) serves as initial program state of the new interaction. Before starting the nested interaction the InitialIO functions are evaluated. NestIO returns the final program state of the new interaction and the original IOState, such that the original interaction reappears.

```
QuitIO        ::    !(IOState s)  ->    IOState s
```

Closes all devices that are held in the IOState argument. The resulting IOState will cause StartIO or NestIO to terminate.

```
ChangeIOState ::   ![(IOState s) -> (IOState s)] !(IOState s)   ->   IOState s
```

Applies all functions in its first argument in consecutive order to the second (IOState) argument.

3.2 Definition of the I/O system (deltaIOSystem)

```
definition module deltaIOSystem
from deltaPicture import Picture, Rectangle, DrawFunction

::    * DialogState * s * io
::    DialogInfo

::    IOSystem * s * io      :==   [DeviceSystem s io]
::    DeviceSystem * s * io  =     TimerSystem   [TimerDef   s io]
                             |     MenuSystem    [MenuDef    s io]
                             |     WindowSystem  [WindowDef  s io]
                             |     DialogSystem  [DialogDef  s io]
```

The timer device responds only to timer events. A timer event occurs as soon as a certain TimerInterval has expired since the last time it was 'sampled'. A TimerInterval is defined as a number of ticks. A macro TicksPerSecond is defined in deltaTimer.dcl. The timer event causes one of the programmer-defined TimerFunctions to be evaluated. The TimerState argument of a TimerFunction indicates how many times the TimerInterval has passed since the last timer event for that timer. When the TimerInterval of one of the timers is smaller than 1 the corresponding TimerFunction is called as often as possible.

```
::    TimerDef * s * io   =  Timer TimerId SelectState TimerInterval (TimerFunction s io)
::    TimerId                 :==   Int
::    TimerInterval           :==   Int
::    TimerFunction * s * io  :==   TimerState -> ( s -> * ( io -> (s, io) ) )
::    TimerState              :==   Int
```

The menu device consists of several PullDownMenus. PullDownMenus are logically grouped into a menu bar, in the same order as they are specified. They are selected by pressing the mouse on their MenuTitle in the menu bar. Menus contain MenuElements. The corresponding MenuFunctions are executed when these elements are selected.

```
::    MenuDef * s * io    =  PullDownMenu MenuId MenuTitle SelectState [MenuElement s io]
::    MenuElement * s * io
   =  MenuItem       MenuItemId ItemTitle KeyShortcut SelectState           (MenuFunction s io)
   |  CheckMenuItem  MenuItemId ItemTitle KeyShortcut SelectState MarkState (MenuFunction s io)
   |  SubMenuItem    MenuId     ItemTitle SelectState [MenuElement s io]
   |  MenuItemGroup  MenuItemGroupId                  [MenuElement s io]
   |  MenuRadioItems MenuItemId                       [RadioElement s io]
   |  MenuSeparator
::    RadioElement * s * io
   =  MenuRadioItem MenuItemId ItemTitle KeyShortcut SelectState (MenuFunction s io)
::    MenuFunction * s * io   :==   s -> *(io -> (s, io))
::    MenuId                  :==   Int
::    MenuTitle               :==   String
::    MenuItemId              :==   Int
::    MenuItemGroupId         :==   Int
::    KeyShortcut             =     Key KeyCode | NoKey
```

The window device consists of several ScrollWindows or FixedWindows. A ScrollWindow is defined by the following arguments:

- WindowId: the number by which the programmer refers to the window.
- WindowPos: the position of the upper-left corner of the window.
- WindowTitle: the title of the window.
- ScrollBarDefs: the horizontal and vertical scroll bars (in that order).
- PictureDomain: the range of the drawing operations in the window.
- MinimumWindowSize: the smallest dimensions of the window.
- InitialWindowSize: the initial dimensions of the window.
- UpdateFunction: the function to redraw parts (UpdateArea) of the window.
- An attribute list that may contain the following window attributes:
  - Activate: the way to respond to activation of the window.

– Deactivate: the way to respond to de-activation of the window.
– GoAway: the way to respond when the window is closed.
– Keyboard: the way the window responds to keyboard input.
– Mouse: the way the window responds to mouse events.
– Cursor: the shape of the cursor (mouse pointer) inside the window.
– StandByWindow: when this attribute is present the window will be a so-called stand-by window.

A FixedWindow has a fixed size, which is defined by its PictureDomain. Therefore it has no scroll bars and no size parameters. When the PictureDomain of a FixedWindow is or becomes greater than one of the screen's dimensions it becomes a ScrollWindow.

```
::   WindowDef * s * io
   = ScrollWindow WindowId WindowPos WindowTitle  ScrollBarDef ScrollBarDef PictureDomain
                MinimumWindowSize InitialWindowSize (UpdateFunction s) [WindowAttribute s io]
   | FixedWindow WindowId WindowPos WindowTitle PictureDomain
                                                     (UpdateFunction s) [WindowAttribute s io]

::   WindowId              :==   Int
::   WindowPos             :==   (!Int, !Int)
::   WindowTitle           :==   String
::   ScrollBarDef          =     ScrollBar ThumbValue ScrollValue
::   ThumbValue            =     Thumb Int
::   ScrollValue           =     Scroll Int
::   MinimumWindowSize     :==   (!Int, !Int)
::   InitialWindowSize     :==   (!Int, !Int)
::   UpdateFunction * s    :==   UpdateArea -> (s -> (s, [DrawFunction]))
::   UpdateArea            :==   [Rectangle]

::   WindowAttribute * s * io =  Activate      (WindowFunction s io)
                              |  Deactivate    (WindowFunction s io)
                              |  GoAway        (WindowFunction s io)
                              |  Mouse         SelectState (MouseFunction s io)
                              |  Keyboard      SelectState (KeyboardFunction s io)
                              |  Cursor        CursorShape
                              |  StandByWindow

::   WindowFunction      * s * io :==  s -> *(io -> (s, io))
::   MouseFunction       * s * io :==  MouseState -> (s -> *(io -> (s, io)))
::   KeyboardFunction    * s * io :==  KeyboardState -> (s -> *(io -> (s, io)))

::   CursorShape    =  StandardCursor   |  BusyCursor       |  IBeamCursor    |
                       CrossCursor      |  FatCrossCursor   |  ArrowCursor    |
                       HiddenCursor
```

