PRELUDE FOR CLEAN*

SIMON János György

2003.
Department of Software Technology
University of Nijmegen

Eötvös Loránd University, Budapest
Supervisor: HORVÁTH Zoltán

*. Sponsored by ERASMUS of the European Community and the Hungarian National Science Research Grant (Project number: OTKA T037742)

User's Guide

Introduction

Haskell and Clean are both state-of-the-art lazy functional languages sharing a lot of common features, Yet there are various slight and significant differences as well. To start with, Haskell is a de-facto standard while Clean incorporates an IDE besides having some extra features. Moreover, an integrated proof system for (core) Clean is near completion. Thus, in order to obtain the best of both worlds, it would be nice if Haskell programs would also be accepted by the Clean compiler.

To make it available to import Haskell modules in a Clean program, a Haskell front-end is needed, which is being constructed by Hajnalka Hegedűs (based on the Clean compiler [3]) and Péter Diviánszky (based on parser combinators). Scanning and parsing Haskell modules will be solved by this extension of the Clean compiler.

However, we still face the fact that Clean and Haskell have different basic libraries. The first has a so-called Standard Environment while the latter uses the Standard Prelude. Despite numerous similar definitions, there are also big differences. For instance, Clean has arrays while Haskell does not. Clean offers uniqueness typing for destructive updates while Haskell requires a monadic approach. Consequently, the aim is to make a Haskell Prelude for the Clean 2.0 compiler (called *Prelude for Clean*) that can be used by the Haskell front-end.

Unfortunately, structural differences and some extra features of Haskell (e.g. deriving) disappoint the hope of importing the Haskell Prelude by the Haskell front-end. The Standard Prelude contains many 'primitives', i.e. functions and types that are not definable in Haskell and are implemented in a low-level, system-dependent way. There are also several functions that are defined in both libraries with the same names but different bodies. Therefore, the *Prelude for Clean,* written in Clean, should be such an interface for Clean users that provides all the functions and type definitions of Haskell Prelude.

All in all, the application of these two extensions, — the Haskell front-end and the Prelude for Clean —, enables us to import Haskell in a Clean module.

Using *Prelude for Clean* without the Haskell front-end.

Until the Haskell front-end will have been finished, the Prelude for Clean can be tested by *Haskell-like modules.* Although these modules are written in Clean, the functions and types of Haskell Prelude are used. These modules can be compiled by the Clean compiler.

In these modules Prelude is to be imported as a standard library: `include StdEnv` should be replaced by `include Preludes`.
For examples of Haskell-like module, see section "Samples" on page 3.

How to install and use the *Prelude for Clean* library

Prelude for Clean should be compiled by the Clean 2.0.1. system, which is available on the included CD (See "What's on CD?" on page 3.).

After installing the Clean environment for Haskell-like modules, include Prelude for Clean in the library path: from the *Environment* menu choose *Edit current...* and append the directory of the *Prelude for Clean* library to the *Paths* section.

Differences

The differences between the functions and classes of the two versions can be found in the Developer's Guide.

Samples

Several samples can be found in the **Prelude for Clean** folder. Copy the library to the hard disk and open the project file (with **prj** extension) of any of the included samples. To rebuild and run the sample, choose *Update and Run* from the Project menu or press *Ctrl-R*.

Included samples:

- numericEnum.icl demonstrates the usage of PreludeList (take, concat) and numeric enumeration functions
- Pascal.icl prints the Pascal triangle
- prgen.icl has instances generated by the help of generics
- prio.icl uses IO monads
- prtext.icl uses PreludeText
- prtext2.icl uses PreludeText and PreludeGeneric

What's on CD?

- **Acrobat Reader:** Adobe Acrobat Reader (to open Prelude for Clean.pdf)
- **Haskell**
  - **HSLibReport:** Haskell Library Report
  - **HSReport:** Haskell Report
- **Prelude for Clean**
  - **Prelude:** the implementation and definition modules of Prelude for Clean
  - sample files
- *Clean 2.0.zip*: Clean 2.0.1 environment (manuals included)
- *Prelude for Clean.pdf*: this documentation in PDF

Developer's Guide

Main approach

I tried to make this prelude reflect Haskell's Prelude as much as possible, therefore only the differences and limitations are mentioned here. For further details about Prelude and Standard Environment, see the Haskell Language Report [1] and Clean Language Report [2].

If Clean has a similar solution (same name and definition) to the Haskell version, the Clean version is used. If no equivalent function was found, the original definition of the function was adopted to Clean.

Primitives of Haskell Prelude that are not definable in Haskell, — indicated by names starting with "prim"—, are defined in a system-dependent manner in module `PreludeBuiltin`. Instance declarations that simply bind primitives to class methods are omitted. Some of the more verbose instances with obvious functionality have been left out for the sake of brevity.
These primitives are defined by the basic functions of Clean.

As the Clean compiler does not support deriving, the derived instances are written in Clean. Some of them are implemented with the help of generics (See in PreludeGeneric.icl). The deriving feature of Haskell can generate instances, yet only for the basic classes (e.g. Eq, Ord, Show...). Thus, although the Clean compiler cannot do it automatically, with the help of generics (and some small extra codes) instances can be generated for any classes in Clean.

'Layers' are used to avoid name conflicts (e.g. in PreludeList.icl). As imports are transitive in Clean, implementing functions with the same name but with a different definition would cause name conflicts.

I/O type of Haskell is implemented by state monads (See in PreludeIO.icl).

General remarks

- Clean style error messages are used instead of the Haskell ones, where it contained `Prelude.functionname:...` (e.g. : `Prelude.hd: hd of []`)
- Function names beginning with `hs__` and operators beginning with `###` are used for the Clean versions to avoid name conflicts
- For a `fun: a a -> a` function the `x 'fun' y` infix notation is not allowed in Clean
- In Haskell `0` and `1` denotes the `zero` and `one` element of Clean. So somewhere they do not only mean integers (`Int` in Clean) but also the general elements of the `Num` class (e.g. the definition of `sum` and `product`, see their implementation in PreludeList.hs)

- **Basic types**
  - Basic types cannot be defined in Clean in contrast to Haskell, where writing down the "definition" of the `Int`, for instance, is possible (See in Prelude.hs). So most of them are imported from the standard library of Clean while others are available without any imports (e.g. `Int`)
  - The definition of the `String` type is `String = [Char]` in Haskell, whereas in Clean the String is a basic type and not a list of characters, so in a Haskell file it will be parsed by the Haskell front-end as `[Char]`
  - Instead of the Haskell String notation (e.g. "`apple`") the Clean version is used (e.g. `['apple']`, which is a list of `Chars`)
  - The `Float` type of Haskell is always parsed as `Real`.
  - Clean does not support Unicode characters.
  - `Integer` and `Double` are not supported in Clean.

- **Lists**
  - Lists can be constructed in Haskell with the following notation:
    `x:xs :: [a]`
    but it should be written in Clean as follows
    `[x:xs] :: [a]`
    so the square brackets are needed
  - `x:y:xs` is a legal list pattern in Haskell that should be written in Clean as `[x:[y:xs]]`
  - The form of list comprehension in Haskell (e.g. `[x | x <- [1..10], isEven x]`) is different from Clean (e.g. `[x \\ x <- [1..10] | isEven x ]`)

- **Class members**
  - Predefinitions for class members cannot be defined in Clean as in Haskell, so the most important ones are members, and the others are macro definitions
  - In some Haskell instantiations (e.g. `instance Enum Float` in Prelude.hs) there is at least one member of the class that is redefined (though it has a predefinition). In these cases the predefined member does NOT work properly, so the usage of additional functions (e.g. `numericEnumFromThenTo` in Prelude.icl) is needed.
  - For derived instances redefining members that already have predefinitions (macro definition) is not allowed.

Prelude.icl

Standard types, classes, instances and related functions

Eq

See it in Haskell and in Clean.

The minimal complete definition is (==), instead of (==) or (/=).

Instances

instance == Trivial
`instance == Bool` is imported from StdBool
`instance == Char` is imported from StdChar
`instance == Int` is imported from StdInt
`instance == Real` is imported from StdReal
instance == (Maybe a) | == a
instance == Ordering
instance == (Either a b) | Eq a & Eq b
instance == [a] | Eq a

Ord

See it in Haskell and in Clean.

The minimal complete definition is (<), instead of (<=) or `compare`.

Instances

instance < Trivial
instance < Bool
`instance < Char` is imported from StdChar
`instance < Int` is imported from StdInt
`instance < Real` is imported from StdReal
instance < (Maybe a) | < a
instance < Ordering
instance < (Either a b) | Ord a & Ord b
instance < [a] | Ord a

Enum

See it in Haskell and in Clean.

The minimal complete definition is `toEnum, fromEnum` and `enumFromThen`, instead of `toEnum, fromEnum.`

Instances
instance Enum Trivial
instance Enum Bool
instance Enum Char
instance Enum Ordering
instance Enum Int
instance Enum Real

Bounded

See it in Haskell and in Clean.

No difference.

Instances
instance Bounded Trivial
instance Bounded Bool
instance Bounded Char
instance Bounded Ordering
instance Bounded Int
instance Bounded (a,b) | Bounded a & Bounded b
instance Bounded (a,b,c) | Bounded a & Bounded b & Bounded c

Num

See it in Haskell and in Clean.

The minimal complete definition is `+,-,*,abs,fromInt,zero,one, signum.`

As `Integer` is not supported yet, `fromInt` should be used instead of `fromInteger.`

In Haskell `zero` and `one` is not requiered for `Num` but the `sum` and `product` functions require them (See in PreludeList.hs).

`(-)` cannot be predefined.

Instances
instance Num Int
instance Num Real

Hs_Real

See it in Haskell and in Clean.

It is renamed due to a name-space bug in the Clean compiler.

Instances
instance Hs_Real Int
instance Hs_Real Real

Integral

See it in Haskell and in Clean.

As `Integer` is not supported yet the `toInteger` function yields an `Int` instead of an `Integer`.

Infix notation for class members is not available .

Though the precedence of `rem, mod` is `infixl 7` in Haskell, the equivalents in Clean are `infix 7`.

Invariants:

```
quotRem x y  =  (x `quot` y, x `rem` y)
divMod  x y  =  (x `div` y, x `mod` y)
(x `quot` y)*y + (x `rem` y) == x
(x `div`  y)*y + (x `mod` y) == x
```

Instances
instance Integral Int

Fractional

See it in Haskell and in Clean.

The minimal complete definition is `FromRational` and `(/)`, instead of `FromRational` and `((/)` or `recip`).

Instances
instance Fractional Real

Floating

See it in Haskell and in Clean.

No difference.

Instances
instance Floating Real

RealFrac

See it in Haskell and in Clean.

No difference.

`properFraction` takes a real fractional number x and returns a pair (n,f) such that x = n+f, and: n is an integral number with the same sign as x; and f is a fraction with the same type and sign as x, and with absolute value less than 1.

Instances
instance RealFrac Real

RealFloat

See it in Haskell and in Clean.

`Int` is used in the definitions of `floatRadix, decodeFloat` and `encodeFloat.`

`atan2` is used only for a briefer definition.

Instances
instance RealFloat Real

Numeric functions

In fact there are no differences in the Haskell and Clean implementations of the following: (See it in Haskell and in Clean.)

```
subtract            :: a a -> a    | Num a
even                :: !a -> Bool  | Integral a
odd                 :: !a -> Bool  | Integral a
gcd                 :: !a !a -> a  | Integral a
lcm                 :: !a !a -> a  | Integral a
(^^) infixr 8       :: !a !b -> a  | Fractional a & Integral b
realToFrac          :: !a -> b     | Hs__Real a & Fractional b
```

In Clean `(^)` is `class (^) infixr 8 a :: !a !a -> a`, whereas in Haskell it is a simple infix operator: `(^) infixr 8 :: !a !b -> a  | Num a & Integral b.`

As `Integer` is not supported yet: `fromIntegral n = fromInt (toInteger n).`

Monadic classes

In fact there are no differences in the Haskell and Clean implementations of the following: (See it in Haskell and in Clean.)

```
class  Functor
sequence            :: [m a] -> m [a]   | Monad m
sequence_           :: [m a] -> m Trivial | Monad m
mapM                :: (a -> m b) [a] -> m [b] | Monad m
mapM_               :: (a -> m b) [a] -> m Trivial | Monad m
(=<<) infixr 1      :: (a -> m b) (m a) -> m b | Monad m
```

Although in `class Monad` the `fail` function has predefinition, it is usually redefined in the instances so predefinition is removed in the Clean version.

Additional functions (like in StdFunc)

In fact there are no differences in the Haskell and Clean implementations of the following: (See it in Haskell and in Clean.)

```
id                  :: !a -> a
const               :: !a b -> a
flip                :: (a b -> c) b a -> c
```

`(.) infixr 9` is not a legal operator name in Clean so it is automatically parsed as the `o` operator of StdFunc. For correct parsing, whitespaces are needed around it in the Haskell source.

`seq :: a b -> b` evaluates its first parameter before returning the second one, which can be implemented in Clean as follows (note the `!` in the signature):

```
(seq) infixr 0 :: !a  b -> b
(seq) x y = y
```

Note the following precedences:

```
($) infixr 0        :: (a -> b)  a -> b
($!) infixr 0       :: (a -> b) !a -> b
```

Basic types and instances

Trivial

See it in Haskell and in Clean.

`()` is not a legala legal type and constructor name in Clean, so by convention `Trivial` is used instead.

Instances

instance == Trivial
instance < Trivial
instance Enum Trivial
instance Bounded Trivial
instance Show Trivial
instance Read Trivial

Bool

See it in Haskell and in Clean.

`not`, `&&` and `||` is imported from StdBool.

`otherwise` is a token in Clean while a function in Haskell.

Instances

`instance == Bool` is imported from StdBool
instance < Bool
instance Enum Bool
instance Bounded Bool

Char

See it in Haskell and in Clean.

Due to the predefinition, the result `enumFrom` yields an infinite list, though it is bounded in Haskell (See in Prelude.hs).

Instances

`instance == Char` and `instance < Char` is imported from StdChar
instance Enum Char
instance Bounded Char
instance Show Char
instance Read Char

Maybe

See it in Haskell and in Clean.

Instances

instance == (Maybe a) | == a
instance < (Maybe a) | < a
instance Functor Maybe
instance Monad Maybe
instance Show (Maybe a) | Show a
instance Read (Maybe a) | Read a

Either

See it in Haskell and in Clean.

Instances

instance == (Either a b) | Eq a & Eq b
instance < (Either a b) | Ord a & Ord b
instance Show (Either a b) | Show a & Show b
instance Read (Either a b) | Read a & Read b

IO

See "PreludeIO.icl" on page 16.

Ordering

Instances

instance == Ordering
instance < Ordering
instance Enum Ordering
instance Bounded Ordering
instance Show Ordering
instance Read Ordering

Lists

Instances

instance == [a] | Eq a
instance < [a] | Ord a
instance Functor []
instance Monad []
instance Show [a] | Show a
instance Read [a] | Read a

Tuples

While `fst` and `snd` are functions in Haskell, they have macro definitions in Clean (See in Prelude.icl).

Instances

instance == (a,b) | Eq a & Eq b
instance < (a,b) | Ord a & Ord b
instance Bounded (a,b) | Bounded a & Bounded b
instance == (a,b,c) | Eq a & Eq b & Eq c
instance < (a,b,c) | Ord a & Ord b & Ord c
instance Bounded (a,b,c) | Bounded a & Bounded b & Bounded c
instance Show (a,b) | Show a & Show b
instance Read (a,b) | Read a & Read b

Instances for standard numeric types

Each of the following instances has built-in definition in Haskell, so the Clean implementation has to be written.

Int

Instances

`instance == Int` and `instance < Int` is imported from StdInt.
instance Num Int
instance Hs_Real Int
instance Integral Int
instance Enum Int
instance Bounded Int
instance Show Int
instance Read Int

Integer

Not supported.

Real (instead of Float)

In `instance Enum Real` the `succ`, `pred`, `enumFrom`, `enumFromTo` and `enumFromThenTo` have different (thus wrong) definition from the generated predefinition. The correct definitions are available as `numericSucc`, `numericPred`, `numericEnumFrom`, `numericEnumFromTo` and `numericEnumFromThenTo`.

Instances

`instance == Real` and `instance < Real` is imported from StdReal.
instance Num Real
instance Enum Real
instance Hs_Real Real
instance Fractional Real
instance Floating Real
instance RealFrac Real
instance RealFloat Real
instance Show Real
instance Read Real

Double

Not supported.

Misc. functions

In fact there are no differences in the Haskell and Clean implementations of the following: (See it in Haskell and in Clean.)

```
curry               :: ((a, b) -> c) a b -> c
uncurry             :: (a b -> c) (a, b) -> c
until               :: !(a -> Bool) (a -> a) a -> a
asTypeOf            :: !a a -> a
undefined           :: a
```

`error` has a built-in definition, substituted by

```
error :: [Char] -> a
error msg   =  abort (toStr msg)
```

PreludeList.icl

As imports are transitive in Clean, implementing - in one module - a Haskell function with the same name but different definition would cause name conflicts, so 'Layers' are used to avoid them. Based on the similarities between Clean and Haskell, the functions can be divided into five groups:

- In fact there are no differences in the Haskell and Clean implementations of the following:

```
map             :: (a -> b) ![a] -> [b]
(++) infixr 5   :: ![a] [a] -> [a]
filter          :: (a -> Bool) [a] -> [a]
last            :: ![a] -> a
init            :: ![a] -> [a]
(!!) infixl 9   :: ![ a]  Int ->  a
foldl           :: (a b -> a) a ![b] -> a
foldr           :: (a b -> b) b ![a] -> b
iterate         :: (a -> a) a -> [a]
repeat          ::  a -> [a]
replicate       :: Int a -> [a]
takeWhile       :: (a -> Bool) ![a] -> [a]
dropWhile       :: (a -> Bool) ![a] -> [a]
span            :: (a -> Bool) ![a] -> ([a],[a])
reverse         :: ![a] -> [a]
and             :: ![Bool] -> Bool
or              :: ![Bool] -> Bool
any             :: (a -> Bool) -> ![a] -> Bool
all             :: (a -> Bool) -> ![a] -> Bool
unzip           :: [(a,b)] -> ([a],[b])
```

- if such a function does **not exist** with that name **in Clean**, it is implemented

```
null      :: [a] -> Bool                      See in PreludeListLayer.icl
foldl1    :: (a a -> a) ![a] -> a             See in PreludeListLayer.icl
foldr1    :: (a a -> a) ![a] -> a             See in PreludeListLayer.icl
scanl1    :: (a a -> a) ![a] -> [a]           See in PreludeListLayer.icl
scanr     :: (a b -> b) b ![a] -> [b]         See in PreludeListLayer.icl
scanr1    :: (a a -> a) ![a] -> [a]           See in PreludeListLayer.icl
cycle     :: [a] -> [a]                       See in PreludeListLayer.icl
break     :: (a -> Bool) [a] -> ([a],[a])     See in PreludeListLayer.icl
lines     :: [Char] -> [[Char]]               See in PreludeListLayer.icl
words     :: [Char] -> [[Char]]               See in PreludeListLayer.icl
unlines   :: [[Char]] -> [Char]               See in PreludeListLayer.icl
unwords   :: [[Char]] -> [Char]               See in PreludeListLayer.icl
notElem   :: a .[a] -> Bool | Eq a            See in PreludeListLayer.icl
lookup    :: a [(a,b)] -> Maybe b | Eq a      See in PreludeListLayer.icl
maximum   :: ![a] -> a | Ord a                See in PreludeListLayer.icl
minimum   :: ![a] -> a | Ord a                See in PreludeListLayer.icl
concatMap :: (a -> [b]) ![a] -> [b]           See in PreludeListLayer.icl
zip3      :: ![a] [b] [c] -> [(a,b,c)]        See in PreludeListLayer.icl
zipWith   :: (a b -> c) ![a] [b] -> [c]       See in PreludeListLayer.icl
zipWith3  :: (a b c ->d) ![a] [b] [c] -> [d]  See in PreludeListLayer.icl
unzip3    :: ![(a,b,c)] -> ([a],[b],[c])      See in PreludeListLayer.icl
```

- it is also possible that a function has **different names** in the two languages

Haskell **Clean**

```
concat     :: ![[a]] -> [a]flatten
head       :: ![a] -> ahd
tail       ::  [a] -> [a]tl
scanl      :: (a b -> a) a ![b] -> [a]scan
replicate  :: Int a -> [a]repeatn
elem       :: a  [a] -> Bool | Eq aisMember
```

- Functions with **different types** in Clean

Haskell **Clean**

```
length  :: ![a] -> Intclass  length m :: !(m a) -> Int
sum     :: ![a] -> a | Num  asum  :: !.[a] -> a | + , zero  a
product :: ![a] -> a | Num aprod :: !.[a] -> a | * , one   a
zip     :: ![a] [b] -> [(a,b)]zip  ::!(![.a],[.b]) -> [(.a,.b)]
```

- The following functions **abort for a negative argument**, but normally yield the same result as the Clean versions (See in PreludeListLayer.icl)

```
take            :: !Int [a] -> [a]
drop            :: Int ![a] -> [a]
splitAt         :: !Int [a] -> ([a],[a])
```

PreludeText.icl

Classes

Read

See it in Haskell and in Clean.

No difference.

Instances

instance Read Int
instance Read Real
instance Read Trivial
instance Read Char
instance Read (Either a b) | Read a & Read b
instance Read Ordering
instance Read Bool
instance Read (Maybe a) | Read a
instance Read [a] | Read a
instance Read (a,b) | Read a & Read b

Show

See it in Haskell and in Clean.

The minimal complete definition is `showsPrec`, instead of `showsPrec` or `shows`.

Instances

instance Show Int
instance Show Real
instance Show Trivial
instance Show Char
instance Show (Either a b) | Show a & Show b
instance Show Ordering
instance Show Bool
instance Show (Maybe a) | Show a
instance Show [a] | Show a
instance Show (a,b) | Show a & Show b

Functions

In fact there are no differences in the Haskell and Clean implementations of the following:

```
reads        :: ReadS a      | Read a
shows        :: a -> ShowS   | Show a
read         :: [Char] -> a  | Read a
showChar     :: Char -> ShowS
showString   :: [Char] -> ShowS
showParen    :: Bool ShowS -> ShowS
readParen    :: Bool (ReadS a) -> ReadS a
lex          :: ReadS [Char]
readCharList :: [Char] -> [([Char],[Char])]
showCharList :: [Char] -> ShowS
```

In Clean `toString (5.0e-2)` yields `.05` while in Haskell `0.05`. For efficiency reasons the Clean version is adopted instead of the original Haskell one.

Redefining `showList` in instance Show Char is not possible, `showCharList` yields the same result as the Haskell version.

Not supported.: `instance Show Integer, instance Read Integer, instance Show Double, instance Read Double`

PreludeIO.icl

See it in Haskell and in Clean.

While IO is handled in a monadic approach in Haskell, the IO type of Haskell is implemented by a state monad in Clean (See in PreludeIO.icl).

The exception handling of Haskell is omitted from Prelude for Clean.

PreludeGeneric.icl

See it in Clean.

This module is added due to the lack of deriving in Clean. As the Clean compiler does not support deriving, the derived instances are written in Clean. Some of them are implemented with the help of generics (See in PreludeGeneric.icl). The deriving feature of Haskell can generate instances, yet only for the basic classes (e.g. Eq, Ord, Show...). Thus, although the Clean compiler cannot do it automatically, with the help of generics (and some small extra codes) instances can be generated for any classes in Clean.

The generic approach is used for instantiation of Eq and Ord of lists and tuples.

References

[1] S. P. Jones, J. Hughes et al.: *The Haskell 98 Report*. http://haskell.org/report, 1998.

[2] R. Plasmeijer, M. van Eekelen: *Clean Language Report Version 2.0*. http://www.cs.kun.nl/~clean/, 2001.

[3] H. Hegedűs: *Haskell to Clean Front End* (MSc Thesis). ELTE, Budapest, 2001.

[4] H. Hegedűs, Z. Horváth, V. Zsók: *A Haskell és a Clean nyelv összehasonlító elemzése*. Informatika a felsőoktatásban 2002: 1075-1084, 2002.

