# A

# Clean 1.1 syntax

**A.1 Clean program**
**A.2 Function definition**
**A.3 Graph definition and expression**
**A.4 Macro definition**
**A.5 Type definition**
**A.6 Class definition**
**A.7 Symbols**
**A.8 Identifiers**
**A.9 Denotations**

In this chapter the context-free syntax of Clean is given. In Section A.1 the construction of a Clean program out of definition and implementation modules is given. Hereafter the syntax for, respectively, defining functions (Section A.2), graphs (Section A.3,A.4), macros (Section A.4) and types (Section A.5) is presented. Overloading is treated in Section A.6. These sections have some production rules in common which are collected in Section A.7,A.8 and A.9.
Notice that the lay-out rule (see Section 3.6) 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 can be found in Section 3.4.

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 10 pts courier** |
| 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 (see also Section 3.4) |
| ~ | is used for concatenation of notions |
| {notion}^str | means the longest expression not containing the string str |

## A.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
              | ClassDef
              | TypeClassInstanceExportDef
              | FunctionTypeDef
              | MacroDef
ImplModDef    = TypeDef
              | ClassDef
              | FunctionDef
              | GraphDef
              | MacroDef
```

## A.2 Function definition

```
FunctionDef   = [FunctionTypeDef] DefOfFunction

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

DefOfFunction = {FunctionAltDef}+
FunctionAltDef
              = FunctionSymb {Pattern}
                {[| Guard] =[>] FunctionBody}+
                [LocalFunctionAltDefs]

Pattern       = [Variable =:] BrackPattern
BrackPattern  = ConstructorSymbol
              | PatternVariable
              | BasicValuePattern
              | ListPattern
              | TuplePattern
              | RecordPattern
              | ArrayPattern
              | (GraphPattern)
GraphPattern  = ConstructorSymbol {Pattern}
              | GraphPattern ConstructorSymb GraphPattern
              | Pattern

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}
              | StringDenot

Guard         = BooleanExpr
BooleanExpr   = GraphExpr

FunctionBody  = {StrictLet}+
                RootExpression ;
                [LocalFunctionDefs]

StrictLet     = let! { {GraphDef}+ } in
```

```
RootExpression          = GraphExpr

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

## A.3 Graph definition and expression

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

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

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
ListExpr                = GraphExpr
ArrayExpr               = GraphExpr

Tuple                   = (GraphExpr,{GraphExpr}-list)

Record                  = {[TypeSymb |][RecordExpr &][{FieldSymbol = GraphExpr}-list]}
RecordSelection         = RecordExpr.[TypeSymb.]FieldSymb
RecordExpr              = GraphExpr

Array                   = {{GraphExpr}-list}
                        | {ArrayExpr & [{ArrayIndex = GraphExpr}-list] [\\ {Qualifier}-list]}
                        | {[ArrayExpr &] GraphExpr \\ {Qualifier}-list}
                        | StringDenot
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] *}
ProcIdExpr              = GraphExpr
```

## A.5 Macro definition

```
MacroDef                = FunctionSymb {Variable} :== FunctionBody #
                          [LocalFunctionAltDefs]
                        | (FunctionSymb) [Fix][Prec] #
                          (FunctionSymb) {Variable} :== FunctionBody #
                          [LocalFunctionAltDefs]
```

## A.6 Type definition

```
TypeDef                 = AlgebraicTypeDef
                        | RecordTypeDef
                        | SynonymTypeDef

AlgebraicTypeDef        = ::TypeLhs = ConstructorDef {|ConstructorDef} #
RecordTypeDef           = ::TypeLhs = {{FieldSymbol :: [Strict] Type}-list} #
SynonymTypeDef          = ::TypeLhs :== Type #
AbstractTypeDef         = ::TypeLhs #

TypeLhs                 = [*]TypeConstructor {E.[*] TypeVariable} {[*] TypeVariable}
TypeConstructor         = TypeSymb

TypeVarDef              = [E.] [UnqTypeAttrib] TypeVariable
UnqTypeAttrib           = *
                        | UniqueTypeVariable:
                        | .

ConstructorDef          = ConstructorSymb {[Strict] BrackType}
                        | (ConstructorSymb) [Fix][Prec] {[Strict] BrackType}
Fix                     = infixl
                        | infixr
                        | infix
Prec                    = Digit
Strict                  = !