The dialog device: dialogs given in the initial I/O system will be opened as modeless dialogs. Use the Open(Modal)Dialog function (deltaDialog.icl) to open dialogs during the interaction. PropertyDialogs are special dialogs that can only be modeless. They are used to change certain properties (default values, preferences etc.) of a program. They have two predefined buttons: the Set and the Reset buttons. A CommandDialog can be any kind of dialog. A PropertyDialog is defined by the following attributes:

- DialogId: a number by which the programmer can refer to the dialog.
- DialogTitle: the title of the dialog.
- A list of attributes that may contain the following dialog attributes:
  – DialogPos: the position of the dialog on the screen.
  – DialogSize: the size of the dialog.
  – DialogMargin: the horizontal and vertical margins between the borders of the dialog and the items.
  – ItemSpace: the horizontal and vertical space between the items of the dialog.
  – StandByDialog: when this attribute is present the dialog will be a so-called stand-by dialog.

  When none of these attributes is specified the dialog is centred on the screen, a size is chosen such that all items fit in the dialog and save default margins, and item spaces are chosen.
- SetFunction/ResetFunction: the button function for the set/reset button.
- A list of DialogItems: other items such as CheckBoxes, Controls etc.

A CommandDialog also has an id, a title, a position, a size and a list of DialogItems. Furthermore it has the following attribute:

- DialogItemId: the item id of the default button.

In the AboutDialog, information about the application (version, authors etc.) can be presented. It may also contain a button that should provide a help facility. The first AboutDialog that is encountered in the initial dialog device becomes the AboutDialog of the application. Attempts to open AboutDialogs with Open(Modal)Dialog are ignored. The AboutDialog will be accessible by the user during the interaction in a system-dependent way.

```
::   DialogDef * s * io
   = PropertyDialog DialogId DialogTitle [DialogAttribute]
                              (SetFunction s io) (ResetFunction s io) [DialogItem s io]
   | CommandDialog  DialogId DialogTitle [DialogAttribute] DialogItemId [DialogItem s io]
   | AboutDialog ApplicationName PictureDomain [DrawFunction] (AboutHelpDef s io)

::   DialogId           :==  Int
::   DialogTitle        :==  String
::   DialogAttribute     =  DialogPos  Measure Measure
                         |  DialogSize Measure Measure
                         |  DialogMargin  Measure Measure
                         |  ItemSpace  Measure Measure
                         |  StandByDialog
::   Measure             =  MM Real  |  Inch Real  |  Pixel Int
::   ApplicationName    :==  String
::   AboutHelpDef  * s * io   =    AboutHelp ItemTitle (AboutHelpFunction s io)
                              |    NoHelp
::   AboutHelpFunction * s * io  :==  s -> *(io -> (s, io))
```

A DialogItem can be a final button (DialogButton), a final button with a user-defined look (DialogIconButton), an unchangeable piece of text (StaticText), a changeable piece of text (DynamicText), an editable text field (EditText), a pop-up menu (DialogPopUp), a group of RadioButtons, a group of CheckBoxes, or a user-defined Control. The ItemPos specifies the position of the item relative to the other items. When the ItemPos is Left, Center or Right the item is placed left-aligned, centred or right-aligned, respectively, beneath all other items.

```
::   DialogItem * s * io
   = DialogButton     DialogItemId ItemPos ItemTitle SelectState (ButtonFunction s io)
   | DialogIconButton DialogItemId ItemPos PictureDomain
                                            IconLook SelectState (ButtonFunction s io)
   | StaticText   DialogItemId ItemPos String
   | DynamicText  DialogItemId ItemPos TextWidth String
   | EditText     DialogItemId ItemPos TextWidth NrEditLines String
   | DialogPopUp  DialogItemId ItemPos SelectState   DialogItemId [RadioItemDef s io]
   | RadioButtons DialogItemId ItemPos RowsOrColumns DialogItemId [RadioItemDef s io]
   | CheckBoxes   DialogItemId ItemPos RowsOrColumns [CheckBoxDef s io]
   | Control DialogItemId ItemPos PictureDomain SelectState ControlState
                                                  ControlLook ControlFeel (DialogFunction s io)

::   DialogItemId  :==   Int
::   ItemPos        =    Left | Center | Right | RightTo DialogItemId |
                         Below DialogItemId | XOffset DialogItemId Measure |
                         YOffset DialogItemId Measure | XY Measure Measure |
                         ItemBox Int Int Int Int
::   IconLook      :==   SelectState -> [DrawFunction]
::   TextWidth     :==   Measure
::   NrEditLines   :==   Int
::   RowsOrColumns =     Rows Int  |  Columns Int

::   RadioItemDef * s * io = RadioItem DialogItemId ItemTitle SelectState (DialogFunction s io)
::   CheckBoxDef * s * io
                  = CheckBox DialogItemId ItemTitle SelectState MarkState (DialogFunction s io)
```

Attributes of a user-defined Control: the ControlState can be a boolean, an integer, a real or a string. The look of the control is defined by the DrawFunctions (see deltaPicture.dcl) returned by the ControlLook function. The ControlFeel defines the way to respond to mouse clicks inside the control's picture domain.

```
::    ControlState  =     BoolCS Bool         │  IntCS Int        │
                          RealCS Real         │  StringCS String  │
                          ListCS [ControlState] │  PairCS ControlState ControlState
::    ControlLook   :==   SelectState -> ControlState -> [DrawFunction]
::    ControlFeel   :==   MouseState  -> ControlState -> (ControlState,[DrawFunction])
```