Appendix 1: Haskell Prelude

Prelude.hs

```
module Prelude (
    module PreludeList, module PreludeText, module PreludeIO,
    Bool(False, True),
    Maybe(Nothing, Just),
    Either(Left, Right),
    Ordering(LT, EQ, GT),
    Char, String, Int, Integer, Float, Double, Rational, IO,
--  List type: []((:), [])
--  Tuple types: (,), (,,), etc.
--  Trivial type: ()
--  Functions: (->)
    Eq((==), (/=)),
    Ord(compare, (<), (<=), (>=), (>), max, min),
    Enum(succ, pred, toEnum, fromEnum, enumFrom, enumFromThen,
         enumFromTo, enumFromThenTo),
    Bounded(minBound, maxBound),
    Num((+), (-), (*), negate, abs, signum, fromInteger),
    Real(toRational),
    Integral(quot, rem, div, mod, quotRem, divMod, toInteger),
    Fractional((/), recip, fromRational),
    Floating(pi, exp, log, sqrt, (**), logBase, sin, cos, tan,
             asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh),
    RealFrac(properFraction, truncate, round, ceiling, floor),
    RealFloat(floatRadix, floatDigits, floatRange, decodeFloat,
              encodeFloat, exponent, significand, scaleFloat, isNaN,
              isInfinite, isDenormalized, isIEEE, isNegativeZero, atan2),
    Monad((>>=), (>>), return, fail),
    Functor(fmap),
    mapM, mapM_, sequence, sequence_, (=<<),
    maybe, either,
    (&&), (||), not, otherwise,
    subtract, even, odd, gcd, lcm, (^), (^^),
    fromIntegral, realToFrac,
    fst, snd, curry, uncurry, id, const, (.), flip, ($), until,
    asTypeOf, error, undefined,
    seq, ($!)
  ) where

import PreludeBuiltin  -- Contains all 'prim' values
import PreludeList
import PreludeText
import PreludeIO
import Ratio( Rational )

infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
infixl 6  +, -
infixr 5  :
infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`

-- Standard types, classes, instances and related functions

-- Equality and Ordered classes


class  Eq a  where
    (==), (/=)       :: a -> a -> Bool

        -- Minimal complete defintion:
        --      (==) or (/=)
    x /= y           =  not (x == y)
    x == y           =  not (x /= y)


class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a

        -- Minimal complete definition:
        --      (<=) or compare
        -- Using compare can be more efficient for complex types.
    compare x y
         | x == y    =  EQ
         | x <= y    =  LT
         | otherwise =  GT

    x <= y           =  compare x y /= GT
    x <  y           =  compare x y == LT
    x >= y           =  compare x y /= LT
    x >  y           =  compare x y == GT

-- note that (min x y, max x y) = (x,y) or (y,x)
    max x y
         | x >= y    =  x
         | otherwise =  y
    min x y
         | x <  y    =  x
         | otherwise =  y

-- Enumeration and Bounded classes


class  Enum a  where
    succ, pred       :: a -> a
    toEnum           :: Int -> a
    fromEnum         :: a -> Int
    enumFrom         :: a -> [a]             -- [n..]
    enumFromThen     :: a -> a -> [a]        -- [n,n'..]
    enumFromTo       :: a -> a -> [a]        -- [n..m]
```

```
    enumFromThenTo   :: a -> a -> a -> [a]   -- [n,n'..m]

        -- Minimal complete definition:
        --      toEnum, fromEnum
    succ             =  toEnum . (+1) . fromEnum
    pred             =  toEnum . (subtract 1) . fromEnum
    enumFrom x       =  map toEnum [fromEnum x ..]
    enumFromTo x y   =  map toEnum [fromEnum x .. fromEnum y]
    enumFromThenTo x y z =
                        map toEnum [fromEnum x, fromEnum y .. fromEnum z]


class  Bounded a  where
    minBound         :: a
    maxBound         :: a

-- Numeric classes


class  (Eq a, Show a) => Num a  where
    (+), (-), (*)    :: a -> a -> a
    negate           :: a -> a
    abs, signum      :: a -> a
    fromInteger      :: Integer -> a

        -- Minimal complete definition:
        --      All, except negate or (-)
    x - y            =  x + negate y
    negate x         =  0 - x


class  (Num a, Ord a) => Real a  where
    toRational       ::  a -> Rational


class  (Real a, Enum a) => Integral a  where
    quot, rem        :: a -> a -> a
    div, mod         :: a -> a -> a
    quotRem, divMod  :: a -> a -> (a,a)
    toInteger        :: a -> Integer

        -- Minimal complete definition:
        --      quotRem, toInteger
    n `quot` d       =  q  where (q,r) = quotRem n d
    n `rem` d        =  r  where (q,r) = quotRem n d
    n `div` d        =  q  where (q,r) = divMod n d
    n `mod` d        =  r  where (q,r) = divMod n d
    divMod n d       =  if signum r == - signum d then (q-1, r+d) else qr
                        where qr@(q,r) = quotRem n d


class  (Num a) => Fractional a  where
    (/)              :: a -> a -> a
    recip            :: a -> a
    fromRational     :: Rational -> a

        -- Minimal complete definition:
        --      fromRational and (recip or (/))
    recip x          =  1 / x
    x / y            =  x * recip y


class  (Fractional a) => Floating a  where
    pi                  :: a
    exp, log, sqrt      :: a -> a
    (**), logBase       :: a -> a -> a
    sin, cos, tan       :: a -> a
    asin, acos, atan    :: a -> a
    sinh, cosh, tanh    :: a -> a
    asinh, acosh, atanh :: a -> a

        -- Minimal complete definition:
        --      pi, exp, log, sin, cos, sinh, cosh
        --      asinh, acosh, atanh
    x ** y           =  exp (log x * y)
    logBase x y      =  log y / log x
    sqrt x           =  x ** 0.5
    tan  x           =  sin  x / cos  x
    tanh x           =  sinh x / cosh x


class  (Real a, Fractional a) => RealFrac a  where
    properFraction   :: (Integral b) => a -> (b,a)
    truncate, round  :: (Integral b) => a -> b
    ceiling, floor   :: (Integral b) => a -> b

        -- Minimal complete definition:
        --      properFraction
    truncate x       =  m  where (m,_) = properFraction x

    round x          =  let (n,r) = properFraction x
                            m     = if r < 0 then n - 1 else n + 1
                        in case signum (abs r - 0.5) of
                             -1 -> n
                             0  -> if even n then n else m
                             1  -> m

    ceiling x        =  if r > 0 then n + 1 else n
                        where (n,r) = properFraction x

    floor x          =  if r < 0 then n - 1 else n
                        where (n,r) = properFraction x


class  (RealFrac a, Floating a) => RealFloat a  where
    floatRadix       :: a -> Integer
    floatDigits      :: a -> Int
    floatRange       :: a -> (Int,Int)
    decodeFloat      :: a -> (Integer,Int)
    encodeFloat      :: Integer -> Int -> a
    exponent         :: a -> Int
    significand      :: a -> a
    scaleFloat       :: Int -> a -> a
    isNaN, isInfinite, isDenormalized, isNegativeZero, isIEEE
                     :: a -> Bool
```

```
    atan2           :: a -> a -> a

        -- Minimal complete definition:
        --      All except exponent, significand,
        --                 scaleFloat, atan2
    exponent x      =  if m == 0 then 0 else n + floatDigits x
                       where (m,n) = decodeFloat x

    significand x   =  encodeFloat m (- floatDigits x)
                       where (m,_) = decodeFloat x

    scaleFloat k x  =  encodeFloat m (n+k)
                       where (m,n) = decodeFloat x

    atan2 y x
      | x>0             =  atan (y/x)
      | x==0 && y>0     =  pi/2
      | x<0  && y>0     =  pi + atan (y/x)
      |(x<=0 && y<0)  ||
       (x<0 && isNegativeZero y) ||
       (isNegativeZero x && isNegativeZero y)
                        = -atan2 (-y) x
      | y==0 && (x<0 || isNegativeZero x)
                        =  pi    -- must be after the previous test on zero y
      | x==0 && y==0    =  y     -- must be after the other double zero tests
      | otherwise       =  x + y -- x or y is a NaN, return a NaN (via +)

-- Numeric functions


subtract        :: (Num a) => a -> a -> a
subtract        =  flip (-)


even, odd       :: (Integral a) => a -> Bool
even n          =  n `rem` 2 == 0
odd             =  not . even


gcd             :: (Integral a) => a -> a -> a
gcd 0 0         =  error "Prelude.gcd: gcd 0 0 is undefined"
gcd x y         =  gcd' (abs x) (abs y)
                   where gcd' x 0  =  x
                         gcd' x y  =  gcd' y (x `rem` y)


lcm             :: (Integral a) => a -> a -> a
lcm _ 0         =  0
lcm 0 _         =  0
lcm x y         =  abs ((x `quot` (gcd x y)) * y)


(^)             :: (Num a, Integral b) => a -> b -> a
x ^ 0           =  1
x ^ n | n > 0   =  f x (n-1) x
                   where f _ 0 y = y
                         f x n y = g x n  where
                                   g x n | even n  = g (x*x) (n `quot` 2)
                                         | otherwise = f x (n-1) (x*y)
_ ^ _           = error "Prelude.^: negative exponent"


(^^)            :: (Fractional a, Integral b) => a -> b -> a
x ^^ n          =  if n >= 0 then x^n else recip (x^(-n))


fromIntegral    :: (Integral a, Num b) => a -> b
fromIntegral    =  fromInteger . toInteger


realToFrac     :: (Real a, Fractional b) => a -> b
realToFrac      =  fromRational . toRational

-- Monadic classes


class  Functor f  where
    fmap            :: (a -> b) -> f a -> f b


class  Monad m  where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a

        -- Minimal complete definition:
        --      (>>=), return
    m >> k  =  m >>= \_ -> k
    fail s  = error s


sequence       :: Monad m => [m a] -> m [a]
sequence       =  foldr mcons (return [])
                      where mcons p q = p >>= \x -> q >>= \y -> return (x:y)


sequence_      :: Monad m => [m a] -> m ()
sequence_      =  foldr (>>) (return ())

-- The xxxM functions take list arguments, but lift the function or
-- list element to a monad type

mapM            :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as       =  sequence (map f as)


mapM_           :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as      =  sequence_ (map f as)


(=<<)           :: Monad m => (a -> m b) -> m a -> m b
f =<< x         =  x >>= f


-- Trivial type
```

```
data  ()  =  ()  deriving (Eq, Ord, Enum, Bounded)

-- Function type


data a -> b  -- No constructor for functions is exported.

-- identity function

id              :: a -> a
id x            =  x

-- constant function

const           :: a -> b -> a
const x _       =  x

-- function composition

(.)             :: (b -> c) -> (a -> b) -> a -> c
f . g           =  \ x -> f (g x)

-- flip f  takes its (first) two arguments in the reverse order of f.

flip            :: (a -> b -> c) -> b -> a -> c
flip f x y      =  f y x


seq :: a -> b -> b
seq = ...       -- Primitive

-- right-associating infix application operators
-- (useful in continuation-passing style)

($), ($!) :: (a -> b) -> a -> b
f $  x    =  f x
f $! x    =  x `seq` f x


-- Boolean type


data  Bool  =  False | True     deriving (Eq, Ord, Enum, Read, Show, Bounded)

-- Boolean functions


(&&), (||)      :: Bool -> Bool -> Bool
True  && x      =  x
False && _      =  False
True  || _      =  True
False || x      =  x


not             :: Bool -> Bool
not True        =  False
not False       =  True


otherwise       :: Bool
otherwise       =  True


-- Character type


data Char = ... 'a' | 'b' ... -- 2^16 unicode values


instance  Eq Char  where
    c == c'          =  fromEnum c == fromEnum c'


instance  Ord Char  where
    c <= c'          =  fromEnum c <= fromEnum c'


instance  Enum Char  where
    toEnum            = primIntToChar
    fromEnum          = primCharToInt
    enumFrom c        = map toEnum [fromEnum c .. fromEnum (maxBound::Char)]
    enumFromThen c c' = map toEnum [fromEnum c, fromEnum c' .. fromEnum lastChar]
                        where lastChar :: Char
                              lastChar | c' < c    = minBound
                                       | otherwise = maxBound


instance  Bounded Char  where
    minBound            =  '\0'
    maxBound            =  '\xffff'


type  String = [Char]


-- Maybe type


data  Maybe a  =  Nothing | Just a      deriving (Eq, Ord, Read, Show)


maybe             :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  =  n
maybe n f (Just x) =  f x


instance  Functor Maybe  where
    fmap f Nothing    =  Nothing
    fmap f (Just x)   =  Just (f x)


instance  Monad Maybe  where
    (Just x) >>= k   =  k x
    Nothing  >>= k   =  Nothing
```

```
    return          =  Just
    fail s          =  Nothing

-- Either type

data  Either a b  =  Left a | Right b   deriving (Eq, Ord, Read, Show)

either              :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)  =  f x
either f g (Right y) =  g y

-- IO type

data  IO a  -- abstract

instance  Functor IO where
   fmap f x           =  x >>= (return . f)

instance Monad IO where
   (>>=)  = ...
   return = ...

   m >> k = m >>= \_ -> k
   fail s = error s

-- Ordering type

data  Ordering  =  LT | EQ | GT
          deriving (Eq, Ord, Enum, Read, Show, Bounded)

-- Standard numeric types.  The data declarations for these types cannot
-- be expressed directly in Haskell since the constructor lists would be
-- far too large.

data  Int  =  minBound ... -1 | 0 | 1 ... maxBound

instance  Eq       Int  where ...

instance  Ord      Int  where ...

instance  Num      Int  where ...

instance  Real     Int  where ...

instance  Integral Int  where ...

instance  Enum     Int  where ...

instance  Bounded  Int  where ...

data  Integer  =  ... -1 | 0 | 1 ...

instance  Eq       Integer  where ...

instance  Ord      Integer  where ...

instance  Num      Integer  where ...

instance  Real     Integer  where ...

instance  Integral Integer  where ...

instance  Enum     Integer  where ...

data  Float

instance  Eq         Float  where ...

instance  Ord        Float  where ...

instance  Num        Float  where ...

instance  Real       Float  where ...

instance  Fractional Float  where ...

instance  Floating   Float  where ...

instance  RealFrac   Float  where ...

instance  RealFloat  Float  where ...

data  Double

instance  Eq         Double  where ...

instance  Ord        Double  where ...

instance  Num        Double  where ...

instance  Real       Double  where ...

instance  Fractional Double  where ...

instance  Floating   Double  where ...

instance  RealFrac   Double  where ...

instance  RealFloat  Double  where ...

-- The Enum instances for Floats and Doubles are slightly unusual.
-- The 'toEnum' function truncates numbers to Int.  The definitions
-- of enumFrom and enumFromThen allow floats to be used in arithmetic
-- series: [0,0.1 .. 1.0].  However, roundoff errors make these somewhat
-- dubious.  This example may have either 10 or 11 elements, depending on
```

```
-- how 0.1 is represented.


instance  Enum Float  where
    succ x           =  x+1
    pred x           =  x-1
    toEnum           =  fromIntegral
    fromEnum         =  fromInteger . truncate   -- may overflow
    enumFrom         =  numericEnumFrom
    enumFromThen     =  numericEnumFromThen
    enumFromTo       =  numericEnumFromTo
    enumFromThenTo   =  numericEnumFromThenTo


instance  Enum Double  where
    succ x           =  x+1
    pred x           =  x-1
    toEnum           =  fromIntegral
    fromEnum         =  fromInteger . truncate   -- may overflow
    enumFrom         =  numericEnumFrom
    enumFromThen     =  numericEnumFromThen
    enumFromTo       =  numericEnumFromTo
    enumFromThenTo   =  numericEnumFromThenTo


numericEnumFrom         :: (Fractional a) => a -> [a]

numericEnumFromThen     :: (Fractional a) => a -> a -> [a]

numericEnumFromTo       :: (Fractional a, Ord a) => a -> a -> [a]

numericEnumFromThenTo   :: (Fractional a, Ord a) => a -> a -> a -> [a]
numericEnumFrom         =  iterate (+1)
numericEnumFromThen n m =  iterate (+(m-n)) n
numericEnumFromTo n m   =  takeWhile (<= m+1/2) (numericEnumFrom n)
numericEnumFromThenTo n n' m = takeWhile p (numericEnumFromThen n n')
                           where
                             p | n' > n    = (<= m + (n'-n)/2)
                               | otherwise = (>= m + (n'-n)/2)

-- Lists

-- This data declaration is not legal Haskell
-- but it indicates the idea

data  [a]  =  [] | a : [a]  deriving (Eq, Ord)


instance Functor [] where
    fmap = map


instance  Monad []  where
    m >>= k          = concat (map k m)
    return x         = [x]
    fail s           = []

-- Tuples


data  (a,b)   =  (a,b)    deriving (Eq, Ord, Bounded)

data  (a,b,c) =  (a,b,c)  deriving (Eq, Ord, Bounded)


-- component projections for pairs:
-- (NB: not provided for triples, quadruples, etc.)

fst             :: (a,b) -> a
fst (x,y)       =  x


snd             :: (a,b) -> b
snd (x,y)       =  y

-- curry converts an uncurried function to a curried function;
-- uncurry converts a curried function to a function on pairs.

curry           :: ((a, b) -> c) -> a -> b -> c
curry f x y     =  f (x, y)


uncurry         :: (a -> b -> c) -> ((a, b) -> c)
uncurry f p     =  f (fst p) (snd p)

-- Misc functions

-- until p f  yields the result of applying f until p holds.

until           :: (a -> Bool) -> (a -> a) -> a -> a
until p f x
     | p x       =  x
     | otherwise =  until p f (f x)

-- asTypeOf is a type-restricted version of const.  It is usually used
-- as an infix operator, and its typing forces its first argument
-- (which is usually overloaded) to have the same type as the second.

asTypeOf        :: a -> a -> a
asTypeOf        =  const

-- error stops execution and displays an error message


error           :: String -> a
error           =  primError

-- It is expected that compilers will recognize this and insert error
-- messages that are more appropriate to the context in which undefined
-- appears.


undefined       :: a
undefined       =  error "Prelude.undefined"
```

PreludeList.hs

```
-- Standard list functions

module PreludeList (
    map, (++), filter, concat,
    head, last, tail, init, null, length, (!!),
    foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1,
    iterate, repeat, replicate, cycle,
    take, drop, splitAt, takeWhile, dropWhile, span, break,
    lines, words, unlines, unwords, reverse, and, or,
    any, all, elem, notElem, lookup,
    Sum, product, maximum, minimum, concatMap,
    zip, zip3, zipWith, zipWith3, unzip, unzip3)
  where

import qualified Char(isSpace)

infixl 9  !!
infixr 5  ++
infix  4  `elem`, `notElem`

-- Map and append

map :: (a -> b) -> [a] -> [a]
map f []     = []
map f (x:xs) = f x : map f xs


(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)


filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs


concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss


-- head and tail extract the first element and remaining elements,
-- respectively, of a list, which must be non-empty.  last and init
-- are the dual functions working from the end of a finite list,
-- rather than the beginning.


head            :: [a] -> a
head (x:_)      =  x
head []         =  error "Prelude.head: empty list"


last            :: [a] -> a
last [x]        =  x
last (_:xs)     =  last xs
last []         =  error "Prelude.last: empty list"


tail            :: [a] -> [a]
tail (_:xs)     =  xs
tail []         =  error "Prelude.tail: empty list"


init            :: [a] -> [a]
init [x]        =  []
init (x:xs)     =  x : init xs
init []         =  error "Prelude.init: empty list"


null            :: [a] -> Bool
null []         =  True
null (_:_)      =  False

-- length returns the length of a finite list as an Int.

length          :: [a] -> Int
length []       =  0
length (_:l)    =  1 + length l

-- List index (subscript) operator, 0-origin

(!!)                :: [a] -> Int -> a
(x:_)  !! 0         =  x
(_:xs) !! n | n > 0 =  xs !! (n-1)
(_:_)  !! _         =  error "Prelude.!!: negative index"
[]     !! _         =  error "Prelude.!!: index too large"

-- foldl, applied to a binary operator, a starting value (typically the
-- left-identity of the operator), and a list, reduces the list using
-- the binary operator, from left to right:
--  foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn
-- foldl1 is a variant that has no starting value argument, and  thus must
-- be applied to non-empty lists.  scanl is similar to foldl, but returns
-- a list of successive reduced values from the left:
--      scanl f z [x1, x2, ...] == [z, z `f` x1, (z `f` x1) `f` x2, ...]
-- Note that  last (scanl f z xs) == foldl f z xs.
-- scanl1 is similar, again without the starting element:
--      scanl1 f [x1, x2, ...] == [x1, x1 `f` x2, ...]


foldl           :: (a -> b -> a) -> a -> [b] -> a
foldl f z []    =  z
foldl f z (x:xs) =  foldl f (f z x) xs


foldl1          :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)  =  foldl f x xs
foldl1 _ []      =  error "Prelude.foldl1: empty list"


scanl           :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q xs    =  q : (case xs of
```

```
                        []   -> []
                        x:xs -> scanl f (f q x) xs


scanl1          :: (a -> a -> a) -> [a] -> [a]
scanl1 f (x:xs)  =  scanl f x xs
scanl1 _ []      =  error "Prelude.scanl1: empty list"

-- foldr, foldr1, scanr, and scanr1 are the right-to-left duals of the
-- above functions.


foldr           :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)


foldr1          :: (a -> a -> a) -> [a] -> a
foldr1 f [x]     =  x
foldr1 f (x:xs)  =  f x (foldr1 f xs)
foldr1 _ []      =  error "Prelude.foldr1: empty list"


scanr            :: (a -> b -> b) -> b -> [a] -> [b]
scanr f q0 []     =  [q0]
scanr f q0 (x:xs) =  f x q : qs
                     where qs@(q:_) = scanr f q0 xs


scanr1          :: (a -> a -> a) -> [a] -> [a]
scanr1 f  [x]    =  [x]
scanr1 f  (x:xs) =  f x q : qs
                    where qs@(q:_) = scanr1 f xs
scanr1 _ []      =  error "Prelude.scanr1: empty list"

-- iterate f x returns an infinite list of repeated applications of f to x:
-- iterate f x == [x, f x, f (f x), ...]

iterate         :: (a -> a) -> a -> [a]
iterate f x      =  x : iterate f (f x)

-- repeat x is an infinite list, with x the value of every element.

repeat          :: a -> [a]
repeat x         =  xs where xs = x:xs

-- replicate n x is a list of length n with x the value of every element

replicate       :: Int -> a -> [a]
replicate n x    =  take n (repeat x)

-- cycle ties a finite list into a circular one, or equivalently,
-- the infinite repetition of the original list.  It is the identity
-- on infinite lists.


cycle           :: [a] -> [a]
cycle []         =  error "Prelude.cycle: empty list"
cycle xs         =  xs' where xs' = xs ++ xs'

-- take n, applied to a list xs, returns the prefix of xs of length n,
-- or xs itself if n > length xs.  drop n xs returns the suffix of xs
-- after the first n elements, or [] if n > length xs.  splitAt n xs
-- is equivalent to (take n xs, drop n xs).


take                    :: Int -> [a] -> [a]
take 0 _                =  []
take _ []               =  []
take n (x:xs) | n > 0   =  x : take (n-1) xs
take _     _            =  error "Prelude.take: negative argument"


drop                    :: Int -> [a] -> [a]
drop 0 xs               =  xs
drop _ []               =  []
drop n (_:xs) | n > 0   =  drop (n-1) xs
drop _     _            =  error "Prelude.drop: negative argument"


splitAt                 :: Int -> [a] -> ([a],[a])
splitAt 0 xs            =  ([],xs)
splitAt _ []            =  ([],[])
splitAt n (x:xs) | n > 0 =  (x:xs',xs'') where (xs',xs'') = splitAt (n-1) xs
splitAt _     _         =  error "Prelude.splitAt: negative argument"

-- takeWhile, applied to a predicate p and a list xs, returns the longest
-- prefix (possibly empty) of xs of elements that satisfy p.  dropWhile p xs
-- returns the remaining suffix.  Span p xs is equivalent to
-- (takeWhile p xs, dropWhile p xs), while break p uses the negation of p.


takeWhile               :: (a -> Bool) -> [a] -> [a]
takeWhile p []          =  []
takeWhile p (x:xs)
            | p x       =  x : takeWhile p xs
            | otherwise =  []


dropWhile               :: (a -> Bool) -> [a] -> [a]
dropWhile p []          =  []
dropWhile p xs@(x:xs')
            | p x       =  dropWhile p xs'
            | otherwise =  xs


span, break             :: (a -> Bool) -> [a] -> ([a],[a])
span p []          = ([],[])
span p xs@(x:xs')
            | p x       =  (x:ys,zs)
            | otherwise =  ([],xs)
                           where (ys,zs) = span p xs'

break p                 =  span (not . p)

-- lines breaks a string up into a list of strings at newline characters.
```

```
-- The resulting strings do not contain newlines.  Similary, words
-- breaks a string up into a list of words, which were delimited by
-- white space.  unlines and unwords are the inverse operations.
-- unlines joins lines with terminating newlines, and unwords joins
-- words with separating spaces.


lines ""          :: String -> [String]
lines ""          =  []
lines s           =  let (l, s') = break (== '\n') s
                       in  l : case s' of
                                 []      -> []
                                 (_:s'') -> lines s''


words             :: String -> [String]
words s           =  case dropWhile Char.isSpace s of
                       "" -> []
                     s' -> w : words s''
                           where (w, s'') = break Char.isSpace s'


unlines           :: [String] -> String
unlines           =  concatMap (++ "\n")


unwords           :: [String] -> String
unwords []        =  ""
unwords ws        =  foldr1 (\w s -> w ++ ' ':s) ws

-- reverse xs returns the elements of xs in reverse order.  xs must be finite.

reverse           :: [a] -> [a]
reverse           =  foldl (flip (:)) []

-- and returns the conjunction of a Boolean list.  For the result to be
-- True, the list must be finite; False, however, results from a False
-- value at a finite index of a finite or infinite list.  or is the
-- disjunctive dual of and.

and, or           :: [Bool] -> Bool
and               =  foldr (&&) True
or                =  foldr (||) False

-- Applied to a predicate and a list, any determines if any element
-- of the list satisfies the predicate.  Similarly, for all.

any, all          :: (a -> Bool) -> [a] -> Bool
any p             =  or . map p
all p             =  and . map p

-- elem is the list membership predicate, usually written in infix form,
-- e.g., x `elem` xs.  notElem is the negation.

elem, notElem     :: (Eq a) => a -> [a] -> Bool
elem x            =  any (== x)
notElem x         =  all (/= x)

-- lookup key assocs looks up a key in an association list.

lookup            :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key []     =  Nothing
lookup key ((x,y):xys)
    | key == x    =  Just y
    | otherwise   =  lookup key xys

-- sum and product compute the sum or product of a finite list of numbers.

sum, product      :: (Num a) => [a] -> a
sum               =  foldl (+) 0
product           =  foldl (*) 1

-- maximum and minimum return the maximum or minimum value from a list,
-- which must be non-empty, finite, and of an ordered type.

maximum, minimum :: (Ord a) => [a] -> a
maximum []        =  error "Prelude.maximum: empty list"
maximum xs        =  foldl1 max xs

minimum []        =  error "Prelude.minimum: empty list"
minimum xs        =  foldl1 min xs


concatMap         :: (a -> [b]) -> [a] -> [b]
concatMap f       =  concat . map f

-- zip takes two lists and returns a list of corresponding pairs.  If one
-- input list is short, excess elements of the longer list are discarded.
-- zip3 takes three lists and returns a list of triples.  Zips for larger
-- tuples are in the List library


zip               :: [a] -> [b] -> [(a,b)]
zip               =  zipWith (,)


zip3              :: [a] -> [b] -> [c] -> [(a,b,c)]
zip3              =  zipWith3 (,,)

-- The zipWith family generalises the zip family by zipping with the
-- function given as the first argument, instead of a tupling function.
-- For example, zipWith (+) is applied to two lists to produce the list
-- of corresponding sums.


zipWith           :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)
                  =  z a b : zipWith z as bs
zipWith _ _ _     =  []


zipWith3          :: (a->b->c->d) -> [a]->[b]->[c]->[d]
zipWith3 z (a:as) (b:bs) (c:cs)
                  =  z a b c : zipWith3 z as bs cs
zipWith3 _ _ _ _  =  []
```

```
-- unzip transforms a list of pairs into a pair of lists.

unzip           :: [(a,b)] -> ([a],[b])
unzip           =  foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

unzip3          :: [(a,b,c)] -> ([a],[b],[c])
unzip3          =  foldr (\(a,b,c) ~(as,bs,cs) -> (a:as,b:bs,c:cs))
                         ([],[],[])
```

