Haskell 2010 Revised Library Report

This is the library report for the revised Haskell 2010 language report.

Contents

1 `Control.Monad` 11

1.1 Functor and monad classes 11

1.2 Functions 13

1.2.1 Naming conventions 13

1.2.2 Basic `Monad` functions 14

1.2.3 Generalisations of list functions 15

1.2.4 Conditional execution of monadic expressions 16

1.2.5 Monadic lifting operators 17

2 `Data.Array` 19

2.1 Immutable non-strict arrays 19

2.2 Array construction 20

2.3 Accessing arrays 21

2.4 Incremental array updates 22

2.5 Derived arrays 22

3 `Data.Bits` 23

4 `Data.Bool` 27

5 `Data.Char` 29

5.1 Characters and strings 30

5.2 Character classification 30

5.3 Subranges 32

5.4 Unicode general categories 33

5.5 Case conversion 35

5.6 Single digit characters 35

5.7 Numeric representations 35

5.8 String representations 36

6 `Data.Complex` 37

6.1 Rectangular form 37

6.2 Polar form 38

6.3 Conjugate 38

7 `Data.Either` 39

8 `Data.Enum` 41

9 `Data.Eq` 45

10 `Data.Function` 49

11 `Data.Int` 51

12 `Data.Ix` 55

12.1 The Ix class 55

13 `Data.List` **57**

13.1 Basic functions . . . 57

13.2 List transformations . . . 58

13.3 Reducing lists (folds) . . . 59

13.4 Special folds . . . 60

13.5 Building lists . . . 61

13.5.1 Scans . . . 61

13.5.2 Accumulating maps . . . 62

13.5.3 Infinite lists . . . 62

13.5.4 Unfolding . . . 63

13.6 Sublists . . . 63

13.6.1 Extracting sublists . . . 63

13.6.2 Predicates . . . 66

13.7 Searching lists . . . 67

13.7.1 Searching by equality . . . 67

13.7.2 Searching with a predicate . . . 67

13.8 Indexing lists . . . 67

13.9 Zipping and unzipping lists . . . 68

13.10 Special lists . . . 70

13.10.1 Functions on strings . . . 70

13.10.2 `Set` operations . . . 71

13.10.3 Ordered lists . . . 72

13.11 Generalized functions . . . 72

13.11.1 The `By` operations . . . 72

13.11.2 The "generic" operations . . . 74

14 Data.Maybe **75**

14.1 The Maybe type and operations 75

15 Data.Ord **79**

16 Data.Ratio **83**

17 Data.String **85**

18 Data.Tuple **87**

19 Data.Word **89**

20 Foreign **93**

21 Foreign.C **95**

22 Foreign.C.Error **97**

22.1 Haskell representation of errno values 98

22.1.1 Common errno symbols 98

22.1.2 Errno functions 102

22.1.3 Guards for IO operations that may fail 102

23 Foreign.C.String **107**

23.1 C strings 107

23.1.1 Using a locale-dependent encoding 108

23.1.2 Using 8-bit characters 109

23.2 C wide strings 111

24 `Foreign.C.Types` 113

24.1 Representations of C types 113

24.1.1 Integral types 113

24.1.2 Numeric types 120

24.1.3 Floating types 120

24.1.4 Other types 121

25 `Foreign.ForeignPtr` 123

25.1 Finalised data pointers 123

25.1.1 Basic operations 124

25.1.2 Low-level operations 125

25.1.3 Allocating managed memory 126

26 `Foreign.Marshal` 127

27 `Foreign.Marshal.Alloc` 129

27.1 Memory allocation 130

27.1.1 Local allocation 130

27.1.2 Dynamic allocation 130

28 `Foreign.Marshal.Array` 133

28.1 Marshalling arrays 133

28.1.1 Allocation 133

28.1.2 Marshalling 134

28.1.3 Combined allocation and marshalling 135

28.1.4 Copying 135

28.1.5 Finding the length 136

28.1.6 Indexing 136

29 `Foreign.Marshal.Error` 137

30 `Foreign.Marshal.Utils` 139

30.1 General marshalling utilities 139

30.1.1 Combined allocation and marshalling 139

30.1.2 Marshalling of Boolean values (non-zero corresponds to True) 140

30.1.3 Marshalling of Maybe values 140

30.1.4 Marshalling lists of storable objects 140

30.1.5 Haskellish interface to memcpy and memmove 140

31 `Foreign.Ptr` 143

31.1 Data pointers 143

31.2 Function pointers 144

31.3 Integral types with lossless conversion to and from pointers . . . 146

32 `Foreign.StablePtr` 149

32.1 Stable references to Haskell values 149

32.1.1 The C-Side interface 150

33 `Foreign.Storable` 151

34 `Numeric` 155

34.1 Showing 155

34.2 Reading 157

34.3 Miscellaneous 158

35 `Prelude` **159**

35.1 Standard types, classes and related functions 160

35.1.1 Basic data types 160

35.1.2 Basic type classes 164

35.1.3 Numbers 170

35.1.4 Monads and functors 178

35.1.5 Miscellaneous functions 180

35.2 List operations 181

35.3 Converting to and from String 188

35.3.1 Converting to String 188

35.3.2 Converting from String 190

35.4 Basic Input and output 192

35.4.1 Simple I/O operations 192

35.4.2 Exception handling in the I/O monad 194

36 `System.Environment` **197**

37 `System.Exit` **199**

38 `System.IO` **201**

38.1 The IO monad 201

38.2 Files and handles 202

38.3 Standard handles 203

38.4 Opening and closing files 203

38.4.1 Opening files 203

38.4.2 Closing files 204

38.4.3 Special cases 204

38.4.4 File locking 205
38.5 Operations on handles 205
38.5.1 Determining and changing the size of a file 205
38.5.2 Detecting the end of input 206
38.5.3 Buffering operations 206
38.5.4 Repositioning handles 208
38.5.5 Handle properties 209
38.5.6 Terminal operations 210
38.5.7 Showing handle state 210
38.6 Text input and output 210
38.6.1 Text input 210
38.6.2 Text output 212
38.6.3 Special cases for standard input and output 213

39 `System.IO.Error` 215

39.1 I/O errors 215
39.2 Classifying I/O errors 216
39.3 Attributes of I/O errors 217
39.4 Types of I/O error 217
39.5 Throwing and catching I/O errors 218

40 `Text.Read` 221

41 `Text.Show` 225

Chapter 1

Control.Monad

```
module Control.Monad (
    Functor(fmap), Monad((>>=), (>>), return, fail), MonadPlus(mzero, mplus),
    mapM, mapM_, forM, forM_, sequence, sequence_, (=<<), (>=>), (<=<), forever,
    void, join, msum, filterM, mapAndUnzipM, zipWithM, zipWithM_, foldM, foldM_,
    replicateM, replicateM_, guard, when, unless, liftM, liftM2, liftM3, liftM4,
    liftM5, ap
  ) where
```

The Control.Monad module provides the `Functor`, `Monad` and `MonadPlus` classes, together with some useful operations on monads.

1.1 Functor and monad classes

`class Functor f where`

> The Functor class is used for types that can be mapped over. Instances of Functor should satisfy the following laws:
>
> ```
> fmap id == id
> fmap (f . g) == fmap f . fmap g
> ```
>
> The instances of `Functor` for lists, `Maybe` and `IO` satisfy these laws.
>
> *Methods*

```
fmap ::  (a -> b) -> f a -> f b
```

```
instance Functor Maybe
instance Functor IO
instance Functor List
instance Ix i => Functor Array i
```

```
class Monad m where
```

The Monad class defines the basic operations over a monad, a concept from a branch of mathematics known as category theory. From the perspective of a Haskell programmer, however, it is best to think of a monad as an abstract datatype of actions. Haskell's do expressions provide a convenient syntax for writing monadic expressions.

Minimal complete definition: ¿¿= and return.

Instances of `Monad` should satisfy the following laws:

```
return a >>= k == k a
m >>= return == m
m >>= (x -> k x >>= h) == (m >>= k) >>= h
```

Instances of both `Monad` and `Functor` should additionally satisfy the law:

```
fmap f xs == xs >>= return . f
```

The instances of `Monad` for lists, `Maybe` and `IO` defined in the Prelude satisfy these laws.

Methods

```
(>>=) ::  m a -> (a -> m b) -> m b
```

Sequentially compose two actions, passing any value produced by the first as an argument to the second.

```
(>>) ::  m a -> m b -> m b
```

Sequentially compose two actions, discarding any value produced by the first, like sequencing operators (such as the semicolon) in imperative languages.

```
return ::  a -> m a
```

Inject a value into the monadic type.

```
fail ::  String -> m a
```

Fail with a message. This operation is not part of the mathematical definition of a monad, but is invoked on pattern-match failure in a do expression.

```
instance Monad Maybe
instance Monad IO
instance Monad List
```

```
class Monad m => MonadPlus m where
```

Monads that also support choice and failure.

Methods

```
mzero ::  m a
```

the identity of mplus. It should also satisfy the equations

```
mzero >>= f = mzero
v >> mzero = mzero
```

```
mplus ::  m a -> m a -> m a
```

an associative operation

```
instance MonadPlus Maybe
instance MonadPlus List
```

1.2 Functions

1.2.1 Naming conventions

The functions in this library use the following naming conventions:

- A postfix 'M' always stands for a function in the Kleisli category: The monad type constructor m is added to function results (modulo currying) and nowhere else. So, for example,

```
filter :: (a -> Bool) -> [a] -> [a]
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

- A postfix ''' changes the result type from (m a) to (m ()). Thus, for example:

```
sequence :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
```

- A prefix 'm' generalizes an existing function to a monadic form. Thus, for example:

```
sum :: Num a => [a] -> a
msum :: MonadPlus m => [m a] -> m a
```

1.2.2 Basic `Monad` functions

`mapM :: Monad m => (a -> m b) -> [a] -> m [b]`

`mapM f` is equivalent to `sequence . map f`.

`mapM_ :: Monad m => (a -> m b) -> [a] -> m ()`

`mapM_ f` is equivalent to `sequence_ . map f`.

`forM :: Monad m => [a] -> (a -> m b) -> m [b]`

`forM` is `mapM` with its arguments flipped

`forM_ :: Monad m => [a] -> (a -> m b) -> m ()`

`forM_` is `mapM_` with its arguments flipped

`sequence :: Monad m => [m a] -> m [a]`

Evaluate each action in the sequence from left to right, and collect the results.

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

Evaluate each action in the sequence from left to right, and ignore the results.

`(=<<) :: Monad m => (a -> m b) -> m a -> m b`

Same as `>>=`, but with the arguments interchanged.

`(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c`

Left-to-right Kleisli composition of monads.

`(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c`

Right-to-left Kleisli composition of monads. `(>=>)`, with the arguments flipped

`forever :: Monad m => m a -> m b`

`forever act` repeats the action infinitely

`void :: Functor f => f a -> f ()`

`void value` discards or ignores the result of evaluation, such as the return value of an `IO` action.

1.2.3 Generalisations of list functions

`join :: Monad m => m (m a) -> m a`

The `join` function is the conventional monad join operator. It is used to remove one level of monadic structure, projecting its bound argument into the outer level.

`msum :: MonadPlus m => [m a] -> m a`

This generalizes the list-based `concat` function.

`filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]`

This generalizes the list-based `filter` function.

`mapAndUnzipM :: Monad m => (a -> m (b, c)) -> [a] -> m ([b], [c])`

The `mapAndUnzipM` function maps its first argument over a list, returning the result as a pair of lists. This function is mainly used with complicated data structures or a state-transforming monad.

```
zipWithM :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
```

The `zipWithM` function generalizes `zipWith` to arbitrary monads.

```
zipWithM_ :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m ()
```

`zipWithM_` is the extension of `zipWithM` which ignores the final result

```
foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
```

The `foldM` function is analogous to `foldl`, except that its result is encapsulated in a monad. Note that `foldM` works from left-to-right over the list arguments. This could be an issue where `(>>)` and the "folded function" are not commutative.

```
  foldM f a1 [x1, x2, ..., xm]
==
  do
    a2 <- f a1 x1
    a3 <- f a2 x2
    ...
    f am xm
```

If right-to-left evaluation is required, the input list should be reversed.

```
foldM_ :: Monad m => (a -> b -> m a) -> a -> [b] -> m ()
```

Like `foldM`, but discards the result.

```
replicateM :: Monad m => Int -> m a -> m [a]
```

`replicateM n act` performs the action `n` times, gathering the results.

```
replicateM_ :: Monad m => Int -> m a -> m ()
```

Like `replicateM`, but discards the result.

1.2.4 Conditional execution of monadic expressions

```
guard :: MonadPlus m => Bool -> m ()
```

`guard b` is `return ()` if `b` is `True`, and `mzero` if `b` is `False`

```
when :: Monad m => Bool -> m () -> m ()
```

Conditional execution of monadic expressions. For example,

```
when debug (putStr "Debugging\n")
```

will output the string `Debugging\n` if the Boolean value `debug` is `True`, and otherwise do nothing.

```
unless ::  Monad m => Bool -> m () -> m ()
```

The reverse of `when`.

1.2.5 Monadic lifting operators

```
liftM ::  Monad m => (a1 -> r) -> m a1 -> m r
```

Promote a function to a monad.

```
liftM2 ::  Monad m => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
```

Promote a function to a monad, scanning the monadic arguments from left to right. For example,

```
liftM2 (+) [0,1] [0,2] = [0,2,1,3]
liftM2 (+) (Just 1) Nothing = Nothing
```

```
liftM3 ::  Monad m => (a1 -> a2 -> a3 -> r) -> m a1 -> m a2 -> m a3 -> m r
```

Promote a function to a monad, scanning the monadic arguments from left to right (cf. `liftM2`).

```
liftM4 ::  Monad m => (a1 -> a2 -> a3 -> a4 -> r) -> m a1 -> m a2 -> m a3 -> m a4 -> m r
```

Promote a function to a monad, scanning the monadic arguments from left to right (cf. `liftM2`).

```
liftM5 ::  Monad m =>
          (a1 -> a2 -> a3 -> a4 -> a5 -> r) -> m a1 -> m a2 -> m a3 -> m a4 -> m a5 -> m r
```

Promote a function to a monad, scanning the monadic arguments from left to right (cf. `liftM2`).

```
ap ::  Monad m => m (a -> b) -> m a -> m b
```

In many situations, the `liftM` operations can be replaced by uses of `ap`, which promotes function application.

```
return f `ap` x1 `ap` ... `ap` xn
```

is equivalent to

```
liftMn f x1 x2 ... xn
```

Chapter 2

Data.Array

```
module Data.Array (
    module Data.Ix, Array, array, listArray, accumArray, (!), bounds, indices,
    elems, assocs, (//), accum, ixmap
  ) where
```

2.1 Immutable non-strict arrays

Haskell provides indexable *arrays*, which may be thought of as functions whose domains are isomorphic to contiguous subsets of the integers. Functions restricted in this way can be implemented efficiently; in particular, a programmer may reasonably expect rapid access to the components. To ensure the possibility of such an implementation, arrays are treated as data, not as general functions.

Since most array functions involve the class `Ix`, the contents of the module `Data.Ix` are re-exported from `Data.Array` for convenience:

```
module Data.Ix
```

```
data Array i e
```

> The type of immutable non-strict (boxed) arrays with indices in `i` and elements in `e`.

```
instance Ix i => Functor Array i
instance (Ix i, Eq e) => Eq Array i e
instance (Ix i, Ord e) => Ord Array i e
instance (Ix a, Read a, Read b) => Read Array a b
instance (Ix a, Show a, Show b) => Show Array a b
```

2.2 Array construction

`array`

`::` `Ix i`

`=>` `(i, i)` a pair of bounds, each of the index type of the array. These bounds are the lowest and highest indices in the array, in that order. For example, a one-origin vector of length '10' has bounds '(1,10)', and a one-origin '10' by '10' matrix has bounds '((1,1),(10,10))'.

`->` `[(i, e)]` a list of associations of the form (index, value). Typically, this list will be ex- pressed as a comprehension. An association '(i, x)' defines the value of the array at index `i` to be `x`.

`->` `Array i e`

Construct an array with the specified bounds and containing values for given indices within these bounds.

The array is undefined (i.e. bottom) if any index in the list is out of bounds. If any two associations in the list have the same index, the value at that index is undefined (i.e. bottom).

Because the indices must be checked for these errors, `array` is strict in the bounds argument and in the indices of the association list, but non-strict in the values. Thus, recurrences such as the following are possible:

```
a = array (1,100) ((1,1) : [(i, i * a!(i-1)) | i <- [2..100]])
```

Not every index within the bounds of the array need appear in the association list, but the values associated with indices that do not appear will be undefined (i.e. bottom).

If, in any dimension, the lower bound is greater than the upper bound, then the array is legal, but empty. Indexing an empty array always gives an array-bounds error, but `bounds` still yields the bounds with which the array was constructed.

`listArray :: Ix i => (i, i) -> [e] -> Array i e`

Construct an array from a pair of bounds and a list of values in index order.

`accumArray`

```
::  Ix i
=>  (e -> a -> e)   accumulating function
->  e               initial value
->  (i, i)          bounds of the array
->  [(i, a)]        association list
->  Array i e
```

The `accumArray` function deals with repeated indices in the association list using an accumulating function which combines the values of associations with the same index. For example, given a list of values of some index type, `hist` produces a histogram of the number of occurrences of each index within a specified range:

```
hist :: (Ix a, Num b) => (a,a) -> [a] -> Array a b
hist bnds is = accumArray (+) 0 bnds [(i, 1) | i<-is, inRange bnds i]
```

If the accumulating function is strict, then `accumArray` is strict in the values, as well as the indices, in the association list. Thus, unlike ordinary arrays built with `array`, accumulated arrays should not in general be recursive.

2.3 Accessing arrays

`(!) :: Ix i => Array i e -> i -> e`

The value at the given index in an array.

`bounds :: Array i e -> (i, i)`

The bounds with which an array was constructed.

`indices :: Ix i => Array i e -> [i]`

The list of indices of an array in ascending order.

`elems :: Array i e -> [e]`

The list of elements of an array in index order.

`assocs :: Ix i => Array i e -> [(i, e)]`

The list of associations of an array in index order.

2.4 Incremental array updates

```
(//) ::  Ix i => Array i e -> [(i, e)] -> Array i e
```

Constructs an array identical to the first argument except that it has been updated by the associations in the right argument. For example, if `m` is a 1-origin, `n` by `n` matrix, then

```
m//[((i,i), 0) | i <- [1..n]]
```

is the same matrix, except with the diagonal zeroed. Repeated indices in the association list are handled as for `array`: the resulting array is undefined (i.e. bottom),

```
accum ::  Ix i => (e -> a -> e) -> Array i e -> [(i, a)] -> Array i e
```

`accum f` takes an array and an association list and accumulates pairs from the list into the array with the accumulating function `f`. Thus `accumArray` can be defined using `accum`:

```
accumArray f z b = accum f (array b [(i, z) | i <- range b])
```

2.5 Derived arrays

```
ixmap ::  (Ix i, Ix j) => (i, i) -> (i -> j) -> Array j e -> Array i e
```

`ixmap` allows for transformations on array indices. It may be thought of as providing function composition on the right with the mapping that the original array embodies.

A similar transformation of array values may be achieved using `fmap` from the `Array` instance of the `Functor` class.

Chapter 3

Data.Bits

```
module Data.Bits (
    Bits((.&.),
         (.|.),
         xor,
         complement,
         shift,
         rotate,
         bit,
         setBit,
         clearBit,
         complementBit,
         testBit,
         bitSize,
         isSigned,
         shiftL,
         shiftR,
         rotateL,
         rotateR)
  ) where
```

This module defines bitwise operations for signed and unsigned integers.

`class Num a => Bits a where`

The Bits class defines bitwise operations over integral types.

- Bits are numbered from 0 with bit 0 being the least significant bit.

Minimal complete definition: .&., .—., xor, complement, (shift or (shiftL and shiftR)), (rotate or (rotateL and rotateR)), bitSize and isSigned.

Methods

`(.&.) :: a -> a -> a`

Bitwise "and"

`(.|.) :: a -> a -> a`

Bitwise "or"

`xor :: a -> a -> a`

Bitwise "xor"

`complement :: a -> a`

Reverse all the bits in the argument

`shift :: a -> Int -> a`

`shift x i` shifts `x` left by `i` bits if `i` is positive, or right by `-i` bits otherwise. Right shifts perform sign extension on signed number types; i.e. they fill the top bits with 1 if the `x` is negative and with 0 otherwise.

An instance can define either this unified `shift` or `shiftL` and `shiftR`, depending on which is more convenient for the type in question.

`rotate :: a -> Int -> a`

`rotate x i` rotates `x` left by `i` bits if `i` is positive, or right by `-i` bits otherwise.

For unbounded types like `Integer`, `rotate` is equivalent to `shift`.

An instance can define either this unified `rotate` or `rotateL` and `rotateR`, depending on which is more convenient for the type in question.

`bit :: Int -> a`

`bit i` is a value with the `i`th bit set and all other bits clear

`setBit :: a -> Int -> a`

`x ‘setBit‘ i` is the same as `x .|. bit i`

`clearBit :: a -> Int -> a`

`x ‘clearBit‘ i` is the same as `x .&. complement (bit i)`

`complementBit :: a -> Int -> a`

`x ‘complementBit‘ i` is the same as `x ‘xor‘ bit i`

`testBit :: a -> Int -> Bool`

Return `True` if the nth bit of the argument is 1

`bitSize :: a -> Int`

Return the number of bits in the type of the argument. The actual value of the argument is ignored. The function `bitSize` is undefined for types that do not have a fixed bitsize, like `Integer`.

`isSigned :: a -> Bool`

Return `True` if the argument is a signed type. The actual value of the argument is ignored

`shiftL :: a -> Int -> a`

Shift the argument left by the specified number of bits (which must be non-negative).

An instance can define either this and `shiftR` or the unified `shift`, depending on which is more convenient for the type in question.

`shiftR :: a -> Int -> a`

Shift the first argument right by the specified number of bits (which must be non-negative). Right shifts perform sign extension on signed number types; i.e. they fill the top bits with 1 if the `x` is negative and with 0 otherwise.

An instance can define either this and `shiftL` or the unified `shift`, depending on which is more convenient for the type in question.

`rotateL :: a -> Int -> a`

Rotate the argument left by the specified number of bits (which must be non-negative).

An instance can define either this and `rotateR` or the unified `rotate`, depending on which is more convenient for the type in question.

`rotateR :: a -> Int -> a`

Rotate the argument right by the specified number of bits (which must be non-negative).

An instance can define either this and `rotateL` or the unified `rotate`, depending on which is more convenient for the type in question.