Types of the several dialog item functions:

```
::    SetFunction      * s * io   :== ButtonFunction s io
::    ResetFunction    * s * io   :== ButtonFunction s io
::    DialogFunction   * s * io   :== DialogInfo -> (DialogState s io) -> DialogState s io
::    ButtonFunction   * s * io   :== DialogInfo -> (s -> *(io -> (s, io)))
```

A notice is a simple, modal dialog containing only text and final buttons. It can be used to inform the user about unusual or dangerous situations. A notice is defined by the following attributes:

- A list of Strings: each string is a line of the message of the notice.
- A NoticeButtonDef: the default button of the notice.
- A list of NoticeButtonDefs: the other buttons of the notice.

```
::    NoticeDef           =     Notice [String] NoticeButtonDef [NoticeButtonDef]
::    NoticeButtonDef     =     NoticeButton NoticeButtonId ItemTitle
::    NoticeButtonId      :==   Int
```

Keyboard input: a window may respond to keyboard events. Each such event causes the KeyboardFunction to be evaluated. For certain special keys constants of type KeyCode are provided in deltaSystem.dcl.

```
::    KeyboardState    :==   (!KeyCode, !KeyState, !Modifiers)
::    KeyCode          :==   Char
::    KeyState         =     KeyUp    │  KeyDown │  KeyStillDown
```

Mouse input: windows and controls may respond to mouse events. Each such event causes the MouseFunction to be evaluated.

```
:: MouseState          :==   (!MousePosition, !ButtonState, !Modifiers)
:: MousePosition       :==   (!Int, !Int)
:: ButtonState = ButtonUp │ ButtonDown │ ButtonDoubleDown │ ButtonTripleDown │ ButtonStillDown
```

For each modifier or meta-key (Shift, Option (Alternate), Command, Control) a boolean in Modifiers indicates whether it was pressed (TRUE) or not (FALSE). On keyboards that have no Command key both the third and the fourth boolean in Modifiers become TRUE when Control is pressed.

```
::    Modifiers        :==   (!Bool, !Bool, !Bool, !Bool)
```

Other common types:

```
::    ItemTitle        :==   String
::    SelectState      =     Able │ Unable
::    MarkState        =     Mark │ NoMark
::    PictureDomain    :==   Rectangle
```

3.3 Operations on the timer device (deltaTimer)

```
definition module deltaTimer
import deltaIOSystem, deltaEventIO

TicksPerSecond   :==   …      // system dependent

::    CurrentTime :== (!Int, !Int, !Int)
```

(hours (0–23), minutes (0–59), seconds (0–59))

```
::    CurrentDate :== (!Int, !Int, !Int, !Int)
```

(year, month (1–12), day (1–31), day of week (1–7, Sunday=1, Saturday=7))

```
OpenTimer  ::   !(TimerDef s io)    !(IOState s)  ->    IOState s
CloseTimer ::   !TimerId            !(IOState s)  ->    IOState s
```

Open (install) a new timer, close (remove) an existing timer.

```
EnableTimer   ::   !TimerId   !(IOState s)  ->    IOState s
DisableTimer  ::   !TimerId   !(IOState s)  ->    IOState s
ChangeTimerFunction:: !TimerId   !(TimerFunction s (IOState s)) !(IOState s)  ->    IOState s
SetTimerInterval       ::   !TimerId   !TimerInterval   !(IOState s)  ->    IOState s
GetTimerBlinkInterval  ::                             !(IOState s)  ->(!TimerInterval, !IOState s)
```

Enable, disable, change the TimerFunction and TimerInterval of a Timer. GetTimerBlinkInterval returns the number of ticks that should pass between a blink of the cursor (also called caret time).

```
Wait    ::   !TimerInterval   x    -> x
UWait   ::   !TimerInterval   * x  -> * x
```

Delay the evaluation of the second argument for a certain TimerInterval.

```
GetCurrentTime   ::   !(IOState s)  ->    (!CurrentTime,    !IOState s)
GetCurrentDate   ::   !(IOState s)  ->    (!CurrentDate,    !IOState s)
```

GetCurrentTime / GetCurrentDate return the current time and date.

3.4 Operations on menus (deltaMenu)

```
definition module deltaMenu
import deltaIOSystem, deltaEventIO
```

Menu operations on unknown MenuIds/MenuItemIds are ignored.

```
EnableMenuSystem    ::   !(IOState s)  ->    IOState s
DisableMenuSystem   ::   !(IOState s)  ->    IOState s
```

Enable and disable the complete menu system. Enabling the menu system will make the previously enabled menus and menu items selectable again. Operations on a disabled menu system take effect when the menu system is re-enabled.

```
EnableMenus   ::   ![MenuId] !(IOState s)   ->    IOState s
DisableMenus  ::   ![MenuId] !(IOState s)   ->    IOState s
```

Enable, disable menus. Disabling a menu causes its contents to be unselectable. Enabling a disabled menu with partially selectable contents causes the previously selectable items to become selectable again.

```
InsertMenuItems:: !MenuItemGroupId !Int ![MenuElement s (IOState s)] !(IOState s) -> IOState s
AppendMenuItems:: !MenuItemGroupId !Int ![MenuElement s (IOState s)] !(IOState s) -> IOState s
RemoveMenuItems        ::   ![MenuItemId]            !(IOState s)  ->    IOState s
RemoveMenuGroupItems   ::   !MenuItemGroupId ![Int]  !(IOState s)  ->    IOState s
```

Addition, removal of MenuItems in MenuItemGroups. InsertMenuItems inserts menu items before the item with the specified index, AppendMenuItems inserts them after that item. Items are numbered starting from one. Indices smaller than one or greater than the number of elements cause the elements to be inserted, respectively, before the first and after the last item in the group. Only (Check)MenuItems and MenuSeparators are added to MenuItemGroups. RemoveMenuItems only works on items that are in a MenuItemGroup. RemoveMenuGroupItems removes the items with the specified indices (counting from one) from the specified MenuItemGroup.

```
SelectMenuRadioItem    ::   !MenuItemId !(IOState s)    ->    IOState s
```

Select a MenuRadioItem: the mark will move from the currently selected item in the group to the item with the specified id.

```
EnableMenuItems       ::   ![MenuItemId]          !(IOState s)  ->   IOState s
DisableMenuItems      ::   ![MenuItemId]          !(IOState s)  ->   IOState s
MarkMenuItems         ::   ![MenuItemId]          !(IOState s)  ->   IOState s
UnmarkMenuItems       ::   ![MenuItemId]          !(IOState s)  ->   IOState s
ChangeMenuItemTitles  ::   ![(MenuItemId, String)]  !(IOState s)  ->   IOState s
ChangeMenuItemFunctions:: ![(MenuItemId, MenuFunction s (IOState s))] !(IOState s)->IOState s
```

Enable, disable, mark, unmark, change titles and functions of MenuElements.