PreludeText.hs

```
module PreludeText (
    ReadS, ShowS,
    Read(readsPrec, readList),
    Show(showsPrec, showList),
    reads, shows, show, read, lex,
    showChar, showString, readParen, showParen ) where

-- The instances of Read and Show for
-- Bool, Char, Maybe, Either, Ordering
-- are done via "deriving" clauses in Prelude.hs

import Char(isSpace, isAlpha, isDigit, isAlphaNum,
            showLitChar, readLitChar, lexLitChar)

import Numeric(showSigned, showInt, readSigned, readDec, showFloat,
               readFloat, lexDigits)

type  ReadS a  = String -> [(a,String)]

type  ShowS    = String -> String

class  Read a  where
    readsPrec       :: Int -> ReadS a
    readList        :: ReadS [a]

-- Minimal complete definition:
-- readsPrec
    readList        = readParen False (\r -> [pr | ("[",s)  <- lex r,
                                                   pr       <- readl s])
                      where readl  s = [([],t)   | ("]",t)  <- lex s] ++
                                       [(x:xs,u) | (x,t)    <- reads s,
                                                   (xs,u)   <- readl' t]
                            readl' s = [([],t)   | ("]",t)  <- lex s] ++
                                       [(x:xs,v) | (",",t)  <- lex s,
                                                   (x,u)    <- reads t,
                                                   (xs,v)   <- readl' u]

class  Show a  where
    showsPrec       :: Int -> a -> ShowS
    show        :: a -> String
    showList        :: [a] -> ShowS

-- Mimimal complete definition:
-- show or showsPrec
    showsPrec _ x s   = show x ++ s

    show x        = showsPrec 0 x ""

    showList []       = showString "[]"
    showList (x:xs)   = showChar '[' . shows x . showl xs
                        where showl []     = showChar ']'
                              showl (x:xs) = showChar ',' . shows x .
                                             showl xs

reads           :: (Read a) => ReadS a
reads           =  readsPrec 0

shows           :: (Show a) => a -> ShowS
shows           =  showsPrec 0

read            :: (Read a) => String -> a
read s          =  case [x | (x,t) <- reads s, ("","") <- lex t] of
                        [x] -> x
                        []  -> error "Prelude.read: no parse"
                        _   -> error "Prelude.read: ambiguous parse"


showChar        :: Char -> ShowS
showChar        =  (:)


showString      :: String -> ShowS
showString      =  (++)


showParen       :: Bool -> ShowS -> ShowS
showParen b p   =  if b then showChar '(' . p . showChar ')' else p


readParen       :: Bool -> ReadS a -> ReadS a
readParen b g   =  if b then mandatory else optional
                   where optional r  = g r ++ mandatory r
                         mandatory r = [(x,u) | ("(",s) <- lex r,
                                                (x,t)   <- optional s,
                                                (")",u) <- lex t    ]

-- This lexer is not completely faithful to the Haskell lexical syntax.
-- Current limitations:
--    Qualified names are not handled properly
--    Octal and hexidecimal numerics are not recognized as a single token
--    Comments are not treated properly


lex             :: ReadS String
lex ""          =  [("","")]
lex (c:s)
   | isSpace c  =  lex (dropWhile isSpace s)
lex ('\'':s)    =  [('\'':ch++"'", t) | (ch,'\'':t)  <- lexLitChar s,
                                        ch /= "'" ]
```

```
lex (""":s)      =  [(""":str, t)      | (str,t) <- lexString s]
                    where
                    lexString (""":s) = [("\"",s)]
                    lexString s = [(ch++str, u)
                                   | (ch,t)  <- lexStrItem s,
                                     (str,u) <- lexString t  ]

                    lexStrItem ('\\':'&':s) =  [("\\&",s)]
                    lexStrItem ('\\':c:s) | isSpace c
                                          =  [("\\&",t) |
                                              '\\':t <-
                                                 [dropWhile isSpace s]]
                    lexStrItem s          =  lexLitChar s

lex (c:s) | isSingle c = [([c],s)]
          | isSym c    = [(c:sym,t)       | (sym,t) <- [span isSym s]]
          | isAlpha c  = [(c:nam,t)       | (nam,t) <- [span isIdChar s]]
          | isDigit c  = [(c:ds++fe,t)    | (ds,s)  <- [span isDigit s],
                                            (fe,t)  <- lexFracExp s     ]
          | otherwise  = []    -- bad character
            where
              isSingle c =  c `elem` ",;()[]{}_`"
      isSym c    =  c `elem` "!@#$%&*+./<=>?\\^|:-~"
              isIdChar c =  isAlphaNum c || c `elem` "_'"

              lexFracExp ('.':c:cs) | isDigit c
                                  = [('.':ds++e,u) | (ds,t) <- lexDigits (c:cs),
                                                     (e,u)  <- lexExp t]
              lexFracExp s        = [("",s)]

              lexExp (e:s) | e `elem` "eE"
                           = [(e:c:ds,u) | (c:t)  <- [s], c `elem` "+-",
                                           (ds,u) <- lexDigits t] ++
                             [(e:ds,t)   | (ds,t) <- lexDigits s]
              lexExp s = [("",s)]

instance  Show Int
  where
    showsPrec           = showSigned showInt

instance  Read Int
  where
    readsPrec p         = readSigned readDec

instance  Show Integer
  where
    showsPrec           = showSigned showInt


instance  Read Integer
  where
    readsPrec p         = readSigned readDec

instance  Show Float
  where
    showsPrec p         = showFloat


instance  Read Float
  where
    readsPrec p         = readFloat


instance  Show Double
  where
    showsPrec p         = showFloat


instance  Read Double
  where
    readsPrec p         = readFloat


instance  Show ()
  where
    showsPrec p () = showString "()"


instance Read ()
 where
    readsPrec p    = readParen False
                           (\r -> [((),t) | ("(",s) <- lex r,
                                            (")",t) <- lex s] )

instance  Show Char
  where
    showsPrec p '\'' = showString "'\\''"
    showsPrec p c    = showChar '\'' . showLitChar c . showChar '\''

    showList cs = showChar '"' . showl cs
                  where showl ""       = showChar '"'
                        showl ('"':cs) = showString "\\\"" . showl cs
                        showl (c:cs)   = showLitChar c . showl cs


instance  Read Char
  where
    readsPrec p      = readParen False
                           (\r -> [(c,t) | ('\'':s,t)<- lex r,
                                           (c,"'")  <- readLitChar s])

    readList = readParen False (\r -> [(l,t) | ('"':s, t) <- lex r,
                                               (l,_)      <- readl s ])
        where readl ('"':s)      = [("",s)]
              readl ('\\':'&':s) = readl s
              readl s            = [(c:cs,u) | (c ,t) <- readLitChar s,
                                               (cs,u) <- readl t       ]


instance  (Show a) => Show [a]
  where
    showsPrec p      = showList


instance  (Read a) => Read [a]
  where
```

```
    readsPrec p      = readList

-- Tuples


instance  (Show a, Show b) => Show (a,b)  where
    showsPrec p (x,y) = showChar '(' . shows x . showChar ',' .
                                       shows y . showChar ')'


instance  (Read a, Read b) => Read (a,b)  where
    readsPrec p       = readParen False
                            (\r -> [((x,y), w) | ("(",s) <- lex r,
                                                 (x,t)   <- reads s,
                                                 (",",u) <- lex t,
                                                 (y,v)   <- reads u,
                                                 (")",w) <- lex v ] )

-- Other tuples have similar Read and Show instances
```

PreludeIO.hs

```
module PreludeIO (
    FilePath, IOError, ioError, userError, catch,
    putChar, putStr, putStrLn, print,
    getChar, getLine, getContents, interact,
    readFile, writeFile, appendFile, readIO, readLn
  ) where

import PreludeBuiltin

type  FilePath = String

data IOError    -- The internals of this type are system dependent

instance  Show IOError  where ...

instance  Eq IOError  where ...


ioError          ::  IOError -> IO a
ioError          =   primIOError


userError        ::  String -> IOError
userError        =   primUserError


catch            ::  IO a -> (IOError -> IO a) -> IO a
catch            =   primCatch


putChar          :: Char -> IO ()
putChar          =  primPutChar


putStr           :: String -> IO ()
putStr s         =  mapM_ putChar s


putStrLn         :: String -> IO ()
putStrLn s       =  do putStr s
                       putStr "\n"


print            :: Show a => a -> IO ()
print x          =  putStrLn (show x)


getChar          :: IO Char
getChar          =  primGetChar


getLine          :: IO String
getLine          =  do c <- getChar
                       if c == '\n' then return "" else
                          do s <- getLine
                             return (c:s)


getContents      :: IO String
getContents      =  primGetContents


interact         ::  (String -> String) -> IO ()
interact f       =  do s <- getContents
                       putStr (f s)


readFile         :: FilePath -> IO String
readFile         =  primReadFile


writeFile        :: FilePath -> String -> IO ()
writeFile        =  primWriteFile


appendFile       :: FilePath -> String -> IO ()
appendFile       =  primAppendFile

  -- raises an exception instead of an error

readIO   :: Read a => String -> IO a
readIO s =  case [x | (x,t) <- reads s, ("","") <- lex t] of
              [x] -> return x
              []  -> ioError (userError "Prelude.readIO: no parse")
              _   -> ioError (userError "Prelude.readIO: ambiguous parse")


readLn           :: Read a => IO a
readLn           =  do l <- getLine
                       r <- readIO l
                       return r
```

Char.hs

```
module Char (
    isAscii, isLatin1, isControl, isPrint, isSpace, isUpper, isLower,
    isAlpha, isDigit, isOctDigit, isHexDigit, isAlphaNum,
    digitToInt, intToDigit,
    toUpper, toLower,
    ord, chr,
    readLitChar, showLitChar, lexLitChar,

-- ...and what the Prelude exports
    Char, String
    ) where

import Array  -- used for character name table.
import Numeric (readDec, readOct, lexDigits, readHex)
import UnicodePrims  -- source of primitive Unicode functions.

-- Character-testing operations
isAscii, isControl, isPrint, isSpace, isUpper, isLower,
 isAlpha, isDigit, isOctDigit, isHexDigit, isAlphaNum :: Char -> Bool

isAscii c               =  c < '\x80'

isLatin1 c              =  c <= '\xff'

isControl c             =  c < ' ' || c >= '\DEL' && c <= '\x9f'

isPrint                 =  primUnicodeIsPrint

isSpace c               =  c `elem` " \t\n\r\f\v\xA0"
-- Only Latin-1 spaces recognized

isUpper                 =  primUnicodeIsUpper -- 'A'..'Z'

isLower                 =  primUnicodeIsLower -- 'a'..'z'

isAlpha c               =  isUpper c || isLower c

isDigit c               =  c >= '0' && c <= '9'

isOctDigit c            =  c >= '0' && c <= '7'

isHexDigit c            =  isDigit c || c >= 'A' && c <= 'F' ||
                                        c >= 'a' && c <= 'f'

isAlphaNum              =  primUnicodeIsAlphaNum


-- Digit conversion operations
digitToInt :: Char -> Int
digitToInt c
  | isDigit c           =  fromEnum c - fromEnum '0'
  | c >= 'a' && c <= 'f' =  fromEnum c - fromEnum 'a' + 10
  | c >= 'A' && c <= 'F' =  fromEnum c - fromEnum 'A' + 10
  | otherwise           =  error "Char.digitToInt: not a digit"

intToDigit :: Int -> Char
intToDigit i
  | i >= 0  && i <=  9  =  toEnum (fromEnum '0' + i)
  | i >= 10 && i <= 15  =  toEnum (fromEnum 'a' + i - 10)
  | otherwise           =  error "Char.intToDigit: not a digit"


-- Case-changing operations
toUpper                 :: Char -> Char
toUpper                 =  primUnicodeToUpper

toLower                 :: Char -> Char
toLower                 =  primUnicodeToLower

-- Character code functions
ord                     :: Char -> Int
ord                     =  fromEnum

chr                     :: Int  -> Char
chr                     =  toEnum

-- Text functions
readLitChar             :: ReadS Char
readLitChar ('\\':s)    =  readEsc s
    where
        readEsc ('a':s)  = [('\a',s)]
        readEsc ('b':s)  = [('\b',s)]
        readEsc ('f':s)  = [('\f',s)]
        readEsc ('n':s)  = [('\n',s)]
        readEsc ('r':s)  = [('\r',s)]
        readEsc ('t':s)  = [('\t',s)]
        readEsc ('v':s)  = [('\v',s)]
        readEsc ('\\':s) = [('\\',s)]
        readEsc ('"':s)  = [('"',s)]
        readEsc ('\'':s) = [('\'',s)]
        readEsc ('^':c:s) | c >= '@' && c <= '_'
                         = [(chr (ord c - ord '@'), s)]
        readEsc s@(d:_) | isDigit d
                         = [(chr n, t) | (n,t) <- readDec s]
        readEsc ('o':s)  = [(chr n, t) | (n,t) <- readOct s]
        readEsc ('x':s)  = [(chr n, t) | (n,t) <- readHex s]
        readEsc s@(c:_) | isUpper c
                         = let table = ('\DEL', "DEL") : assocs asciiTab
                           in case [(c,s') | (c, mne) <- table,
                                            ([],s') <- [match mne s]]
                              of (pr:_) -> [pr]
                                 []     -> []
        readEsc _        = []

        match                       :: (Eq a) => [a] -> [a] -> ([a],[a])
        match (x:xs) (y:ys) | x == y  =  match xs ys
        match xs     ys               =  (xs,ys)

readLitChar (c:s)       =  [(c,s)]

showLitChar             :: Char -> ShowS
showLitChar c | c > '\DEL' =  showChar '\\' .
                              protectEsc isDigit (shows (ord c))
showLitChar '\DEL'         =  showString "\\DEL"
```

```
showLitChar '\\'            =  showString "\\\\"
showLitChar c | c >= ' '    =  showChar c
showLitChar '\a'            =  showString "\\a"
showLitChar '\b'            =  showString "\\b"
showLitChar '\f'            =  showString "\\f"
showLitChar '\n'            =  showString "\\n"
showLitChar '\r'            =  showString "\\r"
showLitChar '\t'            =  showString "\\t"
showLitChar '\v'            =  showString "\\v"
showLitChar '\SO'           =  protectEsc (== 'H') (showString "\\SO")
showLitChar c               =  showString ('\\' : asciiTab!c)

protectEsc p f              = f . cont
                              where cont s@(c:_) | p c = "\\&" ++ s
                                    cont s             = s
asciiTab = listArray ('\NUL', ' ')
           ["NUL", "SOH", "STX", "ETX", "EOT", "ENQ", "ACK", "BEL",
            "BS",  "HT",  "LF",  "VT",  "FF",  "CR",  "SO",  "SI",
            "DLE", "DC1", "DC2", "DC3", "DC4", "NAK", "SYN", "ETB",
            "CAN", "EM",  "SUB", "ESC", "FS",  "GS",  "RS",  "US",
            "SP"]

lexLitChar          :: ReadS String
lexLitChar ('\\':s) =  [('\\':esc, t) | (esc,t) <- lexEsc s]
    where
        lexEsc (c:s)     | c `elem` "abfnrtv\\\"'" = [([c],s)]
        lexEsc s@(d:_)   | isDigit d               = lexDigits s
        lexEsc ('^':c:s) | c >= '@' && c <= '_'    = [(['^',c],s)]
        -- Very crude approximation to \XYZ.  Let readers work this out.
        lexEsc s@(c:_)   | isUpper c               = [span isCharName s]
        lexEsc _                                   = []
        isCharName c = isUpper c || isDigit c

lexLitChar (c:s)    =  [([c],s)]
lexLitChar ""       =  []
```

Divmod.hs

```
divmod n m = [pair x y | (x,y) <- zip [n,n,negate n,negate n] [m,negate m,m,negate m] ]
    where
        pair x y = (x `div` y, x `mod` y)

quotrem n m = [pair x y | (x,y) <- zip [n,n,negate n,negate n] [m,negate m,m,negate m] ]
    where
        pair x y = (x `quot` y, x `rem` y)
```