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

TypeConstructor         = TypeSymb
                        | []
                        | ({,}+)
                        | {}
                        | {!}
                        | {#}
                        | (->)
BasicType               = Int
                        | Real
                        | Char
                        | Bool
PredefAbstrType         = World
                        | File
                        | ProcId
                        | Void
ListType                = [Type]
```

```
TupleType   = (([Strict] Type,{[Strict] Type}-list)
ArrayType   = {Type}
            | {!Type}
            | {#BasicType}
ArrowType   = ({BrackType}+ -> Type)
```

## A.6 Class definition

```
ClassDef    = TypeClassDef
            | TypeClassInstanceDef

TypeClassDef = class ClassSymb TypeVariable [ClassContext]
               [[where] {{ {ClassMemberDef}+ }}]
             | class FunctionSymb TypeVariable:: FunctionType#
             | class (FunctionSymb) [Fix][Prec] TypeVariable:: FunctionType#

ClassMemberDef = FunctionTypeDef
               [MacroDef]#

TypeClassInstanceDef | instance ClassSymb [[default ] BrackType[ClassContext]]
                       [[where] {{ {DefOfFunction}+ }}]

TypeClassInstanceExportDef
             = export ClassSymb BasicType-list#
```

## A.7 Symbols

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

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
ClassSymb       = LowerCaseId | UpperCaseId | FunnyId
```

## A.8 Identifiers

```
LowerCaseId = LowerCaseChar~{IdChar}
UpperCaseId = UpperCaseChar~{IdChar}
.ib.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
              | _ | `
```



## A.9 Denotations

```
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

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

Digit        = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
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

CharDel      = '
StringDel    = "
```

# B

# Standard Environment version 1.1

## B.1 Cleans' Standard Environment

The **standard library of Clean** not only contains the well-known functions for arithmetic and manipulation of lists, arrays and the like, but there is also a lot of support for file I/O and window based I/O. The new Clean 1.0 I/O library makes the specification and combination of interactive programs possible on a very high level of abstraction. Notice that this new I/O library is not yet available on all platforms. The old Clean 0.8 library has been converted to Clean 1.0 syntax and is available on all platforms.
In the Clean library there are many modules. Modules which names start with `Std...` are the topmost interface modules of the library to be used by Clean programmers. In this appendix we have printed the names of types, constructors, functions, type-classes in bold to assist the reader in finding a definition.
The types of the funtions in Std... are as general as possible and therefore include uniqueness type information (the funny dots and `u:` etc in the types). For reasons of efficiency also the strictness information derived by the strictness analyser is exported (the exclamation marks in the types). For most programmers this information will often be of no importance, and if this is the case, simply ignore these funny marks.

### B.1 Cleans' Standard Environment

```
definition module StdEnv

import
    StdOverloaded,
    StdClass,

    StdBool,
    StdInt,
    StdReal,
    StdChar,

    StdList,
    StdCharList,
    StdTuple,
    StdArray,
    StdString,
    StdFunc,
    StdMisc,

    StdFile,

    StdEnum
```



### B.1.1 StdOverloaded: predefined overloaded operations

```
definition module StdOverloaded

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

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

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

class (==) infix  4  a  :: !a  !a -> Bool         //  True if arg1 is equal to arg2

class (<)  infix  4  a  :: !a  !a -> Bool         //  True if arg1 is less than arg2

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

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

class length         m  :: !(m a) -> Int          //  Number of elements in arg
                                                  //  used for list like structures (linear time)

class (%)    infixl 9 a  :: !a !(!Int,!Int)    -> a   //  Slice a part from arg1

class (+++) infixr 5 a  :: !a  !a -> a            //  Append args
```

### B.1.2 StdClass: predefined classes

```
definition module StdClass

import StdOverloaded
from StdBool import not

class PlusMin a  | +, -, zero a

class MultDiv a  | *, /, one a

class Arith a    | PlusMin, MultDiv, abs, sign, ~ a

class IncDec a   | + , - , one , zero a
where
    inc :: !a -> a        | + , one a
    inc x :== x + one

    dec :: !a -> a        | - , one a
    dec x :== x - one

class Enum a     | <, IncDec a

class Eq a       | == a
where
    (<>) infix  4:: !a   !a  ->  Bool | Eq a
    (<>) x y :== not (x == y)

class Ord a      | < a
where
    (>)  infix  4:: !a !a -> Bool | Ord a
    (>) x y    :== y < x

    (<=) infix  4::!a !a -> Bool | Ord a
    (<=) x y   :== not (y<x)
```

```
(>=) infix  4::!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
```

### B.1.3 StdBool: operations on Booleans

```
system module StdBool

import StdOverloaded

instance ==          Bool

instance toBool      Bool
instance toString    Bool

instance fromBool    Bool
instance fromBool    {#Char}            //  String :== {#Char}

//  Additional Logical Operators:

not             :: !Bool      -> Bool   //  Not arg1
(||)   infixr 2 :: !Bool Bool -> Bool   //  Conditional or  of arg1 and arg2
(&&)   infixr 3 :: !Bool Bool -> Bool   //  Conditional and of arg1 and arg2

//  Miscellaneous:

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

### B.1.4 StdInt: operations on Integers