3.5 Operations on windows (deltaWindow)

```
definition module deltaWindow
import deltaIOSystem, deltaEventIO, deltaPicture
```

Functions applied on non-existing windows or unknown WindowIds are ignored. Functions that operate on the active (or frontmost) window have the same effect as those operating on a (list of) WindowId(s) but are (slightly) faster.

```
OpenWindows        ::   ![WindowDef s (IOState s)] !(IOState s)   ->   IOState s
```

The windows are opened in the same order as specified in the list of WindowDefinitions. If one of these windows has the same WindowId as a window that is already open, then this window is not opened. Each new window is always the active window and is placed in front of all existing windows.

```
CloseWindows       ::   ![WindowId] !(IOState s) ->   IOState s
```

The windows are closed in the same order as specified in the list.

```
CloseActiveWindow  ::   !(IOState s)  ->   IOState s
```

The active window is closed.

```
GetActiveWindow    ::   !(IOState s)  ->   (!Bool, !WindowId, !IOState s)
```

Returns the id of the active window. If there is no active window the boolean will be FALSE.

```
ActivateWindow     ::   !WindowId !(IOState s)   ->   IOState s
```

Activates the indicated window. No effect, if the window is already active.

```
ChangeUpdateFunction        ::   !WindowId  !(UpdateFunction s) !(IOState s)-> IOState s
ChangeActiveUpdateFunction  ::              !(UpdateFunction s) !(IOState s) -> IOState s
```

Change the update function of the indicated window.

```
ChangeWindowTitle        ::   !WindowId  !WindowTitle !(IOState s)   ->   IOState s
ChangeActiveWindowTitle  ::              !WindowTitle !(IOState s)   ->   IOState s
```

Change the window title of the indicated window.

```
ChangeWindowCursor        ::   !WindowId  !CursorShape !(IOState s)   ->   IOState s
ChangeActiveWindowCursor ::               !CursorShape !(IOState s)   ->   IOState s
```

Change the local cursor shape of a window. When the mouse pointer moves over the content region of the window the cursor will take the indicated shape.

```
::   ScrollBarChange
   =    ChangeThumbs Int Int   // set new horizontal and vertical thumb values
   |    ChangeScrolls Int Int  // set new horizontal and vertical scroll values
   |    ChangeHThumb Int       // set new horizontal thumb value
   |    ChangeVThumb Int       // set new vertical thumb value
   |    ChangeHScroll Int      // set new horizontal scroll value
   |    ChangeVScroll Int      // set new vertical scroll value
   |    ChangeHBar Int Int     // set new horizontal thumb and scroll values
```

```
|     ChangeVBar Int Int  `  // set new vertical thumb and scroll values
```

```
ChangeScrollBar        ::   !WindowId  !ScrollBarChange !s !(IOState s) -> (!s, !IOState s)
ChangeActiveScrollBar  ::              !ScrollBarChange !s !(IOState s) -> (!s, !IOState s)
```

Change the scroll bar(s) of a window according to the ScrollBarChange.

```
ChangePictureDomain         ::   !WindowId  !PictureDomain !s !(IOState s) -> (!s, !IOState s)
ChangeActivePictureDomain   ::              !PictureDomain !s !(IOState s) -> (!s, !IOState s)
```

Change the PictureDomain of the indicated window. The settings of the scroll bars are automatically adjusted. Windows will be resized in cases where the size of the new PictureDomain is smaller than the current size of the window.

```
::   DrawInWindow !WindowId   ![DrawFunction] !(IOState s)   ->   IOState s
::   DrawInActiveWindow       ![DrawFunction] !(IOState s)   ->   IOState s
```

Apply the list of DrawFunctions (see deltaPicture.dcl) to the Picture of the window in the given order.

```
::   DrawInWindowFrame !WindowId !(UpdateFunction s) !s !(IOState s)
                                                              -> (!s, !IOState s)
::   DrawInActiveWindowFrame !(UpdateFunction s) !s !(IOState s)-> (!s,!IOState s)
```

The UpdateFunction has a list of visible rectangles as parameter. Using this list it is possible to return a list of drawing functions that only draw in the visible part of the window.

```
WindowGetFrame         ::   !WindowId  !(IOState s)  ->   (!PictureDomain, !IOState s)
ActiveWindowGetFrame   ::              !(IOState s)  ->   (!PictureDomain, !IOState s)
```

Return the visible part of the Picture of the indicated window in terms of the PictureDomain. Returns ((0, 0), (0, 0)), if the indicated window does not exist.

```
EnableKeyboard            ::   !WindowId  !(IOState s)  ->   IOState s
DisableKeyboard           ::   !WindowId  !(IOState s)  ->   IOState s
EnableActiveKeyboard      ::              !(IOState s)  ->   IOState s
DisableActiveKeyboard     ::              !(IOState s)  ->   IOState s
ChangeKeyboardFunction :: !WindowId  !(KeyboardFunction s (IOState s)) !(IOState s) -> IOState s
ChangeActiveKeyboardFunction ::      !(KeyboardFunction s (IOState s)) !(IOState s) -> IOState s
```

Enable, disable, change KeyboardFunction of a window.

```
EnableMouse            ::   !WindowId  !(IOState s)  ->   IOState s
DisableMouse           ::   !WindowId  !(IOState s)  ->   IOState s
EnableActiveMouse      ::              !(IOState s)  ->   IOState s
DisableActiveMouse     ::              !(IOState s)  ->   IOState s
ChangeMouseFunction    ::   !WindowId  !(MouseFunction s (IOState s)) !(IOState s) -> IOState s
ChangeActiveMouseFunction ::           !(MouseFunction s (IOState s)) !(IOState s) -> IOState s
```

Enable, disable, change MouseFunction of a window.