Numeric.hs

```
module Numeric(fromRat,
               showSigned, showInt,
               readSigned, readInt,
               readDec, readOct, readHex,
               floatToDigits,
               showEFloat, showFFloat, showGFloat, showFloat,
               readFloat, lexDigits) where

import Char
import Ratio
import Array

-- This converts a rational to a floating.  This should be used in the
-- Fractional instances of Float and Double.

fromRat :: (RealFloat a) => Rational -> a
fromRat x =
    if x == 0 then encodeFloat 0 0             -- Handle exceptional cases
    else if x < 0 then - fromRat' (-x)         -- first.
    else fromRat' x

-- Conversion process:
-- Scale the rational number by the RealFloat base until
-- it lies in the range of the mantissa (as used by decodeFloat/encodeFloat).
-- Then round the rational to an Integer and encode it with the exponent
-- that we got from the scaling.
-- To speed up the scaling process we compute the log2 of the number to get
-- a first guess of the exponent.
fromRat' :: (RealFloat a) => Rational -> a
fromRat' x = r
  where b = floatRadix r
        p = floatDigits r
        (minExp0, _) = floatRange r
        minExp = minExp0 - p            -- the real minimum exponent
        xMin = toRational (expt b (p-1))
        xMax = toRational (expt b p)
        p0 = (integerLogBase b (numerator x) -
              integerLogBase b (denominator x) - p) `max` minExp
        f = if p0 < 0 then 1 % expt b (-p0) else expt b p0 % 1
        (x', p') = scaleRat (toRational b) minExp xMin xMax p0 (x / f)
        r = encodeFloat (round x') p'

-- Scale x until xMin <= x < xMax, or p (the exponent) <= minExp.
scaleRat :: Rational -> Int -> Rational -> Rational ->
            Int -> Rational -> (Rational, Int)
scaleRat b minExp xMin xMax p x =
    if p <= minExp then
        (x, p)
    else if x >= xMax then
        scaleRat b minExp xMin xMax (p+1) (x/b)
    else if x < xMin  then
        scaleRat b minExp xMin xMax (p-1) (x*b)
    else
        (x, p)

-- Exponentiation with a cache for the most common numbers.
minExpt = 0::Int
maxExpt = 1100::Int
expt :: Integer -> Int -> Integer
expt base n =
    if base == 2 && n >= minExpt && n <= maxExpt then
        expts!n
    else
        base^n
```

```
expts :: Array Int Integer
expts = array (minExpt,maxExpt) [(n,2^n) | n <- [minExpt .. maxExpt]]

-- Compute the (floor of the) log of i in base b.
-- Simplest way would be just divide i by b until it's smaller then b,
-- but that would be very slow!  We are just slightly more clever.
integerLogBase :: Integer -> Integer -> Int
integerLogBase b i =
     if i < b then
        0
     else
        -- Try squaring the base first to cut down the number of divisions.
        let l = 2 * integerLogBase (b*b) i
            doDiv :: Integer -> Int -> Int
            doDiv i l = if i < b then l else doDiv (i `div` b) (l+1)
        in  doDiv (i `div` (b^l)) l


-- Misc utilities to show integers and floats

showSigned    :: Real a => (a -> ShowS) -> Int -> a -> ShowS
showSigned showPos p x | x < 0 = showParen (p > 6)
                                          (showChar '-' . showPos (-x))
                       | otherwise = showPos x

-- showInt is used for positive numbers only
showInt    :: Integral a => a -> ShowS
showInt n r | n < 0 = error "Numeric.showInt: can't show negative numbers"
            | otherwise =
              let (n',d) = quotRem n 10
                  r'     = toEnum (fromEnum '0' + fromIntegral d) : r
              in  if n' == 0 then r' else showInt n' r'


readSigned :: (Real a) => ReadS a -> ReadS a
readSigned readPos = readParen False read'
                     where read' r  = read'' r ++
                                      [(-x,t) | ("-",s) <- lex r,
                                                (x,t)   <- read'' s]
                           read'' r = [(n,s)  | (str,s) <- lex r,
                                                (n,"")  <- readPos str]


-- readInt reads a string of digits using an arbitrary base.
-- Leading minus signs must be handled elsewhere.

readInt :: (Integral a) => a -> (Char -> Bool) -> (Char -> Int) -> ReadS a
readInt radix isDig digToInt s =
   [(foldl1 (\n d -> n * radix + d) (map (fromIntegral . digToInt) ds), r)
          | (ds,r) <- nonnull isDig s ]

-- Unsigned readers for various bases
readDec, readOct, readHex :: (Integral a) => ReadS a
readDec = readInt 10 isDigit digitToInt
readOct = readInt  8 isOctDigit digitToInt
readHex = readInt 16 isHexDigit digitToInt


showEFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showFFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showGFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showFloat      :: (RealFloat a) => a -> ShowS

showEFloat d x =  showString (formatRealFloat FFExponent d x)
showFFloat d x =  showString (formatRealFloat FFFixed d x)
showGFloat d x =  showString (formatRealFloat FFGeneric d x)
showFloat      =  showGFloat Nothing

-- These are the format types.  This type is not exported.

data FFFormat = FFExponent | FFFixed | FFGeneric

formatRealFloat :: (RealFloat a) => FFFormat -> Maybe Int -> a -> String
formatRealFloat fmt decs x = s
  where base = 10
        s = if isNaN x then
              "NaN"
            else if isInfinite x then
                if x < 0 then "-Infinity" else "Infinity"
            else if x < 0 || isNegativeZero x then
                '-' : doFmt fmt (floatToDigits (toInteger base) (-x))
            else
                doFmt fmt (floatToDigits (toInteger base) x)
        doFmt fmt (is, e) =
            let ds = map intToDigit is
            in  case fmt of
                FFGeneric ->
                    doFmt (if e < 0 || e > 7 then FFExponent else FFFixed)
                          (is, e)
                FFExponent ->
                    case decs of
                    Nothing ->
                        case ds of
                         ['0'] -> "0.0e0"
                         [d]   -> d : ".0e" ++ show (e-1)
                         d:ds  -> d : '.' : ds ++ 'e':show (e-1)
                    Just dec ->
                        let dec' = max dec 1 in
                        case is of
                         [0] -> '0':'.':take dec' (repeat '0') ++ "e0"
                         _ ->
                          let (ei, is') = roundTo base (dec'+1) is
                              d:ds = map intToDigit
                                         (if ei > 0 then init is' else is')
                          in d:'.':ds  ++ "e" ++ show (e-1+ei)
                FFFixed ->
                    case decs of
                    Nothing ->
                        let f 0 s ds = mk0 s ++ "." ++ mk0 ds
                            f n s "" = f (n-1) (s++"0") ""
                            f n s (d:ds) = f (n-1) (s++[d]) ds
                            mk0 "" = "0"
                            mk0 s = s
                        in  f e "" ds
                    Just dec ->
```

```
                        let dec' = max dec 0 in
                        if e >= 0 then
                            let (ei, is') = roundTo base (dec' + e) is
                                (ls, rs) = splitAt (e+ei) (map intToDigit is')
                            in  (if null ls then "0" else ls) ++
                                (if null rs then "" else '.' : rs)
                        else
                            let (ei, is') = roundTo base dec'
                                                (replicate (-e) 0 ++ is)
                                d : ds = map intToDigit
                                             (if ei > 0 then is' else 0:is')
                            in  d : '.' : ds

roundTo :: Int -> Int -> [Int] -> (Int, [Int])
roundTo base d is = case f d is of
                (0, is) -> (0, is)
                (1, is) -> (1, 1 : is)
  where b2 = base `div` 2
        f n [] = (0, replicate n 0)
        f 0 (i:_) = (if i >= b2 then 1 else 0, [])
        f d (i:is) =
            let (c, ds) = f (d-1) is
                i' = c + i
            in  if i' == base then (1, 0:ds) else (0, i':ds)

--
-- Based on "Printing Floating-Point Numbers Quickly and Accurately"
-- by R.G. Burger and R. K. Dybvig, in PLDI 96.
-- This version uses a much slower logarithm estimator.  It should be improved.

-- This function returns a list of digits (Ints in [0..base-1]) and an
-- exponent.

floatToDigits :: (RealFloat a) => Integer -> a -> ([Int], Int)

floatToDigits _ 0 = ([0], 0)
floatToDigits base x =
    let (f0, e0) = decodeFloat x
        (minExp0, _) = floatRange x
        p = floatDigits x
        b = floatRadix x
        minExp = minExp0 - p            -- the real minimum exponent
        -- Haskell requires that f be adjusted so denormalized numbers
        -- will have an impossibly low exponent.  Adjust for this.
        (f, e) = let n = minExp - e0
                 in  if n > 0 then (f0 `div` (b^n), e0+n) else (f0, e0)

        (r, s, mUp, mDn) =
           if e >= 0 then
               let be = b^e in
               if f == b^(p-1) then
                   (f*be*b*2, 2*b, be*b, b)
               else
                   (f*be*2, 2, be, be)
           else
               if e > minExp && f == b^(p-1) then
                   (f*b*2, b^(-e+1)*2, b, 1)
               else
                   (f*2, b^(-e)*2, 1, 1)
        k =
            let k0 =
                  if b==2 && base==10 then
                      -- logBase 10 2 is slightly bigger than 3/10 so
                      -- the following will err on the low side.  Ignoring
                      -- the fraction will make it err even more.
                      -- Haskell promises that p-1 <= logBase b f < p.
                      (p - 1 + e0) * 3 `div` 10
                  else
                      ceiling ((log (fromInteger (f+1)) +
                                fromInt e * log (fromInteger b)) /
                               log (fromInteger base))
                fixup n =
                    if n >= 0 then
                        if r + mUp <= expt base n * s then n else fixup (n+1)
                    else
                        if expt base (-n) * (r + mUp) <= s then n
                                                           else fixup (n+1)
            in  fixup k0

        gen ds rn sN mUpN mDnN =
            let (dn, rn') = (rn * base) `divMod` sN
                mUpN' = mUpN * base
                mDnN' = mDnN * base
            in  case (rn' < mDnN', rn' + mUpN' > sN) of
                (True,  False) -> dn : ds
                (False, True)  -> dn+1 : ds
                (True,  True)  -> if rn' * 2 < sN then dn : ds else dn+1 : ds
                (False, False) -> gen (dn:ds) rn' sN mUpN' mDnN'
        rds =
            if k >= 0 then
                gen [] r (s * expt base k) mUp mDn
            else
                let bk = expt base (-k)
                in  gen [] (r * bk) s (mUp * bk) (mDn * bk)
    in  (map toInt (reverse rds), k)



-- This floating point reader uses a less restrictive syntax for floating
-- point than the Haskell lexer.  The '.' is optional.

readFloat     :: (RealFloat a) => ReadS a
readFloat r    = [(fromRational ((n%1)*10^^(k-d)),t) | (n,d,s) <- readFix r,
                                                       (k,t)   <- readExp s]
                  where readFix r = [(read (ds++ds'), length ds', t)
                                    | (ds,d) <- lexDigits r,
                                      (ds',t) <- lexFrac d ]

                        lexFrac ('.':ds) = lexDigits ds
                        lexFrac s        = [("",s)]

                        readExp (e:s) | e `elem` "eE" = readExp' s
                        readExp s                     = [(0,s)]

                        readExp' ('-':s) = [(-k,t) | (k,t) <- readDec s]
                        readExp' ('+':s) = readDec s
```

```
                    readExp' s       = readDec s

lexDigits        :: ReadS String
lexDigits        =  nonnull isDigit

nonnull          :: (Char -> Bool) -> ReadS String
nonnull p s      =  [(cs,t) | (cs@(_:_),t) <- [span p s]]
```

Ratio.hs

```
-- Standard functions on rational numbers

module  Ratio (
    Ratio, Rational, (%), numerator, denominator, approxRational ) where

infixl 7  %

prec = 7 :: Int

data  (Integral a)      => Ratio a = !a :% !a  deriving (Eq)
type  Rational          =  Ratio Integer

(%)                     :: (Integral a) => a -> a -> Ratio a
numerator, denominator  :: (Integral a) => Ratio a -> a
approxRational          :: (RealFrac a) => a -> a -> Rational


-- "reduce" is a subsidiary function used only in this module.
-- It normalises a ratio by dividing both numerator
-- and denominator by their greatest common divisor.
--
-- E.g., 12 `reduce` 8    ==  3 :%  2
--       12 `reduce` (-8) ==  3 :% (-2)

reduce _ 0              =  error "Ratio.% : zero denominator"
reduce x y              =  (x `quot` d) :% (y `quot` d)
                           where d = gcd x y

x % y                   =  reduce (x * signum y) (abs y)

numerator (x :% _)      =  x

denominator (_ :% y)    =  y


instance  (Integral a)  => Ord (Ratio a)  where
    (x:%y) <= (x':%y')  =  x * y' <= x' * y
    (x:%y) <  (x':%y')  =  x * y' <  x' * y

instance  (Integral a)  => Num (Ratio a)  where
    (x:%y) + (x':%y')   =  reduce (x*y' + x'*y) (y*y')
    (x:%y) * (x':%y')   =  reduce (x * x') (y * y')
    negate (x:%y)       =  (-x) :% y
    abs (x:%y)          =  abs x :% y
    signum (x:%y)       =  signum x :% 1
    fromInteger x       =  fromInteger x :% 1

instance  (Integral a)  => Real (Ratio a)  where
    toRational (x:%y)   =  toInteger x :% toInteger y

instance  (Integral a)  => Fractional (Ratio a)  where
    (x:%y) / (x':%y')   =  (x*y') % (y*x')
    recip (x:%y)        =  if x < 0 then (-y) :% (-x) else y :% x
    fromRational (x:%y) =  fromInteger x :% fromInteger y

instance  (Integral a)  => RealFrac (Ratio a)  where
    properFraction (x:%y) = (fromIntegral q, r:%y)
                            where (q,r) = quotRem x y

instance  (Integral a)  => Enum (Ratio a)  where
    toEnum           =  fromIntegral
    fromEnum         =  fromInteger . truncate -- May overflow
    enumFrom         =  numericEnumFrom -- These numericEnumXXX functions
    enumFromThen     =  numericEnumFromThen -- are as defined in Prelude.hs
    enumFromTo       =  numericEnumFromTo -- but not exported from it!
    enumFromThenTo   =  numericEnumFromThenTo

instance  (Read a, Integral a)  => Read (Ratio a)  where
    readsPrec p  =  readParen (p > prec)
                              (\r -> [(x%y,u) | (x,s)   <- reads r,
                                                ("%",t) <- lex s,
                                                (y,u)   <- reads t ])

instance  (Integral a)  => Show (Ratio a)  where
    showsPrec p (x:%y)  =  showParen (p > prec)
                           (shows x . showString " % " . shows y)


approxRational x eps    =  simplest (x-eps) (x+eps)
        where simplest x y | y < x      =  simplest y x
                           | x == y     =  xr
                           | x > 0      =  simplest' n d n' d'
                           | y < 0      =  - simplest' (-n') d' (-n) d
                           | otherwise  =  0 :% 1
                                        where xr@(n:%d) = toRational x
                                              (n':%d')  = toRational y

              simplest' n d n' d'       -- assumes 0 < n%d < n'%d'
                        | r == 0    =  q :% 1
                        | q /= q'   =  (q+1) :% 1
                        | otherwise =  (q*n''+d'') :% n''
                                    where (q,r)     =  quotRem n d
                                          (q',r')   =  quotRem n' d'
                                          (n'':%d'') =  simplest' d' r' d r
```

Appendix 2: Prelude for Clean (Definition modules)

Prelude.dcl

See implementation module Prelude.icl on page 47.

```
definition module Prelude

from StdOverloaded import
     class ==(..),      class <(..),    class +(..),        class -(..),        class *(..),    class /(..)
    ,class abs(..),     class zero(..), class one(..),      class exp(..)
    ,class sin(..),     class cos(..),  class asin(..),     class acos(..),     class atan(..)
    ,class sinh(..),    class cosh(..), class asinh(..),    class acosh(..),    class atanh(..)
    ,class fromInt(..), class sign(..), class toReal(..),   class fromReal(..), class ln(..)
    ,class toChar(..),  class fromChar(..),                 class fromString(..), class toString(..)
    ,class +++(..),     class ~(..)

from StdBool        import  not,||,&& ,instance == Bool

import  intInstances
import realInstances
import charInstances

from PreludeList    import  foldr,map,repeat,concat,iterate,takeWhile
from Ratio          import  :: Rational, :: Ratio(..)
from PreludeText    import  class Show(..), :: ShowS(..)

from PreludeLayer   import  from2,from_to,from_then_to
//=============================================================================

// -------- EQ class -----------

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

// -------- ORD class -----------

class Ord a | Eq, < a
where
    compare :: !a !a -> Ordering   | Ord a
    compare x y :== if (x==y) EQ (if (x<y) LT GT)

    (>) 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

// -------- ENUM class -----------

class Enum a
where
    toEnum          :: Int -> a
    fromEnum        :: a -> Int
    enumFromThen    :: a a -> [a]                       // [n,n`..]

    succ    :: !a -> a | Enum a
    succ  x :== toEnum ((fromEnum x) + 1)

    pred    :: !a -> a | Enum a
    pred  x :== toEnum ((fromEnum x) - 1)

    enumFrom :: !a -> [a]  | Enum a                     // [n..]
    enumFrom x :== map toEnum (from2 (fromEnum x))

    enumFromTo :: !a !a -> [a] | Enum a                 // [n..m]
    enumFromTo x y :== map toEnum (from_to (fromEnum x) (fromEnum y))

    enumFromThenTo :: !a !a !a -> [a] | Enum a          // [n,n`..m]
    enumFromThenTo x y z :== map toEnum (from_then_to (fromEnum x) (fromEnum y) (fromEnum z))

// -------- BOUNDED class -----------

class Bounded a
where
    minBound :: a
    maxBound :: a

// -------- NUM class -----------

class Num a | Eq,+,-,*,abs,fromInt,zero,one,Show a
where
    signum      :: a -> a

    negate      ::  a -> a  | Num a
    negate x    :== zero - x

/*  fromInteger     :: Integer -> a
            not supported yet */
class Num a | Eq,+,-,*,abs,fromInt,zero,one,Show a
where
     signum  :: a -> a

     negate  ::  a -> a| Num a
```

```
    negate x:== zero - x
/*  fromInteger :: Integer -> a not supported yet */

// -------- REAL class -----------

class Hs__Real a | Num, Ord a // shoulc be Real : name-space bug clean 2.0
where
    toRational :: a -> Rational

// -------- INTEGRAL class -----------

class  Integral a | Hs__Real,Enum a
where
    quotRem :: a a -> (a,a)
    toInteger:: a -> Int

    (quot)infixl 7 :: a a -> a | Integral a
    (quot) n d  :==  fst (quotRem n d)

    (rem) infixl 7:: a a -> a | Integral a
    (rem) n d    :==  snd (quotRem n d)

    (div) infixl 7 :: a a -> a | Integral a
    (div) n d    :==  fst (divMod n d)

    (mod) infixl 7:: a a -> a | Integral a
    (mod) n d    :==  snd (divMod n d)

    divMod         :: a a -> (a,a) | Integral a
    divMod n d:==
        let (q,r) = quotRem n d
        in
            if ( signum r == ( negate (signum d) )) (q-one, r+d) (q,r)

// -------- FRACTIONAL class -----------

class Fractional a  | Num,/ a
where
    fromRational    :: Rational -> a
    recip:: a -> a| Fractional a
    recip x :==  one / x

// -------- FLOATING class -----------
half :: a | one,/,+ a
//half  = one / (one+one)
minusone :: a | zero,-,one a
//minusone= zero - one
two :: a | one, + a
//two   = one + one

class Floating a | Fractional,exp,sin,cos,asin,acos,atan,sinh,cosh,asinh,acosh,atanh a
where
    pi              :: a
    log             :: a -> a

    (**) infixr 8:: a a -> a | Floating a
    (**) x y:== exp (log x * y)

    tan         :: a -> a| Floating a
    tan  x      :== sin x / cos x

    logBase     :: a a -> a| Floating a
    logBase x y:== log y / log x

    sqrt        :: !a -> a| Floating a
    sqrt x      :==  x ** half

    tanh        :: a -> a| Floating a
    tanh x      :==  sinh x / cosh x

// -------- REALFRAC class -----------

class RealFrac a | Hs__Real,Fractional a
where
    properFraction :: a -> (b,a)| Integral b

    truncate :: a -> b| RealFrac a & Integral b
    truncate x:== fst (properFraction x)

    round  :: a -> b| RealFrac a & Integral b
    round x :==
        let
            (n,r) = properFraction x
            m = if (r < zero) (n - one) (n + one)
        in
            case signum (abs r - half) of
                minusone-> n
                zero        -> if (even n) n m
                one             -> m

    ceiling:: a -> b| RealFrac a & Integral b
    ceiling x :== let (n,r) = properFraction x in
                      if (r > zero) (n + one) n

    floor:: a -> b | RealFrac a & Integral b
    floor x :== let (n,r) = properFraction x in
                if (r < zero) (n - one) n


// -------- REALFLOAT class -----------

class RealFloat a | RealFrac,Floating a
where
    floatRadix:: a -> Int
    floatDigits:: a -> Int
    floatRange:: a -> (Int,Int)
    decodeFloat:: a -> (Int,Int)
    encodeFloat:: Int Int -> a

    isNaN       :: a -> Bool
    isInfinite:: a -> Bool
    isDenormalized:: a -> Bool
    isNegativeZero:: a -> Bool
    isIEEE      :: a -> Bool
```

```
    exponent:: a -> Int| RealFloat a
    exponent x :==
        let (m,n) = decodeFloat x in
        if (m == 0) 0 (n + floatDigits x)

    significand :: a -> a| RealFloat a
    significand x :==
        let (m,_) = decodeFloat x in
        encodeFloat m (negate floatDigits x)

    scaleFloat :: Int a -> a| RealFloat a
    scaleFloat k x :==
        let (m,n) = decodeFloat x in
        encodeFloat m (n+k)

    atan2   :: a a -> a| RealFloat a
    atan2 y x:== atan2` y x

atan2` :: a a -> a | RealFloat a
/*
atan2` :: a a -> a | RealFloat a
atan2` y x
    | (x > zero)  = atan (y/x)
    | x == zero && y > zero  =  pi / two
    | x <  zero && y > zero  =  pi + atan (y/x)
    |       (x <= zero && y < zero)
        || (x <  zero && isNegativeZero y)
        || (isNegativeZero x && isNegativeZero y)
                = negate (atan2 (negate y) x)
    | y == zero && ( x < zero || isNegativeZero x)
                =  pi    // must be after the previous test on zero y
    | x == zero  && y== zero =  y     // must be after the other double zero tests
                             =  x + y // x or y is a NaN, return a NaN (via +)
*/

// ----------- Numeric functions ---------------

subtract :: a a -> a | Num a
even         :: !a -> Bool | Integral a
odd          :: !a -> Bool | Integral a
gcd          :: !a !a -> a| Integral a
lcm           :: !a !a -> a| Integral a
(^) infixr 8:: !a !b -> a| Num a & Integral b
(^^) infixr 8 :: !a !b -> a| Fractional a & Integral b
fromIntegral :: !a -> b| Integral a & Num b
realToFrac  :: !a -> b| Hs__Real a & Fractional b

class  Functor f
where
    fmap :: (a -> b) (f a) -> f b

class  Monad m
where
    (>>=) infixl 1:: (m a) (a -> m b) -> m b
    return      :: a -> m a
    fail        :: [Char] -> m a

    (>>) infixl 1:: (m a) (m b)-> m b  | Monad m
    (>>) m k:==  m >>= \_ -> k

sequence:: [m a] -> m [a]| Monad m
sequence_:: [m a] -> m Trivial| Monad m

// The xxxM functions take list arguments, but lift the function or list element to a monad type

mapM :: (a -> m b) [a] -> m [b]| Monad m
mapM_:: (a -> m b) [a] -> m Trivial| Monad m
(=<<) infixr 1 :: (a -> m b) (m a) -> m b| Monad m

// ----------------------------------

id      :: !a -> a
const:: !a b -> a

//(.) infixr 9 :: (b -> c) (a -> b) -> (a -> c) : automatically parsed as "o"

flip :: (a b -> c) b a -> c
(seq) infixr 0:: !a  b -> b
($) infixr 0 :: (a -> b) a -> b
($!) infixr 0:: (a -> b) !a -> b

// ----- Trivial data type ----------

:: Trivial = Trivial

instance == Trivial
instance <  Trivial
instance Enum Trivial
instance Bounded Trivial

// --------- BOOL type ------------

//:: Bool  =  False | True
//instance EqBool
instance < Bool
instance Enum Bool
instance Bounded Bool
/* See in PreludeGeneric:
instance Show Bool
instance Read Bool
*/

/*
not           :: !Bool->Bool
(||)infixr 2  :: !Bool Bool->Bool
(&&)infixr 3  :: !Bool Bool->Bool
otherwise     :: Bool
*/

// -------------- CHAR type -----------------------
```

```
//instance EqChar
//instance OrdChar
instance EnumChar
instance Bounded Char

// -------------- MAYBE type -----------------------

:: Maybe a =Just a
           |   Nothing

instance == (Maybe a)    | ==   a
instance < (Maybe a)     | <    a
/* See in PreludeGeneric:
instance Show  (Maybe a) | Show a
instance Read  (Maybe a) | Read a
*/

maybe :: b (a -> b) (Maybe a) -> b

instance Functor Maybe
instance Monad Maybe

// -------------- EITHER type -----------------------

:: Either a b  =  Left a
               | Right b

either :: (a -> c) (b -> c) (Either a b) -> c

/* See in PreludeGeneric:
instance ==    (Either a b)  | == a & == b
instance <     (Either a b)  | < a & < b
instance Show  (Either a b)  | Show a & Show b
instance Read  (Either a b)  | Read a & Read b
*/

/*
// -------------- IO type -----------------------
... can be found in PreludeIO

*/

// ------------- ORDERING type ---------------------
:: Ordering = LT | EQ | GT

instance == Odering
instance < Ordering
instance Enum Ordering
instance Bounded Ordering
/* See in PreludeGeneric:
instance Show Ordering
instance Read Ordering
*/

// ---------------- INSTANCES FOR BASIC CLASSES -----------------------------

// --- Int ---
// instance Eq Int
// instance OrdInt
instance NumInt
instance Hs_RealInt
instance EnumInt
instance IntegralInt
instance BoundedInt

// --- Real (Float)  ---
// instance Eq Int
// instance OrdInt
instance NumReal
instance Hs_RealReal
instance FractionalReal
instance FloatingReal
instance RealFracReal
//!!! instance RealFloat Real

numericEnumFrom:: a -> [a]| Fractional a
numericEnumFromThen:: a a -> [a]| Fractional a
numericEnumFromTo:: a a -> [a]| Fractional, Ord a
numericEnumFromThenTo:: a a a -> [a]| Fractional, Ord a

numericSucc :: a -> a | +, one a
numericPred :: a -> a | -, one a

instance EnumReal
// -------- LIST -----------------------
instance == [a] | Eq a
instance <  [a] | Ord a
instance Functor []
instance Monad   []

/* Tuples
data  (a,b)   =  (a,b)
data  (a,b,c) =  (a,b,c)
*/
/* See in PreludeGeneric:
instance Eq  (a,b)
instance Ord (a,b)
instance Eq  (a,b,c)
instance Ord (a,b,c)
instance Show (a,b)| Show a & Show b
nstance  Read (a,b)| Read a & Read b

*/
instance Bounded (a,b)   | Bounded a & Bounded b
instance Bounded (a,b,c) | Bounded a & Bounded b & Bounded c


// fst :: !(!.a,.b) -> .a
```

```
fst tuple :== t1 where (t1, _) = tuple
// snd  :: !(.a,!.b) -> .b
snd tuple :== t2 where (_, t2) = tuple

curry:: ((a, b) -> c) a b -> c
uncurry:: (a b -> c) (a, b) -> c


// until p f  yields the result of applying f until p holds.
until :: !(a -> Bool) (a -> a) a -> a

// asTypeOf is a type-restricted version of const.  It is usually used
// as an infix operator, and its typing forces its first argument
// (which is usually overloaded) to have the same type as the second.
asTypeOf  :: !a a -> a

//error stops execution and displays an error message
error :: [Char] -> a

// It is expected that compilers will recognize this and insert error
// messages that are more appropriate to the context in which undefined
// appears.
undefined       :: a

// ==================== END OF PRELUDE ============================================
```