```
instance Bits Int
instance Bits Int16
instance Bits Int32
instance Bits Int64
instance Bits Int8
instance Bits Word
instance Bits Word16
instance Bits Word32
instance Bits Word64
instance Bits Word8
instance Bits CChar
instance Bits CInt
instance Bits CIntMax
instance Bits CIntPtr
instance Bits CLLong
instance Bits CLong
instance Bits CPtrdiff
instance Bits CSChar
instance Bits CShort
instance Bits CSigAtomic
instance Bits CSize
instance Bits CUChar
instance Bits CUInt
instance Bits CUIntMax
instance Bits CUIntPtr
instance Bits CULLong
instance Bits CULong
instance Bits CUShort
instance Bits CWchar
instance Bits IntPtr
instance Bits WordPtr
instance Bits Integer
```

Chapter 4

Data.Bool

```
module Data.Bool (
    Bool(False, True), (&&), (||), not, otherwise
  ) where
```

```
data Bool
```

Constructors

```
=  False
|  True
```

```
instance Eq Bool
instance Ix Bool
instance Ord Bool
instance Storable Bool
```

```
(&&) ::  Bool -> Bool -> Bool
```

Boolean "and", lazy in the second argument

Specification:

```
True && x = x
False && _ = False
```

```
(||) ::  Bool -> Bool -> Bool
```

Boolean "or", lazy in the second argument

Specification:

```
True || _ = True
False || x = x
```

`not :: Bool -> Bool`

Boolean "not"

Specification:

```
not True = False
not False = True
```

`otherwise :: Bool`

`otherwise` is defined as the value `True`. It helps to make guards more readable, e.g.

```
f x | x < 0     = ...
    | otherwise = ...
```

Specification:

```
otherwise = True
```

Chapter 5

Data.Char

```
module Data.Char (
    Char, String, isControl, isSpace, isLower, isUpper, isAlpha, isAlphaNum,
    isPrint, isDigit, isOctDigit, isHexDigit, isLetter, isMark, isNumber,
    isPunctuation, isSymbol, isSeparator, isAscii, isLatin1, isAsciiUpper,
    isAsciiLower,
    GeneralCategory(UppercaseLetter,
                    LowercaseLetter,
                    TitlecaseLetter,
                    ModifierLetter,
                    OtherLetter,
                    NonSpacingMark,
                    SpacingCombiningMark,
                    EnclosingMark,
                    DecimalNumber,
                    LetterNumber,
                    OtherNumber,
                    ConnectorPunctuation,
                    DashPunctuation,
                    OpenPunctuation,
                    ClosePunctuation,
                    InitialQuote,
                    FinalQuote,
                    OtherPunctuation,
                    MathSymbol,
                    CurrencySymbol,
                    ModifierSymbol,
                    OtherSymbol,
                    Space,
                    LineSeparator,
```

```
                    ParagraphSeparator,
                    Control,
                    Format,
                    Surrogate,
                    PrivateUse,
                    NotAssigned),
    generalCategory, toUpper, toLower, toTitle, digitToInt, intToDigit, ord,
    chr, showLitChar, lexLitChar, readLitChar
  ) where
```

5.1 Characters and strings

```
data Char
```

The character type `Char` is an enumeration whose values represent Unicode (or equivalently ISO/IEC 10646) characters (see `http://www.unicode.org/` for details). This set extends the ISO 8859-1 (Latin-1) character set (the first 256 charachers), which is itself an extension of the ASCII character set (the first 128 characters). A character literal in Haskell has type `Char`. To convert a `Char` to or from the corresponding `Int` value defined by Unicode, use `Prelude.toEnum` and `Prelude.fromEnum` from the `Prelude.Enum` class respectively (or equivalently `ord` and `chr`).

```
instance Bounded Char
instance Enum Char
instance Eq Char
instance Ix Char
instance Ord Char
instance Storable Char
instance Read Char
instance Show Char
```

```
type String = [Char]
```

A `String` is a list of characters. String constants in Haskell are values of type `String`.

5.2 Character classification

Unicode characters are divided into letters, numbers, marks, punctuation, symbols, separators (including spaces) and others (including control characters).

`isControl :: Char -> Bool`

Selects control characters, which are the non-printing characters of the Latin-1 subset of Unicode.

`isSpace :: Char -> Bool`

Returns `True` for any Unicode space character, and the control characters `\t`, `\n`, `\r`, `\f`, `\v`.

`isLower :: Char -> Bool`

Selects lower-case alphabetic Unicode characters (letters).

`isUpper :: Char -> Bool`

Selects upper-case or title-case alphabetic Unicode characters (letters). Title case is used by a small number of letter ligatures like the single-character form of Lj.

`isAlpha :: Char -> Bool`

Selects alphabetic Unicode characters (lower-case, upper-case and title-case letters, plus letters of caseless scripts and modifiers letters). This function is equivalent to `Data.Char.isLetter`.

`isAlphaNum :: Char -> Bool`

Selects alphabetic or numeric digit Unicode characters.

Note that numeric digits outside the ASCII range are selected by this function but not by isDigit. Such digits may be part of identifiers but are not used by the printer and reader to represent numbers.

`isPrint :: Char -> Bool`

Selects printable Unicode characters (letters, numbers, marks, punctuation, symbols and spaces)

`isDigit :: Char -> Bool`

Selects ASCII digits, i.e. `'0'..'9'`.

`isOctDigit :: Char -> Bool`

Selects ASCII octal digits, i.e. `'0'..'7'`.

`isHexDigit :: Char -> Bool`

Selects ASCII hexadecimal digits, i.e. `'0'..'9'`, `'a'..'f'`, `'A'..'F'`.

`isLetter :: Char -> Bool`

Selects alphabetic Unicode characters (lower-case, upper-case and title-case letters, plus letters of caseless scripts and modifiers letters). This function is equivalent to `Data.Char.isAlpha`.

`isMark :: Char -> Bool`

Selects Unicode mark characters, e.g. accents and the like, which combine with preceding letters.

`isNumber :: Char -> Bool`

Selects Unicode numeric characters, including digits from various scripts, Roman numerals, etc.

`isPunctuation :: Char -> Bool`

Selects Unicode punctuation characters, including various kinds of connectors, brackets and quotes.

`isSymbol :: Char -> Bool`

Selects Unicode symbol characters, including mathematical and currency symbols.

`isSeparator :: Char -> Bool`

Selects Unicode space and separator characters.

5.3 Subranges

`isAscii :: Char -> Bool`

Selects the first 128 characters of the Unicode character set, corresponding to the ASCII character set.

`isLatin1 :: Char -> Bool`

Selects the first 256 characters of the Unicode character set, corresponding to the ISO 8859-1 (Latin-1) character set.

`isAsciiUpper :: Char -> Bool`

Selects ASCII upper-case letters, i.e. characters satisfying both `isAscii` and `isUpper`.

`isAsciiLower :: Char -> Bool`

Selects ASCII lower-case letters, i.e. characters satisfying both `isAscii` and `isLower`.

5.4 Unicode general categories

`data GeneralCategory`

Unicode General Categories (column 2 of the UnicodeData table) in the order they are listed in the Unicode standard.

Constructors

```
=  UppercaseLetter        Lu: Letter, Uppercase
|  LowercaseLetter        Ll: Letter, Lowercase
|  TitlecaseLetter        Lt: Letter, Titlecase
|  ModifierLetter         Lm: Letter, Modifier
|  OtherLetter            Lo: Letter, Other
|  NonSpacingMark         Mn: Mark, Non-Spacing
|  SpacingCombiningMark   Mc: Mark, Spacing Combining
|  EnclosingMark          Me: Mark, Enclosing
|  DecimalNumber          Nd: Number, Decimal
|  LetterNumber           Nl: Number, Letter
|  OtherNumber            No: Number, Other
|  ConnectorPunctuation   Pc: Punctuation, Connector
|  DashPunctuation        Pd: Punctuation, Dash
|  OpenPunctuation        Ps: Punctuation, Open
|  ClosePunctuation       Pe: Punctuation, Close
|  InitialQuote           Pi: Punctuation, Initial quote
|  FinalQuote             Pf: Punctuation, Final quote
|  OtherPunctuation       Po: Punctuation, Other
|  MathSymbol             Sm: Symbol, Math
|  CurrencySymbol         Sc: Symbol, Currency
|  ModifierSymbol         Sk: Symbol, Modifier
|  OtherSymbol            So: Symbol, Other
|  Space                  Zs: Separator, Space
|  LineSeparator          Zl: Separator, Line
|  ParagraphSeparator     Zp: Separator, Paragraph
|  Control                Cc: Other, Control
|  Format                 Cf: Other, Format
|  Surrogate              Cs: Other, Surrogate
|  PrivateUse             Co: Other, Private Use
|  NotAssigned            Cn: Other, Not Assigned
```

```
instance Bounded GeneralCategory
instance Enum GeneralCategory
instance Eq GeneralCategory
instance Ix GeneralCategory
instance Ord GeneralCategory
instance Read GeneralCategory
instance Show GeneralCategory
```

```
generalCategory ::  Char -> GeneralCategory
```

The Unicode general category of the character.

5.5 Case conversion

`toUpper :: Char -> Char`

Convert a letter to the corresponding upper-case letter, if any. Any other character is returned un- changed.

`toLower :: Char -> Char`

Convert a letter to the corresponding lower-case letter, if any. Any other character is returned un- changed.

`toTitle :: Char -> Char`

Convert a letter to the corresponding title-case or upper-case letter, if any. (Title case differs from upper case only for a small number of ligature letters.) Any other character is returned unchanged.

5.6 Single digit characters

`digitToInt :: Char -> Int`

Convert a single digit `Char` to the corresponding `Int`. This function fails unless its argument satisfies `isHexDigit`, but recognises both upper and lower-case hexadecimal digits (i.e. `'0'..'9'`, `'a'..'f'`, `'A'..'F'`).

`intToDigit :: Int -> Char`

Convert an `Int` in the range `0..15` to the corresponding single digit `Char`. This function fails on other inputs, and generates lower-case hexadecimal digits

5.7 Numeric representations

`ord :: Char -> Int`

The `Prelude.fromEnum` method restricted to the type `Data.Char.Char`

`chr :: Int -> Char`

The `Prelude.toEnum` method restricted to the type `Data.Char.Char`.

5.8 String representations

`showLitChar :: Char -> ShowS`

Convert a character to a string using only printable characters, using Haskell source-language escape conventions. For example:

```
showLitChar 'n' s = "\n" ++ s
```

`lexLitChar :: ReadS String`

Read a string representation of a character, using Haskell source-language escape conventions. For example:

```
lexLitChar "\nHello" = [("\n", \haddocktt{Hello})]
```

`readLitChar :: ReadS Char`

Read a string representation of a character, using Haskell source-language escape conventions, and convert it to the character that it encodes. For example:

```
readLitChar "\nHello" = [('n', \haddocktt{Hello})]
```

Chapter 6

Data.Complex

```
module Data.Complex (
    Complex((:+)), realPart, imagPart, mkPolar, cis, polar, magnitude, phase,
    conjugate
  ) where
```

6.1 Rectangular form

```
data Complex a
```

Complex numbers are an algebraic type.

For a complex number `z`, `abs z` is a number with the magnitude of `z`, but oriented in the positive real direction, whereas `signum z` has the phase of `z`, but unit magnitude.

Constructors

`=` `!a :+ !a` forms a complex number from its real and imaginary rectangular components.

```
instance Eq a => Eq Complex a
instance RealFloat a => Floating Complex a
instance RealFloat a => Fractional Complex a
instance RealFloat a => Num Complex a
instance Read a => Read Complex a
instance Show a => Show Complex a
```

```
realPart :: Complex a -> a
```

Extracts the real part of a complex number.

```
imagPart :: Complex a -> a
```

Extracts the imaginary part of a complex number.

6.2 Polar form

```
mkPolar :: Floating a => a -> a -> Complex a
```

Form a complex number from polar components of magnitude and phase.

```
cis :: Floating a => a -> Complex a
```

cis t is a complex value with magnitude 1 and phase t (modulo 2 * pi).

```
polar :: RealFloat a => Complex a -> (a, a)
```

The function polar takes a complex number and returns a (magnitude, phase) pair in canonical form: the magnitude is nonnegative, and the phase in the range (-pi, pi]; if the magnitude is zero, then so is the phase.

```
magnitude :: RealFloat a => Complex a -> a
```

The nonnegative magnitude of a complex number.

```
phase :: RealFloat a => Complex a -> a
```

The phase of a complex number, in the range (-pi, pi]. If the magnitude is zero, then so is the phase.

6.3 Conjugate

```
conjugate :: Num a => Complex a -> Complex a
```

The conjugate of a complex number.

Chapter 7

Data.Either

```
module Data.Either (
    Either(Left, Right), either
  ) where
```

```
data Either a b
```

Constructors

```
=  Left a
|  Right b
```

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

Case analysis for the `Either` type. If the value is `Left a`, apply the first function to `a`; if it is `Right b`, apply the second function to `b`.

Specification:

```
either f g (Left x)  =  f x
either f g (Right y) =  g y
```

Chapter 8

Data.Enum

```
module Data.Enum (
    Enum(enumFrom,
         enumFromThen,
         enumFromThenTo,
         enumFromTo,
         fromEnum,
         pred,
         succ,
         toEnum),
    Bounded(maxBound, minBound)
  ) where
```

```
class Enum a where
```

Methods

```
succ ::  a -> a

pred ::  a -> a

toEnum ::  Int -> a

fromEnum ::  a -> Int

enumFrom ::  a -> [a]
```

```
    enumFromThen ::  a -> a -> [a]

    enumFromTo ::  a -> a -> [a]

    enumFromThenTo ::  a -> a -> a -> [a]


instance Enum GeneralCategory
instance Enum Int
instance Enum Int16
instance Enum Int32
instance Enum Int64
instance Enum Int8
instance Enum Word
instance Enum Word16
instance Enum Word32
instance Enum Word64
instance Enum Word8
instance Enum CChar
instance Enum CClock
instance Enum CDouble
instance Enum CFloat
instance Enum CInt
instance Enum CIntMax
instance Enum CIntPtr
instance Enum CLLong
instance Enum CLong
instance Enum CPtrdiff
instance Enum CSChar
instance Enum CShort
instance Enum CSigAtomic
instance Enum CSize
instance Enum CTime
instance Enum CUChar
instance Enum CUInt
instance Enum CUIntMax
instance Enum CUIntPtr
instance Enum CULLong
instance Enum CULong
instance Enum CUShort
instance Enum CWchar
instance Enum IntPtr
instance Enum WordPtr
instance Enum Char
instance Enum Double
instance Enum Float
instance Enum Integer
instance Enum IOMode
instance Enum SeekMode
instance Integral a => Enum Ratio a
```

```
class Bounded a where
```

Methods

```
minBound ::  a

maxBound ::  a
```

```
instance Bounded GeneralCategory
instance Bounded Int
instance Bounded Int16
instance Bounded Int32
instance Bounded Int64
instance Bounded Int8
instance Bounded Word
instance Bounded Word16
instance Bounded Word32
instance Bounded Word64
instance Bounded Word8
instance Bounded CChar
instance Bounded CInt
instance Bounded CIntMax
instance Bounded CIntPtr
instance Bounded CLLong
instance Bounded CLong
instance Bounded CPtrdiff
instance Bounded CSChar
instance Bounded CShort
instance Bounded CSigAtomic
instance Bounded CSize
instance Bounded CUChar
instance Bounded CUInt
instance Bounded CUIntMax
instance Bounded CUIntPtr
instance Bounded CULLong
instance Bounded CULong
instance Bounded CUShort
instance Bounded CWchar
instance Bounded IntPtr
instance Bounded WordPtr
instance Bounded Char
```

Chapter 9

Data.Eq

```
module Data.Eq (
    Eq((/=), (==))
  ) where
```

```
class Eq a where
```

Methods

```
(==) ::  a -> a -> Bool
```

```
(/=) ::  a -> a -> Bool
```

```
instance Eq Bool
instance Eq GeneralCategory
instance Eq Int
instance Eq Int16
instance Eq Int32
instance Eq Int64
instance Eq Int8
instance Eq Ordering
instance Eq Word
instance Eq Word16
instance Eq Word32
instance Eq Word64
instance Eq Word8
instance Eq Errno
instance Eq CChar
instance Eq CClock
instance Eq CDouble
instance Eq CFloat
instance Eq CInt
instance Eq CIntMax
instance Eq CIntPtr
instance Eq CLLong
instance Eq CLong
instance Eq CPtrdiff
instance Eq CSChar
instance Eq CShort
instance Eq CSigAtomic
instance Eq CSize
instance Eq CTime
instance Eq CUChar
instance Eq CUInt
instance Eq CUIntMax
instance Eq CUIntPtr
instance Eq CULLong
instance Eq CULong
instance Eq CUShort
instance Eq CWchar
instance Eq IntPtr
instance Eq WordPtr
instance Eq Char
instance Eq Double
instance Eq Float
instance Eq Integer
instance Eq ExitCode
instance Eq BufferMode
instance Eq Handle
instance Eq HandlePosn
instance Eq IOMode
instance Eq SeekMode
instance Eq IOErrorType
instance Eq ()
instance Eq a => Eq Complex a
instance Eq a => Eq Maybe a
instance Eq ForeignPtr a
instance Eq FunPtr a
instance Eq Ptr a
instance Eq StablePtr a
instance Integral a => Eq Ratio a
instance (Ix i, Eq e) => Eq Array i e
instance (Eq a, Eq b) => Eq (a, b)
instance (Eq a, Eq b, Eq c) => Eq (a, b, c)
```

Chapter 10

Data.Function

```
module Data.Function (
    id, const, (.), flip, ($)
  ) where
```

```
id ::  a -> a

const ::  a -> b -> a

(.)  ::  (b -> c) -> (a -> b) -> a -> c

flip ::  (a -> b -> c) -> b -> a -> c

($) ::  (a -> b) -> a -> b
```

Chapter 11

Data.Int

```
module Data.Int (
    Int, Int8, Int16, Int32, Int64
  ) where
```

This module provides signed integer types of unspecified width (`Int`) and fixed widths (`Int8`, `Int16`, `Int32` and `Int64`). All arithmetic is performed modulo 2^n, where n is the number of bits in the type.

For coercing between any two integer types, use `fromIntegral`. Coercing word types (see `Word`) to and from integer types preserves representation, not sign.

The rules that hold for `Enum` instances over a bounded type such as `Int` (see the section of the Haskell language report dealing with arithmetic sequences) also hold for the `Enum` instances over the various Int types defined here.

Right and left shifts by amounts greater than or equal to the width of the type result in either zero or -1, depending on the sign of the value being shifted. This is contrary to the behaviour in C, which is undefined; a common interpretation is to truncate the shift count to the width of the type, for example 1 ¡¡ 32 == 1 in some C implementations.

`data Int`

> A fixed-precision integer type with at least the range [-2^29 .. 2^29-1]. The exact range for a given implementation can be determined by using `minBound` and `maxBound` from the `Bounded` class.