3.6 Operations on dialogs (deltaDialog)

```
definition module deltaDialog
import deltaIOSystem, deltaEventIO
```

Functions applied on non-existent dialogs or unknown ids are ignored.

```
::   DialogChange s :== (DialogState s (IOState s)) -> (DialogState s (IOState s))
```

OpenDialog opens a PropertyDialog or CommandDialog as a modeless dialog.

```
OpenDialog :: !(DialogDef s (IOState s)) !(IOState s) -> IOState s
```

OpenModalDialog opens a CommandDialog as a modal dialog. The function terminates when the dialog is closed (by means of Close(Active)Dialog). Attempts to open property dialogs with this function are ignored.

```
OpenModalDialog ::  !(DialogDef s (IOState s)) !s !(IOState s) -> (!s, !IOState s)
```

Close(Active)Dialog closes the indicated dialog.

```
CloseDialog         ::    !DialogId  !(IOState s) -> IOState s
CloseActiveDialog   ::               !(IOState s) -> IOState s
```

OpenNotice opens a notice and returns the id of the selected notice button.

```
OpenNotice :: !NoticeDef !(IOState s) -> (!NoticeButtonId, !IOState s)
```

A Beep is the simplest kind of notice.

```
Beep ::  !(IOState s) -> IOState s
```

With the following functions the state of dialog items can be changed. They have no effect when an id is specified for an item for which the state change is invalid.

```
EnableDialogItems      :: ![DialogItemId]
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
DisableDialogItems     :: ![DialogItemId]
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
MarkCheckBoxes         :: ![DialogItemId]
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
UnmarkCheckBoxes       :: ![DialogItemId]
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
SelectDialogRadioItem  :: !DialogItemId
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeEditText         :: !DialogItemId !String
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeDynamicText      :: !DialogItemId !String
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeIconLook         :: !DialogItemId !IconLook
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
```

Functions to change state, look and feel (behaviour) of Controls. When the id is not the id of a Control the functions have no effect.

```
ChangeControlState  ::    !DialogItemId !ControlState
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeControlLook   ::    !DialogItemId !ControlLook
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeControlFeel   ::    !DialogItemId !ControlFeel
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
```

Functions to change the functions related to dialog items.

```
ChangeDialogFunction   ::    !DialogItemId !(DialogFunction s (IOState s))
                                !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeButtonFunction   ::    !DialogItemId !(ButtonFunction s (IOState s))
                                !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeSetFunction      ::    !(SetFunction s (IOState s))
                                !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeResetFunction    ::    !(ResetFunction s (IOState s))
                                !(DialogState s (IOState s)) -> DialogState s (IOState s)
```

Get(Active)DialogInfo returns the current settings (DialogInfo) of the indicated dialog. The boolean indicates whether the indicated dialog is open. When it is FALSE a dummy DialogInfo is returned.

```
GetDialogInfo          ::    !DialogId  !(IOState s)  ->    (!Bool, !DialogInfo, !IOState s)
GetActiveDialogInfo    ::               !(IOState s)  ->    (!Bool, !DialogInfo, !IOState s)
```

DialogStateGetDialogInfo returns the DialogInfo corresponding to a DialogState.

```
DialogStateGetDialogInfo ::    !(DialogState s (IOState s))
                                                    ->    (!DialogInfo, !DialogState s (IOState s))
```

The following functions return the current settings of certain dialog items (those that can be changed by the user). When the corresponding item does not exist a run-time error occurs. The id passed to GetSelectedRadioItemId must be the id of a DialogPopUp or a group of RadioButtons. The function CheckBoxesMarked returns the settings of a group of CheckBoxes. The id passed to it must be the id of such a group.

```
GetEditText             ::   !DialogItemId !DialogInfo  ->  String
GetSelectedRadioItemId  ::   !DialogItemId !DialogInfo  ->  DialogItemId
CheckBoxesMarked        ::   !DialogItemId !DialogInfo  ->  [(DialogItemId, Bool)]
CheckBoxMarked          ::   !DialogItemId !DialogInfo  ->  Bool
GetControlState         ::   !DialogItemId !DialogInfo  ->  ControlState
```

ChangeDialog can be used to modify open dialogs.

```
ChangeDialog  ::   !DialogId  ![DialogChange s]  !(IOState s)  ->   IOState s
```

3.7 The file selector dialogs (deltaFileSelect)

```
definition module deltaFileSelect
import deltaFile, deltaEventIO
```

With the functions defined in this module, standard file selector dialogs can be opened, which provide a user-friendly way to select input or output files. The lay-out of these dialogs depends on the operating system.

```
SelectInputFile :: !Files !(IOState s)    -> (!Bool, !String, !Files, !IOState s)
```

Opens a dialog in which the user can traverse the file system to indicate an existing file. The Boolean result indicates whether the user has pressed the Open button (TRUE) or the Cancel button (FALSE). The String result is the complete path name of the selected file. If the Cancel button has been pressed, an empty string will be returned.

```
SelectOutputFile:: !String !String !Files !(IOState s) ->  (!Bool, !String, !Files, !IOState s)
```

Opens a dialog in which the user can specify the name of a file that should be created in a certain directory. The first parameter is the prompt of the dialog (default: "Save As:"). The second parameter is the default file name. The Boolean result indicates whether the user has pressed the Save button (TRUE) or the Cancel button (FALSE). The String result is the complete path name of the selected file. If the Cancel button has been pressed, an empty string will be returned. When a file with the indicated name already exists in the indicated directory a confirm dialog will be opened after pressing Save.