PreludeList.dcl

See implementation module PreludeList.icl on page 53.

```
definition module PreludeList

from Prelude import class Eq(..),   class Num (..), class Ord (..), :: Ordering(..), :: Maybe(..)
                   ,class ==(..),   class <(..),    class +(..),    class -(..), class *(..), class abs(..)
    ,class zero(..),class one(..),  class exp(..),  class sin(..),  class cos(..),  class asin(..)
    ,class acos(..),class atan(..), class sinh(..), class cosh(..), class asinh(..),class acosh(..)
   ,class fromInt(..),              class sign(..), class toChar(..),              class fromChar(..)
    , class Show(..), ::ShowS

// ============================================================================

map             :: (a -> b) ![a] -> [b]
(++) infixr 5   :: ![a] [a] -> [a]
filter          :: (a -> Bool) [a] -> [a]
concat          :: ![[a]] -> [a]
head            :: ![a] -> a
last            :: ![a] -> a
tail            ::  [a] -> [a]
init            :: ![a] -> [a]
null            :: [a] -> Bool
length          :: ![a] -> Int
(!!) infixl 9   :: ![ a]  Int ->  a
foldl           :: (a b -> a) a ![b] -> a
foldl1          :: (a a -> a) ![a] -> a
foldr           :: (a b -> b) b ![a] -> b
foldr1          :: (a a -> a) ![a] -> a
scanl           :: (a b -> a) a ![b] -> [a]
scanl1          :: (a a -> a) ![a] -> [a]
scanr           :: (a b -> b) b ![a] -> [b]
scanr1          :: (a a -> a) ![a] -> [a]
iterate         :: (a -> a) a -> [a]
repeat          ::  a -> [a]
replicate       :: Int a -> [a]
cycle           :: [a] -> [a]
take            :: !Int [a] -> [a]
drop            :: Int ![a] -> [a]
splitAt         :: !Int [a] -> ([a],[a])
takeWhile       :: (a -> Bool) ![a] -> [a]
dropWhile       :: (a -> Bool) ![a] -> [a]
span            :: (a -> Bool) ![a] -> ([a],[a])
break           :: (a -> Bool) [a] -> ([a],[a])
lines           :: [Char]  -> [[Char]]
words           :: [Char]  -> [[Char]]
unlines         :: [[Char]] -> [Char]
unwords         :: [[Char]] -> [Char]
reverse         :: ![a] -> [a]
and             :: ![Bool] -> Bool
or              :: ![Bool] -> Bool
any             :: (a -> Bool) -> ![a] -> Bool
all             :: (a -> Bool) -> ![a] -> Bool

elem            :: a  [a] -> Bool | Eq a
notElem         :: a .[a] -> Bool | Eq a
lookup          :: a [(a,b)] -> Maybe b | Eq a
sum             :: ![a] -> a | Num  a
product         :: ![a] -> a | Num a
maximum         :: ![a] -> a | Ord a
minimum         :: ![a] -> a | Ord a

concatMap       :: (a -> [b]) ![a] -> [b]
zip             :: ![a] [b] -> [(a,b)]
zip3            :: ![a] [b] [c] -> [(a,b,c)]
zipWith         :: (a b -> c) ![a] [b] -> [c]
zipWith3        :: (a b c -> d) ![a] [b] [c] -> [d]
unzip           :: [(a,b)] -> ([a],[b])
unzip3          :: ![(a,b,c)] -> ([a],[b],[c])
```

PreludeListLayer.dcl

See implementation module PreludeListLayer.icl on page 54.

```
definition module PreludeListLayer

from Prelude import class Eq(..), class Num (..), class Ord (..), :: Ordering(..), :: Maybe(..), class Show
         ,class ==(..),  class <(..)
         ,class +(..),    class -(..),    class *(..),    class abs(..)
         ,class zero(..),class one(..),  class exp(..),  class sin(..),  class cos(..),  class asin(..)
         ,class acos(..),class atan(..), class sinh(..), class cosh(..), class asinh(..),class acosh(..)
         ,class fromInt(..),             class sign(..), class toChar(..),class fromChar(..)

hs_map              :: (a -> b) ![a] -> [b]
(###++) infixr 5    :: ![a] [a] -> [a]
hs_filter           :: (a -> Bool) [a] -> [a]
hs_concat           :: ![[a]] -> [a]
hs_head             :: ![a] -> a
hs_last             :: ![a] -> a
hs_tail             :: [a] -> [a]
hs_init             :: ![a] -> [a]
hs_null             :: [a] -> Bool
hs_length           :: ![a] -> Int
(###!!) infixl 9    :: ![ a]  Int ->  a
hs_foldl            :: (a b -> a) a ![b] -> a
hs_foldl1           :: (a a -> a) ![a] -> a
hs_foldr            :: (a b -> b) b ![a] -> b
hs_foldr1           :: (a a -> a) ![a] -> a
hs_scanl            :: (a b -> a) a ![b] -> [a]
hs_scanl1           :: (a a -> a) ![a] -> [a]
hs_scanr            :: (a b -> b) b ![a] -> [b]
hs_scanr1           :: (a a -> a) ![a] -> [a]
hs_iterate          :: (a -> a) a -> [a]
hs_repeat           ::  a -> [a]
hs_replicate        :: Int a -> [a]
hs_cycle            :: [a] -> [a]
hs_take             :: !Int [a] -> [a]
hs_drop             :: Int ![a] -> [a]
hs_splitAt          :: !Int [a] -> ([a],[a])
hs_takeWhile        :: (a -> Bool) ![a] -> [a]
hs_dropWhile        :: (a -> Bool) ![a] -> [a]
hs_span             :: (a -> Bool) ![a] -> ([a],[a])
hs_break            :: (a -> Bool) [a] -> ([a],[a])
hs_lines            :: [Char] -> [[Char]]
hs_words            :: [Char] -> [[Char]]
hs_unlines          :: [[Char]] -> [Char]
hs_unwords          :: [[Char]] -> [Char]
hs_reverse          :: ![a] -> [a]
hs_and              :: ![Bool] -> Bool
hs_or               :: ![Bool] -> Bool
hs_any              :: (a -> Bool) -> ![a] -> Bool
hs_all              :: (a -> Bool) -> ![a] -> Bool

hs_elem             :: a  [a] -> Bool | Eq a
hs_notElem          :: a ,[a] -> Bool | Eq a
hs_lookup           :: a [(a,b)] -> Maybe b | Eq a
hs_sum              :: ![a] -> a | Num a
hs_product          :: ![a] -> a | Num a
hs_maximum          :: ![a] -> a | Ord a
hs_minimum          :: ![a] -> a | Ord a

hs_concatMap        :: (a -> [b]) ![a] -> [b]
hs_zip              :: ![a] [b] -> [(a,b)]
hs_zip3             :: ![a] [b] [c] -> [(a,b,c)]
hs_zipWith          :: (a b -> c) ![a] [b] -> [c]
hs_zipWith3         :: (a b c -> d) ![a] [b] [c] -> [d]
hs_unzip            :: [(a,b)] -> ([a],[b])
hs_unzip3           :: ![(a,b,c)] -> ([a],[b],[c])
```

PreludeText.dcl

See implementation module PreludeText.icl on page 58.

```
definition module PreludeText

from PreludeList import ++
from StdFunc     import o
from Prelude     import :: Trivial(..)

// ------------------------------------------------------------------------------------

:: ReadS a  :== [Char] -> [(a,[Char])]
:: ShowS    :== [Char] -> [Char]

// -------- READ class ---------
class   Read a
where
    readsPrec   :: Int -> ReadS a
    readList    :: ReadS [a] | Read a
    readList    :== readList_

readList_ :: ReadS [a] | Read a
/*
readList_ = readParen False (\r -> [pr  \\  (['['],s)  <- lex r,    pr  <- readl s])
        where
            readl s =   [([],t)     \\  ([']'],t)  <- lex s]
                        ++
                        [([x:xs],u) \\  (x,t)    <- reads s, (xs,u) <- readl` t]
            readl` s=   [([],t)     \\  ([']'],t) <- lex s]
                        ++
                        [([x:xs],v) \\  ([','],t)  <- lex s, (x,u)    <- reads t, (xs,v)   <- readl` u]
*/

class   Show a
where
    showsPrec   :: Int a    -> ShowS

    show        :: a -> [Char]  | Show a
    show x      :== show_ x

    showList    :: [a] -> ShowS | Show a
    showList xs :== showList_ xs

showList_ :: [a] -> ShowS   | Show a
show_     :: a -> ShowS| Show a
/*
showList_ []        = showString ['[]']
showList_ [x:xs]    = showChar '[' o shows x o showl xs
    where
        showl []        = showChar ']'
        showl [x:xs]    = showChar ',' o shows x o showl xs
show_ x = showsPrec 0 x []
*/

// -------------

reads       :: ReadS a      | Read a
shows       :: a -> ShowS   | Show a

read        :: [Char] -> a  | Read a
showChar    :: Char -> ShowS
showString  :: [Char] -> ShowS

showParen   :: Bool ShowS -> ShowS
readParen   :: Bool (ReadS a) -> ReadS a
lex         :: ReadS [Char]

instance Show Int
instance Read Int
instance Show Real
instance Read Real

instance Show Trivial
instance Read Trivial
instance Show Char
instance Read Char

/* See in PreludeGeneric:
instance Show (a,b)| Show a & Show b
nstance  Read (a,b)| Read a & Read b
*/

readCharList :: [Char] -> [([Char],[Char])]
showCharList :: [Char] -> ShowS     // writes a string between ""
```

PreludeIO.dcl

See implementation module PreludeIO.icl on page 60.

```
definition module PreludeIO

from Prelude     import class Monad(..), class Functor(..), :: Trivial, :: Maybe(..)
from PreludeText import class Show(..),  :: ShowS(..)

import StdFile

:: StateMonad  state a  =   MkState (state -> *(a, state))

applyStateMonad     :: (StateMonad .state a) !.state -> (a, !.state)
appStateMonad       :: (*IOState -> *IOState) -> IO Trivial
accStateMonad       :: (*IOState -> *(a, *IOState)) -> IO a

instance Monad (StateMonad .state)
instance Monad IO
instance Functor IO

:: *IOState :== (*World, *File, *(Maybe *File), Maybe Error)
            //  (world, console, opened file,   errors)

:: IO a

:: Error    :== [Char]
:: FilePath :== [Char]
:: FileMode :== Int

doIO        :: (IO a) *World -> (a, *World) // used for testing

putChar     :: Char     -> IO Trivial
putStr      :: [Char]   -> IO Trivial
putStrLn    :: [Char]   -> IO Trivial
print       :: a        -> IO Trivial | Show a

getChar     :: IO Char
getLine     :: IO [Char]
getContents :: IO [Char]
interact    :: ([Char] -> [Char]) -> IO Trivial

readFile    :: FilePath -> IO [Char]
writeFile   :: FilePath [Char] -> IO Trivial
appendFile  :: FilePath [Char] -> IO Trivial
```

PreludeGeneric.dcl

See implementation module PreludeGeneric.icl on page 63.

```
definition module PreludeGeneric

import Prelude
from PreludeText    import class Show(..), ::ShowS, class Read(..), ::ReadS

:: Unit             = Unit                          // unit type; empty
:: Con a            = Con String a                  // Constructor
:: Pair a b         = Pair a b                      // product
//:: Either a b          = Left a | Right b       // disjoint sum

instance == Unit
instance == (Con a)       | Eq a
instance == (Pair a b)    | Eq a & Eq b
instance == (Either a b)  | Eq a & Eq b

instance < Unit
instance < (Con a)        | Ord a
instance < (Pair a b)     | Ord a & Ord b
instance < (Either a b)   | Ord a & Ord b

//instance == [a]           | Eq  a
//instance < [a]            | Ord a
instance == (a,b)         | Eq  a & Eq  b
instance < (a,b)          | Ord a & Ord b
instance == (a,b,c)       | Eq  a & Eq  b & Eq  c
instance < (a,b,c)        | Ord a & Ord b & Ord c

instance Show Unit
instance Show (Con a)       | Show a
instance Show (Pair a b)    | Show a & Show b
instance Show (Either a b)  | Show a & Show b

instance Read Unit
instance Read (Con a)       | Read a
instance Read (Pair a b)    | Read a & Read b
instance Read (Either a b)  | Read a & Read b

shown   :: Int a      -> [Char] |Show a
readsn  :: Int -> ReadS a  | Read a

class toGen source gen :: source -> gen

lexConstr :: String a -> ReadS a

instance Show Ordering
instance Show Bool
instance Show (Maybe a)  | Show a
instance Show [a]        | Show a
instance Show (a,b)      | Show a & Show b

instance Read Ordering
instance Read Bool
instance Read (Maybe a)  | Read a
instance Read [a]        | Read a
instance Read  (a,b)     | Read a & Read b
```

Char.dcl

See implementation module Char.icl on page 66.

```
definition module Char

from StdChar        import  isSpace,isAlpha,isDigit,isOctDigit,isHexDigit,isAlphanum,isUpper,isLower
from PreludeText    import  :: ShowS(..), :: ReadS(..)

ord :: Char -> Int
chr :: Int  -> Char

readLitChar :: ReadS Char
showLitChar :: Char -> ShowS
lexLitChar  :: ReadS [Char]
```

Numeric.dcl

See implementation module Numeric.icl on page 67.

```
definition module Numeric

from PreludeText    import  :: ReadS(..), :: ShowS(..)
import intInstances

showSigned  :: (a -> ShowS) Int a -> ShowS  | Hs__Real a
showInt     :: a -> ShowS   |   Integral a
readSigned  :: (ReadS a) -> ReadS a |   Hs__Real a
readInt     :: a (Char -> Bool) (Char -> Int) -> ReadS a | Integral a
readDec     :: ReadS a  | Integral a
readOct     :: ReadS a  | Integral a
readHex     :: ReadS a  | Integral a

lexDigits   :: ReadS [Char]
```

PreludeLayer.dcl

See implementation module PreludeLayer.icl on page 68.

```
definition module PreludeLayer
from StdBool import not

from2           :: !Int -> [Int]
from_to         :: !Int !Int -> [Int]
from_then       :: !Int !Int -> [Int]
from_then_to    :: !Int !Int !Int -> [Int]

// ---------- Conflicting classes and instances used in Clean --------------
/*
`quot` is integer division truncated toward zero,
while the result of `div` is truncated toward negative infinity.
*/
class   (hs__rem) infix 7   a :: !a !a -> a
instance hs__rem Int

class   (hs__quot) infix 7  a :: !a !a -> a
instance hs__quot Int

// -------------

hs__abort   :: ![Char] -> .a

// for PreludeIO
toStr       :: [Char] -> String
fromStr     :: String -> [Char]
removeNL    :: String -> String
```

Preludes.dcl

See implementation module Preludes.icl on page 68.

```
definition module Preludes

import PreludeList
import Prelude
import PreludeText
import PreludeGeneric
import PreludeIO
```

Ratio.dcl

See implementation module Ratio.icl on page 68.

```
definition module Ratio

from Prelude import class Integral, class Enum, class Hs__Real, class Ord, class Num, class Eq
     , class <, class one, class zero, class fromInt, class abs, class *, class -
     , class +, class ==, class Show
:: Ratio a = (:%) !a  !a
:: Rational :== Ratio Int

(%) infixl 7 :: a a -> Ratio a | Integral a
```

charInstances.dcl

See implementation module charInstances.icl on page 68.

```
definition module charInstances

from Prelude    import  class ==(..),       class <(..),        class fromChar(..),     class toChar(..)
                      , class Enum(..),     class Bounded(..),  class Ord(..) ,         class Eq(..), >=, not
from StdChar    import    instance ==       Char
                      , instance <        Char
                      , instance fromChar Char
                      , instance toChar   Char

from Prelude    import  //instance Eq        Char
                          //instance Ord       Char
                        instance Enum     Char
                      , instance Bounded  Char
```

intInstances.dcl

See implementation module intInstances.icl on page 68.

```
definition module intInstances

from Prelude import  class ==(..), class <(..),   class fromInt(..), class /(..),     class +(..)
                   , class -(..)  ,class *(..),   class abs(..),     class zero(..),  class one(..),
                   , class sign(..), class toReal(..), class fromReal(..),    class Integral(..)
                   ,class Enum(..),class Hs__Real(..), class Ord(..), class Num(..),class Eq(..)
                     ,class Bounded(..),class fromChar(..), class toChar(..), class Show
                    ,:: Rational, >
from Ratio          import :: Ratio

from StdInt import   instance ==        Int
                    ,instance <         Int
                    ,instance fromInt   Int
                    ,instance /         Int
                    ,instance +         Int
                    ,instance -         Int
                    ,instance *         Int
                    ,instance abs       Int
                    ,instance zero      Int
                    ,instance one       Int
                    ,instance sign      Int

from StdReal import  instance toReal    Int
                    ,instance fromReal  Int

from StdChar import  instance fromChar  Int
                    ,instance toChar    Int

from Prelude import  instance Num        Int
                    ,instance Hs__Real   Int
                    ,instance Enum       Int
                    ,instance Integral   Int
                    ,instance Bounded    Int
```

realInstances.dcl

See implementation module realInstances.icl on page 69.

```
definition module realInstances

from Prelude import class toReal,   class sign,     class fromReal,     class /,    class toReal,   class sign
                   ,class ln,       class *,        class ==,           class -,    class <,        class abs
                   ,class zero,     class one,      class +,            class exp,  class sin,      class cos
                   ,class asin,     class acos,     class atan,         class sinh, class cosh,     class asinh
                   ,class acosh,    class atanh,    class fromInt
                   ,class Num,      class Hs__Real, class Fractional,   class Floating, class RealFrac
                    ,class Eq,       class Ord,      class ~(..),        class toString, class Show(..),
                   ::ShowS
from StdReal import   instance ==    Real
                    , instance <     Real
                    , instance +     Real
                    , instance -     Real
                    , instance *     Real
                    , instance /     Real
                    , instance abs   Real
                    , instance zero  Real
                    , instance one   Real
                    , instance exp   Real
                    , instance sin   Real
                    , instance cos   Real
                    , instance asin  Real
                    , instance acos  Real
                    , instance atan  Real
                    , instance sinh  Real
                    , instance cosh  Real
                    , instance asinh Real
                    , instance acosh Real
                    , instance atanh Real
                    , instance sign  Real
                    , instance ln    Real
                    , instance ~     Real

from StdString      import instance toString Real

from StdInt import    instance fromInt Real

from Prelude import   instance Num        Real
                    , instance Hs__Real   Real
                    , instance Fractional Real
                    , instance Floating   Real
                    , instance RealFrac   Real
```

Appendix 3: Prelude for Clean (Implementation modules)