```
instance Bits Int
instance Bounded Int
instance Enum Int
instance Eq Int
instance Ix Int
instance Ord Int
instance Storable Int
instance Integral Int
instance Num Int
instance Real Int
instance Read Int
instance Show Int
```

```
data Int8
```

8-bit signed integer type

```
instance Bits Int8
instance Bounded Int8
instance Enum Int8
instance Eq Int8
instance Ix Int8
instance Ord Int8
instance Storable Int8
instance Integral Int8
instance Num Int8
instance Real Int8
instance Read Int8
instance Show Int8
```

```
data Int16
```

16-bit signed integer type

```
instance Bits Int16
instance Bounded Int16
instance Enum Int16
instance Eq Int16
instance Ix Int16
instance Ord Int16
instance Storable Int16
instance Integral Int16
instance Num Int16
instance Real Int16
instance Read Int16
instance Show Int16
```

```
data Int32
```

32-bit signed integer type

```
instance Bits Int32
instance Bounded Int32
instance Enum Int32
instance Eq Int32
instance Ix Int32
instance Ord Int32
instance Storable Int32
instance Integral Int32
instance Num Int32
instance Real Int32
instance Read Int32
instance Show Int32
```

```
data Int64
```

64-bit signed integer type

```
instance Bits Int64
instance Bounded Int64
instance Enum Int64
instance Eq Int64
instance Ix Int64
instance Ord Int64
instance Storable Int64
instance Integral Int64
instance Num Int64
instance Real Int64
instance Read Int64
instance Show Int64
```

Chapter 12

Data.Ix

```
module Data.Ix (
    Ix(range, index, inRange, rangeSize)
  ) where
```

12.1 The Ix class

```
class Ord a => Ix a where
```

The `Ix` class is used to map a contiguous subrange of values in a type onto integers. It is used primarily for array indexing (see the array package).

The first argument `(l,u)` of each of these operations is a pair specifying the lower and upper bounds of a contiguous subrange of values.

An implementation is entitled to assume the following laws about these operations:

- `inRange (l,u) i == elem i (range (l,u))`
- `range (l,u) !! index (l,u) i == i, when inRange (l,u) i`
- `map (index (l,u)) (range (l,u))) == [0..rangeSize (l,u)-1]`
- `rangeSize (l,u) == length (range (l,u))`

Minimal complete instance: `range`, `index` and `inRange`.

Methods

`range :: (a, a) -> [a]`

The list of values in the subrange defined by a bounding pair.

`index :: (a, a) -> a -> Int`

The position of a subscript in the subrange.

`inRange :: (a, a) -> a -> Bool`

Returns `True` the given subscript lies in the range defined the bounding pair.

`rangeSize :: (a, a) -> Int`

The size of the subrange defined by a bounding pair.

```
instance Ix Bool
instance Ix GeneralCategory
instance Ix Int
instance Ix Int16
instance Ix Int32
instance Ix Int64
instance Ix Int8
instance Ix Ordering
instance Ix Word
instance Ix Word16
instance Ix Word32
instance Ix Word64
instance Ix Word8
instance Ix Char
instance Ix Integer
instance Ix IOMode
instance Ix SeekMode
instance Ix ()
instance (Ix a, Ix b) => Ix (a, b)
instance (Ix a1, Ix a2, Ix a3) => Ix (a1, a2, a3)
instance (Ix a1, Ix a2, Ix a3, Ix a4) => Ix (a1, a2, a3, a4)
instance (Ix a1, Ix a2, Ix a3, Ix a4, Ix a5) => Ix (a1, a2, a3, a4, a5)
```

Chapter 13

Data.List

```
module Data.List (
    (++), head, last, tail, init, null, length, map, reverse, intersperse,
    intercalate, transpose, subsequences, permutations, foldl, foldl', foldl1,
    foldl1', foldr, foldr1, concat, concatMap, and, or, any, all, sum, product,
    maximum, minimum, scanl, scanl1, scanr, scanr1, mapAccumL, mapAccumR,
    iterate, repeat, replicate, cycle, unfoldr, take, drop, splitAt, takeWhile,
    dropWhile, span, break, stripPrefix, group, inits, tails, isPrefixOf,
    isSuffixOf, isInfixOf, elem, notElem, lookup, find, filter, partition, (!!),
    elemIndex, elemIndices, findIndex, findIndices, zip, zip3, zip4, zip5, zip6,
    zip7, zipWith, zipWith3, zipWith4, zipWith5, zipWith6, zipWith7, unzip,
    unzip3, unzip4, unzip5, unzip6, unzip7, lines, words, unlines, unwords, nub,
    delete, (\\), union, intersect, sort, insert, nubBy, deleteBy,
    deleteFirstsBy, unionBy, intersectBy, groupBy, sortBy, insertBy, maximumBy,
    minimumBy, genericLength, genericTake, genericDrop, genericSplitAt,
    genericIndex, genericReplicate
  ) where
```

13.1 Basic functions

```
(++) ::  [a] -> [a] -> [a]
```

Append two lists, i.e.,

```
[x1, ..., xm] ++ [y1, ..., yn] == [x1, ..., xm, y1, ..., yn]
[x1, ..., xm] ++ [y1, ...] == [x1, ..., xm, y1, ...]
```

If the first list is not finite, the result is the first list.

`head :: [a] -> a`

Extract the first element of a list, which must be non-empty.

`last :: [a] -> a`

Extract the last element of a list, which must be finite and non-empty.

`tail :: [a] -> [a]`

Extract the elements after the head of a list, which must be non-empty.

`init :: [a] -> [a]`

Return all the elements of a list except the last one. The list must be non-empty.

`null :: [a] -> Bool`

Test whether a list is empty.

`length :: [a] -> Int`

O(n). `length` returns the length of a finite list as an `Int`. It is an instance of the more general `genericLength`, the result type of which may be any kind of number.

13.2 List transformations

`map :: (a -> b) -> [a] -> [b]`

`map f xs` is the list obtained by applying `f` to each element of `xs`, i.e.,

```
map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
map f [x1, x2, ...] == [f x1, f x2, ...]
```

`reverse :: [a] -> [a]`

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

`intersperse :: a -> [a] -> [a]`

The `intersperse` function takes an element and a list and 'intersperses' that element between the elements of the list. For example,

```
intersperse ',' "abcde" == "a,b,c,d,e"
```

`intercalate :: [a] -> [[a]] -> [a]`

`intercalate xs xss` is equivalent to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

`transpose :: [[a]] -> [[a]]`

The `transpose` function transposes the rows and columns of its argument. For example,

```
transpose [[1,2,3],[4,5,6]] == [[1,4],[2,5],[3,6]]
```

`subsequences :: [a] -> [[a]]`

The `subsequences` function returns the list of all subsequences of the argument.

```
subsequences "abc" == ["","a","b","ab","c","ac","bc","abc"]
```

`permutations :: [a] -> [[a]]`

The `permutations` function returns the list of all permutations of the argument.

```
permutations "abc" == ["abc","bac","cba","bca","cab","acb"]
```

13.3 Reducing lists (folds)

`foldl :: (a -> b -> a) -> a -> [b] -> a`

`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
```

The list must be finite.

`foldl' :: (a -> b -> a) -> a -> [b] -> a`

A strict version of `foldl`.

`foldl1 :: (a -> a -> a) -> [a] -> a`

`foldl1` is a variant of `foldl` that has no starting value argument, and thus must be applied to non- empty lists.

`foldl1' :: (a -> a -> a) -> [a] -> a`

A strict version of `foldl1`

`foldr :: (a -> b -> b) -> b -> [a] -> b`

`foldr`, applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left:

```
foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)
```

`foldr1 :: (a -> a -> a) -> [a] -> a`

`foldr1` is a variant of `foldr` that has no starting value argument, and thus must be applied to non- empty lists.

13.4 Special folds

`concat :: [[a]] -> [a]`

Concatenate a list of lists.

`concatMap :: (a -> [b]) -> [a] -> [b]`

Map a function over a list and concatenate the results.

`and :: [Bool] -> Bool`

`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 :: [Bool] -> Bool`

`or` returns the disjunction of a Boolean list. For the result to be `False`, the list must be finite; `True`, however, results from a `True` value at a finite index of a finite or infinite list.

`any :: (a -> Bool) -> [a] -> Bool`

Applied to a predicate and a list, `any` determines if any element of the list satisfies the predicate. For the result to be `False`, the list must be finite; `True`, however, results from a `True` value for the predicate applied to an element at a finite index of a finite or infinite list.

`all :: (a -> Bool) -> [a] -> Bool`

Applied to a predicate and a list, `all` determines if all elements of the list satisfy the predicate. For the result to be `True`, the list must be finite; `False`, however, results from a `False` value for the predicate applied to an element at a finite index of a finite or infinite list.

`sum :: Num a => [a] -> a`

The `sum` function computes the sum of a finite list of numbers.

`product :: Num a => [a] -> a`

The `product` function computes the product of a finite list of numbers

`maximum :: Ord a => [a] -> a`

`maximum` returns the maximum value from a list, which must be non-empty, finite, and of an ordered type. It is a special case of `maximumBy`, which allows the programmer to supply their own comparison function.

`minimum :: Ord a => [a] -> a`

`minimum` returns the minimum value from a list, which must be non-empty, finite, and of an ordered type. It is a special case of `minimumBy`, which allows the programmer to supply their own comparison function.

13.5 Building lists

13.5.1 Scans

`scanl :: (a -> b -> a) -> a -> [b] -> [a]`

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

`scanl1` is a variant of `scanl` that has no starting value argument:

```
scanl1 f [x1, x2, ...] == [x1, x1 `f` x2, ...]
```

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
```

`scanr` is the right-to-left dual of `scanl`. Note that

```
head (scanr f z xs) == foldr f z xs.
```

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

`scanr1` is a variant of `scanr` that has no starting value argument.

13.5.2 Accumulating maps

```
mapAccumL :: (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])
```

The `mapAccumL` function behaves like a combination of `map` and `foldl`; it applies a function to each element of a list, passing an accumulating parameter from left to right, and returning a final value of this accumulator together with the new list.

```
mapAccumR :: (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])
```

The `mapAccumR` function behaves like a combination of `map` and `foldr`; it applies a function to each element of a list, passing an accumulating parameter from right to left, and returning a final value of this accumulator together with the new list.

13.5.3 Infinite lists

```
iterate :: (a -> a) -> a -> [a]
```

`iterate f x` returns an infinite list of repeated applications of `f` to `x`:

```
iterate f x == [x, f x, f (f x), ...]
```

```
repeat :: a -> [a]
```

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

`replicate :: Int -> a -> [a]`

`replicate n x` is a list of length `n` with `x` the value of every element. It is an instance of the more general `Data.List.genericReplicate`, in which `n` may be of any integral type.

`cycle :: [a] -> [a]`

`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.

13.5.4 Unfolding

`unfoldr :: (b -> Maybe (a, b)) -> b -> [a]`

The `unfoldr` function is a 'dual' to `foldr`: while `foldr` reduces a list to a summary value, `unfoldr` builds a list from a seed value. The function takes the element and returns `Nothing` if it is done producing the list or returns `Just (a,b)`, in which case, `a` is a prepended to the list and `b` is used as the next element in a recursive call. For example,

```
iterate f == unfoldr (x -> Just (x, f x))
```

In some cases, `unfoldr` can undo a `foldr` operation:

```
unfoldr f' (foldr f z xs) == xs
```

if the following holds:

```
f' (f x y) = Just (x,y)
f' z = Nothing
```

A simple use of unfoldr:

```
unfoldr (b -> if b == 0 then Nothing else Just (b, b-1)) 10
  [10,9,8,7,6,5,4,3,2,1]
```

13.6 Sublists

13.6.1 Extracting sublists

`take :: Int -> [a] -> [a]`

`take n`, applied to a list `xs`, returns the prefix of `xs` of length `n`, or `xs` itself if `n > length xs`:

```
take 5 "Hello World!" == \haddocktt{Hello}
take 3 [1,2,3,4,5] == [1,2,3]
take 3 [1,2] == [1,2]
take 3 [] == []
take (-1) [1,2] == []
take 0 [1,2] == []
```

It is an instance of the more general `genericTake`, in which `n` may be of any integral type.

`drop :: Int -> [a] -> [a]`

`drop n xs` returns the suffix of `xs` after the first `n` elements, or `[]` if `n > length xs`:

```
drop 6 "Hello World!" == "World!"
drop 3 [1,2,3,4,5] == [4,5]
drop 3 [1,2] == []
drop 3 [] == []
drop (-1) [1,2] == [1,2]
drop 0 [1,2] == [1,2]
```

It is an instance of the more general `genericDrop`, in which `n` may be of any integral type.

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

`splitAt n xs` returns a tuple where first element is `xs` prefix of length `n` and second element is the remainder of the list:

```
splitAt 6 "Hello World!" == ("Hello ","World!")
splitAt 3 [1,2,3,4,5] == ([1,2,3],[4,5])
splitAt 1 [1,2,3] == ([1],[2,3])
splitAt 3 [1,2,3] == ([1,2,3],[])
splitAt 4 [1,2,3] == ([1,2,3],[])
splitAt 0 [1,2,3] == ([],[1,2,3])
splitAt (-1) [1,2,3] == ([],[1,2,3])
```

It is equivalent to `(take n xs, drop n xs)`. `splitAt` is an instance of the more general `genericSplitAt`, in which `n` may be of any integral type.

`takeWhile :: (a -> Bool) -> [a] -> [a]`

`takeWhile`, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy `p`:

```
takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
takeWhile (< 9) [1,2,3] == [1,2,3]
takeWhile (< 0) [1,2,3] == []
```

```
dropWhile ::  (a -> Bool) -> [a] -> [a]
```

`dropWhile p xs` returns the suffix remaining after `takeWhile p xs`:

```
dropWhile (< 3) [1,2,3,4,5,1,2,3] == [3,4,5,1,2,3]
dropWhile (< 9) [1,2,3] == []
dropWhile (< 0) [1,2,3] == [1,2,3]
```

```
span ::  (a -> Bool) -> [a] -> ([a], [a])
```

`span`, applied to a predicate `p` and a list `xs`, returns a tuple where first element is longest prefix (possibly empty) of `xs` of elements that satisfy `p` and second element is the remainder of the list:

```
span (< 3) [1,2,3,4,1,2,3,4] == ([1,2],[3,4,1,2,3,4])
span (< 9) [1,2,3] == ([1,2,3],[])
span (< 0) [1,2,3] == ([],[1,2,3])
```

`span p xs` is equivalent to `(takeWhile p xs, dropWhile p xs)`

```
break ::  (a -> Bool) -> [a] -> ([a], [a])
```

`break`, applied to a predicate `p` and a list `xs`, returns a tuple where first element is longest prefix (possibly empty) of `xs` of elements that do not satisfy `p` and second element is the remainder of the list:

```
break (> 3) [1,2,3,4,1,2,3,4] == ([1,2,3],[4,1,2,3,4])
break (< 9) [1,2,3] == ([],[1,2,3])
break (> 9) [1,2,3] == ([1,2,3],[])
```

`break p` is equivalent to `span (not .  p)`.

```
stripPrefix ::  Eq a => [a] -> [a] -> Maybe [a]
```

The `stripPrefix` function drops the given prefix from a list. It returns `Nothing` if the list did not start with the prefix given, or `Just` the list after the prefix, if it does.

```
stripPrefix "foo" "foobar" == Just "bar"
stripPrefix "foo" "foo" == Just ""
stripPrefix "foo" "barfoo" == Nothing
stripPrefix "foo" "barfoobaz" == Nothing
```

```
group ::  Eq a => [a] -> [[a]]
```

The `group` function takes a list and returns a list of lists such that the concatenation of the result is equal to the argument. Moreover, each sublist in the result contains only equal elements. For example,

```
group \haddocktt{Mississippi} = [\haddocktt{M},"i","ss","i","ss","i","pp","i"]
```

It is a special case of `groupBy`, which allows the programmer to supply their own equality test.

```
inits ::  [a] -> [[a]]
```

The `inits` function returns all initial segments of the argument, shortest first. For example,

```
inits "abc" == ["","a","ab","abc"]
```

```
tails ::  [a] -> [[a]]
```

The tails function returns all final segments of the argument, longest first. For example,

```
tails "abc" == ["abc", "bc", "c",""]
```

13.6.2 Predicates

```
isPrefixOf ::  Eq a => [a] -> [a] -> Bool
```

The `isPrefixOf` function takes two lists and returns `True` iff the first list is a prefix of the second.

```
isSuffixOf ::  Eq a => [a] -> [a] -> Bool
```

The `isSuffixOf` function takes two lists and returns `True` iff the first list is a suffix of the second. Both lists must be finite.

```
isInfixOf ::  Eq a => [a] -> [a] -> Bool
```

The `isInfixOf` function takes two lists and returns `True` iff the first list is contained, wholly and intact, anywhere within the second.

Example:

```
isInfixOf \haddocktt{Haskell} "I really like Haskell." == True
isInfixOf \haddocktt{Ial} "I really like Haskell." == False
```

13.7 Searching lists

13.7.1 Searching by equality

```
elem :: Eq a => a -> [a] -> Bool
```

`elem` is the list membership predicate, usually written in infix form, e.g., `x `elem` xs`. For the result to be `False`, the list must be finite; `True`, however, results from an element equal to `x` found at a finite index of a finite or infinite list.

```
notElem :: Eq a => a -> [a] -> Bool
```

`notElem` is the negation of `elem`.

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

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

13.7.2 Searching with a predicate

```
find :: (a -> Bool) -> [a] -> Maybe a
```

The `find` function takes a predicate and a list and returns the first element in the list matching the predicate, or `Nothing` if there is no such element.

```
filter :: (a -> Bool) -> [a] -> [a]
```

`filter`, applied to a predicate and a list, returns the list of those elements that satisfy the predicate; i.e.,

```
filter p xs = [ x | x <- xs, p x]
```

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
```

The `partition` function takes a predicate a list and returns the pair of lists of elements which do and do not satisfy the predicate, respectively; i.e.,

```
partition p xs == (filter p xs, filter (not . p) xs)
```

13.8 Indexing lists

These functions treat a list `xs` as a indexed collection, with indices ranging from `0` to `length xs - 1`.

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

List index (subscript) operator, starting from 0. It is an instance of the more general `genericIndex`, which takes an index of any integral type.

`elemIndex :: Eq a => a -> [a] -> Maybe Int`

The `elemIndex` function returns the index of the first element in the given list which is equal (by `==`) to the query element, or `Nothing` if there is no such element.

`elemIndices :: Eq a => a -> [a] -> [Int]`

The `elemIndices` function extends `elemIndex`, by returning the indices of all elements equal to the query element, in ascending order.

`findIndex :: (a -> Bool) -> [a] -> Maybe Int`

The `findIndex` function takes a predicate and a list and returns the index of the first element in the list satisfying the predicate, or `Nothing` if there is no such element.

`findIndices :: (a -> Bool) -> [a] -> [Int]`

The `findIndices` function extends `findIndex`, by returning the indices of all elements satisfying the predicate, in ascending order.

13.9 Zipping and unzipping lists

`zip :: [a] -> [b] -> [(a, b)]`

`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 :: [a] -> [b] -> [c] -> [(a, b, c)]`

`zip3` takes three lists and returns a list of triples, analogous to `zip`.

`zip4 :: [a] -> [b] -> [c] -> [d] -> [(a, b, c, d)]`

The `zip4` function takes four lists and returns a list of quadruples, analogous to `zip`.

`zip5 :: [a] -> [b] -> [c] -> [d] -> [e] -> [(a, b, c, d, e)]`

The `zip5` function takes five lists and returns a list of five-tuples, analogous to `zip`.

```
zip6 ::  [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [(a, b, c, d, e, f)]
```

The `zip6` function takes six lists and returns a list of six-tuples, analogous to `zip`.

```
zip7 ::  [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g] -> [(a, b, c, d, e, f, g)]
```

The `zip7` function takes seven lists and returns a list of seven-tuples, analogous to `zip`.

```
zipWith ::  (a -> b -> c) -> [a] -> [b] -> [c]
```

`zipWith` generalises `zip` 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.

```
zipWith3 ::  (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
```

The `zipWith3` function takes a function which combines three elements, as well as three lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
zipWith4 ::  (a -> b -> c -> d -> e) -> [a] -> [b] -> [c] -> [d] -> [e]
```

The `zipWith4` function takes a function which combines four elements, as well as four lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
zipWith5 ::  (a -> b -> c -> d -> e -> f) -> [a] -> [b] -> [c] -> [d] -> [e] -> [f]
```

The `zipWith5` function takes a function which combines five elements, as well as five lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
zipWith6 ::  (a -> b -> c -> d -> e -> f -> g)
          -> [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g]
```

The `zipWith6` function takes a function which combines six elements, as well as six lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
zipWith7 ::  (a -> b -> c -> d -> e -> f -> g -> h)
          -> [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g] -> [h]
```

The `zipWith7` function takes a function which combines seven elements, as well as seven lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
unzip :: [(a, b)] -> ([a], [b])
```

`unzip` transforms a list of pairs into a list of first components and a list of second components.

```
unzip3 :: [(a, b, c)] -> ([a], [b], [c])
```

The `unzip3` function takes a list of triples and returns three lists, analogous to `unzip`.