3.8 Predefined Controls (deltaControls)

```
definition module deltaControls
import deltaIOSystem, deltaEventIO
```

General scrolling list and slider bar definition. These predefined dialog items are implemented entirely in Concurrent Clean as a user-defined Control.

```
::    NrVisible        :==   Int
::    SliderDirection  =     Horizontal | Vertical
::    SliderPos        :==   Int
::    SliderMax        :==   Int
```

A ScrollingList is defined by the following attributes:

- Id, ItemPos and SelectState (like other dialog items).
- The minimum width of the scrolling list (Measure).
  This attribute is important only when ChangeScrollingList is used. Use a minimum width of zero when the scrolling list is not changed.
- The number of items that is visible in the list (NrVisible).
- The item that is initially selected (ItemTitle).

- The list of items ([ItemTitle]).
- A DialogFunction that is called whenever a new item is selected.

The function ScrollingList returns a DialogItem (a Control) that can be used in any dialog definition.

```
ScrollingList ::    !DialogItemId !ItemPos !Measure !SelectState !NrVisible !ItemTitle
              ![ItemTitle] !(DialogFunction s (IOState s))    ->    DialogItem s (IOState s)
```

With ChangeScrollingList the items in the scrolling list can be changed. Its arguments are the id of the scrolling list, the newly selected item and the new list of items. When the id is not the id of a ScrollingList a run-time error is generated.

```
ChangeScrollingList ::  !DialogItemId !ItemTitle ![ItemTitle]
                         !(DialogState s (IOState s))   ->    DialogState s (IOState s)
```

GetScrollingListItem returns the currently selected item in the scrolling list with the indicated id from the DialogInfo parameter. When the id is not the id of a ScrollingList a run-time error is generated.

```
GetScrollingListItem   ::    !DialogItemId !DialogInfo   ->    ItemTitle
```

A SliderBar is defined by the following attributes:

- Id, ItemPos and SelectState, like other DialogItems.
- SliderDirection: Horizontal or Vertical.
- SliderPos: the initial position of the slider. This position is always adjusted between 0 and SliderMax.
- SliderMax: the slider can take on positions between 0 and SliderMax.
- A DialogFunction that is called whenever the slider moves.

```
SliderBar ::  !DialogItemId !ItemPos !SelectState !SliderDirection !SliderPos
         !SliderMax !(DialogFunction s (IOState s))   ->    DialogItem s (IOState s)
```

ChangeSliderBar moves the slider of the indicated slider bar to the new position. The position is always adjusted between 0 and SliderMax.

```
ChangeSliderBar     ::    !DialogItemId !SliderPos !(DialogState s (IOState s))
                                                    ->    DialogState s (IOState s)
```

GetSliderPosition returns the current slider position of the slider bar with the indicated id from the DialogInfo parameter. When the id is not the id of a SliderBar a run-time error is generated.

```
GetSliderPosition   ::    !DialogItemId !DialogInfo   ->    SliderPos
```

3.9 Miscellaneous operations (deltaIOState)

```
definition module deltaIOState
import deltaIOSystem, deltaEventIO
```

```
SetGlobalCursor        ::    !CursorShape  !(IOState s)  ->    IOState s
```

Sets the shape of the cursor (mouse pointer) globally. This shape overrules the local cursor shapes of windows. Attempts to set the global cursor to HiddenCursor are ignored.

```
ResetCursor            ::    !(IOState s)  ->    IOState s
```

Undoes the effect of SetGlobalCursor. It resets the cursor to the standard shape outside windows with a local cursor shape and to the local shape inside such windows.

```
ObscureCursor          ::    !(IOState s)  ->    IOState s
```

Hides the cursor until the next time that the mouse is moved.

```
SetDoubleDownDistance  ::    !Int  !(IOState s)  ->    IOState s
```

Set the maximum distance (in pixels) between two mouse clicks such that they will be treated as a ButtonDoubleDown or ButtonTripleDown.

3.10 Operations on pictures (deltaPicture)

```
definition module deltaPicture
import deltaFont

::    * Picture

::    DrawFunction  :== Picture -> Picture
```

The predefined figures that can be drawn:

```
::    Point            :==   (!Int, !Int)
::    Line             :==   (!Point, !Point)
::    Curve            :==   (!Oval, !Int, !Int)
::    Rectangle        :==   (!Point, !Point)
::    RoundRectangle   :==   (!Rectangle, !Int, !Int)
::    Oval             :==   Rectangle
::    Circle           :==   (!Point, !Int)
::    Wedge            :==   (!Oval, !Int, !Int)
::    Polygon          :==   (!Point, !PolygonShape)
::    PolygonShape     :==   [Vector]
::    Vector           :==   (!Int, !Int)
```

The pen attributes influence the way figures are drawn. The PenMode also influences the way text is drawn. The Not... modes do not work when text is drawn. When the PenMode is a Not... mode text is drawn in OrMode.

```
::    PenSize    :==    (!Int, !Int)
::    PenMode     =     CopyMode       |  OrMode         |  XorMode      |
                        ClearMode      |  NotCopyMode    |  NotOrMode    |
                        NotXorMode     |  NotClearMode   |  HiliteMode
::    PenPattern =      BlackPattern   |  DkGreyPattern  |  GreyPattern     |
                        LtGreyPattern  |  WhitePattern
```

The predefined colours:

```
::    Colour =    RGB Real Real Real               |
                  BlackColour   |  WhiteColour     |  BlueColour     |  CyanColour     |
                  RedColour     |  GreenColour     |  YellowColour   |  MagentaColour


   MinRGB   :==   0.0
   MaxRGB   :==   1.0
```

Rules setting the attributes of a Picture:

SetPenSize (w, h) sets the PenSize to w pixels wide and h pixels high.
SetPenMode sets the drawing mode of the pen.
SetPenPattern sets the pattern of the pen.
SetPenNormal sets the PenSize to (1,1), the PenMode to CopyMode and the PenPattern to BlackPattern.

```
SetPenSize    ::    !PenSize      !Picture    ->    Picture
SetPenMode    ::    !PenMode      !Picture    ->    Picture
SetPenPattern ::    !PenPattern   !Picture    ->    Picture
SetPenNormal  ::                  !Picture    ->    Picture
```

Colour: there are two types of Colour: RGB colours and basic colours. An RGB colour defines the amount of red (r), green (g) and blue (b) in a certain colour by the tuple (r,g,b). These are Real values and each of them must be between MinRGB and MaxRGB (0.0 and 1.0). The colour black is defined by (MinRGB, MinRGB, MinRGB) and white by (MaxRGB, MaxRGB, MaxRGB). Given an RGB colour, all amounts are adjusted between MinRGB and MaxRGB.