```
system module StdInt

import StdOverloaded

instance +           Int
instance -           Int
instance zero        Int

instance *           Int
instance /           Int
instance one         Int

instance ^           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     {#Char}                //  String :== {#Char}

// 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 6    :: !Int !Int   ->  Int    //  Bitwise Or of arg1 and arg2
(bitand)  infix 6    :: !Int !Int   ->  Int    //  Bitwise And of arg1 and arg2
(bitxor)  infix 6    :: !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
```

### B.1.5 StdReal: operations on Reals

```
system module StdReal

import StdOverloaded

instance +           Real
instance -           Real
instance zero        Real

instance *           Real
instance /           Real
instance one         Real

instance ^           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    {#Char}   //  String :== {#Char}

//  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    //  Cconvert Real into Int by taking entier
```

### B.1.6 StdChar: operations on Characters

```
system module StdChar

import StdOverloaded

instance +           Char
instance -           Char
instance zero        Char
instance one         Char

instance ==          Char
```

```
instance <          Char

instance toInt      Char
instance toChar     Char
instance toString   Char

instance fromChar   Int
instance fromChar   Char
instance fromChar   {#Char}         //  String :== {#Char}

//  Additional conversions:

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
```

### B.1.7 StdList: operations on Lists

```
definition module StdList

import StdClass

instance ==   [a] | Eq a

instance <    [a] | Ord a

instance toString   [a]   | ToChar a          // Convert [e to Char] into String
instance fromString [a]   | FromChar a        // Convert String into [Char to e]

instance length  []
instance %       [a]

//  List Operators:

(!)    infixl 9   :: [.a] Int -> .a                 //  Get nth element of the list
(++)   infixr 5   :: ![.a] u:[.a] -> u:[.a]         //  Append args
flatten           :: ![.[a]] -> [a]                 //  e0 ++ e1 ++ ... ++ en
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 elem of the list
drop        :: Int !u:[.a] -> u:[.a]                //  Drop first arg1 elem from the list
takeWhile   :: (a -> .Bool) ![.a] -> .[a]           //  Take elements while pred holds
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..length list - 1]
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)...en)
foldr       :: (.a -> .(.b -> .b)) !.b ![.a] -> .b  //  op e0 (op e1(...(op r en)...)

// for efficiency reasons, foldl and folr are defined as macros,
// so that applications of these functions will be inlined !

// foldl  :: (.a -> .(.b -> .a)) !.a ![.b] -> .a  //  op(...(op (op (op r e0) e1)...en)
foldl op r l :== foldl r l
where
    foldl r []        = r
    foldl r [a:x]  = foldl (op r a) x

// foldr  :: (.a -> .(.b -> .b)) !.b ![.a] -> .b  //  op e0 (op e1(...(op r en)...)
foldr op r l :== foldr r l
where
    foldr r []        = r
    foldr r [a:x]  = op a (foldr r x)

scan        :: (a -> .(.b -> a)) a ![.b] -> .[a]    //  [r,op r e0,op (op r e0) e1,...

//  On Booleans

and         :: ![.Bool] -> Bool                     //  e0 && e1 ... && en
or          :: ![.Bool] -> Bool                     //  e0 || e1 ... || en
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

//  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 giving 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]

//  Overloaded definition of sum, product, average

sum    :: !.[a] -> a | + , zero  a            //  sum of list elements, sum [] = zero
prod   :: !.[a] -> a | * , one   a            //  product of list elements, prod [] = one
avg    :: !.[a] -> a | / , IncDec a           //  average of list elements, avg [] gives error!
```

### B.1.8 StdCharList: operations on lists of characters

```
definition module StdCharList

//  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 arg1n

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
```

### B.1.9 StdTuple: operations on Tuples

```
definition module StdTuple

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
```

## B.1.10 StdArray: operations on Arrays

```
definition module StdArray

import _SystemArray

system module _SystemArray

/*
Warning:
1) Arrays currently get a special treatment in the Clean compiler.
   This means that you shouldn't rename the functions declared here,
   and that you shouldn't make other instances of ArrayClass
2) The structure of this module will change in a future release
*/

class Array a
where

   select       :: ! .(a .e) !Int -> .e                 | select_u e
   uselect      :: ! u:(a e) !Int -> (e, ! u:(a e))     | uselect_u e

   size         :: ! .(a .e)  -> Int                    | size_u e
   usize        :: ! u:(a .e) -> (!Int, ! u:(a .e))     | usize_u e

   update       :: !* (a .e)  !Int .e -> * (a .e)       | update_u e
   createArray  :: !Int e -> * (a e)                    | createArray_u e

instance Array  {} default, {!}, {#}

class ArrayElem e | select_u, uselect_u, size_u, usize_u, update_u, createArray_u, defaultArrayvalu