```
unzip4 :: [(a, b, c, d)] -> ([a], [b], [c], [d])
```

The `unzip4` function takes a list of quadruples and returns four lists, analogous to `unzip`.

```
unzip5 :: [(a, b, c, d, e)] -> ([a], [b], [c], [d], [e])
```

The `unzip5` function takes a list of five-tuples and returns five lists, analogous to `unzip`.

```
unzip6 :: [(a, b, c, d, e, f)] -> ([a], [b], [c], [d], [e], [f])
```

The `unzip6` function takes a list of six-tuples and returns six lists, analogous to `unzip`.

```
unzip7 :: [(a, b, c, d, e, f, g)] -> ([a], [b], [c], [d], [e], [f], [g])
```

The `unzip7` function takes a list of seven-tuples and returns seven lists, analogous to `unzip`.

13.10 Special lists

13.10.1 Functions on strings

```
lines :: String -> [String]
```

`lines` breaks a string up into a list of strings at newline characters. The resulting strings do not contain newlines.

`words :: String -> [String]`

`words` breaks a string up into a list of words, which were delimited by white space.

`unlines :: [String] -> String`

`unlines` is an inverse operation to `lines`. It joins lines, after appending a terminating newline to each.

`unwords :: [String] -> String`

`unwords` is an inverse operation to `words`. It joins words with separating spaces.

13.10.2 `Set` operations

`nub :: Eq a => [a] -> [a]`

O(n^2). The `nub` function removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element. (The name `nub` means 'essence'.) It is a special case of `nubBy`, which allows the programmer to supply their own equality test.

`delete :: Eq a => a -> [a] -> [a]`

`delete x` removes the first occurrence of `x` from its list argument. For example,

```
delete 'a' "banana" == "bnana"
```

It is a special case of `deleteBy`, which allows the programmer to supply their own equality test.

`(\\) :: Eq a => [a] -> [a] -> [a]`

The `\\` function is list difference ((non-associative). In the result of `xs \\ ys`, the first occurrence of each element of `ys` in turn (if any) has been removed from `xs`. Thus

```
(xs ++ ys) \\ xs == ys.
```

It is a special case of `deleteFirstsBy`, which allows the programmer to supply their own equality test.

`union :: Eq a => [a] -> [a] -> [a]`

The `union` function returns the list union of the two lists. For example,

```
"dog" `union` "cow" == "dogcw"
```

Duplicates, and elements of the first list, are removed from the the second list, but if the first list contains duplicates, so will the result. It is a special case of `unionBy`, which allows the programmer to supply their own equality test.

```
intersect :: Eq a => [a] -> [a] -> [a]
```

The `intersect` function takes the list intersection of two lists. For example,

```
[1,2,3,4] `intersect` [2,4,6,8] == [2,4]
```

If the first list contains duplicates, so will the result.

```
[1,2,2,3,4] `intersect` [6,4,4,2] == [2,2,4]
```

It is a special case of `intersectBy`, which allows the programmer to supply their own equality test.

13.10.3 Ordered lists

```
sort :: Ord a => [a] -> [a]
```

The `sort` function implements a stable sorting algorithm. It is a special case of `sortBy`, which allows the programmer to supply their own comparison function.

```
insert :: Ord a => a -> [a] -> [a]
```

The `insert` function takes an element and a list and inserts the element into the list at the last position where it is still less than or equal to the next element. In particular, if the list is sorted before the call, the result will also be sorted. It is a special case of `insertBy`, which allows the programmer to supply their own comparison function.

13.11 Generalized functions

13.11.1 The `By` operations

By convention, overloaded functions have a non-overloaded counterpart whose name is suffixed with 'By'.

User-supplied equality (replacing an Eq context)

The predicate is assumed to define an equivalence.

`nubBy :: (a -> a -> Bool) -> [a] -> [a]`

The `nubBy` function behaves just like `nub`, except it uses a user-supplied equality predicate instead of the overloaded `==` function.

`deleteBy :: (a -> a -> Bool) -> a -> [a] -> [a]`

The `deleteBy` function behaves like `delete`, but takes a user-supplied equality predicate.

`deleteFirstsBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]`

The `deleteFirstsBy` function takes a predicate and two lists and returns the first list with the first occurrence of each element of the second list removed.

`unionBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]`

The `unionBy` function is the non-overloaded version of `union`.

`intersectBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]`

The `intersectBy` function is the non-overloaded version of `intersect`.

`groupBy :: (a -> a -> Bool) -> [a] -> [[a]]`

The `groupBy` function is the non-overloaded version of `group`.

User-supplied comparison (replacing an Ord context)

The function is assumed to define a total ordering.

`sortBy :: (a -> a -> Ordering) -> [a] -> [a]`

The `sortBy` function is the non-overloaded version of `sort`.

`insertBy :: (a -> a -> Ordering) -> a -> [a] -> [a]`

The non-overloaded version of `insert`.

`maximumBy :: (a -> a -> Ordering) -> [a] -> a`

The `maximumBy` function takes a comparison function and a list and returns the greatest element of the list by the comparison function. The list must be finite and non-empty.

`minimumBy :: (a -> a -> Ordering) -> [a] -> a`

The `minimumBy` function takes a comparison function and a list and returns the least element of the list by the comparison function. The list must be finite and non-empty.

13.11.2 The "generic" operations

The prefix 'generic' indicates an overloaded function that is a generalized version of a `Prelude` function.

`genericLength :: Num i => [b] -> i`

The `genericLength` function is an overloaded version of `length`. In particular, instead of returning an `Int`, it returns any type which is an instance of `Num`. It is, however, less efficient than `length`.

`genericTake :: Integral i => i -> [a] -> [a]`

The `genericTake` function is an overloaded version of `take`, which accepts any `Integral` value as the number of elements to take.

`genericDrop :: Integral i => i -> [a] -> [a]`

The `genericDrop` function is an overloaded version of `drop`, which accepts any `Integral` value as the number of elements to drop.

`genericSplitAt :: Integral i => i -> [b] -> ([b], [b])`

The `genericSplitAt` function is an overloaded version of `splitAt`, which accepts any `Integral` value as the position at which to split.

`genericIndex :: Integral a => [b] -> a -> b`

The `genericIndex` function is an overloaded version of `!!`, which accepts any `Integral` value as the index.

`genericReplicate :: Integral i => i -> a -> [a]`

The `genericReplicate` function is an overloaded version of `replicate`, which accepts any `Integral` value as the number of repetitions to make.

Chapter 14

Data.Maybe

```
module Data.Maybe (
    Maybe(Nothing, Just), maybe, isJust, isNothing, fromJust, fromMaybe,
    listToMaybe, maybeToList, catMaybes, mapMaybe
  ) where
```

14.1 The Maybe type and operations

```
data Maybe a
```

The `Maybe` type encapsulates an optional value. A value of type `Maybe a` either contains a value of type `a` (represented as `Just a`), or it is empty (represented as `Nothing`). Using `Maybe` is a good way to deal with errors or exceptional cases without resorting to drastic measures such as `error`.

The `Maybe` type is also a monad. It is a simple kind of error monad, where all errors are represented by `Nothing`. A richer error monad can be built using the `Data.Either.Either` type

Constructors

```
=  Nothing
|  Just a
```

```
instance Functor Maybe
instance Monad Maybe
instance MonadPlus Maybe
instance Eq a => Eq Maybe a
instance Ord a => Ord Maybe a
instance Read a => Read Maybe a
instance Show a => Show Maybe a
```

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

The `maybe` function takes a default value, a function, and a `Maybe` value. If the `Maybe` value is `Nothing`, the function returns the default value. Otherwise, it applies the function to the value inside the `Just` and returns the result.

Specification:

```
maybe n f Nothing  = n
maybe n f (Just x) = f x
```

`isJust :: Maybe a -> Bool`

The `isJust` function returns `True` iff its argument is of the form `Just _`.

Specification:

```
isJust (Just _) = True
isJust Nothing  = False
```

`isNothing :: Maybe a -> Bool`

The `isNothing` function returns `True` iff its argument is `Nothing`.

Specification:

```
isNothing = not . isJust
```

`fromJust :: Maybe a -> a`

The `fromJust` function extracts the element out of a `Just` and throws an error if its argument is `Nothing`.

Specification:

```
fromJust (Just a) = a
fromJust Nothing  = error "Maybe.fromJust: Nothing"
```

`fromMaybe :: a -> Maybe a -> a`

The `fromMaybe` function takes a default value and and `Maybe` value. If the `Maybe` is `Nothing`, it returns the default values; otherwise, it returns the value contained in the `Maybe`.

Specification:

```
fromMaybe d Nothing  = d
fromMaybe _ (Just a) = a
```

`listToMaybe :: [a] -> Maybe a`

The `listToMaybe` function returns `Nothing` on an empty list or `Just a` where `a` is the first element of the list.

Specification:

```
listToMaybe []    = Nothing
listToMaybe (a:_) = Just a
```

`maybeToList :: Maybe a -> [a]`

The `maybeToList` function returns an empty list when given `Nothing` or a singleton list when not given `Nothing`.

Specification:

```
maybeToList Nothing  = []
maybeToList (Just a) = [a]
```

`catMaybes :: [Maybe a] -> [a]`

The `catMaybes` function takes a list of `Maybe`s and returns a list of all the `Just` values.

Specification:

```
catMaybes ms = [ m | Just m <- ms ]
```

```
mapMaybe ::  (a -> Maybe b) -> [a] -> [b]
```

The `mapMaybe` function is a version of `map` which can throw out elements. In particular, the functional argument returns something of type `Maybe b`. If this is `Nothing`, no element is added on to the result list. If it just `Just b`, then `b` is included in the result list.

Specification:

```
mapMaybe f = catMaybes . map f
```

Chapter 15

Data.Ord

```
module Data.Ord (
    Ordering(EQ, GT, LT), Ord((<), (<=), (>), (>=), compare, max, min)
  ) where
```

```
data Ordering
```

Constructors

```
=  LT
|  EQ
|  GT
```

```
instance Eq Ordering
instance Ix Ordering
instance Ord Ordering
```

```
class Eq a => Ord a where
```

Methods

```
compare ::  a -> a -> Ordering
```

```
(<) ::  a -> a -> Bool
```

```
(<=) ::  a -> a -> Bool
```

```
(>=) ::  a -> a -> Bool
```

```
(>) ::  a -> a -> Bool
```

```
max ::  a -> a -> a
```

```
min ::  a -> a -> a
```

```
instance Ord Bool
instance Ord GeneralCategory
instance Ord Int
instance Ord Int16
instance Ord Int32
instance Ord Int64
instance Ord Int8
instance Ord Ordering
instance Ord Word
instance Ord Word16
instance Ord Word32
instance Ord Word64
instance Ord Word8
instance Ord CChar
instance Ord CClock
instance Ord CDouble
instance Ord CFloat
instance Ord CInt
instance Ord CIntMax
instance Ord CIntPtr
instance Ord CLLong
instance Ord CLong
instance Ord CPtrdiff
instance Ord CSChar
instance Ord CShort
instance Ord CSigAtomic
instance Ord CSize
instance Ord CTime
instance Ord CUChar
instance Ord CUInt
instance Ord CUIntMax
instance Ord CUIntPtr
instance Ord CULLong
instance Ord CULong
instance Ord CUShort
instance Ord CWchar
instance Ord IntPtr
instance Ord WordPtr
instance Ord Char
instance Ord Double
instance Ord Float
instance Ord Integer
instance Ord ExitCode
instance Ord BufferMode
instance Ord IOMode
instance Ord SeekMode
instance Ord ()
instance Ord a => Ord Maybe a
instance Ord ForeignPtr a
instance Ord FunPtr a
instance Ord Ptr a
instance Integral a => Ord Ratio a
instance (Ix i, Ord e) => Ord Array i e
instance (Ord a, Ord b) => Ord (a, b)
instance (Ord a, Ord b, Ord c) => Ord (a, b, c)
instance (Ord a, Ord b, Ord c, Ord d) => Ord (a, b, c, d)
instance (Ord a, Ord b, Ord c, Ord d, Ord e) => Ord (a, b, c, d, e)
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f) => Ord (a, b, c, d, e, f)
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g) => Ord (a, b, c, d, e, f, g)
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g, Ord h) => Ord (a, b, c, d, e, f, g, h)
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g, Ord h, Ord i) => Ord (a, b, c, d, e, f, g, h, i)
```

Chapter 16

Data.Ratio

```
module Data.Ratio (
    Ratio, Rational, (%), numerator, denominator, approxRational
  ) where
```

```
data Ratio a
```

Rational numbers, with numerator and denominator of some `Integral` type.

```
instance Integral a => Enum Ratio a
instance Integral a => Eq Ratio a
instance Integral a => Ord Ratio a
instance Integral a => Fractional Ratio a
instance Integral a => Num Ratio a
instance Integral a => Real Ratio a
instance Integral a => RealFrac Ratio a
instance (Integral a, Read a) => Read Ratio a
instance Integral a => Show Ratio a
```

```
type Rational = Ratio Integer
```

Arbitrary-precision rational numbers, represented as a ratio of two `Integer` values. A rational number may be constructed using the `%` operator.

`(%) :: Integral a => a -> a -> Ratio a`

Forms the ratio of two integral numbers.

`numerator :: Ratio a -> a`

Extract the numerator of the ratio in reduced form: the numerator and denominator have no common factor and the denominator is positive.

`denominator :: Ratio a -> a`

Extract the denominator of the ratio in reduced form: the numerator and denominator have no common factor and the denominator is positive.

`approxRational :: RealFrac a => a -> a -> Rational`

`approxRational`, applied to two real fractional numbers `x` and `epsilon`, returns the simplest rational number within `epsilon` of `x`. A rational number `y` is said to be simpler than another `y'` if

- `abs (numerator y) <= abs (numerator y')`, and
- `denominator y <= denominator y'`.

Any real interval contains a unique simplest rational; in particular, note that `0/1` is the simplest rational of all.

Chapter 17

Data.String

```
module Data.String (
    String
  ) where
```

```
type String = [Char]
```

A `String` is a list of characters. String constants in Haskell are values of type `String`.

Chapter 18

Data.Tuple

```
module Data.Tuple (
    fst, snd, curry, uncurry
  ) where
```

```
fst ::  (a, b) -> a

snd ::  (a, b) -> b

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

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

Chapter 19

Data.Word

```
module Data.Word (
    Word, Word8, Word16, Word32, Word64
  ) where
```

This module provides unsigned integer types of unspecified width (`Word`) and fixed widths (`Word8`, `Word16`, `Word32` and `Word64`). All arithmetic is performed modulo 2^n, where n is the number of bits in the type.

For coercing between any two integer types, use `fromIntegral`. Coercing word types to and from integer types preserves representation, not sign.

The rules that hold for `Enum` instances over a bounded type such as `Int` (see the section of the Haskell language report dealing with arithmetic sequences) also hold for the `Enum` instances over the various `Word` types defined here.

Right and left shifts by amounts greater than or equal to the width of the type result in a zero result. This is contrary to the behaviour in C, which is undefined; a common interpretation is to truncate the shift count to the width of the type, for example 1 ¡¡ 32 == 1 in some C implementations.

```
data Word
```

A Word is an unsigned integral type, with the same size as Int.

```
instance Bits Word
instance Bounded Word
instance Enum Word
instance Eq Word
instance Ix Word
instance Ord Word
instance Storable Word
instance Integral Word
instance Num Word
instance Real Word
instance Read Word
instance Show Word
```

```
data Word8
```

8-bit unsigned integer type

```
instance Bits Word8
instance Bounded Word8
instance Enum Word8
instance Eq Word8
instance Ix Word8
instance Ord Word8
instance Storable Word8
instance Integral Word8
instance Num Word8
instance Real Word8
instance Read Word8
instance Show Word8
```

```
data Word16
```

16-bit unsigned integer type

```
instance Bits Word16
instance Bounded Word16
instance Enum Word16
instance Eq Word16
instance Ix Word16
instance Ord Word16
instance Storable Word16
instance Integral Word16
instance Num Word16
instance Real Word16
instance Read Word16
instance Show Word16
```

```
data Word32
```

32-bit unsigned integer type

```
instance Bits Word32
instance Bounded Word32
instance Enum Word32
instance Eq Word32
instance Ix Word32
instance Ord Word32
instance Storable Word32
instance Integral Word32
instance Num Word32
instance Real Word32
instance Read Word32
instance Show Word32
```

```
data Word64
```

64-bit unsigned integer type

```
instance Bits Word64
instance Bounded Word64
instance Enum Word64
instance Eq Word64
instance Ix Word64
instance Ord Word64
instance Storable Word64
instance Integral Word64
instance Num Word64
instance Real Word64
instance Read Word64
instance Show Word64
```

Chapter 20

Foreign

```
module Foreign (
    module Data.Bits, module Data.Int, module Data.Word, module Foreign.Ptr,
    module Foreign.ForeignPtr, module Foreign.StablePtr,
    module Foreign.Storable, module Foreign.Marshal
  ) where
```

The module `Foreign` combines the interfaces of all modules providing language-independent marshalling support, namely

```
module Data.Bits
module Data.Int
module Data.Word
module Foreign.Ptr
module Foreign.ForeignPtr
module Foreign.StablePtr
module Foreign.Storable
module Foreign.Marshal
```

Chapter 21

Foreign.C

```
module Foreign.C (
    module Foreign.C.Types, module Foreign.C.String, module Foreign.C.Error
  ) where
```

The module `Foreign.C` combines the interfaces of all modules providing C-specific marshalling support, namely

```
module Foreign.C.Types
module Foreign.C.String
module Foreign.C.Error
```

Chapter 22

Foreign.C.Error

```
module Foreign.C.Error (
    Errno(Errno), eOK, e2BIG, eACCES, eADDRINUSE, eADDRNOTAVAIL, eADV,
    eAFNOSUPPORT, eAGAIN, eALREADY, eBADF, eBADMSG, eBADRPC, eBUSY, eCHILD,
    eCOMM, eCONNABORTED, eCONNREFUSED, eCONNRESET, eDEADLK, eDESTADDRREQ,
    eDIRTY, eDOM, eDQUOT, eEXIST, eFAULT, eFBIG, eFTYPE, eHOSTDOWN,
    eHOSTUNREACH, eIDRM, eILSEQ, eINPROGRESS, eINTR, eINVAL, eIO, eISCONN,
    eISDIR, eLOOP, eMFILE, eMLINK, eMSGSIZE, eMULTIHOP, eNAMETOOLONG, eNETDOWN,
    eNETRESET, eNETUNREACH, eNFILE, eNOBUFS, eNODATA, eNODEV, eNOENT, eNOEXEC,
    eNOLCK, eNOLINK, eNOMEM, eNOMSG, eNONET, eNOPROTOOPT, eNOSPC, eNOSR, eNOSTR,
    eNOSYS, eNOTBLK, eNOTCONN, eNOTDIR, eNOTEMPTY, eNOTSOCK, eNOTTY, eNXIO,
    eOPNOTSUPP, ePERM, ePFNOSUPPORT, ePIPE, ePROCLIM, ePROCUNAVAIL,
    ePROGMISMATCH, ePROGUNAVAIL, ePROTO, ePROTONOSUPPORT, ePROTOTYPE, eRANGE,
    eREMCHG, eREMOTE, eROFS, eRPCMISMATCH, eRREMOTE, eSHUTDOWN, eSOCKTNOSUPPORT,
    eSPIPE, eSRCH, eSRMNT, eSTALE, eTIME, eTIMEDOUT, eTOOMANYREFS, eTXTBSY,
    eUSERS, eWOULDBLOCK, eXDEV, isValidErrno, getErrno, resetErrno,
    errnoToIOError, throwErrno, throwErrnoIf, throwErrnoIf_, throwErrnoIfRetry,
    throwErrnoIfRetry_, throwErrnoIfMinus1, throwErrnoIfMinus1_,
    throwErrnoIfMinus1Retry, throwErrnoIfMinus1Retry_, throwErrnoIfNull,
    throwErrnoIfNullRetry, throwErrnoIfRetryMayBlock,
    throwErrnoIfRetryMayBlock_, throwErrnoIfMinus1RetryMayBlock,
    throwErrnoIfMinus1RetryMayBlock_, throwErrnoIfNullRetryMayBlock,
    throwErrnoPath, throwErrnoPathIf, throwErrnoPathIf_, throwErrnoPathIfNull,
    throwErrnoPathIfMinus1, throwErrnoPathIfMinus1_
  ) where
```

The module `Foreign.C.Error` facilitates C-specific error handling of `errno`.

22.1 Haskell representation of errno values

```
newtype Errno
```

Haskell representation for `errno` values. The implementation is deliberately exposed, to allow users to add their own definitions of `Errno` values.