Prelude.icl

See definiton module Prelude.dcl on page 35.

```
implementation module Prelude

from StdOverloaded import
     class ==(..),  class <(..),    class +(..),    class -(..),    class *(..),    class /(..)
    ,class abs(..), class zero(..), class one(..),  class exp(..),  class ln(..)
    ,class sin(..), class cos(..),  class asin(..), class acos(..), class atan(..)
    ,class sinh(..),class cosh(..), class asinh(..),class acosh(..), class atanh(..)
    ,class fromInt(..),             class sign(..), class toChar(..),               class fromChar(..)
    ,class toReal(..), class fromReal(..)

from StdBool        import  not,||,&&,instance == Bool
from StdFunc        import  o
from StdMisc        import  abort

import  intInstances
import realInstances
import charInstances

from Ratio          import  :: Rational, :: Ratio(..)
from PreludeLayer   import  class hs_quot(..), class hs_rem(..),  instance hs_quot Int, instance hs_rem Int
                          , from2,from_to,from_then_to,toStr
from PreludeText    import  class Show(..), :: ShowS(..)
from PreludeList    import  foldr,repeat,concat,iterate,takeWhile,map

// ============================================================================================

// -------- EQ class -----------

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

// -------- ORD class -----------

class Ord a | Eq, < a
where
    compare :: !a !a -> Ordering   | Ord a
    compare x y :== if (x==y) EQ (if (x<y) LT GT)

    (>) 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

// -------- ENUM class -----------

class Enum a
where
    toEnum          :: Int -> a
    fromEnum        :: a -> Int
    enumFromThen    :: a a -> [a]                       // [n,n`..]

    succ    :: !a -> a | Enum a
    succ  x :== toEnum ((fromEnum x) + 1)

    pred    :: !a -> a | Enum a
    pred  x :== toEnum ((fromEnum x) - 1)

    enumFrom :: !a -> [a]  | Enum a                     // [n..]
    enumFrom x :== map toEnum (from2 (fromEnum x))

    enumFromTo :: !a !a -> [a] | Enum a                 // [n..m]
    enumFromTo x y :== map toEnum (from_to (fromEnum x) (fromEnum y))

    enumFromThenTo :: !a !a !a -> [a] | Enum a          // [n,n`..m]
    enumFromThenTo x y z :== map toEnum (from_then_to (fromEnum x) (fromEnum y) (fromEnum z))

// -------- BOUNDED class -----------

class Bounded a
where
    minBound :: a
    maxBound :: a

// -------- NUM class -----------

class Num a | Eq,+,-,*,abs,fromInt,zero,one,Show a
where
    signum      :: a -> a

    negate      ::  a -> a  | Num a
    negate x    :== zero - x

/*  fromInteger     :: Integer -> a
                not supported yet */
// -------- REAL class -----------
```

```
class Hs__Real a | Num, Ord a                   // shoulc be Real : name-space bug clean 2.0
where
    toRational :: a -> Rational

// -------- INTEGRAL class -----------

class  Integral a | Hs__Real,Enum a
where
    quotRem     :: a a -> (a,a)
    toInteger   :: a -> Int

    (quot)  infixl 7:: a a -> a | Integral a    // `quot` is integer division truncated toward zero
    (quot) n d      :==  fst (quotRem n d)

    (rem) infixl 7  :: a a -> a | Integral a
    (rem) n d       :==  snd (quotRem n d)

    (div) infixl 7  :: a a -> a | Integral a
    (div) n d       :==  fst (divMod n d)// `div` is integer division truncated toward negative infinity

    (mod) infixl 7  :: a a -> a | Integral a
    (mod) n d       :==  snd (divMod n d)

    divMod          :: a a -> (a,a) | Integral a
    divMod n d      :==
        let (q,r) = quotRem n d
        in
            if ( signum r == ( negate (signum d) )) (q-one, r+d) (q,r)

// -------- FRACTIONAL class -----------

class Fractional a  | Num,/ a
where
    fromRational    :: Rational -> a
    recip           :: a -> a   | Fractional a
    recip x         :==  one / x

// -------- FLOATING class -----------

half :: a | one,/,+ a
half        = one / (one+one)

minusone :: a | zero,-,one a
minusone    = zero - one

two :: a | one, + a
two         = one + one

class Floating a | Fractional,exp,sin,cos,asin,acos,atan,sinh,cosh,asinh,acosh,atanh a
where
    pi              :: a
    log             :: a -> a

    (**) infixr 8   :: a a -> a     | Floating a
    (**) x y        :== exp (log x * y)

    tan             :: a -> a       | Floating a
    tan  x          :== sin x / cos x

    logBase         :: a a -> a     | Floating a
    logBase x y     :== log y / log x

    sqrt            :: !a -> a      | Floating a
    sqrt x          :==  x ** half

    tanh            :: a -> a       | Floating a
    tanh x          :==  sinh x / cosh x

// -------- REALFRAC class -----------

class RealFrac a | Hs__Real,Fractional a
where
    properFraction :: a -> (b,a)    | Integral b

    truncate    :: a -> b       | RealFrac a & Integral b
    truncate x  :== fst (properFraction x)

    round  :: a -> b        | RealFrac a & Integral b
    round x :==
        let
            (n,r) = properFraction x
            m = if (r < zero) (n - one) (n + one)
        in
            case signum (abs r - half) of
                minusone        -> n
                zero            -> if (even n) n m
                one             -> m

    ceiling :: a -> b       | RealFrac a & Integral b
    ceiling x :== let (n,r) = properFraction x in
                      if (r > zero) (n + one) n

    floor   :: a -> b | RealFrac a & Integral b
    floor x :== let (n,r) = properFraction x in
                if (r < zero) (n - one) n

// -------- REALFLOAT class -----------

class RealFloat a | RealFrac,Floating a
where
    floatRadix      :: a -> Int
    floatDigits     :: a -> Int
    floatRange      :: a -> (Int,Int)
    decodeFloat     :: a -> (Int,Int)
    encodeFloat     :: Int Int -> a

    isNaN           :: a -> Bool
    isInfinite      :: a -> Bool
    isDenormalized  :: a -> Bool
    isNegativeZero  :: a -> Bool
    isIEEE          :: a -> Bool

    exponent    :: a -> Int | RealFloat a
    exponent x :==
```

```
        let (m,n) = decodeFloat x in
        if (m == 0) 0 (n + floatDigits x)

    significand :: a -> a       | RealFloat a
    significand x :==
        let (m,_) = decodeFloat x in
        encodeFloat m (negate floatDigits x)

    scaleFloat :: Int a -> a    | RealFloat a
    scaleFloat k x :==
        let (m,n) = decodeFloat x in
        encodeFloat m (n+k)

    atan2       :: a a -> a | RealFloat a
    atan2 y x   :== atan2` y x

atan2` :: a a -> a | RealFloat a
atan2` y x
    | (x > zero)                = atan (y/x)
    | x == zero && y > zero     =  pi / two
    | x <  zero && y > zero     =  pi + atan (y/x)
    |       (x <= zero && y < zero)
        ||  (x <  zero && isNegativeZero y)
        ||  (isNegativeZero x && isNegativeZero y)
                                = negate (atan2 (negate y) x)
    | y == zero && ( x < zero || isNegativeZero x)
                                =  pi    // must be after the previous test on zero y
    | x == zero  && y== zero    =  y     // must be after the other double zero tests
                                =  x + y // x or y is a NaN, return a NaN (via +)

// ----------- Numeric functions ---------------

subtract :: a a -> a | Num a
subtract a b =  flip (-) a b

even :: !a -> Bool | Integral a
even n = n rem two == zero

odd :: !a -> Bool | Integral a
odd n = not (even n)

gcd :: !a !a -> a   | Integral a
gcd x y | x == y && x == zero   =  abort "gcd 0 0 is undefined"
gcd x y =  gcd` (abs x) (abs y)
    where
        gcd` v z |  z == zero   =  v
                                =  gcd` z (v rem z)

lcm :: !a !a -> a   | Integral a
lcm z _     | z == zero =  zero
lcm _ z     | z == zero =  zero
lcm x y     =  abs ((x quot (gcd x y)) * y)


(^) infixr 8 :: !a !b -> a  | Num a & Integral b
(^) _ n | n == zero = one
        | n <  zero = abort "^: negative exponent"
(^) x n | n >  zero =  f x (n-one) x
    where
        f _ z y | z == zero = y
        f x n y             = g x n
        where
            g x n | even n  = g (x*x) (n quot two)
                            = f x (n-one) (x*y)

(^^) infixr 8 :: !a !b -> a | Fractional a & Integral b
(^^) x n    = if (n >= zero) (x^n) (recip (x^(negate n)))

fromIntegral ::  !a -> b    | Integral a & Num b
fromIntegral n = fromInt (toInteger n)

realToFrac  :: !a -> b  | Hs_Real a & Fractional b
realToFrac n = fromRational (toRational n)

// --------------- MONADIC classes ---------------------

class  Functor f
where
    fmap :: (a -> b) (f a) -> f b

class  Monad m
where
    (>>=) infixl 1  :: (m a) (a -> m b)     -> m b
    return          :: a -> m a

    fail            :: [Char] -> m a

    (>>) infixl 1   :: (m a) (m b)-> m b  | Monad m
    (>>) m k        :==  m >>= (\_ -> k)

sequence :: [m a] -> m [a]  | Monad m
sequence mxs =  foldr mcons (return []) mxs
    where mcons p q = p >>= \x -> q >>= \y -> return [x:y]

sequence_ :: [m a] -> m Trivial | Monad m
sequence_ mxs = foldr (>>) (return Trivial) mxs

// The xxxM functions take list arguments, but lift the function or list element to a monad type

mapM :: (a -> m b) [a] -> m [b] | Monad m
mapM f as =  sequence (map f as)

mapM_ :: (a -> m b) [a] -> m Trivial | Monad m
mapM_ f as =  sequence_ (map f as)

(=<<) infixr 1 :: (a -> m b) (m a) -> m b | Monad m
(=<<) f x =  x >>= f

id :: !a -> a
id x =  x

const :: !a b -> a
const x _  =  x

/*
```

```
"." is not a legal operator name in Clean, automatically parsed as "o"
(.) infixr 9 :: (b -> c) (a -> b) -> (a -> c)
(.) f g =  \ x -> f (g x)
*/

flip :: (a b -> c) b a -> c
flip f x y  =  f y x

(seq) infixr 0 :: !a  b -> b
(seq) x y = y

($) infixr 0 :: (a -> b) a -> b
($) f x =  f x

($!) infixr 0 :: (a -> b) !a -> b
($!) f x = x seq f x

// ----- Trivial data type ----------

:: Trivial = Trivial

instance == Trivial
    where (==) x y = True
instance < Trivial
    where (<)  x y = False

instance Enum Trivial
where
    toEnum  _ = Trivial
    fromEnum _ = 0
    enumFromThen _ _ = repeat Trivial

instance Bounded Trivial
where
    maxBound    = Trivial
    minBound    = Trivial

// --------- BOOL type ------------

//:: Bool  =  False | True

fromBool :: Bool -> Int
fromBool False  =  0
fromBool True   =  1

toBool :: Int -> Bool
toBool 0    =  False
toBool _    =  True

instance < Bool
    where   (<) x y  =  (fromBool x) <  (fromBool y)

instance Enum Bool
where
    toEnum  o   = toBool o
    fromEnum o =    fromBool o
    enumFromThen o o`
        = map toEnum ( from_then_to (fromEnum o) (fromEnum o`) (fromEnum lastElem))
        where
            lastElem :: Bool
            lastElem | o` < o   = minBound
                     | otherwise = maxBound

instance Bounded Bool
where
    minBound = False
    maxBound = True

// -------------- CHAR type ----------------------

instance Enum Char
where
    toEnum c    = toChar c
    fromEnum c  = fromChar c
//  in Haskell: enumFrom c        = map toEnum [fromEnum c .. fromEnum (maxBound::Char)]
    enumFromThen c c`
        = map toEnum ( from_then_to (fromEnum c) (fromEnum c`) (fromEnum lastElem))
        where
            lastElem :: Char
            lastElem | c` < c   = minBound
                     | otherwise = maxBound

instance  Bounded Char
where
    minBound    =  '\0'
    maxBound    =  '\255'


// -------------- MAYBE type ----------------------

:: Maybe a  =   Just a
            |   Nothing

instance == (Maybe a) | == a
where
    (==) Nothing Nothing    = True
    (==) (Just x) (Just y)  = x == y
    (==) _       _          = False

instance < (Maybe a) | < a
where
    (<) Nothing (Just x)    = True
    (<) (Just x) (Just y)   = x < y
    (<) _       _           = False

maybe :: b (a -> b) (Maybe a) -> b
maybe n f Nothing  =  n
maybe n f (Just x) =  f x

instance  Functor Maybe
where
    fmap f Nothing    =  Nothing
    fmap f (Just x)   =  Just (f x)
```

```
instance Monad Maybe
where
    (>>=) Nothing  k = Nothing
    (>>=) (Just x) k = k x

    return x    =  Just x
    fail s      =  Nothing

// -------------- EITHER type -----------------------

:: Either a b   =  Left a
                | Right b

either :: (a -> c) (b -> c) (Either a b) -> c
either f g (Left x)  =  f x
either f g (Right y) =  g y

/*
// -------------- IO type -----------------------
... can be found in PreludeIO
*/

// ------------- ORDERING type ---------------------
:: Ordering = LT | EQ | GT

fromOrdering :: Ordering -> Int
fromOrdering LT =   0
fromOrdering EQ =   1
fromOrdering GT =   2

toOrdering :: Int -> Ordering
toOrdering 0    =   LT
toOrdering 1    =   EQ
toOrdering _    =   GT

instance == Ordering
where   (==) x y =  (fromOrdering x) == (fromOrdering y)

instance < Ordering
where   (<) x y  =  (fromOrdering x) <  (fromOrdering y)

instance Enum Ordering
where
    toEnum  o   = toOrdering o
    fromEnum o =    fromOrdering o
    enumFromThen o o`
        = map toEnum ( from_then_to (fromEnum o) (fromEnum o`) (fromEnum lastElem))
        where
            lastElem :: Ordering
            lastElem | o` < o   = minBound
                     | otherwise = maxBound
instance Bounded Ordering
where
    minBound = LT
    maxBound = GT

// ============================ INSTANCES FOR BASIC CLASSES ==================================

// --- Int ---
// Eq, Ord

instance Num Int
where
    signum x = sign x

instance Hs_Real Int
where
    toRational :: Int -> Rational
    toRational n = (:%) n 1

instance Enum Int
where
    toEnum n    =   n
    fromEnum n  =   n
    enumFromThen n m = [n : enumFromThen m (m+(m-n)) ]

instance Integral Int
where
    quotRem x y = (x hs_quot y, x hs_rem y)
    toInteger n = n

instance Bounded Int
    where
        minBound = (-2147483647) // 32 bits
        maxBound =   2147483648

// --- Real (Float) ---
// Eq, Ord

instance Num Real
where
    signum n = toReal (sign n)

instance Hs_Real Real
where
    toRational :: Real -> Rational
    toRational r = toRat r 1
        where
            toRat :: Real Int -> Rational
            toRat r denomin
                | (r - toReal ri) == 0.0    =   (ri / gcdrd) :% (denomin / gcdrd )
                                            =   toRat (r*10.0) (denomin*10)
                where
                    ri      =   toInt r
                    gcdrd   =   gcd ri denomin

            toInt :: Real -> Int
            toInt r = fromReal r

instance Fractional Real
where
    fromRational :: Rational -> Real
    fromRational (n :% m) = toReal (n/m)
```

```
instance Floating Real
where
    pi  =   3.14159265358979323846264338327795
    log x = ln x

instance RealFrac Real
where
    properFraction :: Real -> (b,Real) | Integral b
    properFraction r
        | abs r < abs (toReal fromRealr)
            |   r > 0.0
                =   (toEnum (fromRealr-1),  r - toReal (fromRealr-1))
                =   (toEnum (fromRealr+1),  r - toReal (fromRealr+1))

        =   (toEnum fromRealr,      r - toReal fromRealr )
        where
            fromRealr   :: Int
            fromRealr   =   fromReal r

// !!! instance RealFloat Real

instance  Enum Real
where
    toEnum f            = fromIntegral f
    fromEnum f          = (fromInt o truncate) f    // may overflow
    enumFromThen f1 f2  = numericEnumFromThen f1 f2

numericEnumFrom :: a -> [a] | Fractional a
numericEnumFrom x = iterate ((+) one) x

numericEnumFromThen :: a a -> [a] | Fractional a
numericEnumFromThen n m = iterate ((+) (m-n)) n

numericEnumFromTo :: a a -> [a] | Fractional, Ord a
numericEnumFromTo n m = takeWhile ((>=) (m + half) ) (numericEnumFrom n)

numericEnumFromThenTo :: a a a -> [a] | Fractional, Ord a
numericEnumFromThenTo n n` m
    = takeWhile p (numericEnumFromThen n n`)
    where
        p   | n` > n    = ((>=) (m + (n`-n)/two))
            | otherwise = ((<=) (m + (n`-n)/two))

numericSucc :: a -> a | +, one a
numericSucc x = x + one

numericPred :: a -> a | -, one a
numericPred x = x - one

// -------- LIST -----------------------

instance == [a] | Eq a
    where
    (==) :: ![a]    ![a]    ->  Bool | Eq a
    (==) [] []          = True
    (==) [] _           = False
    (==) [_:_] []       = False
    (==) [a:as] [b:bs]
        | a == b        = as == bs
                        = False

instance < [a] | Ord a
    where
    (<) :: ![a] ![a] -> Bool | Ord a
    (<) [] []           = False
    (<) [] _            = True
    (<) [_:_] []        = False
    (<) [a:as] [b:bs]
        | a < b         = True
        | a > b         = False
                        = as < bs

instance Functor []
where
    fmap f xs = map f xs

instance  Monad []
where
    (>>=) m k       = concat (map k m)
    return x        = [x]
    fail s          = []

/* Tuples
data  (a,b)   =  (a,b)
data  (a,b,c) =  (a,b,c)
*/

/* See in PreludeGeneric:
instance Eq  (a,b)
instance Ord (a,b)
instance Eq  (a,b,c)
instance Ord (a,b,c)
*/

instance Bounded (a,b) | Bounded a & Bounded b
where
     minBound = (minBound,minBound)
     maxBound = (maxBound,maxBound)

instance Bounded (a,b,c) | Bounded a & Bounded b & Bounded c
where
     minBound = (minBound,minBound,minBound)
     maxBound = (maxBound,maxBound,maxBound)

// ------------------------------------------------------------------------------------

// fst      :: !(!.a,.b) -> .a
fst tuple :== t1 where (t1, _) = tuple
// snd      :: !(.a,!.b) -> .b
snd tuple :== t2 where (_, t2) = tuple

curry   :: ((a, b) -> c) a b -> c           // the same as in Clean
curry f x y      =  f (x, y)
```

```
uncurry :: (a b -> c) (a, b) -> c
uncurry f (x,y)      =  f x y


// until p f  yields the result of applying f until p holds.
until :: !(a -> Bool) (a -> a) a -> a
until p f x
    | p x   =  x
            =  until p f (f x)

// asTypeOf is a type-restricted version of const.  It is usually used
// as an infix operator, and its typing forces its first argument
// (which is usually overloaded) to have the same type as the second.

asTypeOf  :: !a a -> a
asTypeOf x y =  const x y

//error stops execution and displays an error message

error :: [Char] -> a
error msg   =  abort (toStr msg)

// It is expected that compilers will recognize this and insert error
// messages that are more appropriate to the context in which undefined
// appears.

undefined   :: a
undefined   =  error ['Prelude.undefined']

// ====================== END OF PRELUDE ================================================
```

PreludeList.icl

See definiton module PreludeList.dcl on page 40.

```
implementation module PreludeList

from Prelude import class Eq(..), class Num (..), class Ord (..), :: Ordering(..), :: Maybe(..)
           , class ==(..),  class <(..)
           ,class +(..),    class -(..),    class *(..),    class abs(..)
           ,class zero(..),class one(..),  class exp(..),  class sin(..),  class cos(..),  class asin(..)
           ,class acos(..),class atan(..), class sinh(..), class cosh(..), class asinh(..),class acosh(..)
           ,class fromInt(..),             class sign(..), class toChar(..),           class fromChar(..)

from PreludeListLayer import
    hs__map,    ###++,     hs__filter, hs__concat,     hs__head,      hs__last,       hs__tail,      hs__init,
    hs__null,   hs__length, ###!!,     hs__foldl,      hs__foldr,     hs__foldl1,     hs__foldr1,    hs__scanl,
    hs__scanr,  hs__scanl1, hs__scanr1, hs__iterate,    hs__repeat,    hs__replicate,  hs__cycle,     hs__takeWhile,
    hs__dropWhile,hs__span, hs__break,  hs__words,      hs__unlines,   hs__unwords,    hs__reverse,   hs__and,
    hs__or,     hs__any,    hs__all,    hs__elem,       hs__notElem,   hs__lookup,     hs__sum,       hs__product,
    hs__maximum,hs__minimum,hs__concatMap,hs__zip,      hs__zip3,      hs__zipWith,    hs__zipWith3,  hs__unzip,
    hs__unzip3, hs__take,   hs__drop,   hs__splitAt,hs__lines
// ============================================================================================

map :: (a -> b) ![a] -> [b]
map f xs        = hs__map f xs

(++) infixr 5   :: ![a] [a] -> [a]
(++) a b            = (###++) a b

filter  :: (a -> Bool) [a] -> [a]
filter p xs     = hs__filter p xs

concat  :: ![[a]] -> [a]
concat a        = hs__concat a

head    :: ![a] -> a
head xs             = hs__head xs

last    :: ![a] -> a
last xs         = hs__last xs

tail    ::  [a] -> [a]
tail xs             = hs__tail xs

init    :: ![a] -> [a]
init xs         = hs__init xs

null :: [a] -> Bool
null xs             = hs__null xs

length  :: ![a] -> Int
length xs       = hs__length xs

(!!) infixl 9 :: ![ a]  Int ->  a
(!!) li n       = (###!!) li n

foldl :: (a b -> a) a ![b] -> a
foldl f e xs    = hs__foldl f e xs

foldl1 :: (a a -> a) ![a] -> a
foldl1 f xs = hs__foldl1 f xs

foldr :: (a b -> b) b ![a] -> b
foldr f e xs    = hs__foldr f e xs

foldr1 :: (a a -> a) ![a] -> a
foldr1 f xs     = hs__foldr1 f xs

scanl :: (a b -> a) a ![b] -> [a]
scanl f q xs   = hs__scanl f q xs

scanl1 :: (a a -> a) ![a] -> [a]
scanl1 f xs     = hs__scanl1 f xs

scanr :: (a b -> b) b ![a] -> [b]
scanr f q0 xs  = hs__scanr f q0 xs

scanr1 :: (a a -> a) ![a] -> [a]
```

```
scanr1 f xs    = hs__scanr1 f xs

iterate :: (a -> a) a -> [a]
iterate f x     = hs__iterate f x

repeat ::  a -> [a]
repeat x           = hs__repeat x

replicate :: Int a -> [a]
replicate n x   = hs__replicate n x

cycle :: [a] -> [a]
cycle xs        = hs__cycle xs

take :: !Int [a] -> [a]
take n xs       = hs__take n xs

drop :: Int ![a] -> [a]
drop n xs       = hs__drop n xs

splitAt :: !Int [a] -> ([a],[a])
splitAt n xs    = hs__splitAt n xs

takeWhile :: (a -> Bool) ![a] -> [a]
takeWhile p xs  = hs__takeWhile p xs

dropWhile :: (a -> Bool) ![a] -> [a]
dropWhile p xs  = hs__dropWhile p xs

span    :: (a -> Bool) ![a] -> ([a],[a])
span p xs       = hs__span p xs

break :: (a -> Bool) [a] -> ([a],[a])
break p xs      = hs__break p xs

lines :: [Char] -> [[Char]]
lines xs           = hs__lines xs

words :: [Char] -> [[Char]]
words xs           = hs__words xs

unlines :: [[Char]] -> [Char]
unlines xs      = hs__unlines xs

unwords :: [[Char]] -> [Char]
unwords xs      = hs__unwords xs

reverse :: ![a] -> [a]
reverse xs      = hs__reverse xs

and :: ![Bool] -> Bool
and xs          = hs__and xs

or :: ![Bool] -> Bool
or  xs             = hs__or xs

any ::  (a -> Bool) -> ![a] -> Bool
any xs          = hs__any xs

all ::  (a -> Bool) -> ![a] -> Bool
all xs          = hs__all xs

elem :: a  [a] -> Bool | Eq a
elem e list     = hs__elem e list

notElem :: a .[a] -> Bool | Eq a
notElem e list = hs__notElem e list

lookup:: a [(a,b)] -> Maybe b | Eq a
lookup a xs     = hs__lookup a xs

sum:: ![a] -> a | Num  a
sum xs          = hs__sum xs

product :: ![a] -> a | Num a
product xs      = hs__product xs

maximum ::  ![a] -> a | Ord a
maximum xs      = hs__maximum xs

minimum ::  ![a] -> a | Ord a
minimum xs      = hs__minimum xs

concatMap :: (a-> [b]) ![a] -> [b]
concatMap f x   = hs__concatMap f x

zip::![a] [b] -> [(a,b)]
zip  x y           = hs__zip x y

/* in Clean:
zip::!(![.a],[.b]) -> [(.a,.b)]
zip (x,y) = zip2 x y
*/

zip3 :: ![a] [b] [c] -> [(a,b,c)]
zip3 al bl cl   = hs__zip3 al bl cl

zipWith :: (a b -> c) ![a] [b] -> [c]
zipWith f al bl = hs__zipWith f al bl

zipWith3 :: (a b c -> d) ![a] [b] [c] -> [d]
zipWith3 z a b c = hs__zipWith3 z a b c

unzip :: [(a,b)] -> ([a],[b])
unzip xs           = hs__unzip xs

unzip3 :: ![(a,b,c)] -> ([a],[b],[c])
unzip3 xs       = hs__unzip3 xs
```