SetPenColour sets the colour of the pen.
SetBackColour sets the background colour.

```
SetPenColour  ::    !Colour !Picture    ->    Picture
SetBackColour ::    !Colour !Picture    ->    Picture
```

Fonts: the initial font of a Picture is the default font (see deltaFont.dcl).

SetFont sets a new complete Font in the Picture.
SetFontName sets a new font without changing the style or size.
SetFontStyle sets a new style without changing font or size.
SetFontSize sets a new size without changing font or style. The size is always adjusted between MinFontSize and MaxFontSize (see deltaFont.dcl).

```
SetFont       ::   !Font          !Picture   ->   Picture
SetFontName   ::   !FontName      !Picture   ->   Picture
SetFontStyle  ::   ![FontStyle]   !Picture   ->   Picture
SetFontSize   ::   !FontSize      !Picture   ->   Picture
```

PictureCharWidth (PictureStringWidth) yield the width of the given Char (String) given the current font of the Picture. PictureFontMetrics yields the FontInfo of the current font.

```
PictureCharWidth    ::   !Char   !Picture   ->   (!Int,      !Picture)
PictureStringWidth  ::   !String !Picture   ->   (!Int,      !Picture)
PictureFontMetrics  ::           !Picture   ->   (!FontInfo,   !Picture)
```

Absolute and relative pen move operations without drawing.

```
MovePenTo  ::   !Point  !Picture   ->   Picture
MovePen    ::   !Vector !Picture   ->   Picture
```

Absolute and relative pen move operations with drawing.

```
LinePenTo  ::   !Point  !Picture   ->   Picture
LinePen    ::   !Vector !Picture   ->   Picture
```

DrawChar (DrawString) draws a character (string) in the current font. The baseline of the characters is the *y*-coordinate of the pen. The new position of the pen is directly after the (last) character (of the string).

```
DrawChar   ::   !Char      !Picture   ->   Picture
DrawString ::   !String !Picture   ->   Picture
```

All following rules do not change the position of the pen after drawing.

Draw(C)Point draws the pixel (in the given colour) in the Picture.
Draw(C)Line draws the line (in the given colour) in the Picture.
Draw(C)Curve draws the curve (in the given colour) in the Picture. A Curve is a part of an Oval o starting from angle a up to angle b (both in degrees modulo 360): (o, a, b). Angles are always taken anticlockwise, starting from 3 o'clock.

```
DrawPoint   ::   !Point   !Picture   ->   Picture
DrawLine    ::   !Line    !Picture   ->   Picture
DrawCurve   ::   !Curve   !Picture   ->   Picture
DrawCPoint  ::   !Point   !Colour  !Picture   ->   Picture
DrawCLine   ::   !Line    !Colour  !Picture   ->   Picture
DrawCCurve  ::   !Curve   !Colour  !Picture   ->   Picture
```

A Rectangle is defined by two diagonal corner points.
DrawRectangle draws the edges of the rectangle.
FillRectangle draws the edges and interior of the rectangle.
EraseRectangle erases the edges and interior of the rectangle.
InvertRectangle inverts the edges and interior of the rectangle.
MoveRectangleTo moves the contents of the rectangle to a new top-left corner.
MoveRectangle moves the contents of the rectangle over the given vector.
CopyRectangleTo copies the contents of the rectangle to a new top-left corner.
CopyRectangle moves the contents of the rectangle over the given vector, but leaves the original untouched.

```
DrawRectangle    ::    !Rectangle !Picture    ->    Picture
FillRectangle    ::    !Rectangle !Picture    ->    Picture
EraseRectangle   ::    !Rectangle !Picture    ->    Picture
InvertRectangle  ::    !Rectangle !Picture    ->    Picture

MoveRectangleTo  ::    !Rectangle !Point   !Picture    ->    Picture
MoveRectangle    ::    !Rectangle !Vector  !Picture    ->    Picture
CopyRectangleTo  ::    !Rectangle !Point   !Picture    ->    Picture
CopyRectangle    ::    !Rectangle !Vector  !Picture    ->    Picture
```

Rounded corner rectangles: a RoundRectangle with enclosing Rectangle r and corner curvatures x and y is defined by the tuple (r, x, y). x and y define the horizontal and vertical diameter of the corner curves. They are always adjusted between 0 and the width (height) of r.

```
DrawRoundRectangle     ::    !RoundRectangle  !Picture    ->    Picture
FillRoundRectangle     ::    !RoundRectangle  !Picture    ->    Picture
EraseRoundRectangle    ::    !RoundRectangle  !Picture    ->    Picture
InvertRoundRectangle   ::    !RoundRectangle  !Picture    ->    Picture
```

An Oval is defined by its enclosing Rectangle.

```
DrawOval    ::    !Oval    !Picture    ->    Picture
FillOval    ::    !Oval    !Picture    ->    Picture
EraseOval   ::    !Oval    !Picture    ->    Picture
InvertOval  ::    !Oval    !Picture    ->    Picture
```

A Circle with centre c (a Point) and radius r (an Int) is defined by the tuple (c, r).

```
DrawCircle    ::    !Circle  !Picture    ->    Picture
FillCircle    ::    !Circle  !Picture    ->    Picture
EraseCircle   ::    !Circle  !Picture    ->    Picture
InvertCircle  ::    !Circle  !Picture    ->    Picture
```

A Wedge is a part of an Oval o starting from angle a up to angle b (both in degrees modulo 360): (o, a, b). Angles are taken anticlockwise, starting from 3 o'-clock.

```
DrawWedge     ::    !Wedge   !Picture    ->    Picture
FillWedge     ::    !Wedge   !Picture    ->    Picture
EraseWedge    ::    !Wedge   !Picture    ->    Picture
InvertWedge   ::    !Wedge   !Picture    ->    Picture
```

A Polygon is a figure drawn by a number of lines without taking the pen off the Picture, starting from and ending at some Point p. The PolygonShape is a list of Vectors ([v_1,...,v_n]) that defines how the Polygon is drawn: (p, [v_1,...,v_n]).