Constructors

```
=  Errno CInt
```

```
instance Eq Errno
```

22.1.1 Common errno symbols

Different operating systems and/or C libraries often support different values of `errno`. This module defines the common values, but due to the open definition of `Errno` users may add definitions which are not predefined.

```
eOK ::  Errno

e2BIG ::  Errno

eACCES ::  Errno

eADDRINUSE ::  Errno

eADDRNOTAVAIL ::  Errno

eADV ::  Errno

eAFNOSUPPORT ::  Errno

eAGAIN ::  Errno

eALREADY ::  Errno

eBADF ::  Errno

eBADMSG ::  Errno

eBADRPC ::  Errno

eBUSY ::  Errno

eCHILD ::  Errno

eCOMM ::  Errno

eCONNABORTED ::  Errno
```

```
eCONNREFUSED ::  Errno

eCONNRESET ::  Errno

eDEADLK ::  Errno

eDESTADDRREQ ::  Errno

eDIRTY ::  Errno

eDOM ::  Errno

eDQUOT ::  Errno

eEXIST ::  Errno

eFAULT ::  Errno

eFBIG ::  Errno

eFTYPE ::  Errno

eHOSTDOWN ::  Errno

eHOSTUNREACH ::  Errno

eIDRM ::  Errno

eILSEQ ::  Errno

eINPROGRESS ::  Errno

eINTR ::  Errno

eINVAL ::  Errno

eIO ::  Errno

eISCONN ::  Errno

eISDIR ::  Errno

eLOOP ::  Errno

eMFILE ::  Errno

eMLINK ::  Errno

eMSGSIZE ::  Errno

eMULTIHOP ::  Errno

eNAMETOOLONG ::  Errno

eNETDOWN ::  Errno
```

```
eNETRESET :: Errno

eNETUNREACH :: Errno

eNFILE :: Errno

eNOBUFS :: Errno

eNODATA :: Errno

eNODEV :: Errno

eNOENT :: Errno

eNOEXEC :: Errno

eNOLCK :: Errno

eNOLINK :: Errno

eNOMEM :: Errno

eNOMSG :: Errno

eNONET :: Errno

eNOPROTOOPT :: Errno

eNOSPC :: Errno

eNOSR :: Errno

eNOSTR :: Errno

eNOSYS :: Errno

eNOTBLK :: Errno

eNOTCONN :: Errno

eNOTDIR :: Errno

eNOTEMPTY :: Errno

eNOTSOCK :: Errno

eNOTTY :: Errno

eNXIO :: Errno

eOPNOTSUPP :: Errno

ePERM :: Errno

ePFNOSUPPORT :: Errno
```

```
ePIPE :: Errno

ePROCLIM :: Errno

ePROCUNAVAIL :: Errno

ePROGMISMATCH :: Errno

ePROGUNAVAIL :: Errno

ePROTO :: Errno

ePROTONOSUPPORT :: Errno

ePROTOTYPE :: Errno

eRANGE :: Errno

eREMCHG :: Errno

eREMOTE :: Errno

eROFS :: Errno

eRPCMISMATCH :: Errno

eRREMOTE :: Errno

eSHUTDOWN :: Errno

eSOCKTNOSUPPORT :: Errno

eSPIPE :: Errno

eSRCH :: Errno

eSRMNT :: Errno

eSTALE :: Errno

eTIME :: Errno

eTIMEDOUT :: Errno

eTOOMANYREFS :: Errno

eTXTBSY :: Errno

eUSERS :: Errno

eWOULDBLOCK :: Errno

eXDEV :: Errno
```

22.1.2 Errno functions

`isValidErrno :: Errno -> Bool`

Yield `True` if the given `Errno` value is valid on the system. This implies that the `Eq` instance of `Errno` is also system dependent as it is only defined for valid values of `Errno`.

`getErrno :: IO Errno`

Get the current value of `errno` in the current thread.

`resetErrno :: IO ()`

Reset the current thread's `errno` value to `eOK`.

`errnoToIOError`

`::`	`String`	the location where the error occured
`->`	`Errno`	the error number
`->`	`Maybe Handle`	optional handle associated with the error
`->`	`Maybe String`	optional filename associated with the error
`->`	`IOError`	

Construct an `IOError` based on the given `Errno` value. The optional information can be used to improve the accuracy of error messages.

`throwErrno`

`::`	`String`	textual description of the error location
`->`	`IO a`	

Throw an `IOError` corresponding to the current value of `getErrno`.

22.1.3 Guards for IO operations that may fail

`throwErrnoIf`

`::`	`(a -> Bool)`	predicate to apply to the result value of the `IO` operation
`->`	`String`	textual description of the location
`->`	`IO a`	the `IO` operation to be executed
`->`	`IO a`	

Throw an `IOError` corresponding to the current value of `getErrno` if the result value of the `IO` action meets the given predicate.

`throwErrnoIf_ :: (a -> Bool) -> String -> IO a -> IO ()`

as `throwErrnoIf`, but discards the result of the `IO` action after error handling.

`throwErrnoIfRetry :: (a -> Bool) -> String -> IO a -> IO a`

as `throwErrnoIf`, but retry the `IO` action when it yields the error code `eINTR` - this amounts to the standard retry loop for interrupted POSIX system calls.

`throwErrnoIfRetry_ :: (a -> Bool) -> String -> IO a -> IO ()`

as `throwErrnoIfRetry`, but discards the result.

`throwErrnoIfMinus1 :: Num a => String -> IO a -> IO a`

Throw an `IOError` corresponding to the current value of `getErrno` if the `IO` action returns a result of `-1`.

`throwErrnoIfMinus1_ :: Num a => String -> IO a -> IO ()`

as `throwErrnoIfMinus1`, but discards the result.

`throwErrnoIfMinus1Retry :: Num a => String -> IO a -> IO a`

Throw an `IOError` corresponding to the current value of `getErrno` if the `IO` action returns a result of `-1`, but retries in case of an interrupted operation.

`throwErrnoIfMinus1Retry_ :: Num a => String -> IO a -> IO ()`

as `throwErrnoIfMinus1`, but discards the result.

`throwErrnoIfNull :: String -> IO (Ptr a) -> IO (Ptr a)`

Throw an `IOError` corresponding to the current value of `getErrno` if the `IO` action returns `nullPtr`.

`throwErrnoIfNullRetry :: String -> IO (Ptr a) -> IO (Ptr a)`

Throw an `IOError` corresponding to the current value of `getErrno` if the `IO` action returns `nullPtr`, but retry in case of an interrupted operation.

`throwErrnoIfRetryMayBlock`

::	(a -> Bool)	predicate to apply to the result value of the IO operation
->	String	textual description of the location
->	IO a	the IO operation to be executed
->	IO b	action to execute before retrying if an immediate retry would block
->	IO a	

as throwErrnoIfRetry, but additionally if the operation yields the error code eAGAIN or eWOULDBLOCK, an alternative action is executed before retrying.

```
throwErrnoIfRetryMayBlock_ ::  (a -> Bool) -> String -> IO a -> IO b -> IO ()
```

as throwErrnoIfRetryMayBlock, but discards the result.

```
throwErrnoIfMinus1RetryMayBlock ::  Num a => String -> IO a -> IO b -> IO a
```

as throwErrnoIfMinus1Retry, but checks for operations that would block.

```
throwErrnoIfMinus1RetryMayBlock_ ::  Num a => String -> IO a -> IO b -> IO ()
```

as throwErrnoIfMinus1RetryMayBlock, but discards the result.

```
throwErrnoIfNullRetryMayBlock ::  String -> IO (Ptr a) -> IO b -> IO (Ptr a)
```

as throwErrnoIfNullRetry, but checks for operations that would block.

```
throwErrnoPath ::  String -> FilePath -> IO a
```

as throwErrno, but exceptions include the given path when appropriate.

```
throwErrnoPathIf ::  (a -> Bool) -> String -> FilePath -> IO a -> IO a
```

as throwErrnoIf, but exceptions include the given path when appropriate.

```
throwErrnoPathIf_ ::  (a -> Bool) -> String -> FilePath -> IO a -> IO ()
```

as throwErrnoIf_, but exceptions include the given path when appropriate.

```
throwErrnoPathIfNull ::  String -> FilePath -> IO (Ptr a) -> IO (Ptr a)
```

as throwErrnoIfNull, but exceptions include the given path when appropriate.

`throwErrnoPathIfMinus1 :: Num a => String -> FilePath -> IO a -> IO a`

as `throwErrnoIfMinus1`, but exceptions include the given path when appropriate.

`throwErrnoPathIfMinus1_ :: Num a => String -> FilePath -> IO a -> IO ()`

as `throwErrnoIfMinus1_`, but exceptions include the given path when appropriate.

Chapter 23

Foreign.C.String

```
module Foreign.C.String (
    CString, CStringLen, peekCString, peekCStringLen, newCString, newCStringLen,
    withCString, withCStringLen, charIsRepresentable, castCharToCChar,
    castCCharToChar, castCharToCUChar, castCUCharToChar, castCharToCSChar,
    castCSCharToChar, peekCAString, peekCAStringLen, newCAString,
    newCAStringLen, withCAString, withCAStringLen, CWString, CWStringLen,
    peekCWString, peekCWStringLen, newCWString, newCWStringLen, withCWString,
    withCWStringLen
  ) where
```

Utilities for primitive marshalling of C strings.

The marshalling converts each Haskell character, representing a Unicode code point, to one or more bytes in a manner that, by default, is determined by the current locale. As a consequence, no guarantees can be made about the relative length of a Haskell string and its corresponding C string, and therefore all the marshalling routines include memory allocation. The translation between Unicode and the encoding of the current locale may be lossy.

23.1 C strings

```
type CString = Ptr CChar
```

A C string is a reference to an array of C characters terminated by NUL.

`type CStringLen = (Ptr CChar, Int)`

A string with explicit length information in bytes instead of a terminating NUL (allowing NUL characters in the middle of the string).

23.1.1 Using a locale-dependent encoding

Currently these functions are identical to their `CAString` counterparts; eventually they will use an encoding determined by the current locale.

`peekCString :: CString -> IO String`

Marshal a NUL terminated C string into a Haskell string.

`peekCStringLen :: CStringLen -> IO String`

Marshal a C string with explicit length into a Haskell string.

`newCString :: String -> IO CString`

Marshal a Haskell string into a NUL terminated C string.

- the Haskell string may *not* contain any NUL characters
- new storage is allocated for the C string and must be explicitly freed using `free` or `finalizerFree`.

`newCStringLen :: String -> IO CStringLen`

Marshal a Haskell string into a C string (ie, character array) with explicit length information.

- new storage is allocated for the C string and must be explicitly freed using `free` or `finalizerFree`.

`withCString :: String -> (CString -> IO a) -> IO a`

Marshal a Haskell string into a NUL terminated C string using temporary storage.

- the Haskell string may *not* contain any NUL characters
- the memory is freed when the subcomputation terminates (either normally or via an exception), so the pointer to the temporary storage must *not* be used after this.

`withCStringLen :: String -> (CStringLen -> IO a) -> IO a`

Marshal a Haskell string into a C string (ie, character array) in temporary storage, with explicit length information.

- the memory is freed when the subcomputation terminates (either normally or via an exception), so the pointer to the temporary storage must *not* be used after this.

`charIsRepresentable :: Char -> IO Bool`

Determines whether a character can be accurately encoded in a `CString`. Unrepresentable characters are converted to '?'.

Currently only Latin-1 characters are representable.

23.1.2 Using 8-bit characters

These variants of the above functions are for use with C libraries that are ignorant of Unicode. These functions should be used with care, as a loss of information can occur.

`castCharToCChar :: Char -> CChar`

Convert a Haskell character to a C character. This function is only safe on the first 256 characters.

`castCCharToChar :: CChar -> Char`

Convert a C byte, representing a Latin-1 character, to the corresponding Haskell character.

`castCharToCUChar :: Char -> CUChar`

Convert a Haskell character to a C `unsigned char`. This function is only safe on the first 256 characters.

`castCUCharToChar :: CUChar -> Char`

Convert a C `unsigned char`, representing a Latin-1 character, to the corresponding Haskell character.

`castCharToCSChar :: Char -> CSChar`

Convert a Haskell character to a C `signed char`. This function is only safe on the first 256 characters.

`castCSCharToChar :: CSChar -> Char`

Convert a C `signed char`, representing a Latin-1 character, to the corresponding Haskell character.

`peekCAString :: CString -> IO String`

Marshal a NUL terminated C string into a Haskell string.

`peekCAStringLen :: CStringLen -> IO String`

Marshal a C string with explicit length into a Haskell string.

`newCAString :: String -> IO CString`

Marshal a Haskell string into a NUL terminated C string.

- the Haskell string may *not* contain any NUL characters
- new storage is allocated for the C string and must be explicitly freed using `free` or `finalizerFree`.

`newCAStringLen :: String -> IO CStringLen`

Marshal a Haskell string into a C string (ie, character array) with explicit length information.

- new storage is allocated for the C string and must be explicitly freed using `free` or `finalizerFree`.

`withCAString :: String -> (CString -> IO a) -> IO a`

Marshal a Haskell string into a NUL terminated C string using temporary storage.

- the Haskell string may *not* contain any NUL characters
- the memory is freed when the subcomputation terminates (either normally or via an exception), so the pointer to the temporary storage must *not* be used after this.

`withCAStringLen :: String -> (CStringLen -> IO a) -> IO a`

Marshal a Haskell string into a C string (ie, character array) in temporary storage, with explicit length information.

- the memory is freed when the subcomputation terminates (either normally or via an exception), so the pointer to the temporary storage must *not* be used after this.

23.2 C wide strings

These variants of the above functions are for use with C libraries that encode Unicode using the C `wchar_t` type in a system-dependent way. The only encodings supported are

- UTF-32 (the C compiler defines `__STDC_ISO_10646__`), or
- UTF-16 (as used on Windows systems).

```
type CWString = Ptr CWchar
```

A C wide string is a reference to an array of C wide characters terminated by NUL.

```
type CWStringLen = (Ptr CWchar, Int)
```

A wide character string with explicit length information in `CWchar`s instead of a terminating NUL (allowing NUL characters in the middle of the string).

```
peekCWString ::  CWString -> IO String
```

Marshal a NUL terminated C wide string into a Haskell string.

```
peekCWStringLen ::  CWStringLen -> IO String
```

Marshal a C wide string with explicit length into a Haskell string.

```
newCWString ::  String -> IO CWString
```

Marshal a Haskell string into a NUL terminated C wide string.

- the Haskell string may *not* contain any NUL characters
- new storage is allocated for the C wide string and must be explicitly freed using `free` or `finalizerFree`.

```
newCWStringLen ::  String -> IO CWStringLen
```

Marshal a Haskell string into a C wide string (ie, wide character array) with explicit length information.

- new storage is allocated for the C wide string and must be explicitly freed using `free` or `finalizerFree`.

`withCWString :: String -> (CWString -> IO a) -> IO a`

Marshal a Haskell string into a NUL terminated C wide string using temporary storage.

- the Haskell string may *not* contain any NUL characters
- the memory is freed when the subcomputation terminates (either normally or via an exception), so the pointer to the temporary storage must *not* be used after this.

`withCWStringLen :: String -> (CWStringLen -> IO a) -> IO a`

Marshal a Haskell string into a NUL terminated C wide string using temporary storage.

- the Haskell string may *not* contain any NUL characters
- the memory is freed when the subcomputation terminates (either normally or via an exception), so the pointer to the temporary storage must *not* be used after this.

Chapter 24

Foreign.C.Types

```
module Foreign.C.Types (
    CChar, CSChar, CUChar, CShort, CUShort, CInt, CUInt, CLong, CULong,
    CPtrdiff, CSize, CWchar, CSigAtomic, CLLong, CULLong, CIntPtr, CUIntPtr,
    CIntMax, CUIntMax, CClock, CTime, CFloat, CDouble, CFile, CFpos, CJmpBuf
  ) where
```

24.1 Representations of C types

24.1.1 Integral types

These types are are represented as `newtype`s of types in `Data.Int` and `Data.Word`, and are instances of `Eq`, `Ord`, `Num`, `Read`, `Show`, `Enum`, `Storable`, `Bounded`, `Real`, `Integral` and `Bits`.

```
data CChar
```

Haskell type representing the C `char` type.

```
instance Bits CChar
instance Bounded CChar
instance Enum CChar
instance Eq CChar
instance Ord CChar
instance Storable CChar
instance Integral CChar
instance Num CChar
instance Real CChar
instance Read CChar
instance Show CChar
```

```
data CSChar
```

Haskell type representing the C `signed char` type.

```
instance Bits CSChar
instance Bounded CSChar
instance Enum CSChar
instance Eq CSChar
instance Ord CSChar
instance Storable CSChar
instance Integral CSChar
instance Num CSChar
instance Real CSChar
instance Read CSChar
instance Show CSChar
```

```
data CUChar
```

Haskell type representing the C `unsigned char` type.

```
instance Bits CUChar
instance Bounded CUChar
instance Enum CUChar
instance Eq CUChar
instance Ord CUChar
instance Storable CUChar
instance Integral CUChar
instance Num CUChar
instance Real CUChar
instance Read CUChar
instance Show CUChar
```

```
data CShort
```

Haskell type representing the C `short` type.

```
instance Bits CShort
instance Bounded CShort
instance Enum CShort
instance Eq CShort
instance Ord CShort
instance Storable CShort
instance Integral CShort
instance Num CShort
instance Real CShort
instance Read CShort
instance Show CShort
```

```
data CUShort
```

Haskell type representing the C `unsigned short` type.

```
instance Bits CUShort
instance Bounded CUShort
instance Enum CUShort
instance Eq CUShort
instance Ord CUShort
instance Storable CUShort
instance Integral CUShort
instance Num CUShort
instance Real CUShort
instance Read CUShort
instance Show CUShort
```

```
data CInt
```

Haskell type representing the C `int` type.

```
instance Bits CInt
instance Bounded CInt
instance Enum CInt
instance Eq CInt
instance Ord CInt
instance Storable CInt
instance Integral CInt
instance Num CInt
instance Real CInt
instance Read CInt
instance Show CInt
```

```
data CUInt
```

Haskell type representing the C `unsigned int` type.

```
instance Bits CUInt
instance Bounded CUInt
instance Enum CUInt
instance Eq CUInt
instance Ord CUInt
instance Storable CUInt
instance Integral CUInt
instance Num CUInt
instance Real CUInt
instance Read CUInt
instance Show CUInt
```

```
data CLong
```

Haskell type representing the C `long` type.

```
instance Bits CLong
instance Bounded CLong
instance Enum CLong
instance Eq CLong
instance Ord CLong
instance Storable CLong
instance Integral CLong
instance Num CLong
instance Real CLong
instance Read CLong
instance Show CLong
```

```
data CULong
```

Haskell type representing the C `unsigned long` type.

```
instance Bits CULong
instance Bounded CULong
instance Enum CULong
instance Eq CULong
instance Ord CULong
instance Storable CULong
instance Integral CULong
instance Num CULong
instance Real CULong
instance Read CULong
instance Show CULong
```

```
data CPtrdiff
```

Haskell type representing the C `ptrdiff_t` type.

```
instance Bits CPtrdiff
instance Bounded CPtrdiff
instance Enum CPtrdiff
instance Eq CPtrdiff
instance Ord CPtrdiff
instance Storable CPtrdiff
instance Integral CPtrdiff
instance Num CPtrdiff
instance Real CPtrdiff
instance Read CPtrdiff
instance Show CPtrdiff
```

```
data CSize
```

Haskell type representing the C `size_t` type.

```
instance Bits CSize
instance Bounded CSize
instance Enum CSize
instance Eq CSize
instance Ord CSize
instance Storable CSize
instance Integral CSize
instance Num CSize
instance Real CSize
instance Read CSize
instance Show CSize
```

```
data CWchar
```

Haskell type representing the C `wchar_t` type.

```
instance Bits CWchar
instance Bounded CWchar
instance Enum CWchar
instance Eq CWchar
instance Ord CWchar
instance Storable CWchar
instance Integral CWchar
instance Num CWchar
instance Real CWchar
instance Read CWchar
instance Show CWchar
```

```
data CSigAtomic
```

Haskell type representing the C `sig_atomic_t` type.

```
instance Bits CSigAtomic
instance Bounded CSigAtomic
instance Enum CSigAtomic
instance Eq CSigAtomic
instance Ord CSigAtomic
instance Storable CSigAtomic
instance Integral CSigAtomic
instance Num CSigAtomic
instance Real CSigAtomic
instance Read CSigAtomic
instance Show CSigAtomic
```

```
data CLLong
```

Haskell type representing the C `long long` type.

```
instance Bits CLLong
instance Bounded CLLong
instance Enum CLLong
instance Eq CLLong
instance Ord CLLong
instance Storable CLLong
instance Integral CLLong
instance Num CLLong
instance Real CLLong
instance Read CLLong
instance Show CLLong
```

```
data CULLong
```

Haskell type representing the C `unsigned long long` type.

```
instance Bits CULLong
instance Bounded CULLong
instance Enum CULLong
instance Eq CULLong
instance Ord CULLong
instance Storable CULLong
instance Integral CULLong
instance Num CULLong
instance Real CULLong
instance Read CULLong
instance Show CULLong
```

```
data CIntPtr
```

```
instance Bits CIntPtr
instance Bounded CIntPtr
instance Enum CIntPtr
instance Eq CIntPtr
instance Ord CIntPtr
instance Storable CIntPtr
instance Integral CIntPtr
instance Num CIntPtr
instance Real CIntPtr
instance Read CIntPtr
instance Show CIntPtr


data CUIntPtr


instance Bits CUIntPtr
instance Bounded CUIntPtr
instance Enum CUIntPtr
instance Eq CUIntPtr
instance Ord CUIntPtr
instance Storable CUIntPtr
instance Integral CUIntPtr
instance Num CUIntPtr
instance Real CUIntPtr
instance Read CUIntPtr
instance Show CUIntPtr


data CIntMax


instance Bits CIntMax
instance Bounded CIntMax
instance Enum CIntMax
instance Eq CIntMax
instance Ord CIntMax
instance Storable CIntMax
instance Integral CIntMax
instance Num CIntMax
instance Real CIntMax
instance Read CIntMax
instance Show CIntMax


data CUIntMax
```

```
instance Bits CUIntMax
instance Bounded CUIntMax
instance Enum CUIntMax
instance Eq CUIntMax
instance Ord CUIntMax
instance Storable CUIntMax
instance Integral CUIntMax
instance Num CUIntMax
instance Real CUIntMax
instance Read CUIntMax
instance Show CUIntMax
```

24.1.2 Numeric types

```
data CClock
```

Haskell type representing the C `clock_t` type.

```
instance Enum CClock
instance Eq CClock
instance Ord CClock
instance Storable CClock
instance Num CClock
instance Real CClock
instance Read CClock
instance Show CClock
```

```
data CTime
```

Haskell type representing the C `time_t` type.

```
instance Enum CTime
instance Eq CTime
instance Ord CTime
instance Storable CTime
instance Num CTime
instance Real CTime
instance Read CTime
instance Show CTime
```

24.1.3 Floating types

```
data CFloat
```

Haskell type representing the C `float` type.

```
instance Enum CFloat
instance Eq CFloat
instance Ord CFloat
instance Storable CFloat
instance Floating CFloat
instance Fractional CFloat
instance Num CFloat
instance Real CFloat
instance RealFloat CFloat
instance RealFrac CFloat
instance Read CFloat
instance Show CFloat
```

```
data CDouble
```

Haskell type representing the C `double` type.

```
instance Enum CDouble
instance Eq CDouble
instance Ord CDouble
instance Storable CDouble
instance Floating CDouble
instance Fractional CDouble
instance Num CDouble
instance Real CDouble
instance RealFloat CDouble
instance RealFrac CDouble
instance Read CDouble
instance Show CDouble
```

24.1.4 Other types

```
data CFile
```

Haskell type representing the C `FILE` type.

```
data CFpos
```

Haskell type representing the C `fpos_t` type.

```
data CJmpBuf
```

Haskell type representing the C `jmp_buf` type.