PreludeListLayer.icl

See definiton module PreludeListLayer.dcl on page 41.

```
implementation module PreludeListLayer

from Prelude import class Eq(..), class Num (..), class Ord (..), :: Ordering(..), :: Maybe(..), class Show
        ,max,min,/=
        ,class ==(..),  class <(..),    class +(..),    class -(..),    class *(..),    class abs(..)
        ,class zero(..),class one(..),  class exp(..),  class sin(..),  class cos(..),  class asin(..)
        ,class acos(..),class atan(..), class sinh(..), class cosh(..), class asinh(..),class acosh(..)
        ,class fromInt(..),             class sign(..), class toChar(..),               class fromChar(..)

from StdInt         import  instance + Int, instance < Int
from StdChar        import  instance == Char

from StdMisc import abort

from StdList import map, ++,filter,flatten,last,init,!!,foldr,foldl,repeat,repeatn,iterate
                    ,take,takeWhile,drop,splitAt,dropWhile,span,isEmpty,reverse,and,or,any,all,
                    zip,unzip

from StdChar import isSpace
from StdFunc import o
from StdBool import not

// --------------------------------------------------------------------

hs__map :: (a -> b) ![a] -> [b]
hs__map f xs =  map f xs

(###++) infixr 5 :: ![a] [a] -> [a]
(###++) a b = (++) a b

hs__filter :: (a -> Bool) [a] -> [a]
hs__filter p xs = filter p xs

hs__concat :: ![[a]] -> [a]
hs__concat a = flatten a

hs__head :: ![a] -> a
hs__head [a:x]  = a
hs__head []     = abort "head of []"

hs__last :: ![a] -> a
hs__last xs =   last xs

hs__tail :: [a] -> [a]
hs__tail [a:x]  = x
hs__tail []     = abort "tail of []"

hs__init :: ![a] -> [a]
hs__init xs =   init xs

hs__null :: [a] -> Bool
hs__null []     = True
hs__null [x:xs]   = False

hs__length  :: ![a] -> Int
hs__length []       =   0
hs__length [_:l]    =   1 + hs__length l

(###!!) infixl 9 :: ![ a]  Int ->  a
(###!!) li n    =   (!!) li n

hs__foldl :: (a b -> a) a ![b] -> a
hs__foldl f e xs    =   foldl f e xs

hs__foldl1 :: (a a -> a) ![a] -> a
hs__foldl1 f [x:xs] = foldl f x xs
hs__foldl1 _ []     = abort "foldl1 of []"

hs__foldr :: (a b -> b) b ![a] -> b
hs__foldr f e xs    =   foldr f e xs

hs__foldr1 :: (a a -> a) ![a] -> a
hs__foldr1 f [x]    = x
hs__foldr1 f [x:xs] = f x (hs__foldr1 f xs)
hs__foldr1 _ []     = abort "foldr1 of []"

hs__scanl :: (a b -> a) a ![b] -> [a]
hs__scanl f q xs      = [ q : t]
    where t = case xs of
                []          -> []
                [x:xs]  -> hs__scanl f (f q x) xs

hs__scanl1 :: (a a -> a) ![a] -> [a]
hs__scanl1 f [x:xs] = hs__scanl f x xs
hs__scanl1 _ []     = abort "scanl1 of []"

hs__scanr :: (a b -> b) b ![a] -> [b]
hs__scanr f q0 []     = [q0]
hs__scanr f q0 [x:xs] = [f x q : qs ]
    where
        qs  = hs__scanr f q0 xs
        q   = hs__head qs

hs__scanr1 :: (a a -> a) ![a] -> [a]
hs__scanr1 f [x]        = [x]
hs__scanr1 f [x:xs] = [f x q : qs ]
    where
        qs  = hs__scanr1 f xs
        q   = hs__head qs
hs__scanr1 _ []     = abort "scanr1 of []"

hs__iterate :: (a -> a) a -> [a]
hs__iterate f x =   iterate f x

hs__repeat ::  a -> [a]
hs__repeat x    =   repeat x

hs__replicate :: Int a -> [a]
hs__replicate n x = repeatn n x

hs__cycle :: [a] -> [a]
hs__cycle [] = abort "cycle of []"
hs__cycle xs = xs1
            where xs1   =   xs ++ xs1

hs__take :: !Int [a] -> [a]
```

```
hs__take n xs
    | n<0   = abort "take: negative argument"
            = take n xs

hs__drop :: Int ![a] -> [a]
hs__drop n xs
    | n<0   = abort "drop: negative argument"
            = drop n xs

hs__splitAt :: !Int [a] -> ([a],[a])
hs__splitAt n xs
    | n<0   = abort "splitAt: negative argument"
            = splitAt n xs

hs__takeWhile ::(a -> Bool) ![a] -> [a]
hs__takeWhile f xs =    takeWhile f xs

hs__dropWhile :: (a -> Bool) ![a] -> [a]
hs__dropWhile f xs = dropWhile f xs

hs__span    :: (a -> Bool) ![a] -> ([a],[a])
hs__span p xs       = span p xs

hs__break :: (a -> Bool) [a] -> ([a],[a])
hs__break p l = span (not o p) l

hs__lines :: [Char] -> [[Char]]
hs__lines []    = []
hs__lines s     = [ w : remain ]
    where
        (w,s1) = hs__break ((==) '\n') s
        remain =
            case s1 of
                []      -> []
                [_:s2]  -> hs__lines s2
            //  ^ '\n'

hs__words :: [Char] -> [[Char]]
hs__words s
    | isEmpty s1    = []
                    = [w : hs__words s2]
        where
            (w,s2)  = hs__break isSpace s1
            s1          = dropWhile isSpace s

hs__unlines :: [[Char]] -> [Char]
hs__unlines  lines = hs__foldr1 ( \ x s -> x ++ ['\n'] ++ s ) lines

hs__unwords :: [[Char]] -> [Char]
hs__unwords words = hs__foldr1 ( \ x s -> x ++ [' '] ++ s ) words

hs__reverse :: ![a] -> [a]
hs__reverse xs  =   reverse xs

hs__and :: ![Bool] -> Bool
hs__and xs  =   and xs

hs__or :: ![Bool] -> Bool
hs__or  xs  =   or xs

hs__any ::  (a -> Bool) -> ![a] -> Bool
hs__any xs  =   any xs

hs__all ::  (a -> Bool) -> ![a] -> Bool
hs__all xs  =   all xs

hs__elem :: a  [a] -> Bool | Eq a
hs__elem e list = hs__any ((==) e ) list

hs__notElem :: a .[a] -> Bool | Eq a
hs__notElem e list = hs__all ((/=) e) list

hs__lookup:: a [(a,b)] -> Maybe b | Eq a
hs__lookup key []   =   Nothing
hs__lookup key [(x,y) : xys]
    | x == key  =   Just y
                =   hs__lookup key xys

hs__sum:: ![a] -> a | Num  a
hs__sum xs = accsum xs zero
where
    //accsum :: ![a] !a -> a | + a;
    accsum [x:xs] n = accsum xs (n + x)
    accsum []     n = n

hs__product :: ![a] -> a | Num a
hs__product xs = accprod one xs
  where
    accprod n [x:xs] = accprod (n * x) xs
    accprod n []     = n

hs__maximum ::  ![a] -> a | Ord a
hs__maximum []  = abort "maximum of []"
hs__maximum xs  = hs__foldl1 max xs

hs__minimum ::  ![a] -> a | Ord a
hs__minimum []  = abort "minimum of []"
hs__minimum xs = hs__foldl1 min xs

hs__concatMap :: (a-> [b]) ![a] -> [b]
hs__concatMap f x = flatten (map f x)

hs__zip::![a] [b] -> [(a,b)]
hs__zip  x y = zip (x,y)

/* in Clean:
zip::!([.a],[.b]) -> [(.a,.b)]
zip (x,y) = zip2 x y
*/

hs__zip3 :: ![a] [b] [c] -> [(a,b,c)]
hs__zip3 al bl cl = hs__zipWith3 (\ax bx cx -> (ax,bx,cx)) al bl cl

hs__zipWith :: (a b -> c) ![a] [b] -> [c]
hs__zipWith z [a:as] [b:bs] =   [z a b : hs__zipWith z as bs ]
hs__zipWith _ _      _      =   []
```

```
hs__zipWith3 :: (a b c -> d) ![a] [b] [c] -> [d]
hs__zipWith3 z [a:as] [b:bs] [c:cs] =   [z a b c : hs__zipWith3 z as bs cs]
hs__zipWith3 _ _      _      _      =   []

hs__unzip :: [(a,b)] -> ([a],[b])
hs__unzip xs    = unzip xs

hs__unzip3 :: ![(a,b,c)] -> ([a],[b],[c])
hs__unzip3 []        =   ([],[],[])
hs__unzip3 [(a,b,c): abcs] = ([a:as],[b:bs],[c:cs])
    where   (as,bs,cs)  =   hs__unzip3 abcs
```

PreludeText.icl

See definiton module PreludeText.dcl on page 42.

```
implementation module PreludeText

from PreludeList import ++, dropWhile,elem,span,map,!!, length
from StdFunc     import o
from StdChar     import isAlphanum

from StdReal        import instance fromReal {#Char}
from StdList        import instance fromString []
from Ratio          import %
from Char           import isSpace,isAlpha,isDigit,isOctDigit,isHexDigit,isUpper
                          ,showLitChar,readLitChar,lexLitChar
from Numeric        import showSigned, showInt, readSigned, readDec, lexDigits

import Prelude
// ------------------------------------------------------------------------------------------------------------

:: ReadS a  :== [Char] -> [(a,[Char])]
:: ShowS    :== [Char] -> [Char]

// -------- READ class ---------
class   Read a
where
    readsPrec   :: Int -> ReadS a

    readList    :: ReadS [a] | Read a
    readList    :== readList_

readList_ :: ReadS [a] | Read a
readList_ = readParen False (\r -> [pr  \\  (['['],s)  <- lex r,    pr  <- readl s])
        where
            readl s  =  [([],t)     \\  ([']'],t)  <- lex s]
                        ++
                        [([x:xs],u) \\  (x,t)    <- reads s, (xs,u) <- readl` t]
            readl` s =  [([],t)     \\  ([']'],t) <- lex s]
                        ++
                        [([x:xs],v) \\  ([','],t)  <- lex s, (x,u)    <- reads t, (xs,v)   <- readl` u]

class   Show a
where
    showsPrec   :: Int a    -> ShowS
    //showsPrec _ x s   :== show x ++ s

    show        :: a -> [Char]  | Show a
    show x      :== show_ x

    showList    :: [a] -> ShowS | Show a
    showList xs :== showList_ xs

showList_ :: [a] -> ShowS   | Show a
showList_ []        = showString ['[]']
showList_ [x:xs]    = showChar '[' o shows x o showl xs
    where
        showl []        = showChar ']'
        showl [x:xs]    = showChar ',' o shows x o showl xs

show_     :: a -> ShowS | Show a
show_ x   = showsPrec 0 x
// -------------

reads   :: ReadS a      | Read a
reads   = readsPrec 0

shows   :: a -> ShowS   | Show a
shows a = showsPrec 0 a

read    :: [Char] -> a  | Read a
read s  =
    case [x \\ (x,t) <- reads s, ([],[]) <- lex t] of
            [x] -> x
            []  -> error ['read: no parse']
            _   -> error ['read: ambiguous parse']

showChar    :: Char -> ShowS
showChar c = \cs ->  [c:cs]

showString  :: [Char] -> ShowS
showString str  = \cs  -> str ++ cs

showParen   :: Bool ShowS -> ShowS
showParen b p   =  if b (showChar '(' o p o showChar ')') p

readParen   :: Bool (ReadS a) -> ReadS a
readParen b g   =
    if b mandatory optional
    where
        optional r  = g r ++ mandatory r
        mandatory r = [(x,u)    \\  (['('],s) <- lex r, (x,t)   <- optional s,  ([')'],u) <- lex t  ]

lex             :: ReadS [Char]
lex = lex`
    where
        lex` []         =   [([],[])]
        lex` [c:s]
            | isSpace c  =   lex (dropWhile isSpace s)
        lex` ['\'':s]   =   [ (['\'':ch++['\'']], t)    \\ (ch,['\'':t])  <- lexLitChar s   //'c':character
                                                        |   ch /= ['\'']    ]
        lex` ['"':s]    =   [(['"':str], t) \\ (str,t) <- lexString s]                      // "string"
            where
                lexString ['"':s]  = [(['\"'],s)]                                       // empty string
                lexString s        = [(ch++str, u) \\  (ch,t)  <- lexStrItem s, (str,u) <- lexString t ]
                lexStrItem ['\\':['&':s]]   =  [(['\\&'],s)]
                                                        // '\&' is a null charachter (used for : "\65\&9" == "a9")
                lexStrItem ['\\':[c:s]]
                    /* Strings could be written in several lines, an '\' should be at the end of the line and
                       another one at the beginning of the next line */
                    | isSpace c         =   [   (['\\&'],t) \\  ['\\':t] <- [dropWhile isSpace s]   ]
                lexStrItem s            =   lexLitChar s
        lex` [c:s]
```

```
            | isSingle c    =   [([c],s)]
            | isSym c       =   [([c:sym],t)    \\ (sym,t) <- [span isSym s]   ]
            | isAlpha c     =   [([c:nam],t)    \\ (nam,t) <- [span isIdChar s]    ]
            | isDigit c     =   [([c:ds++fe],t) \\ (ds,s)  <- [span isDigit s], (fe,t)  <- lexFracExp s ]
                            =   []    // bad character
        where
            isSingle c  = elem c [',;()[]{}_`']
            isSym c     = elem c ['!@#$%&*+-./<=>?\\^|:~']
            isIdChar c  = isAlphanum c || elem c ['_','\''] //2.0['_\'']
            lexFracExp ['.':[c:cs]]
                | isDigit c = [   ([['.':ds++e],u) \\  (ds,t)  <- lexDigits [c:cs] , (e,u) <- lexExp t ]
            lexFracExp s    = [([],s)]

            lexExp [e:s]
                | elem e ['eE']
                    =   [([e:[c:ds]],u) \\  [c:t]  <- [s], (ds,u) <- lexDigits t
                                        |   elem c ['+-']   ] // e.g. e-56
                        ++
                        [([e:ds],t)     \\ (ds,t) <- lexDigits s ]  // e.g. e56
            lexExp s = [([],s)]

// ****** INT *******

instance Show Int
where showsPrec p n =   showSigned showInt p n

instance Read Int
where readsPrec p   =   readSigned readDec

// ****** REAL *******

instance Show Real
where
    showsPrec :: Int Real -> ShowS
    showsPrec p r
        | (-1.0) < r && r < 0.0 =   showString ( ['-0'] ++ fromString (toString (~r)))
        |    0.0 < r && r < 1.0 =   showString ( ['0'] ++ fromString (toString r))
                                =   showString (fromString (toString r))

instance Read Real
 where readsPrec p = readFloat

// ****** TRIVIAL *******

instance Show Trivial
  where    showsPrec p Trivial = showString ['()']

instance Read Trivial
where
    readsPrec p    = readParen False (\r -> [(Trivial,t) \\ (['('],s) <- lex r, ([')'],t) <- lex s ] )

// ****** CHAR *******

instance  Show Char
    where
        showsPrec p '\'' = showString ['\'','\\','\'','\'']
        showsPrec p c    = showChar '\'' o showLitChar c o showChar '\''

//instead of showList member of Show (which is predefined)
showCharList :: [Char] -> ShowS
showCharList cs = showChar '"' o showl cs
    where
        showl []        = showChar '"'
        showl ['"':cs]  = showString ['\\"'] o showl cs
        showl [c:cs]    = showLitChar c o showl cs


instance  Read Char
where
    readsPrec p = readParen False   (\r -> [(c,t) \\ (['\'':s],t)<- lex r, (c,['\'']) <- readLitChar s] )

//instead of readList member of Read (which is predefined)
readCharList :: [Char] -> [([Char],[Char])]
readCharList cs
    = readParen False (\r -> [(l,t) \\ (['"':s], t) <- lex r,   (l,_) <- readl s ]) cs
    where
        readl ['"':s]           = [([],s)]
        readl ['\\':['&':s]]    = readl s
        readl s                 = [([c:cs],u) \\ (c ,t) <- readLitChar s, (cs,u) <- readl t ]

/* See in PreludeGeneric:
instance Show (a,b)| Show a & Show b
nstance  Read (a,b)| Read a & Read b
*/
// ------------------------ from NUMERIC library ----------------------------------------------------

readFloat
    = \r -> [   (   fromRational (n % (toEnum 1) ) * (toEnum  10) ^^ (k-d)
            , t)
        \\  (n,d,s) <- readFix r,   (k,t)   <- readExp s
        ]
    where
        readFix r = [(read (ds++ds`), length ds`, t)    \\  (ds,d) <- lexDigits r,  (ds`,t) <- lexFrac d ]

        lexFrac ['.':ds] = lexDigits ds
        lexFrac s        = [([],s)]

        readExp [e:s] | elem e ['eE']   = readExp` s
        readExp s                       = [(0,s)]

        readExp` ['-':s]    = [(negate k,t) \\ (k,t) <- readDec s]
        readExp` ['+':s]    = readDec s
        readExp` s          = readDec s
```

PreludeIO.icl

See definiton module PreludeIO.dcl on page 43.

```
implementation module PreludeIO

from Prelude        import  class Monad(..), class Functor(..), :: Trivial(..), :: Maybe(..), class ==(..), class +++(..)
from PreludeText    import  class Show(..), :: ShowS(..), show, show_
from StdString      import  instance +++ {#Char}
from StdInt         import  instance == Int

import StdFile,StdArray

from StdFunc        import  o
from Prelude        import  error

from PreludeList    import  ++
from PreludeLayer   import  toStr,fromStr,removeNL

// --------------------------------------------------------------------
:: StateMonad  state a  =   MkState (state -> *(a, state))

applyStateMonad     :: (StateMonad .state a) !.state -> (a, !.state)
applyStateMonad stMon =: (MkState stFun) st = stFun st

appStateMonad       :: (*IOState -> *IOState) -> IO Trivial
appStateMonad stCh  = IO (MkState (\st -> (Trivial, stCh st)))

accStateMonad       :: (*IOState -> *(a, *IOState)) -> IO a
accStateMonad stFun = IO (MkState stFun)

// instance for monads containing state
instance Monad (StateMonad .state)
where
    return val          = (MkState (\st -> (val,st)))

    (>>=) mon1 mon2     = MkState mon3
        where mon3 st = (val2, st2)
                where (val1, st1) = applyStateMonad mon1 st
                      tmon        = mon2 val1
                      (val2, st2) = applyStateMonad tmon st1
    fail s = error s

instance Monad IO
where
    return val  =   IO (MkState (\st -> (val,st)))
    (>>=) (IO mon1_) mon2       = IO (MkState mon3)
        where
            mon3 st = (val2, st2)
            where
                (val1, st1) = applyStateMonad mon1_ st
                tmon        = (conv mon2) val1
                (val2, st2) = applyStateMonad tmon st1

    fail s = error s

instance Functor IO
where
    fmap f x = x >>= (return o f)

conv :: (a -> IO b) -> (a -> StateMonad *IOState b)
conv f  =   \a -> fromIO (f a)

fromIO :: (IO a) -> (StateMonad *IOState a)
fromIO (IO a)   = a


// --------------------------------------------------------------------

:: *IOState :== (*World, *File, *(Maybe *File), Maybe Error)
            //  (world, console, opened file,   errors)

:: IO a = IO (StateMonad *IOState a)

:: Error    :== [Char]
:: FilePath :== [Char]
:: FileMode :== Int

// --------------------------------------------------------------------
getErrorStr :: (Maybe Error) -> String
getErrorStr Nothing     = "\n\n\nNo errors\n"
getErrorStr (Just str)  = "\n\n\n" +++ toStr str +++ "\n"

// adds new error message to the exsisting list of errors
mkError :: (Maybe Error)  [Char] -> Maybe Error
mkError Nothing     cs  = (Just cs)
mkError (Just cs1) cs2
    # newcs = cs1 ++ ['\n'] ++ cs2
    = (Just newcs)

// opens console, applies the given monad to an initial state, lists the errors
// occured while processing the monad, and prompts for exit
doIO        :: (IO a) *World -> (a, *World)
doIO (IO mon) world
    # (MkState stfun)                   = mon
      (console, world)                  = stdio world
      (val, (world, console, f, err))   = stfun (world, console, Nothing, Nothing)
      console                           = fwrites (getErrorStr err) console
      (_, world)                        = fclose console world
    = (val, world)

// ----- for console ------------------------------------------------------------

// --- Output ---

put_Char :: Char *IOState -> *IOState
put_Char c state =: (w, con, f, err)
    # con   = fwritec c con
    = (w, con, f, err)

putChar :: Char -> IO Trivial
putChar c = appStateMonad (put_Char c)
```

```
put_Str :: [Char] *IOState -> *IOState
put_Str c state =: (w, con, f, err)
    # con   = fwrites (toStr c) con
    = (w, con, f, err)

putStr  :: [Char] -> IO Trivial
putStr cs = appStateMonad (put_Str cs)

putStrLn :: [Char] -> IO Trivial
putStrLn cs = appStateMonad (put_Str ( cs ++ ['\n']))

print :: a -> IO Trivial | Show a
print a = appStateMonad (put_Str ( (show a) ++ ['\n']))

// --- Input ---

get_Char :: *IOState -> (Char, *IOState)
get_Char state =: (w, con, f, err)
    # (ok, c, con)  = freadc con
    | ok    = (c, (w, con, f, err))
            = (c, (w, con, f, mkError err ['Cannot get char.']))

getChar :: IO Char
getChar = accStateMonad get_Char

get_Line :: *IOState -> ([Char], *IOState)
get_Line state =: (w, c, f, err)
    # (str, c)      = freadline c
    | size str == 0
        =   (fromStr str, (w, c, f,mkError err ['cannot get line.']))
        #   cs = fromStr (removeNL str)
        = (cs, (w, c, f, err))

getLine :: IO [Char]
getLine = accStateMonad get_Line

get_Contents :: *IOState -> ([Char], *IOState)
get_Contents state =: (w, con, f, err)
    #   (con,content) = get_Contents` con []
    = (content, (w, con, f, err))

get_Contents` :: *File [Char] -> (*File ,[Char])
get_Contents` con buf
    #   (ok, c, con)    = freadc con
    | ok    =   get_Contents` con (buf ++ [c])
            =   (con,buf)

getContents :: IO [Char]
getContents = accStateMonad get_Contents

interact_ :: ([Char] -> [Char]) *IOState -> *IOState
interact_ strfun state =: (w, con, f, err)
    #   (con,content)   = get_Contents` con []
        con             = fwrites (toStr (strfun content)) con
    = (w, con, f, err)

interact :: ([Char] -> [Char]) -> IO Trivial
interact strfun = appStateMonad (interact_ strfun)

// ----- for file ------------------------------------------------------------

// opens a file; if there already is an open file, gives an error message
open_file :: FilePath Int *IOState -> *IOState
open_file fn m state =: (w, c, (Just f), err)
    # err = mkError err ['there is already an opened file.']            //!!! baj lehet belole
    = (w, c, Nothing, err)
open_file fn m state =: (w, c, Nothing, err)
    # (ok, f, w) = fopen (toStr fn) m w
    | ok
    = (w, c, (Just f), err)
        # err = mkError err (['Unable to open file '] ++ fn ++ ['.'])
        = (w, c, Nothing, err)

// closes the opened file; gives an error message if there's no open file
close_file  :: *IOState -> (Trivial, *IOState)
close_file state =: (w, c, Nothing, err)
    # err = mkError err ['No file to close.']
    = (Trivial, (w, c, Nothing, err))
close_file state =: (w, c, (Just f), err)
    # (ok, w)   = fclose f w
    | ok
    = (Trivial, (w, c, Nothing, err))
        # err = mkError err ['Unable to close file.']
        = (Trivial, (w, c, Nothing, err))

// --- Input ---

read_File :: FilePath *IOState -> ([Char], *IOState)
read_File fp state
    #   state                   =   open_file fp FReadText state
        (w, con, f, err)    =   state
    =   fun w con f err
        /*
        case f of
            (Just fl)
                #   (fl,content) = get_Contents` fl []
                ->  (content,(w,con,(Just fl),err,cl))
            Nothing     ->  ([], state)
        */
    where
        fun w con (Just fl) err
            #   (fl,content)    = get_Contents` fl []
                (_,state)       = close_file (w,con,(Just fl),err)
            =   (content,state)
        fun w con Nothing err   = ([], (w,con,Nothing,err))

readFile :: FilePath -> IO [Char]
readFile fp = accStateMonad (read_File fp)

// --- Output ---

write_File :: FilePath [Char] *IOState -> *IOState
write_File fp cs state
    #   state                   =   open_file fp FWriteText state
        (w, con, f, err)    =   state
    =   fun w con f err
```

```
    where
        fun w con (Just fl) err
            #   fl                  = fwrites (toStr cs) fl
                (_,state)           = close_file (w,con,(Just fl),err)
            =   state
        fun w con Nothing err   = (w,con,Nothing,err)


writeFile :: FilePath [Char] -> IO Trivial
writeFile fp cs = appStateMonad (write_File fp cs)

append_File :: FilePath [Char] *IOState -> *IOState
append_File fp cs state
    #   state                   =   open_file fp FAppendText state
        (w, con, f, err)    =   state
    =   fun w con f err
    where
        fun w con (Just fl) err
            #   fl                  = fwrites (toStr cs) fl
                (_,state)           = close_file (w,con,(Just fl),err)
            =   state
        fun w con Nothing err   = (w,con,Nothing,err)

appendFile :: FilePath [Char] -> IO Trivial
appendFile fp cs = appStateMonad (append_File fp cs)
```