ScalePolygon scales the polygon. Non-positive scale factors are allowed.
MovePolygonTo changes the starting point into the given Point and
MovePolygon moves the starting point by the given Vector.

```
ScalePolygon  ::    !Int     !Polygon    ->    Polygon
MovePolygonTo ::    !Point   !Polygon    ->    Polygon
MovePolygon   ::    !Vector  !Polygon    ->    Polygon

DrawPolygon   ::    !Polygon    !Picture    ->    Picture
FillPolygon   ::    !Polygon    !Picture    ->    Picture
ErasePolygon  ::    !Polygon    !Picture    ->    Picture
InvertPolygon ::    !Polygon    !Picture    ->    Picture
```

3.11 Operations on fonts (deltaFont)

```
definition module deltaFont

::    Font

::    FontName    :==   String
::    FontStyle   :==   String
::    FontSize    :==   Int
::    FontInfo    :==   (!Int, !Int, !Int, !Int)
```

```
MinFontSize   :==   …         // system dependent
MaxFontSize   :==   …         // system dependent
```

SelectFont creates the font as specified by the name, the stylistic variations and size. The size is always adjusted between MinFontSize and MaxFontSize. The Boolean result is TRUE if the font is available and need not be scaled. In cases where the font is not available, the default font is chosen in the specified style and size.

```
SelectFont    ::    !FontName ![FontStyle] !FontSize  ->    (!Bool, !Font)
```

DefaultFont returns the default font, specified by name, style and size.

```
DefaultFont   ::    (!FontName, ![FontStyle], !FontSize)
```

FontNames returns the FontNames of all available fonts.
FontStyles returns the available FontStyles for a certain font.
FontSizes returns all FontSizes of a font that are available without scaling.
In cases where the font is not available, the styles or sizes of the default font are returned.

```
FontNames   ::    [FontName]
FontStyles  ::    !FontName   ->    [FontStyle]
FontSizes   ::    !FontName   ->    [FontSize]
```

FontCharWidth(s) and FontStringWidth(s) return the width(s) in terms of pixels of given character(s) or string(s) in a certain Font.

```
FontCharWidth       ::    !Char       !Font    ->    Int
FontCharWidths      ::    ![Char]     !Font    ->    [Int]
FontStringWidth     ::    !String     !Font    ->    Int
FontStringWidths    ::    ![String]   !Font    ->    [Int]
```

FontMetrics yields the FontInfo in terms of pixels of a given Font. The FontInfo is a 4-tuple (ascent, descent, maxwidth, leading), which defines the font metrics:
- ascent is the maximum height of a character measured from the base line.
- descent is the maximum depth of a character measured from the base line.
- maxwidth is the width of the widest character.
- leading is the vertical distance between two lines of the same font.

The full height of a line is the sum of the ascent, descent and leading.

```
FontMetrics   ::    !Font    ->    FontInfo
```

3.12 System-dependent constants and functions (deltaSystem)

definition module **deltaSystem**

Keyboard constants (of type KeyCode):

```
UpKey       :== ...              BackSpKey   :== ...
DownKey     :== ...              DelKey      :== ...
LeftKey     :== ...              TabKey      :== ...
RightKey    :== ...              ReturnKey   :== ...
PgUpKey     :== ...              EnterKey    :== ...
PgDownKey   :== ...              EscapeKey   :== ...
BeginKey    :== ...              HelpKey     :== ...
EndKey      :== ...
```

Separator between directory and filenames in a pathname (of type Char):

```
DirSeparator  :== ...
```

Constants to check which of the meta-keys (modifiers) is down (of type Modifiers):

```
ShiftOnly      :== (TRUE,FALSE,FALSE,FALSE)
OptionOnly     :== (FALSE,TRUE,FALSE,FALSE)
CommandOnly    :== ... // depends on whether the keyboard of the system
ControlOnly    :== ... // has a Command key.
```

The minimum and maximum sizes of a picture domain (of type Int):

```
MinPictureDomain    :== ...
MaxPictureDomain    :== ...
```

The functions HomePath and ApplicationPath prefix the filename given to them with the full pathnames of the 'home' and 'application' directory. These functions have been added for compatibility with the Sun version of the Clean system. In the 'home' directory settings-files (containing preferences, options etc.) should be stored. In the 'application' directory (i.e. the directory in which the application resides) files that are used read-only by the application (such as help files) should be stored. On the Macintosh these functions just return the filename given to them, which means that the file will be stored in the same folder as the application.

```
HomePath         ::    !String  -> String
ApplicationPath  ::    !String  -> String
```

Functions to retrieve the horizontal and vertical screen resolution.

```
MMToHorPixels    ::    !Real    ->    Int
MMToVerPixels    ::    !Real    ->    Int
InchToHorPixels  ::    !Real    ->    Int
InchToVerPixels  ::    !Real    ->    Int
```

The maximum width and height for the indicated type of window such that the window will fit on the screen. For FixedWindows this is the maximum size of the PictureDomain such that the window does not become a ScrollWindow.

```
MaxScrollWindowSize    ::    (!Int, !Int)
MaxFixedWindowSize     ::    (!Int, !Int)
```