Chapter 25

Foreign.ForeignPtr

```
module Foreign.ForeignPtr (
    ForeignPtr, FinalizerPtr, FinalizerEnvPtr, newForeignPtr, newForeignPtr_,
    addForeignPtrFinalizer, newForeignPtrEnv, addForeignPtrFinalizerEnv,
    withForeignPtr, finalizeForeignPtr, unsafeForeignPtrToPtr, touchForeignPtr,
    castForeignPtr, mallocForeignPtr, mallocForeignPtrBytes,
    mallocForeignPtrArray, mallocForeignPtrArray0
  ) where
```

25.1 Finalised data pointers

```
data ForeignPtr a
```

The type `ForeignPtr` represents references to objects that are maintained in a foreign language, i.e., that are not part of the data structures usually managed by the Haskell storage manager. The essential difference between `ForeignPtr`s and vanilla memory references of type `Ptr a` is that the former may be associated with finalizers. A finalizer is a routine that is invoked when the Haskell storage manager detects that - within the Haskell heap and stack - there are no more references left that are pointing to the `ForeignPtr`. Typically, the finalizer will, then, invoke routines in the foreign language that free the resources bound by the foreign object.

The `ForeignPtr` is parameterised in the same way as `Ptr`. The type argument of `ForeignPtr` should normally be an instance of class `Storable`.

```
instance Eq ForeignPtr a
instance Ord ForeignPtr a
instance Show ForeignPtr a
```

```
type FinalizerPtr a = FunPtr (Ptr a -> IO ())
```

A finalizer is represented as a pointer to a foreign function that, at finalisation time, gets as an argument a plain pointer variant of the foreign pointer that the finalizer is associated with.

```
type FinalizerEnvPtr env a = FunPtr (Ptr env -> Ptr a -> IO ())
```

25.1.1 Basic operations

```
newForeignPtr :: FinalizerPtr a -> Ptr a -> IO (ForeignPtr a)
```

Turns a plain memory reference into a foreign pointer, and associates a finalizer with the reference. The finalizer will be executed after the last reference to the foreign object is dropped. There is no guarantee of promptness, however the finalizer will be executed before the program exits.

```
newForeignPtr_ :: Ptr a -> IO (ForeignPtr a)
```

Turns a plain memory reference into a foreign pointer that may be associated with finalizers by using `addForeignPtrFinalizer`.

```
addForeignPtrFinalizer :: FinalizerPtr a -> ForeignPtr a -> IO ()
```

This function adds a finalizer to the given foreign object. The finalizer will run *before* all other finalizers for the same object which have already been registered.

```
newForeignPtrEnv :: FinalizerEnvPtr env a -> Ptr env -> Ptr a -> IO (ForeignPtr a)
```

This variant of `newForeignPtr` adds a finalizer that expects an environment in addition to the finalized pointer. The environment that will be passed to the finalizer is fixed by the second argument to `newForeignPtrEnv`.

```
addForeignPtrFinalizerEnv :: FinalizerEnvPtr env a -> Ptr env -> ForeignPtr a -> IO ()
```

Like `addForeignPtrFinalizerEnv` but allows the finalizer to be passed an additional environment parameter to be passed to the finalizer. The environment passed to the finalizer is fixed by the second argument to `addForeignPtrFinalizerEnv`.

`withForeignPtr :: ForeignPtr a -> (Ptr a -> IO b) -> IO b`

This is a way to look at the pointer living inside a foreign object. This function takes a function which is applied to that pointer. The resulting `IO` action is then executed. The foreign object is kept alive at least during the whole action, even if it is not used directly inside. Note that it is not safe to return the pointer from the action and use it after the action completes. All uses of the pointer should be inside the `withForeignPtr` bracket. The reason for this unsafeness is the same as for `unsafeForeignPtrToPtr` below: the finalizer may run earlier than expected, because the compiler can only track usage of the `ForeignPtr` object, not a `Ptr` object made from it.

This function is normally used for marshalling data to or from the object pointed to by the `ForeignPtr`, using the operations from the `Storable` class.

`finalizeForeignPtr :: ForeignPtr a -> IO ()`

Causes the finalizers associated with a foreign pointer to be run immediately.

25.1.2 Low-level operations

`unsafeForeignPtrToPtr :: ForeignPtr a -> Ptr a`

This function extracts the pointer component of a foreign pointer. This is a potentially dangerous operations, as if the argument to `unsafeForeignPtrToPtr` is the last usage occurrence of the given foreign pointer, then its finalizer(s) will be run, which potentially invalidates the plain pointer just obtained. Hence, `touchForeignPtr` must be used wherever it has to be guaranteed that the pointer lives on - i.e., has another usage occurrence.

To avoid subtle coding errors, hand written marshalling code should preferably use `withForeignPtr` rather than combinations of `unsafeForeignPtrToPtr` and `touchForeignPtr`. However, the latter routines are occasionally preferred in tool generated marshalling code.

`touchForeignPtr :: ForeignPtr a -> IO ()`

This function ensures that the foreign object in question is alive at the given place in the sequence of IO actions. In particular `withForeignPtr` does a `touchForeignPtr` after it executes the user action.

Note that this function should not be used to express dependencies between finalizers on `ForeignPtr`s. For example, if the finalizer for a `ForeignPtr F1` calls `touchForeignPtr` on a second `ForeignPtr F2`, then the only guarantee is that the finalizer for `F2` is never started before the finalizer for `F1`. They might be started together if for example both `F1` and `F2` are otherwise unreachable.

In general, it is not recommended to use finalizers on separate objects with ordering constraints between them. To express the ordering robustly requires explicit synchronisation between finalizers.

`castForeignPtr :: ForeignPtr a -> ForeignPtr b`

This function casts a `ForeignPtr` parameterised by one type into another type.

25.1.3 Allocating managed memory

`mallocForeignPtr :: Storable a => IO (ForeignPtr a)`

Allocate some memory and return a `ForeignPtr` to it. The memory will be released automatically when the `ForeignPtr` is discarded.

`mallocForeignPtr` is equivalent to

```
do { p <- malloc; newForeignPtr finalizerFree p }
```

although it may be implemented differently internally: you may not assume that the memory returned by `mallocForeignPtr` has been allocated with `malloc`.

`mallocForeignPtrBytes :: Int -> IO (ForeignPtr a)`

This function is similar to `mallocForeignPtr`, except that the size of the memory required is given explicitly as a number of bytes.

`mallocForeignPtrArray :: Storable a => Int -> IO (ForeignPtr a)`

This function is similar to `mallocArray`, but yields a memory area that has a finalizer attached that releases the memory area. As with `mallocForeignPtr`, it is not guaranteed that the block of memory was allocated by `malloc`.

`mallocForeignPtrArray0 :: Storable a => Int -> IO (ForeignPtr a)`

This function is similar to `mallocArray0`, but yields a memory area that has a finalizer attached that releases the memory area. As with `mallocForeignPtr`, it is not guaranteed that the block of memory was allocated by `malloc`.

Chapter 26

Foreign.Marshal

```
module Foreign.Marshal (
    module Foreign.Marshal.Alloc, module Foreign.Marshal.Array,
    module Foreign.Marshal.Error, module Foreign.Marshal.Utils, unsafeLocalState
  ) where
```

```
module Foreign.Marshal.Alloc
module Foreign.Marshal.Array
module Foreign.Marshal.Error
module Foreign.Marshal.Utils
```

```
unsafeLocalState ::  IO a -> a
```

Sometimes an external entity is a pure function, except that it passes arguments and/or results via pointers. The function `unsafeLocalState` permits the packaging of such entities as pure functions.

The only IO operations allowed in the IO action passed to `unsafeLocalState` are (a) local allocation (`alloca`, `allocaBytes` and derived operations such as `withArray` and `withCString`), and (b) pointer operations (`Foreign.Storable` and `Foreign.Ptr`) on the pointers to local storage, and (c) foreign functions whose only observable effect is to read and/or write the locally allocated memory. Passing an IO operation that does not obey these rules results in undefined behaviour.

It is expected that this operation will be replaced in a future revision of Haskell.

Chapter 27

Foreign.Marshal.Alloc

```
module Foreign.Marshal.Alloc (
    alloca, allocaBytes, malloc, mallocBytes, realloc, reallocBytes, free,
    finalizerFree
  ) where
```

The module `Foreign.Marshal.Alloc` provides operations to allocate and deallocate blocks of raw mem- ory (i.e., unstructured chunks of memory outside of the area maintained by the Haskell storage manager). These memory blocks are commonly used to pass compound data structures to foreign functions or to pro- vide space in which compound result values are obtained from foreign functions.

If any of the allocation functions fails, a value of `nullPtr` is produced. If `free` or `reallocBytes` is applied to a memory area that has been allocated with `alloca` or `allocaBytes`, the behaviour is undefined. Any further access to memory areas allocated with `alloca` or `allocaBytes`, after the computation that was passed to the allocation function has terminated, leads to undefined behaviour. Any further access to the memory area referenced by a pointer passed to `realloc`, `reallocBytes`, or `free` entails undefined behaviour.

All storage allocated by functions that allocate based on a *size in bytes* must be sufficiently aligned for any of the basic foreign types that fits into the newly allocated storage. All storage allocated by functions that allocate based on a specific type must be sufficiently aligned for that type. Array allocation routines need to obey the same alignment constraints for each array element.

27.1 Memory allocation

27.1.1 Local allocation

```
alloca :: Storable a => (Ptr a -> IO b) -> IO b
```

`alloca f` executes the computation `f`, passing as argument a pointer to a temporarily allocated block of memory sufficient to hold values of type `a`.

The memory is freed when `f` terminates (either normally or via an exception), so the pointer passed to `f` must *not* be used after this.

```
allocaBytes :: Int -> (Ptr a -> IO b) -> IO b
```

`allocaBytes n f` executes the computation `f`, passing as argument a pointer to a temporarily allocated block of memory of `n` bytes. The block of memory is sufficiently aligned for any of the basic foreign types that fits into a memory block of the allocated size.

The memory is freed when `f` terminates (either normally or via an exception), so the pointer passed to `f` must *not* be used after this.

27.1.2 Dynamic allocation

```
malloc :: Storable a => IO (Ptr a)
```

Allocate a block of memory that is sufficient to hold values of type `a`. The size of the area allocated is determined by the `sizeOf` method from the instance of `Storable` for the appropriate type.

The memory may be deallocated using `free` or `finalizerFree` when no longer required.

```
mallocBytes :: Int -> IO (Ptr a)
```

Allocate a block of memory of the given number of bytes. The block of memory is sufficiently aligned for any of the basic foreign types that fits into a memory block of the allocated size.

The memory may be deallocated using `free` or `finalizerFree` when no longer required.

```
realloc :: Storable b => Ptr a -> IO (Ptr b)
```

Resize a memory area that was allocated with `malloc` or `mallocBytes` to the size needed to store values of type `b`. The returned pointer may refer to an entirely different memory area, but will be suitably aligned to hold values of type `b`. The contents of the referenced memory area will be the

same as of the original pointer up to the minimum of the original size and the size of values of type `b`.

If the argument to `realloc` is `nullPtr`, `realloc` behaves like `malloc`.

```
reallocBytes ::  Ptr a -> Int -> IO (Ptr a)
```

Resize a memory area that was allocated with `malloc` or `mallocBytes` to the given size. The returned pointer may refer to an entirely different memory area, but will be sufficiently aligned for any of the basic foreign types that fits into a memory block of the given size. The contents of the referenced memory area will be the same as of the original pointer up to the minimum of the original size and the given size.

If the pointer argument to `reallocBytes` is `nullPtr`, `reallocBytes` behaves like `malloc`. If the requested size is `0`, `reallocBytes` behaves like `free`.

```
free ::  Ptr a -> IO ()
```

Free a block of memory that was allocated with `malloc`, `mallocBytes`, `realloc`, `reallocBytes`, `new` or any of the `newX` functions in `Foreign.Marshal.Array` or `Foreign.C.String`.

```
finalizerFree ::  FinalizerPtr a
```

A pointer to a foreign function equivalent to `free`, which may be used as a finalizer (cf `ForeignPtr`) for storage allocated with `malloc`, `mallocBytes`, `realloc` or `reallocBytes`.

Chapter 28

Foreign.Marshal.Array

```
module Foreign.Marshal.Array (
    mallocArray, mallocArray0, allocaArray, allocaArray0, reallocArray,
    reallocArray0, peekArray, peekArray0, pokeArray, pokeArray0, newArray,
    newArray0, withArray, withArray0, withArrayLen, withArrayLen0, copyArray,
    moveArray, lengthArray0, advancePtr
  ) where
```

The module `Foreign.Marshal.Array` provides operations for marshalling Haskell lists into monolithic arrays and vice versa. Most functions come in two flavours: one for arrays terminated by a special termination element and one where an explicit length parameter is used to determine the extent of an array. The typical example for the former case are C's NUL terminated strings. However, please note that C strings should usually be marshalled using the functions provided by `Foreign.C.String` as the Unicode encoding has to be taken into account. All functions specifically operating on arrays that are terminated by a special termination element have a name ending on `0`—e.g., `mallocArray` allocates space for an array of the given size, whereas `mallocArray0` allocates space for one more element to ensure that there is room for the terminator.

28.1 Marshalling arrays

28.1.1 Allocation

```
mallocArray ::  Storable a => Int -> IO (Ptr a)
```

Allocate storage for the given number of elements of a storable type (like `malloc`, but for multiple elements).

```
mallocArray0 ::  Storable a => Int -> IO (Ptr a)
```

Like `mallocArray`, but add an extra position to hold a special termination element.

```
allocaArray ::  Storable a => Int -> (Ptr a -> IO b) -> IO b
```

Temporarily allocate space for the given number of elements (like `alloca`, but for multiple elements).

```
allocaArray0 ::  Storable a => Int -> (Ptr a -> IO b) -> IO b
```

Like `allocaArray`, but add an extra position to hold a special termination element.

```
reallocArray ::  Storable a => Ptr a -> Int -> IO (Ptr a)
```

Adjust the size of an array.

```
reallocArray0 ::  Storable a => Ptr a -> Int -> IO (Ptr a)
```

Adjust the size of an array including an extra position for the end marker.

28.1.2 Marshalling

```
peekArray ::  Storable a => Int -> Ptr a -> IO [a]
```

Convert an array of given length into a Haskell list.

```
peekArray0 ::  (Storable a, Eq a) => a -> Ptr a -> IO [a]
```

Convert an array terminated by the given end marker into a Haskell list.

```
pokeArray ::  Storable a => Ptr a -> [a] -> IO ()
```

Write the list elements consecutive into memory

```
pokeArray0 ::  Storable a => a -> Ptr a -> [a] -> IO ()
```

Write the list elements consecutive into memory and terminate them with the given marker element

28.1.3 Combined allocation and marshalling

`newArray :: Storable a => [a] -> IO (Ptr a)`

Write a list of storable elements into a newly allocated, consecutive sequence of storable values (like `new`, but for multiple elements).

`newArray0 :: Storable a => a -> [a] -> IO (Ptr a)`

Write a list of storable elements into a newly allocated, consecutive sequence of storable values, where the end is fixed by the given end marker

`withArray :: Storable a => [a] -> (Ptr a -> IO b) -> IO b`

Temporarily store a list of storable values in memory (like `with`, but for multiple elements).

`withArray0 :: Storable a => a -> [a] -> (Ptr a -> IO b) -> IO b`

Like `withArray`, but a terminator indicates where the array ends

`withArrayLen :: Storable a => [a] -> (Int -> Ptr a -> IO b) -> IO b`

Like `withArray`, but the action gets the number of values as an additional parameter

`withArrayLen0 :: Storable a => a -> [a] -> (Int -> Ptr a -> IO b) -> IO b`

Like `withArrayLen`, but a terminator indicates where the array ends

28.1.4 Copying

(argument order: destination, source)

`copyArray :: Storable a => Ptr a -> Ptr a -> Int -> IO ()`

Copy the given number of elements from the second array (source) into the first array (destination); the copied areas may *not* overlap

`moveArray :: Storable a => Ptr a -> Ptr a -> Int -> IO ()`

Copy the given number of elements from the second array (source) into the first array (destination); the copied areas *may* overlap

28.1.5 Finding the length

```
lengthArray0 ::  (Storable a, Eq a) => a -> Ptr a -> IO Int
```

Return the number of elements in an array, excluding the terminator

28.1.6 Indexing

```
advancePtr ::  Storable a => Ptr a -> Int -> Ptr a
```

Advance a pointer into an array by the given number of elements

Chapter 29

Foreign.Marshal.Error

```
module Foreign.Marshal.Error (
    throwIf, throwIf_, throwIfNeg, throwIfNeg_, throwIfNull, void
  ) where
```

`throwIf`

```
    ::  (a -> Bool)       error condition on the result of the IO action
    ->  (a -> String)     computes an error message from erroneous results of
                          the IO action
    ->  IO a              the IO action to be executed
    ->  IO a
```

Execute an `IO` action, throwing a `userError` if the predicate yields `True` when applied to the result returned by the `IO` action. If no exception is raised, return the result of the computation.

`throwIf_ :: (a -> Bool) -> (a -> String) -> IO a -> IO ()`

Like `throwIf`, but discarding the result

`throwIfNeg :: (Ord a, Num a) => (a -> String) -> IO a -> IO a`

Guards against negative result values

`throwIfNeg_ :: (Ord a, Num a) => (a -> String) -> IO a -> IO ()`

Like `throwIfNeg`, but discarding the result

`throwIfNull :: String -> IO (Ptr a) -> IO (Ptr a)`

Guards against null pointers

`void :: IO a -> IO ()`

Discard the return value of an `IO` action.

Chapter 30

Foreign.Marshal.Utils

```
module Foreign.Marshal.Utils (
    with, new, fromBool, toBool, maybeNew, maybeWith, maybePeek, withMany,
    copyBytes, moveBytes
  ) where
```

30.1 General marshalling utilities

30.1.1 Combined allocation and marshalling

`with :: Storable a => a -> (Ptr a -> IO b) -> IO b`

> `with val f` executes the computation `f`, passing as argument a pointer to a temporarily allocated block of memory into which `val` has been marshalled (the combination of `alloca` and `poke`).
>
> The memory is freed when `f` terminates (either normally or via an exception), so the pointer passed to `f` must not be used after this.