PreludeGeneric.icl

See definiton module PreludeGeneric.dcl on page 44.

```
implementation module PreludeGeneric

from StdBool        import not, &&, ||
from StdList        import ++, instance fromString [x], instance toString [x]

from Prelude        import :: Either(..), :: Ordering(..), :: Maybe(..), class fromString(..), class toString(..)
                         , instance == [a]
from PreludeText    import class Show(..), class Read(..), ::ReadS, ::ShowS, show, showString, show_
                        ,read, reads, lex, showList, showList_, readList, readList_
from Char           import lexLitChar, readLitChar

import  intInstances
import charInstances
// ------------ Types used to encode the generic representation type of kind * ----------------------

:: Unit             = Unit                      // unit type; empty
:: Con a            = Con String a              // Constructor
:: Pair a b         = Pair a b                  // product
//:: Either a b         = Left a | Right b      // disjoint sum

class toGen source gen :: source -> gen

// ------ Instances of Eq --------

instance == Unit
where
    (==)  _ _ = True

instance == (Con a)    | Eq a
where
    (==) (Con _ x) (Con _ y)        =  x == y       // no test on equality of strings

instance == (Pair a b) | Eq a & Eq b
where
    (==) (Pair x1 y1) (Pair x2 y2)  = (x1 == x2) && (y1 == y2)

instance == (Either a b) | Eq a & Eq b
where
    (==) (Left x)    (Left y)   = x == y
    (==) (Right x)   (Right y)  = x == y
    (==) _           _          = False

// ------ Instances of Ord --------
instance < Unit
where
    (<)  _ _ = True

instance < (Con a) | Ord a
where
    (<) (Con _ x) (Con _ y)         =  x < y

instance < (Pair a b) | Ord a & Ord b
where
    (<) (Pair x1 y1) (Pair x2 y2) = f (x1,y1) (x2,y2)
    where
        f (x1,y1) (x2,y2)
          |  x1 == x2   = y1 < y2
                        = x1 < x2

instance < (Either a b) | Ord a & Ord b
where
    (<) (Left x)     (Left y)   = x < y
    (<) (Right x)    (Right y)  = x < y
    (<) (Left _)     (Right _)  = True
    (<) (Right _)    (Left _)   = False

// ------ Instances of Show --------
instance Show Unit
where
    showsPrec _ Unit = \cs -> cs

instance Show (Con a)    | Show a
where
    showsPrec n (Con name x) = showString( openbr ++ fromString name ++ [' '] ++ show x ++ closebr)
        where
            openbr  | n > 0     = ['(']
                    =           []
            closebr | n > 0     = [')']
                    =           []

instance Show (Pair a b) | Show a & Show b
where
    showsPrec _ (Pair x y) = showString( show x ++ [' '] ++ show y)

instance Show (Either a b) | Show a & Show b
where
    showsPrec _ (Left  a) = showString (show a)
    showsPrec _ (Right b) = showString (show b)

// ------ Instances of Read --------
instance Read Unit
where
    readsPrec _ = \cs -> [(Unit,cs)]

instance Read (Con a) | Read a
where
    readsPrec 0 = \cs -> [(Con (toString name) x,tail_)
                              \\     (name,cs3)   <- lex  cs
                              ,  (' ', cs4)   <- readLitChar cs3
                              ,  ( x , tail_) <- reads       cs4
                         ]
    readsPrec _ = \cs -> [(Con (toString name) x,tail_)
                              \\     ('(', cs2)   <- readLitChar cs
                              ,  (name,cs3)   <- lex  cs2
                              ,  (' ', cs4)   <- readLitChar cs3
                              ,  ( x , cs5)   <- reads       cs4
                              ,  (')', tail_) <- readLitChar cs5
                         ]
```

```
instance Read  (Pair a b) | Read a & Read b
where
    readsPrec _ = \cs ->    [ (Pair x y, tail_) \\  ( x ,cs2) <- reads cs
                                                ,   (' ',cs3) <- readLitChar cs2
                                                ,   ( y ,tail_) <- reads cs3
                            ]

instance Read (Either a b) | Read a & Read b
where
    readsPrec _ = \cs ->    [(Left x, res1) \\ (x, res1) <- reads cs]
                      ++    [(Right y, res2) \\ (y, res2) <- reads cs]


// ----------------------

lexConstr :: String a -> ReadS a
lexConstr const x = \cs -> [(x,res) \\ (le, res) <- lex cs | le == constr ]
    where
        constr :: [Char]
        constr = fromString const

// ----------------------

shown   :: Int a      -> [Char] |Show a
shown n x = (showsPrec n x) []

readsn  :: Int -> ReadS a  | Read a
readsn n = \cs ->  readsPrec n cs

// ************************** Basic type's generic encodings ******************************

// ------ LIST -------------

:: GenList a :== Either (Con Unit) (Con (Pair a [a]))

instance toGen [a] (Either (Con Unit) (Con (Pair a [a])))
where
    toGen []        = Left (Con "Nil" Unit)
    toGen [x: xs]   = Right (Con "Cons" (Pair x xs))

ListToGen :: [a] -> GenList a
ListToGen l = toGen l

// ------ TUPLE -------------

:: GenTuple a b :== Pair a b

instance toGen (a,b) (Pair a b)
where
    toGen (x,y)     = Pair x y

TupleToGen :: (a,b) -> (GenTuple a b)
TupleToGen l = toGen l

// ------ TUPLE -------------

:: GenTriple a b c :== Pair a (Pair b c)

instance toGen (a,b,c) (Pair a (Pair b c))
where
    toGen (x,y,z)   = Pair x (Pair y z)

TripleToGen :: (a,b,c) -> (GenTriple a b c)
TripleToGen l = toGen l

// ------- Instantiations of basic types (Eq, Ord)  ---------------------------------------
/*
instance == [a] | Eq a
where   (==) xs1 xs2    = (ListToGen xs1) == (ListToGen xs2)

instance < [a] | Ord a
where   (<) xs1 xs2     = (ListToGen xs1) < (ListToGen xs2)
*/
instance == (a,b) | Eq a & Eq b
where   (==) xs1 xs2 = (TupleToGen xs1) == (TupleToGen xs2)

instance < (a,b) | Ord a & Ord b
where   (<) xs1 xs2  = (TupleToGen xs1) < (TupleToGen xs2)

instance == (a,b,c) | Eq a & Eq b & Eq c
where   (==) xs1 xs2 = (TripleToGen xs1) == (TripleToGen xs2)

instance < (a,b,c) | Ord a & Ord b & Ord c
where   (<) xs1 xs2  = (TripleToGen xs1) < (TripleToGen xs2)

// ----- ORDERING -----------------

instance Show Ordering
where
    showsPrec _ LT  =   showString ['LT']
    showsPrec _ EQ  =   showString ['EQ']
    showsPrec _ GT  =   showString ['GT']

instance Read Ordering
where
    readsPrec _ = \cs ->        [(o1,res1) \\ (o1,res1) <- (lexConstr "LT" LT) cs]
                          ++    [(o1,res1) \\ (o1,res1) <- (lexConstr "EQ" EQ) cs]
                          ++    [(o1,res1) \\ (o1,res1) <- (lexConstr "GT" GT) cs]

// ----- BOOL ---------------------

instance Show Bool
where
    showsPrec _ True    =   showString ['True']
    showsPrec _ False   =   showString ['False']

instance Read Bool
where
    readsPrec _ = \cs ->        [(o1,res1) \\ (o1,res1) <- (lexConstr "True" True) cs]
                          ++    [(o1,res1) \\ (o1,res1) <- (lexConstr "False" False) cs]

// ----- MAYBE a  -----------------
```

```
:: GenMaybe a :== Either (Con Unit) (Con a)

instance toGen (Maybe a) (Either (Con Unit) (Con a))
where
    toGen Nothing   = Left  (Con "Nothing" Unit)
    toGen (Just a)  = Right (Con "Just"    a)

MaybeToGen :: (Maybe a) -> GenMaybe a
MaybeToGen l = toGen l

MaybeFromGenMaybe :: (GenMaybe a) -> Maybe a
MaybeFromGenMaybe (Left  (Con "Nothing" Unit))  =   Nothing
MaybeFromGenMaybe (Right (Con "Just"    a))     =   Just a

instance Show (Maybe a) | Show a
where
    showsPrec _ Nothing =   showString ['Nothing']
    showsPrec _ (Just x)=   showString (['Just '] ++ show x)

instance Read (Maybe a) | Read a
where

//  readsPrec _ = \cs -> [(MaybeFromGenMaybe mbg, res) \\ (mbg, res) <- reads cs ]
    readsPrec _ = \cs ->   [(o1,res1) \\ (o1,res1) <- (lexConstr "Nothing" Nothing) cs]
                        ++ [(Just x,res2) \\ (jt,res1) <- lex cs, (x, res2) <- reads res1 | jt == ['Just'] ]
// ----- [a] -----------------

instance Show [a] | Show a
where
    showsPrec _ xs = showList xs

instance Read [a] | Read a
where
    readsPrec _ = readList

// ----- (a,b) -----------------

instance Show (a,b) | Show a & Show b
where
    showsPrec _ (x,y) = showString( ['('] ++ show x ++ [','] ++ show y ++ [')'])

instance Read  (a,b) | Read a & Read b
where
    readsPrec _ = \cs ->    [ ((x,y), tail_) \\     ('(',cs1) <- readLitChar cs
                                                ,   (x,cs2) <- reads cs1
                                                ,   (',',cs3) <- readLitChar cs2
                                                ,   (y,cs4) <- reads cs3
                                                ,   (')',tail_) <- readLitChar cs4
                            ]
```

Char.icl

See definiton module Char.dcl on page 45.

```
implementation module Char

from StdChar        import  isDigit,isUpper
from StdFunc        import  o

from PreludeList    import  length,++,!!,elem,span
from PreludeText    import  class Show(..), :: ShowS(..), :: ReadS(..), showChar, showString, shows, instance Show Int
from Numeric        import  readDec,readOct,readHex,lexDigits
from Preludeimport  ||, from_to

import charInstances
import   intInstances

// Character code functions

ord :: Char -> Int
ord c = fromChar c

chr :: Int  -> Char
chr c = toChar c

// Text functions

readLitChar :: ReadS Char
readLitChar = readLitChar_

readLitChar_ ['\\':s]    =  readEsc s
    where
        readEsc ['a':s]  = [('\a',s)]
        readEsc ['b':s]  = [('\b',s)]
        readEsc ['f':s]  = [('\f',s)]
        readEsc ['n':s]  = [('\n',s)]
        readEsc ['r':s]  = [('\r',s)]
        readEsc ['t':s]  = [('\t',s)]
        readEsc ['v':s]  = [('\v',s)]
        readEsc ['\\':s] = [('\\',s)]
        readEsc ['"':s]  = [('"' ,s)]
        readEsc ['\'':s] = [('\'',s)]
//      readEsc ['^':c:s] | c >= '@' && c <= '_' = [(chr (ord c - ord '@'), s)] // Unicode needed
        readEsc s=:[d:_] | isDigit d    = [(chr n, t) \\ (n,t) <- readDec s]
        readEsc ['o':s]                 = [(chr n, t) \\ (n,t) <- readOct s]
        readEsc ['x':s]                 = [(chr n, t) \\ (n,t) <- readHex s]
        readEsc s=:[c:_] | isUpper c =
            let
                table           = [(127, ['DEL']) : assocs_ asciiTab]
                assocs_ asct    = [(i,a) \\ i <- from_to 0 (length asct - 1), a <- asct]
            in
                case [(toChar c,s`) \\ (c, mne) <- table, ([],s`) <- [match mne s]] of
                    [pr:_]  -> [pr]
                    []      -> []
        readEsc _           = []

        match :: [a] [a] -> ([a],[a])   | Eq a
        match [x:xs] [y:ys] | x == y  =  match xs ys
        match xs     ys               =  (xs,ys)

readLitChar_ [c:s]       =  [(c,s)]

showLitChar :: Char -> ShowS
showLitChar c | c > del    =  showChar '\\' o protectEsc isDigit (shows (ord c))
showLitChar d | d == del   =  showString ['\\DEL']
showLitChar '\\'           =  showString ['\\','\\']
showLitChar c | c >= ' '   =  showChar c
showLitChar '\a'           =  showString ['\\a']
showLitChar '\b'           =  showString ['\\b']
showLitChar '\f'           =  showString ['\\f']
showLitChar '\n'           =  showString ['\\n']
showLitChar '\r'           =  showString ['\\r']
showLitChar '\t'           =  showString ['\\t']
showLitChar '\v'           =  showString ['\\v']
showLitChar s | s == so    =  protectEsc ((==) 'H') (showString ['\\SO'])
showLitChar c              =  showString ['\\' : asciiTab !! (fromChar c) ]


protectEsc p f             = f o cont
    where
        cont s=:[c:_]   | p c   = ['\\&'] ++ s
        cont s                  = s

del =   toChar 127  // '\DEL'
so  =   toChar 14   // '\SO'

asciiTab = [['NUL'], ['SOH'], ['STX'], ['ETX'], ['EOT'], ['ENQ'], ['ACK'], ['BEL'],
            ['BS'],  ['HT'],  ['LF'],  ['VT'],  ['FF'],  ['CR'],  ['SO'],  ['SI'],
            ['DLE'], ['DC1'], ['DC2'], ['DC3'], ['DC4'], ['NAK'], ['SYN'], ['ETB'],
            ['CAN'], ['EM'],  ['SUB'], ['ESC'], ['FS'],  ['GS'],  ['RS'],  ['US'],
            ['SP']]

lexLitChar  :: ReadS [Char]
lexLitChar  = lexLitChar`
    where
        lexLitChar` ['\\':s] =  [(['\\':esc], t) \\ (esc,t) <- lexEsc s]
        lexLitChar` [c:s]    =  [([c],s)]
        lexLitChar` []       =  []

        lexEsc [c:s]     | elem c ['abfnrtv\\\"\'']  = [([c],s)]
        lexEsc s=:[d:_]  | isDigit d                = lexDigits s
        lexEsc s=:[c:_]  | isUpper c                = [span isCharName s]
        lexEsc                                      = []
 //  lexEsc ['^':[c:s]] | c >= '@' && c <= '_'    = [(['^',c],s)] Unicode: Very crude approximation to \XYZ.
        isCharName c = isUpper c || isDigit c
```

Numeric.icl

See definiton module Numeric.dcl on page 45.

```
implementation module Numeric

from StdFunc        import  o
from StdChar        import  digitToInt

from PreludeText    import  :: ReadS(..), :: ShowS(..), readParen, showParen, showChar, lex, read, class Read(..)
from PreludeList    import  span,foldl1,length,elem,++,map
from Char           import  isDigit,isOctDigit,isHexDigit
from Ratio          import  %, ::Ratio

from Prelude import negate, >, error, fromIntegral

import  intInstances
import charInstances

showSigned  :: (a -> ShowS) Int a -> ShowS  | Hs__Real a
showSigned showPos p x
    | x < zero  = showParen (p > 6) (showChar '-' o showPos ( negate x))
                = showPos x

showInt     :: a -> ShowS   |   Integral a
showInt n
    | n < zero  = error ['showInt: can\'t show negative numbers']
                = \r -> let
                            (n`,d)  = quotRem n ten
                            r`      = [toEnum (fromEnum '0' + fromIntegral d) : r]
                            ten     = toEnum 10
                        in
                            if (n` == zero) r` (showInt n` r`)

readSigned :: (ReadS a) -> ReadS a  |   Hs__Real a
readSigned readPos = readParen False read`
    where
        read` r     = (read`` r) ++ [(negate x,t)   \\  (['-'],s) <- lex r, (x,t)   <- read`` s ]
        read`` r    = [(n,s) \\ (str,s) <- lex r,   (n,[])  <- readPos str ]

readInt :: a (Char -> Bool) (Char -> Int) -> ReadS a | Integral a
readInt radix isDig digToInt = \s ->
    [(foldl1 (\n d -> n * radix + d) (map (fromIntegral o digToInt) ds), r)
        \\  (ds,r) <- nonnull isDig s ]

readDec :: ReadS a  | Integral a
readDec = readInt (toEnum 10) isDigit digitToInt

readOct :: ReadS a | Integral a
readOct = readInt (toEnum 8) isOctDigit digitToInt

readHex :: ReadS a | Integral a
readHex = readInt (toEnum 16) isHexDigit digitToInt

lexDigits   :: ReadS [Char]
lexDigits   =  nonnull isDigit

nonnull     :: (Char -> Bool) -> ReadS [Char]
nonnull p   = \s ->   [(cs,t) \\ (cs=:[_:_],t) <- [span p s]]
```

PreludeLayer.icl

See definiton module PreludeLayer.dcl on page 45.

```
implementation module PreludeLayer

import StdEnv

from2 :: !Int -> [Int]
from2 n = [n : from2 (n+1)]

from_to :: !Int !Int -> [Int]
from_to n e
    | n <= e    = [n : from_to (n+1) e]
                = []

from_then :: !Int !Int -> [Int]
from_then n1 n2 = [n1 : from_by n2 (n2-n1)]
    where
        from_by :: Int Int -> [Int]
        from_by n s = [n : from_by (n+s) s]

from_then_to :: !Int !Int !Int -> [Int]
from_then_to n1 n2 e
    | n1 <= n2
        = from_by_to n1 (n2-n1) e
        = from_by_down_to n1 (n2-n1) e
    where
        from_by_to :: !Int !Int !Int -> [Int]
        from_by_to n s e
            | n<=e  = [n : from_by_to (n+s) s e]
                    = []
        from_by_down_to :: !Int !Int !Int -> [Int]
        from_by_down_to n s e
            | n>=e
                = [n : from_by_down_to (n+s) s e]
                = []

// ---------- Conflicting classes and instances used in Clean --------------
hs__abort :: ![Char] -> .a
hs__abort x = abort (toString x)

class (hs__rem) infix 7 a :: !a !a -> a
instance hs__rem Int where (hs__rem) x y = x rem y

class (hs__quot) infix 7    a :: !a !a -> a

instance hs__quot Int where (hs__quot) x y = x / y

// -----
// for PreludeIO

toStr :: [Char] -> String
toStr cs = toString cs

fromStr :: String -> [Char]
fromStr str = fromString str

// removes newline from the end of the string
removeNL :: String -> String
removeNL str = { ch \\ ch <-: str | not (ch == '\n') }
```

Preludes.icl

See definiton module Preludes.dcl on page 45.

```
implementation module Preludes
```

Ratio.icl

See definiton module Ratio.dcl on page 45.

```
from Prelude import class Integral(..), class Enum, class Hs__Real(..), class Ord(..), class Num(..), class Eq(..)
                  , class <, class one, class zero(..), class fromInt, class abs, class *(..), class -
                  , class +, class ==(..), error, gcd, class abs(..), quot, class Show

:: Ratio a = (:%) !a !a
:: Rational :== Ratio Int

(%) infixl 7 :: a a -> Ratio a | Integral a
(%) x y =   reduce (x * signum y) (abs y)
    where
        reduce _ z | z == zero  =  error ['Ratio (%): zero denominator']
        reduce x y              =  (:%) (x quot d) (y quot d)
            where   d = gcd x y
```

charInstances.icl

See definiton module charInstances.dcl on page 46.

```
implementation module charInstances
```

intInstances.icl

See definiton module intInstances.dcl on page 46.

```
implementation module intInstances
```

realInstances.icl

See definiton module realInstances.dcl on page 46.

```
implementation module realInstances
```

Preludes.icl

See definiton module Preludes.dcl on page 45.

```
implementation module Preludes
```

Appendix 4: Samples

numericEnum.icl

```
/*
* This module demonstrates the usage of PreludeList(take, concat) and numeric enumeration functions
*/

module numericEnum
import Preludes

from StdInt import instance fromInt Real

Start :: [Real]
Start = concat [take 5 (numericEnumFrom 4.5), take 5 (numericEnumFromThenTo 4.5 4.8 5.9)]
```

Pascal.icl

```
/*
 * This module prints the Pascal triangle
 */

module Pascal
import Preludes
import intInstances
from StdList import instance fromString [], instance toString []
from StdChar import instance fromChar Char, instance toChar Char

pascal :: [[Int]]
pascal  = iterate (\row -> zipWith (+) ([0]++row) (row++[0])) [1]

layn :: [[Char]] -> [Char]
layn  cs = lay 1 cs
    where lay _ []     = []
          lay n [x:xs] = rjustify 4 (show n) ++ [') '] ++ x ++ ['\n'] ++ lay (n+1) xs

rjustify :: Int  [Char] -> [Char]
rjustify n s = space (n - length s) ++ s
cjustify n s = space halfm ++ s ++ space (m - halfm)
    where m     = n - length s
          halfm = m div 2

space :: Int -> [Char]
space n = copy n ' '

copy :: Int  a -> [a]
copy n x  = take n (repeat x)

showPascal = (layn o map show o take 14) pascal

Start :: *World -> *World
Start world
    #
        (con,world) = stdio world
        con         = con <<< toString showPascal
        (_, con)    = freadline con
        (_,world)   = fclose con world
    = world
```

prgen.icl

```
/*
 * These instances are generated by the help of generics
 */

module prgen
import Preludes

Start = [   (1,2,4) /= (1,2,5)
        ,   (1,2,5) < (2,4,8)
        ,   (2,6,4) < (2,5,3)
        ]
```

prio.icl

```
/*
 * Module to demostrate IO monads.
 *
 * Enter lines, Press Ctrl-Z, and the lines will be mapped to upper
 *
 */

module prio
import Preludes

from PreludeLayer import toStr
from StdOverloaded import class +++(..)
from StdString import instance +++ {#Char}
from StdChar import toUpper

Start :: *World -> *World
Start world
    #
        (c,world)   = doIO (interact (map toUpper)) world
        (con,world) = stdio world
        (_, con)    = freadline con
        (_,world)   = fclose con world
    = world
```

prtext.icl

```
module prtext
import PreludeText, StdFile
from StdInt import instance fromInt Real
from StdString import class fromString(..), class toString(..)
from StdOverloaded import class %(..)
import StdArray,StdString
from StdString import instance % {#Char}
from Prelude import snd
from StdList import instance fromString [], instance toString []
from StdChar import instance fromChar Char, instance toChar Char
from StdInt  import instance - Int

//for testing instances of Read, Show

readInput :: !*File -> ([Char],!*File)
readInput con
    #   con         =   con <<< "Enter input : "
        (inpu,con)  =   freadline con
        input       =   inpu % (0,(size inpu) - 2 ) // erase the "\n" from the end of inpu
    = (fromString input,con)


instance <<< [(Real,[Char])]
    where
        (<<<) :: !*File [(Real,[Char])] -> !*File
        (<<<) f []      = f
        (<<<) f [x:xs]  = f <<< x <<< xs


instance <<< (Real,[Char])
    where
        (<<<) :: !*File (Real,[Char]) -> !*File
        (<<<) f (n,remain)  = f <<< "Result: " <<< toString (show n) <<< "\nRemain of input: "
                                    <<< (toString remain) <<< "\n"


Start :: *World -> *World
Start world
    #   (con,world) = stdio world
        con = con <<< "Enter an positive Real to be parsed. To quit: empty input\n"
        con = loop con
        (inpu,con)  = freadline con // to see the results
    =   snd (fclose con world)

    where
        loop con
            #   (inpu,con) = readInput con
            = case (inpu) of
                    []  ->  con
                    inp ->  loop (con <<< readsReal inp <<< "\n")

        readsReal :: ReadS Real
        readsReal = reads
```

prtext2.icl

```
/*
 * This module demonstrate the usage of PreludeText and Read (Maybe Int)
 */

module prtext2

import Preludes
import intInstances, charInstances

from StdFile import class <<<(..), instance <<< {#Char}, freadline, class FileSystem(..), instance FileSystem World
import StdArray
from StdOverloaded import class %(..)
from StdString import instance % {#Char}, class fromString(..), class toString(..)
from StdList import instance fromString [], instance toString []


readInput :: !*File -> ([Char],!*File)
readInput con
    #   con         =   con <<< "Enter input : "
        (inpu,con)  =   freadline con
        input       =   inpu % (0,(size inpu) - 2 ) // erase the "\n" from the end of inpu
    = (fromString input,con)


instance <<< [(Maybe Int,[Char])]
    where
        (<<<) f []      = f
        (<<<) f [x:xs]  = f <<< x <<< xs


instance <<< (Maybe Int,[Char])
    where
        (<<<) f (n,remain)  = f <<< "Result: " <<< toString (show n) <<< "\nRemain of input: " <<<
                                    (toString remain) <<< "\n"


Start :: *World -> *World
Start world
    #   (con,world) = stdio world
        con = con <<< "Enter an Maybe Int to be parsed ('Just 5' or 'Nothing'). To quit: empty input\n"
        con = loop con
        (inpu,con)  = freadline con // to see the results
    =   snd (fclose con world)

    where
        loop con
            #   (inpu,con) = readInput con
            = case (inpu) of
                    []  ->  con
                    inp ->  loop (con <<< readsMaybeReal inp <<< "\n")


        readsMaybeReal :: ReadS (Maybe Int)
```

```
readsMaybeReal = reads
```

Contents

Prelude for Clean 1

User's Guide 2

Introduction 2

Using Prelude for Clean without the Haskell front-end 2

How to install and use the Prelude for Clean library 2

Differences 3

Samples 3

What's on CD? 3

Developer's Guide 4

Main approach 4

General remarks 4

Prelude.icl 5
Standard types, classes, instances and related functions 5
Numeric functions 8
Monadic classes 8
Additional functions (like in StdFunc) 8
Basic types and instances 9
Instances for standard numeric types 11
Misc. functions 11

PreludeList.icl 12
..... 12

PreludeText.icl 14
Classes 14
Functions 14

PreludeIO.icl 16
..... 16

PreludeGeneric.icl 16

References 17
..... 17

Appendix 1: Haskell Prelude 18

Prelude.hs 18
PreludeList.hs 24
PreludeText.hs 27
PreludeIO.hs 29

Char.hs 30
Divmod.hs 31
Numeric.hs 31
Ratio.hs 34

Appendix 2: Prelude for Clean (Definition modules) 35

Prelude.dcl 35
PreludeList.dcl 40
PreludeListLayer.dcl 41
PreludeText.dcl 42
PreludeIO.dcl 43
PreludeGeneric.dcl 44
Char.dcl 45
Numeric.dcl 45
PreludeLayer.dcl 45
Preludes.dcl 45
Ratio.dcl 45
charInstances.dcl 46
intInstances.dcl 46
realInstances.dcl 46

Appendix 3: Prelude for Clean (Implementation modules) 47

Prelude.icl 47
PreludeList.icl 53
PreludeListLayer.icl 54
PreludeText.icl 58
PreludeIO.icl 60
PreludeGeneric.icl 63
Char.icl 66
Numeric.icl 67
PreludeLayer.icl 68
Preludes.icl 68
Ratio.icl 68
charInstances.icl 68
intInstances.icl 68
realInstances.icl 69
Preludes.icl 69

Appendix 4: Samples 70

numericEnum.icl 70
Pascal.icl 70
prgen.icl 70
prio.icl 70
prtext.icl 71
prtext2.icl 71

Contents **73**