// Operation on unboxed arrays

class select_u e        :: !   { #.e } !Int -> .e
class uselect_u e       :: ! u:{ # e } !Int -> (!e, ! u:{ #e })
class size_u e          :: !   { #.e } -> Int
class usize_u e         :: ! u:{ #.e } -> (!Int, ! u:{ #.e })
class update_u e        :: ! * { #.e } !Int !.e -> * { #.e }
class createArray_u e   :: !Int !e -> *{ #e }


instance select_u          a, Int, Real, Char, Bool, File
instance uselect_u         a, Int, Real, Char, Bool, File
instance size_u            a, Int, Real, Char, Bool, File
instance usize_u           a, Int, Real, Char, Bool, File
instance update_u          a, Int, Real, Char, Bool, File
instance createArray_u     a, Int, Real, Char, Bool, File

class      defaultArrayvalue e :: .e
instance   defaultArrayvalue Int, Real, Char, Bool, File, a
```

## B.1.11 StdString: operations on Strings

```
system module StdString

import StdOverloaded

:: String :== {#Char}
```



```
instance ==             {#Char}

instance <              {#Char}

instance toString       {#Char}
instance toInt          {#Char}
instance toReal         {#Char}

instance fromString     {#Char}

instance %              {#Char}

instance +++            {#Char}

//  additional operator

(:=) infixl 1 :: !String !(!Int,!Char) -> String  //  non-destructive update of the i-th element
```

## B.1.12 StdFunc: operations on polymorphic functions

```
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 x) else x
until        :: !(a -> .Bool) (a -> a) a ->  a              // until (p x) x else (f 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 :: w:(St .s .a) v:(.a -> .(St .s .b)) -> u:(St .s .b), [u <= v, u <= w]
return            :: u:a -> u:(St .s u:a)
```

## B.1.13 StdMisc: miscellaneous functions

```
system module StdMisc

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

## B.1.14 StdFile: File based I/O

```
system module StdFile

import StdString

//  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 FWriteR 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. */
```

## B.1.15 StdEnum: handling dot-dot expressions

The definitions listed in `StdEnum` are used by the Clean compiler to handle dot-dot expressions. Dot-dot expressions can be used for objects of type `Int`, `Char` and `Real`. Dot-dot expressions can also be used of objects of arbitrary user-defined types provided that the indicated classes have been instantiated for objects of that type.

```
definition module StdEnum

import _SystemEnum

/*
   This module must be imported if dotdot expressions are used

       [from .. ]        -> _from from
       [from .. to]      -> _from_to from to
       [from, then .. ]  -> _from_then from then
       [from, then .. ]  -> _from_then_to from then to
*/


system module _SystemEnum

from StdClass import Enum
from StdBool import not

from            ::  a         -> [a] | IncDec , Ord a
from_to         :: !a !a      -> [a] | Enum a
from_then       ::  a  a      -> [a] | Enum a
from_then_to    :: !a !a !a   -> [a] | Enum a

lteq a b   :== not (b < a)
minus a b  :== a - b

implementation module _SystemEnum

import StdEnv

lteq a b  :== not (b < a)
minus a b     :== a - b
```

```
from ::  a -> [a] | IncDec , Ord a
from n = [n | _from (inc n)]

from_to :: !a !a -> [a] | Enum a
from_to n e
| n <= e       = [n | _from_to (inc n) e]
| otherwise    = []

from_then ::  a  a -> [a] | Enum a
from_then n1 n2 = [n1 | _from_by n2 (n2-n1)]
where
    from_by :: a a -> [a] | Enum a
    from_by n s = [n | _from_by (n+s) s]

from_then_to : !a !a !a -> [a] | Enum a
from_then_to n1 n2 e
| n1 <= n2     = _from_by_to n1 (n2-n1) e
| otherwise    = _from_by_down_to n1 (n2-n1) e
where
    from_by_to :: !a !a !a -> [a] | Enum a
    from_by_to n s e
        | n <= e       = [n | _from_by_to (n+s) s e]
        | otherwise    = []

    from_by_down_to :: !a !a !a -> [a] | Enum a
    from_by_down_to n s e
        | n >= e       = [n | _from_by_down_to (n+s) s e]
        | otherwise    = []

from_to :: !Int !Int -> [Int]
from_to n e
| n <= e       = [n | from_to (inc n) e]
| otherwise    = []
```