`new :: Storable a => a -> IO (Ptr a)`

> Allocate a block of memory and marshal a value into it (the combination of `malloc` and `poke`). The size of the area allocated is determined by the `sizeOf` method from the instance of `Storable` for the appropriate type.
>
> The memory may be deallocated using `free` or `finalizerFree` when no longer required.

30.1.2 Marshalling of Boolean values (non-zero corresponds to True)

`fromBool :: Num a => Bool -> a`

Convert a Haskell `Bool` to its numeric representation

`toBool :: Num a => a -> Bool`

Convert a Boolean in numeric representation to a Haskell value

30.1.3 Marshalling of Maybe values

`maybeNew :: (a -> IO (Ptr a)) -> Maybe a -> IO (Ptr a)`

Allocate storage and marshal a storable value wrapped into a `Maybe`

- the `nullPtr` is used to represent `Nothing`

`maybeWith :: (a -> (Ptr b -> IO c) -> IO c) -> Maybe a -> (Ptr b -> IO c) -> IO c`

Converts a `withXXX` combinator into one marshalling a value wrapped into a `Maybe`, using `nullPtr` to represent `Nothing`.

`maybePeek :: (Ptr a -> IO b) -> Ptr a -> IO (Maybe b)`

Convert a peek combinator into a one returning `Nothing` if applied to a `nullPtr`

30.1.4 Marshalling lists of storable objects

`withMany :: (a -> (b -> res) -> res) -> [a] -> ([b] -> res) -> res`

Replicates a `withXXX` combinator over a list of objects, yielding a list of marshalled objects

30.1.5 Haskellish interface to memcpy and memmove

(argument order: destination, source)

`copyBytes :: Ptr a -> Ptr a -> Int -> IO ()`

Copies the given number of bytes from the second area (source) into the first (destination); the copied areas may *not* overlap

```
moveBytes ::  Ptr a -> Ptr a -> Int -> IO ()
```

Copies the given number of bytes from the second area (source) into the first (destination); the copied areas *may* overlap

Chapter 31

Foreign.Ptr

```
module Foreign.Ptr (
    Ptr, nullPtr, castPtr, plusPtr, alignPtr, minusPtr, FunPtr, nullFunPtr,
    castFunPtr, castFunPtrToPtr, castPtrToFunPtr, freeHaskellFunPtr, IntPtr,
    ptrToIntPtr, intPtrToPtr, WordPtr, ptrToWordPtr, wordPtrToPtr
  ) where
```

The module `Foreign.Ptr` provides typed pointers to foreign entities. We distinguish two kinds of pointers: pointers to data and pointers to functions. It is understood that these two kinds of pointers may be represented differently as they may be references to data and text segments, respectively.

31.1 Data pointers

```
data Ptr a
```

A value of type `Ptr a` represents a pointer to an object, or an array of objects, which may be marshalled to or from Haskell values of type `a`.

The type `a` will often be an instance of class `Storable` which provides the marshalling operations. However this is not essential, and you can provide your own operations to access the pointer. For example you might write small foreign functions to get or set the fields of a C `struct`.

```
instance Eq Ptr a
instance Ord Ptr a
instance Storable Ptr a
instance Show Ptr a
```

`nullPtr :: Ptr a`

The constant `nullPtr` contains a distinguished value of `Ptr` that is not associated with a valid memory location

`castPtr :: Ptr a -> Ptr b`

The `castPtr` function casts a pointer from one type to another.

`plusPtr :: Ptr a -> Int -> Ptr b`

Advances the given address by the given offset in bytes.

`alignPtr :: Ptr a -> Int -> Ptr a`

Given an arbitrary address and an alignment constraint, `alignPtr` yields the next higher address that fulfills the alignment constraint. An alignment constraint `x` is fulfilled by any address divisible by `x`. This operation is idempotent.

`minusPtr :: Ptr a -> Ptr b -> Int`

Computes the offset required to get from the second to the first argument. We have

```
p2 == p1 `plusPtr` (p2 `minusPtr` p1)
```

31.2 Function pointers

`data FunPtr a`

A value of type `FunPtr a` is a pointer to a function callable from foreign code. The type `a` will normally be a *foreign type*, a function type with zero or more arguments where

- the argument types are *marshallable foreign types*, i.e. `Char`, `Int`, `Double`, `Float`, `Bool`, `Int8`, `Int16`, `Int32`, `Int64`, `Word8`, `Word16`, `Word32`, `Word64`, `Ptr a`, `FunPtr a`, `Foreign.StablePtr.StablePtr a` or a renaming of any of these using `newtype`.
- the return type is either a marshallable foreign type or has the form `IO t` where `t` is a marshallable foreign type or `()`.

A value of type `FunPtr a` may be a pointer to a foreign function, either returned by another foreign function or imported with a a static address import like

```
foreign import ccall "stdlib.h &free"
  p_free :: FunPtr (Ptr a -> IO ())
```

or a pointer to a Haskell function created using a *wrapper* stub declared to produce a `FunPtr` of the correct type. For example:

```
type Compare = Int -> Int -> Bool
foreign import ccall "wrapper"
  mkCompare :: Compare -> IO (FunPtr Compare)
```

Calls to wrapper stubs like `mkCompare` allocate storage, which should be released with `freeHaskellFunPtr` when no longer required.

To convert `FunPtr` values to corresponding Haskell functions, one can define a dynamic stub for the specific foreign type, e.g.

```
type IntFunction = CInt -> IO ()
foreign import ccall "dynamic"
  mkFun :: FunPtr IntFunction -> IntFunction
```

```
instance Eq FunPtr a
instance Ord FunPtr a
instance Storable FunPtr a
instance Show FunPtr a
```

`nullFunPtr :: FunPtr a`

The constant `nullFunPtr` contains a distinguished value of `FunPtr` that is not associated with a valid memory location

`castFunPtr :: FunPtr a -> FunPtr b`

Casts a `FunPtr` to a `FunPtr` of a different type.

`castFunPtrToPtr :: FunPtr a -> Ptr b`

Casts a `FunPtr` to a `Ptr`.

Note: this is valid only on architectures where data and function pointers range over the same set of addresses, and should only be used for bindings to external libraries whose interface already relies on this assumption.

`castPtrToFunPtr :: Ptr a -> FunPtr b`

Casts a `Ptr` to a `FunPtr`.

Note: this is valid only on architectures where data and function pointers range over the same set of addresses, and should only be used for bindings to external libraries whose interface already relies on this assumption.

```
freeHaskellFunPtr :: FunPtr a -> IO ()
```

Release the storage associated with the given `FunPtr`, which must have been obtained from a wrapper stub. This should be called whenever the return value from a foreign import wrapper function is no longer required; otherwise, the storage it uses will leak.

31.3 Integral types with lossless conversion to and from pointers

```
data IntPtr
```

A signed integral type that can be losslessly converted to and from `Ptr`. This type is also compatible with the C99 type `intptr_t`, and can be marshalled to and from that type safely.

```
instance Bits IntPtr
instance Bounded IntPtr
instance Enum IntPtr
instance Eq IntPtr
instance Ord IntPtr
instance Storable IntPtr
instance Integral IntPtr
instance Num IntPtr
instance Real IntPtr
instance Read IntPtr
instance Show IntPtr
```

```
ptrToIntPtr :: Ptr a -> IntPtr
```

casts a `Ptr` to an `IntPtr`

```
intPtrToPtr :: IntPtr -> Ptr a
```

casts an `IntPtr` to a `Ptr`

```
data WordPtr
```

An unsigned integral type that can be losslessly converted to and from `Ptr`. This type is also compatible with the C99 type `uintptr_t`, and can be marshalled to and from that type safely.

```
instance Bits WordPtr
instance Bounded WordPtr
instance Enum WordPtr
instance Eq WordPtr
instance Ord WordPtr
instance Storable WordPtr
instance Integral WordPtr
instance Num WordPtr
instance Real WordPtr
instance Read WordPtr
instance Show WordPtr
```

```
ptrToWordPtr ::  Ptr a -> WordPtr
```

casts a `Ptr` to a `WordPtr`

```
wordPtrToPtr ::  WordPtr -> Ptr a
```

casts a `WordPtr` to a `Ptr`

Chapter 32

Foreign.StablePtr

```
module Foreign.StablePtr (
    StablePtr, newStablePtr, deRefStablePtr, freeStablePtr, castStablePtrToPtr,
    castPtrToStablePtr
  ) where
```

32.1 Stable references to Haskell values

```
data StablePtr a
```

A *stable pointer* is a reference to a Haskell expression that is guaranteed not to be affected by garbage collection, i.e., it will neither be deallocated nor will the value of the stable pointer itself change during garbage collection (ordinary references may be relocated during garbage collection). Consequently, stable pointers can be passed to foreign code, which can treat it as an opaque reference to a Haskell value.

A value of type `StablePtr a` is a stable pointer to a Haskell expression of type `a`.

```
instance Eq StablePtr a
instance Storable StablePtr a
```

```
newStablePtr ::  a -> IO (StablePtr a)
```

Create a stable pointer referring to the given Haskell value.

`deRefStablePtr :: StablePtr a -> IO a`

Obtain the Haskell value referenced by a stable pointer, i.e., the same value that was passed to the corresponding call to `newStablePtr`. If the argument to `deRefStablePtr` has already been freed using `freeStablePtr`, the behaviour of `deRefStablePtr` is undefined.

`freeStablePtr :: StablePtr a -> IO ()`

Dissolve the association between the stable pointer and the Haskell value. Afterwards, if the stable pointer is passed to `deRefStablePtr` or `freeStablePtr`, the behaviour is undefined. However, the stable pointer may still be passed to `castStablePtrToPtr`, but the `Foreign.Ptr.Ptr ()` value returned by `castStablePtrToPtr`, in this case, is undefined (in particular, it may be `nullPtr`). Nevertheless, the call to `castStablePtrToPtr` is guaranteed not to diverge.

`castStablePtrToPtr :: StablePtr a -> Ptr ()`

Coerce a stable pointer to an address. No guarantees are made about the resulting value, except that the original stable pointer can be recovered by `castPtrToStablePtr`. In particular, the address may not refer to an accessible memory location and any attempt to pass it to the member functions of the class `Storable` leads to undefined behaviour.

`castPtrToStablePtr :: Ptr () -> StablePtr a`

The inverse of `castStablePtrToPtr`, i.e., we have the identity

```
sp == castPtrToStablePtr (castStablePtrToPtr sp)
```

for any stable pointer `sp` on which `freeStablePtr` has not been executed yet. Moreover, `castPtrToStablePtr` may only be applied to pointers that have been produced by `castStablePtrToPtr`.

32.1.1 The C-Side interface

The following definition is available to C programs inter-operating with Haskell code when including the header `HsFFI.h`.

```
typedef void *HsStablePtr;
```

Note that no assumptions may be made about the values representing stable pointers. In fact, they need not even be valid memory addresses. The only guarantee provided is that if they are passed back to Haskell land, the function `deRefStablePtr` will be able to reconstruct the Haskell value referred to by the stable pointer.

Chapter 33

Foreign.Storable

```
module Foreign.Storable (
    Storable(sizeOf,
             alignment,
             peekElemOff,
             pokeElemOff,
             peekByteOff,
             pokeByteOff,
             peek,
             poke)
  ) where
```

`class Storable a where`

The member functions of this class facilitate writing values of primitive types to raw memory (which may have been allocated with the above mentioned routines) and reading values from blocks of raw memory. The class, furthermore, includes support for computing the storage requirements and alignment restrictions of storable types.

Memory addresses are represented as values of type `Ptr a`, for some a which is an instance of class `Storable`. The type argument to `Ptr` helps provide some valuable type safety in FFI code (you can't mix pointers of different types without an explicit cast), while helping the Haskell type system figure out which marshalling method is needed for a given pointer.

All marshalling between Haskell and a foreign language ultimately boils down to translating Haskell data structures into the binary representation

of a corresponding data structure of the foreign language and vice versa. To code this marshalling in Haskell, it is necessary to manipulate primitive data types stored in unstructured memory blocks. The class `Storable` facilitates this manipulation on all types for which it is instantiated, which are the standard basic types of Haskell, the fixed size Int types (`Int8`, `Int16`, `Int32`, `Int64`), the fixed size Word types (`Word8`, `Word16`, `Word32`, `Word64`), `StablePtr`, all types from `Foreign.C.Types`, as well as `Ptr`.

Minimal complete definition: `sizeOf`, `alignment`, one of `peek`, `peekElemOff` and `peekByteOff`, and one of `poke`, `pokeElemOff` and `pokeByteOff`.

Methods

`sizeOf :: a -> Int`

Computes the storage requirements (in bytes) of the argument. The value of the argument is not used.

`alignment :: a -> Int`

Computes the alignment constraint of the argument. An alignment constraint `x` is fulfilled by any address divisible by `x`. The value of the argument is not used.

`peekElemOff :: Ptr a -> Int -> IO a`

Read a value from a memory area regarded as an array of values of the same kind. The first argument specifies the start address of the array and the second the index into the array (the first element of the array has index 0). The following equality holds,

```
peekElemOff addr idx = IOExts.fixIO $ \result ->
  peek (addr `plusPtr` (idx * sizeOf result))
```

Note that this is only a specification, not necessarily the concrete implementation of the function.

`pokeElemOff :: Ptr a -> Int -> a -> IO ()`

Write a value to a memory area regarded as an array of values of the same kind. The following equality holds:

```
pokeElemOff addr idx x =
  poke (addr `plusPtr` (idx * sizeOf x)) x
```

`peekByteOff :: Ptr b -> Int -> IO a`

Read a value from a memory location given by a base address and offset. The following equality holds:

```
peekByteOff addr off = peek (addr `plusPtr` off)
```

`pokeByteOff :: Ptr b -> Int -> a -> IO ()`

Write a value to a memory location given by a base address and offset. The following equality holds:

```
pokeByteOff addr off x = poke (addr `plusPtr` off) x
```

`peek :: Ptr a -> IO a`

Read a value from the given memory location.

Note that the peek and poke functions might require properly aligned addresses to function correctly. This is architecture dependent; thus, portable code should ensure that when peeking or poking values of some type `a`, the alignment constraint for `a`, as given by the function `alignment` is fulfilled.

`poke :: Ptr a -> a -> IO ()`

Write the given value to the given memory location. Alignment restrictions might apply; see `peek`.

```
instance Storable Bool
instance Storable Int
instance Storable Int16
instance Storable Int32
instance Storable Int64
instance Storable Int8
instance Storable Word
instance Storable Word16
instance Storable Word32
instance Storable Word64
instance Storable Word8
instance Storable CChar
instance Storable CClock
instance Storable CDouble
instance Storable CFloat
instance Storable CInt
instance Storable CIntMax
instance Storable CIntPtr
instance Storable CLLong
instance Storable CLong
instance Storable CPtrdiff
instance Storable CSChar
instance Storable CShort
instance Storable CSigAtomic
instance Storable CSize
instance Storable CTime
instance Storable CUChar
instance Storable CUInt
instance Storable CUIntMax
instance Storable CUIntPtr
instance Storable CULLong
instance Storable CULong
instance Storable CUShort
instance Storable CWchar
instance Storable IntPtr
instance Storable WordPtr
instance Storable Char
instance Storable Double
instance Storable Float
instance Storable FunPtr a
instance Storable Ptr a
instance Storable StablePtr a
```

Chapter 34

Numeric

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

34.1 Showing

`showSigned`

`::`	`Real a`	
`=>`	`(a -> ShowS)`	a function that can show unsigned values
`->`	`Int`	the precedence of the enclosing context
`->`	`a`	the value to show
`->`	`ShowS`	

Converts a possibly-negative `Real` value to a string

`showIntAtBase :: Integral a => a -> (Int -> Char) -> a -> ShowS`

Shows a non-negative `Integral` number using the base specified by the first argument, and the character representation specified by the second.

`showInt :: Integral a => a -> ShowS`

Show non-negative `Integral` numbers in base 10.

`showHex :: Integral a => a -> ShowS`

Show non-negative `Integral` numbers in base 16.

`showOct :: Integral a => a -> ShowS`

Show non-negative `Integral` numbers in base 8.

`showEFloat :: RealFloat a => Maybe Int -> a -> ShowS`

Show a signed `RealFloat` value using scientific (exponential) notation (e.g. `2.45e2`, `1.5e-3`).

In the call `showEFloat digs val`, if `digs` is `Nothing`, the value is shown to full precision; if `digs` is `Just d`, then at most `d` digits after the decimal point are shown.

`showFFloat :: RealFloat a => Maybe Int -> a -> ShowS`

Show a signed `RealFloat` value using standard decimal notation (e.g. `245000`, `0.0015`).

In the call `showFFloat digs val`, if `digs` is `Nothing`, the value is shown to full precision; if `digs` is `Just d`, then at most `d` digits after the decimal point are shown.

`showGFloat :: RealFloat a => Maybe Int -> a -> ShowS`

Show a signed `RealFloat` value using standard decimal notation for arguments whose absolute value lies between `0.1` and `9,999,999`, and scientific notation otherwise.

In the call `showGFloat digs val`, if `digs` is `Nothing`, the value is shown to full precision; if `digs` is `Just d`, then at most `d` digits after the decimal point are shown.

`showFloat :: RealFloat a => a -> ShowS`

Show a signed RealFloat value to full precision using standard decimal notation for arguments whose absolute value lies between `0.1` and `9,999,999`, and scientific notation otherwise.

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

`floatToDigits` takes a base and a non-negative `RealFloat` number, and returns a list of digits and an exponent. In particular, if `x>=0`, and

```
floatToDigits base x = ([d1,d2,...,dn], e)
```

then

1. `n >= 1`
2. `x = 0.d1d2...dn * (base**e)`
3. `0 <= di <= base-1`

34.2 Reading

NB: `readInt` is the 'dual' of `showIntAtBase`, and `readDec` is the 'dual' of `showInt`. The inconsistent naming is a historical accident.

`readSigned :: Real a => ReadS a -> ReadS a`

Reads a signed `Real` value, given a reader for an unsigned value

`readInt`

`::`	`Num a`	
`=>`	`a`	the base
`->`	`(Char -> Bool)`	a predicate distinguishing valid digits in this base
`->`	`(Char -> Int)`	a function converting a valid digit character to an Int
`->`	`ReadS a`	

Reads an unsigned `Integral` value in an arbitrary base.

`readDec :: Num a => ReadS a`

Read an unsigned number in decimal notation.

`readOct :: Num a => ReadS a`

Read an unsigned number in octal notation.

`readHex :: Num a => ReadS a`

Read an unsigned number in hexadecimal notation. Both upper or lower case letters are allowed

`readFloat :: RealFrac a => ReadS a`

Reads an unsigned `RealFrac` value, expressed in decimal scientific notation

`lexDigits :: ReadS String`

Reads a non-empty string of decimal digits.

34.3 Miscellaneous

```
fromRat ::  RealFloat a => Rational -> a
```

Converts a `Rational` value into any type in class `RealFloat`.

Chapter 35

Prelude

```
module Prelude (
    Bool(False, True), (&&), (||), not, otherwise, Maybe(Nothing, Just), maybe,
    Either(Left, Right), either, Ordering(LT, EQ, GT), Char, String, fst, snd,
    curry, uncurry, Eq((==), (/=)),
    Ord(compare, (<), (<=), (>=), (>), max, min),
    Enum(succ,
         pred,
         toEnum,
         fromEnum,
         enumFrom,
         enumFromThen,
         enumFromTo,
         enumFromThenTo),
    Bounded(minBound, maxBound), Int, Integer, Float, Double, Rational,
    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),
    subtract, even, odd, gcd, lcm, (^), (^^), fromIntegral, realToFrac,
    Functor(fmap), Monad((>>=), (>>), return, fail), mapM, mapM_, sequence,
    sequence_, (=<<), id, const, (.), flip, ($), until, asTypeOf, error,
    undefined, seq, ($!), map, (++), filter, concat, concatMap, 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,
    zip, zip3, zipWith, zipWith3, unzip, unzip3, ShowS,
    Show(showsPrec, show, showList), shows, showChar, showString, showParen,
    ReadS, Read(readsPrec, readList), reads, readParen, read, lex, IO, putChar,
    putStr, putStrLn, print, getChar, getLine, getContents, interact, FilePath,
    readFile, writeFile, appendFile, readIO, readLn, IOError, ioError,
    userError, catch
  ) where
```

35.1 Standard types, classes and related functions

35.1.1 Basic data types

```
data Bool
```

Constructors

```
=  False
|  True
```

```
instance Eq Bool
instance Ix Bool
instance Ord Bool
instance Storable Bool
```

```
(&&) ::  Bool -> Bool -> Bool
```

Boolean "and", lazy in the second argument

Specification:

```
True && x = x
False && _ = False
```

```
(||) ::  Bool -> Bool -> Bool
```

Boolean "or", lazy in the second argument

Specification:

```
True || _ = True
False || x = x
```

```
not ::  Bool -> Bool
```

Boolean "not"

Specification:

```
not True = False
not False = True
```

```
otherwise ::  Bool
```

`otherwise` is defined as the value `True`. It helps to make guards more readable, e.g.

```
f x | x < 0     = ...
    | otherwise = ...
```

Specification:

```
otherwise = True
```

```
data Maybe a
```

The `Maybe` type encapsulates an optional value. A value of type `Maybe a` either contains a value of type `a` (represented as `Just a`), or it is empty (represented as `Nothing`). Using `Maybe` is a good way to deal with errors or exceptional cases without resorting to drastic measures such as `error`.

The `Maybe` type is also a monad. It is a simple kind of error monad, where all errors are represented by `Nothing`. A richer error monad can be built using the `Data.Either.Either` type

Constructors

```
=  Nothing
|  Just a
```

```
instance Functor Maybe
instance Monad Maybe
instance MonadPlus Maybe
instance Eq a => Eq Maybe a
instance Ord a => Ord Maybe a
instance Read a => Read Maybe a
instance Show a => Show Maybe a
```

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

The `maybe` function takes a default value, a function, and a `Maybe` value. If the `Maybe` value is `Nothing`, the function returns the default value. Otherwise, it applies the function to the value inside the `Just` and returns the result.

Specification:

```
maybe n f Nothing  = n
maybe n f (Just x) = f x
```

```
data Either a b
```

Constructors

```
=  Left a
|  Right b
```

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

Case analysis for the `Either` type. If the value is `Left a`, apply the first function to `a`; if it is `Right b`, apply the second function to `b`.

Specification:

```
either f g (Left x)  =  f x
either f g (Right y) =  g y
```

```
data Ordering
```

Constructors

```
=  LT
|  EQ
|  GT
```

```
instance Eq Ordering
instance Ix Ordering
instance Ord Ordering
```

```
data Char
```

The character type `Char` is an enumeration whose values represent Unicode (or equivalently ISO/IEC 10646) characters (see `http://www.unicode.org/` for details). This set extends the ISO 8859-1 (Latin-1) character set (the first 256 charachers), which is itself an extension of the ASCII character set (the first 128 characters). A character literal in Haskell has type `Char`. To convert a `Char` to or from the corresponding `Int` value defined by Unicode, use `Prelude.toEnum` and `Prelude.fromEnum` from the `Prelude.Enum` class respectively (or equivalently `ord` and `chr`).

```
instance Bounded Char
instance Enum Char
instance Eq Char
instance Ix Char
instance Ord Char
instance Storable Char
instance Read Char
instance Show Char
```

```
type String = [Char]
```

A `String` is a list of characters. String constants in Haskell are values of type `String`.

Tuples

```
fst ::  (a, b) -> a

snd ::  (a, b) -> b
```

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

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

35.1.2 Basic type classes

```
class Eq a where
```

Methods

```
(==) ::  a -> a -> Bool
```

```
(/=) ::  a -> a -> Bool
```

```
instance Eq Bool
instance Eq GeneralCategory
instance Eq Int
instance Eq Int16
instance Eq Int32
instance Eq Int64
instance Eq Int8
instance Eq Ordering
instance Eq Word
instance Eq Word16
instance Eq Word32
instance Eq Word64
instance Eq Word8
instance Eq Errno
instance Eq CChar
instance Eq CClock
instance Eq CDouble
instance Eq CFloat
instance Eq CInt
instance Eq CIntMax
instance Eq CIntPtr
instance Eq CLLong
instance Eq CLong
instance Eq CPtrdiff
instance Eq CSChar
instance Eq CShort
instance Eq CSigAtomic
instance Eq CSize
instance Eq CTime
instance Eq CUChar
instance Eq CUInt
instance Eq CUIntMax
instance Eq CUIntPtr
instance Eq CULLong
instance Eq CULong
instance Eq CUShort
instance Eq CWchar
instance Eq IntPtr
instance Eq WordPtr
instance Eq Char
instance Eq Double
instance Eq Float
instance Eq Integer
instance Eq ExitCode
instance Eq BufferMode
instance Eq Handle
instance Eq HandlePosn
instance Eq IOMode
instance Eq SeekMode
instance Eq IOErrorType
instance Eq ()
instance Eq a => Eq Complex a
instance Eq a => Eq Maybe a
instance Eq ForeignPtr a
instance Eq FunPtr a
instance Eq Ptr a
instance Eq StablePtr a
instance Integral a => Eq Ratio a
instance (Ix i, Eq e) => Eq Array i e
instance (Eq a, Eq b) => Eq (a, b)
instance (Eq a, Eq b, Eq c) => Eq (a, b, c)
```

```
class Eq a => Ord a where
```

Methods

```
compare ::  a -> a -> Ordering
```

```
(<) ::  a -> a -> Bool
```

```
(<=) ::  a -> a -> Bool
```

```
(>=) ::  a -> a -> Bool
```

```
(>) ::  a -> a -> Bool
```

```
max ::  a -> a -> a
```

```
min ::  a -> a -> a
```

```
instance Ord Bool
instance Ord GeneralCategory
instance Ord Int
instance Ord Int16
instance Ord Int32
instance Ord Int64
instance Ord Int8
instance Ord Ordering
instance Ord Word
instance Ord Word16
instance Ord Word32
instance Ord Word64
instance Ord Word8
instance Ord CChar
instance Ord CClock
instance Ord CDouble
instance Ord CFloat
instance Ord CInt
instance Ord CIntMax
instance Ord CIntPtr
instance Ord CLLong
instance Ord CLong
instance Ord CPtrdiff
instance Ord CSChar
instance Ord CShort
instance Ord CSigAtomic
instance Ord CSize
instance Ord CTime
instance Ord CUChar
instance Ord CUInt
instance Ord CUIntMax
instance Ord CUIntPtr
instance Ord CULLong
instance Ord CULong
instance Ord CUShort
instance Ord CWchar
instance Ord IntPtr
instance Ord WordPtr
instance Ord Char
instance Ord Double
instance Ord Float
instance Ord Integer
instance Ord ExitCode
instance Ord BufferMode
instance Ord IOMode
instance Ord SeekMode
instance Ord ()
instance Ord a => Ord Maybe a
instance Ord ForeignPtr a
instance Ord FunPtr a
instance Ord Ptr a
instance Integral a => Ord Ratio a
instance (Ix i, Ord e) => Ord Array i e
instance (Ord a, Ord b) => Ord (a, b)
instance (Ord a, Ord b, Ord c) => Ord (a, b, c)
instance (Ord a, Ord b, Ord c, Ord d) => Ord (a, b, c, d)
instance (Ord a, Ord b, Ord c, Ord d, Ord e) => Ord (a, b, c, d, e)
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f) => Ord (a, b, c, d, e, f)
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g) => Ord (a, b, c, d, e, f, g)
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g, Ord h) => Ord (a, b, c, d, e, f, g, h)
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g, Ord h, Ord i) => Ord (a, b, c, d, e, f, g, h, i)
```

```
class Enum a where
```

Methods

```
succ ::  a -> a

pred ::  a -> a

toEnum ::  Int -> a

fromEnum ::  a -> Int

enumFrom ::  a -> [a]

enumFromThen ::  a -> a -> [a]

enumFromTo ::  a -> a -> [a]

enumFromThenTo ::  a -> a -> a -> [a]
```

```
instance Enum GeneralCategory
instance Enum Int
instance Enum Int16
instance Enum Int32
instance Enum Int64
instance Enum Int8
instance Enum Word
instance Enum Word16
instance Enum Word32
instance Enum Word64
instance Enum Word8
instance Enum CChar
instance Enum CClock
instance Enum CDouble
instance Enum CFloat
instance Enum CInt
instance Enum CIntMax
instance Enum CIntPtr
instance Enum CLLong
instance Enum CLong
instance Enum CPtrdiff
instance Enum CSChar
instance Enum CShort
instance Enum CSigAtomic
instance Enum CSize
instance Enum CTime
instance Enum CUChar
instance Enum CUInt
instance Enum CUIntMax
instance Enum CUIntPtr
instance Enum CULLong
instance Enum CULong
instance Enum CUShort
instance Enum CWchar
instance Enum IntPtr
instance Enum WordPtr
instance Enum Char
instance Enum Double
instance Enum Float
instance Enum Integer
instance Enum IOMode
instance Enum SeekMode
instance Integral a => Enum Ratio a


class Bounded a where
```

Methods

```
minBound ::  a
```

```
    maxBound ::  a


instance Bounded GeneralCategory
instance Bounded Int
instance Bounded Int16
instance Bounded Int32
instance Bounded Int64
instance Bounded Int8
instance Bounded Word
instance Bounded Word16
instance Bounded Word32
instance Bounded Word64
instance Bounded Word8
instance Bounded CChar
instance Bounded CInt
instance Bounded CIntMax
instance Bounded CIntPtr
instance Bounded CLLong
instance Bounded CLong
instance Bounded CPtrdiff
instance Bounded CSChar
instance Bounded CShort
instance Bounded CSigAtomic
instance Bounded CSize
instance Bounded CUChar
instance Bounded CUInt
instance Bounded CUIntMax
instance Bounded CUIntPtr
instance Bounded CULLong
instance Bounded CULong
instance Bounded CUShort
instance Bounded CWchar
instance Bounded IntPtr
instance Bounded WordPtr
instance Bounded Char
```

35.1.3 Numbers

Numeric types

`data Int`

> A fixed-precision integer type with at least the range [-2^29 .. 2^29-1]. The exact range for a given implementation can be determined by using `minBound` and `maxBound` from the `Bounded` class.

```
instance Bits Int
instance Bounded Int
instance Enum Int
instance Eq Int
instance Ix Int
instance Ord Int
instance Storable Int
instance Integral Int
instance Num Int
instance Real Int
instance Read Int
instance Show Int


data Integer


instance Bits Integer
instance Enum Integer
instance Eq Integer
instance Ix Integer
instance Ord Integer
instance Integral Integer
instance Num Integer
instance Real Integer
instance Read Integer
instance Show Integer


data Float


instance Enum Float
instance Eq Float
instance Ord Float
instance Storable Float
instance Floating Float
instance Fractional Float
instance Num Float
instance Real Float
instance RealFloat Float
instance RealFrac Float
instance Read Float
instance Show Float


data Double
```

```
instance Enum Double
instance Eq Double
instance Ord Double
instance Storable Double
instance Floating Double
instance Fractional Double
instance Num Double
instance Real Double
instance RealFloat Double
instance RealFrac Double
instance Read Double
instance Show Double
```

```
type Rational = Ratio Integer
```

Arbitrary-precision rational numbers, represented as a ratio of two `Integer` values. A rational number may be constructed using the `%` operator.

Numeric type classes

```
class Num a where
```

Methods

```
(+) ::  a -> a -> a
```

```
(-) ::  a -> a -> a
```

```
(*) ::  a -> a -> a
```

```
negate ::  a -> a
```

```
abs ::  a -> a
```

```
signum ::  a -> a
```

```
fromInteger ::  Integer -> a
```

```
instance Num Int
instance Num Int16
instance Num Int32
instance Num Int64
instance Num Int8
instance Num Word
instance Num Word16
instance Num Word32
instance Num Word64
instance Num Word8
instance Num CChar
instance Num CClock
instance Num CDouble
instance Num CFloat
instance Num CInt
instance Num CIntMax
instance Num CIntPtr
instance Num CLLong
instance Num CLong
instance Num CPtrdiff
instance Num CSChar
instance Num CShort
instance Num CSigAtomic
instance Num CSize
instance Num CTime
instance Num CUChar
instance Num CUInt
instance Num CUIntMax
instance Num CUIntPtr
instance Num CULLong
instance Num CULong
instance Num CUShort
instance Num CWchar
instance Num IntPtr
instance Num WordPtr
instance Num Double
instance Num Float
instance Num Integer
instance RealFloat a => Num Complex a
instance Integral a => Num Ratio a
```

```
class (Num a, Ord a) => Real a where
```

Methods

```
toRational ::  a -> Rational
```

```
instance Real Int
instance Real Int16
instance Real Int32
instance Real Int64
instance Real Int8
instance Real Word
instance Real Word16
instance Real Word32
instance Real Word64
instance Real Word8
instance Real CChar
instance Real CClock
instance Real CDouble
instance Real CFloat
instance Real CInt
instance Real CIntMax
instance Real CIntPtr
instance Real CLLong
instance Real CLong
instance Real CPtrdiff
instance Real CSChar
instance Real CShort
instance Real CSigAtomic
instance Real CSize
instance Real CTime
instance Real CUChar
instance Real CUInt
instance Real CUIntMax
instance Real CUIntPtr
instance Real CULLong
instance Real CULong
instance Real CUShort
instance Real CWchar
instance Real IntPtr
instance Real WordPtr
instance Real Double
instance Real Float
instance Real Integer
instance Integral a => Real Ratio a


class (Real a, Enum a) => Integral a where
```

Methods

```
quot ::  a -> a -> a

rem ::  a -> a -> a

div ::  a -> a -> a
```

```
mod ::  a -> a -> a

quotRem ::  a -> a -> (a, a)

divMod ::  a -> a -> (a, a)

toInteger ::  a -> Integer
```

```
instance Integral Int
instance Integral Int16
instance Integral Int32
instance Integral Int64
instance Integral Int8
instance Integral Word
instance Integral Word16
instance Integral Word32
instance Integral Word64
instance Integral Word8
instance Integral CChar
instance Integral CInt
instance Integral CIntMax
instance Integral CIntPtr
instance Integral CLLong
instance Integral CLong
instance Integral CPtrdiff
instance Integral CSChar
instance Integral CShort
instance Integral CSigAtomic
instance Integral CSize
instance Integral CUChar
instance Integral CUInt
instance Integral CUIntMax
instance Integral CUIntPtr
instance Integral CULLong
instance Integral CULong
instance Integral CUShort
instance Integral CWchar
instance Integral IntPtr
instance Integral WordPtr
instance Integral Integer
```

```
class Num a => Fractional a where
```

Methods

```
(/) ::  a -> a -> a

recip ::  a -> a
```

```
    fromRational ::  Rational -> a


instance Fractional CDouble
instance Fractional CFloat
instance Fractional Double
instance Fractional Float
instance RealFloat a => Fractional Complex a
instance Integral a => Fractional Ratio a


class Fractional a => Floating a where
```

Methods

```
    pi ::  a

    exp ::  a -> a

    log ::  a -> a

    sqrt ::  a -> a

    (**) ::  a -> a -> a

    logBase ::  a -> a -> a

    sin ::  a -> a

    cos ::  a -> a

    tan ::  a -> a

    asin ::  a -> a

    acos ::  a -> a

    atan ::  a -> a

    sinh ::  a -> a

    cosh ::  a -> a

    tanh ::  a -> a

    asinh ::  a -> a

    acosh ::  a -> a

    atanh ::  a -> a
```

```
instance Floating CDouble
instance Floating CFloat
instance Floating Double
instance Floating Float
instance RealFloat a => Floating Complex a
```

```
class (Real a, Fractional a) => RealFrac a where
```

Methods

```
properFraction ::  Integral b => a -> (b, a)

truncate ::  Integral b => a -> b

round ::  Integral b => a -> b

ceiling ::  Integral b => a -> b

floor ::  Integral b => a -> b
```

```
instance RealFrac CDouble
instance RealFrac CFloat
instance RealFrac Double
instance RealFrac Float
instance Integral a => RealFrac Ratio a
```

```
class (RealFrac a, Floating a) => RealFloat a where
```

Methods

```
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 ::  a -> Bool
```

```
    isInfinite ::  a -> Bool

    isDenormalized ::  a -> Bool

    isNegativeZero ::  a -> Bool

    isIEEE ::  a -> Bool

    atan2 ::  a -> a -> a


instance RealFloat CDouble
instance RealFloat CFloat
instance RealFloat Double
instance RealFloat Float
```

Numeric functions

```
subtract ::  Num a => a -> a -> a

even ::  Integral a => a -> Bool

odd ::  Integral a => a -> Bool

gcd ::  Integral a => a -> a -> a

lcm ::  Integral a => a -> a -> a

(^) ::  (Num a, Integral b) => a -> b -> a

(^^) ::  (Fractional a, Integral b) => a -> b -> a

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

realToFrac ::  (Real a, Fractional b) => a -> b
```

35.1.4 Monads and functors

```
class Functor f where
```

The Functor class is used for types that can be mapped over. Instances of Functor should satisfy the following laws:

```
fmap id == id
fmap (f . g) == fmap f . fmap g
```

The instances of `Functor` for lists, `Maybe` and `IO` satisfy these laws.

Methods

```
fmap ::  (a -> b) -> f a -> f b
```

```
instance Functor Maybe
instance Functor IO
instance Functor List
instance Ix i => Functor Array i
```

```
class Monad m where
```

The Monad class defines the basic operations over a monad, a concept from a branch of mathematics known as category theory. From the perspective of a Haskell programmer, however, it is best to think of a monad as an abstract datatype of actions. Haskell's do expressions provide a convenient syntax for writing monadic expressions.

Minimal complete definition: ¿¿= and return.

Instances of `Monad` should satisfy the following laws:

```
return a >>= k == k a
m >>= return == m
m >>= (x -> k x >>= h) == (m >>= k) >>= h
```

Instances of both `Monad` and `Functor` should additionally satisfy the law:

```
fmap f xs == xs >>= return . f
```

The instances of `Monad` for lists, `Maybe` and `IO` defined in the Prelude satisfy these laws.

Methods

```
(>>=) ::  m a -> (a -> m b) -> m b
```

Sequentially compose two actions, passing any value produced by the first as an argument to the second.

```
(>>) ::  m a -> m b -> m b
```

Sequentially compose two actions, discarding any value produced by the first, like sequencing operators (such as the semicolon) in imperative languages.

```
return ::  a -> m a
```

Inject a value into the monadic type.

```
fail ::  String -> m a
```

Fail with a message. This operation is not part of the mathematical definition of a monad, but is invoked on pattern-match failure in a do expression.

```
instance Monad Maybe
instance Monad IO
instance Monad List
```

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

`mapM f` is equivalent to `sequence . map f`.

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
```

`mapM_ f` is equivalent to `sequence_ . map f`.

```
sequence :: Monad m => [m a] -> m [a]
```

Evaluate each action in the sequence from left to right, and collect the results.

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

Evaluate each action in the sequence from left to right, and ignore the results.

```
(=<<) :: Monad m => (a -> m b) -> m a -> m b
```

Same as `>>=`, but with the arguments interchanged.

35.1.5 Miscellaneous functions

```
id :: a -> a

const :: a -> b -> a

(.) :: (b -> c) -> (a -> b) -> a -> c

flip :: (a -> b -> c) -> b -> a -> c

($) :: (a -> b) -> a -> b

until :: (a -> Bool) -> (a -> a) -> a -> a

asTypeOf :: a -> a -> a

error :: String -> a

undefined :: a

seq :: a -> b -> b

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

35.2 List operations

`map :: (a -> b) -> [a] -> [b]`

`map f xs` is the list obtained by applying `f` to each element of `xs`, i.e.,

```
map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
map f [x1, x2, ...] == [f x1, f x2, ...]
```

`(++) :: [a] -> [a] -> [a]`

Append two lists, i.e.,

```
[x1, ..., xm] ++ [y1, ..., yn] == [x1, ..., xm, y1, ..., yn]
[x1, ..., xm] ++ [y1, ...] == [x1, ..., xm, y1, ...]
```

If the first list is not finite, the result is the first list.

`filter :: (a -> Bool) -> [a] -> [a]`

`filter`, applied to a predicate and a list, returns the list of those elements that satisfy the predicate; i.e.,

```
filter p xs = [ x | x <- xs, p x]
```

`concat :: [[a]] -> [a]`

Concatenate a list of lists.

`concatMap :: (a -> [b]) -> [a] -> [b]`

Map a function over a list and concatenate the results.

`head :: [a] -> a`

Extract the first element of a list, which must be non-empty.

`last :: [a] -> a`

Extract the last element of a list, which must be finite and non-empty.

`tail :: [a] -> [a]`

Extract the elements after the head of a list, which must be non-empty.

`init :: [a] -> [a]`

Return all the elements of a list except the last one. The list must be non-empty.

```
null ::  [a] -> Bool
```

Test whether a list is empty.

```
length ::  [a] -> Int
```

O(n). `length` returns the length of a finite list as an `Int`. It is an instance of the more general `genericLength`, the result type of which may be any kind of number.

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

List index (subscript) operator, starting from 0. It is an instance of the more general `genericIndex`, which takes an index of any integral type.

```
foldl ::  (a -> b -> a) -> a -> [b] -> a
```

`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
```

The list must be finite.

```
foldl1 ::  (a -> a -> a) -> [a] -> a
```

`foldl1` is a variant of `foldl` that has no starting value argument, and thus must be applied to non- empty lists.

```
scanl ::  (a -> b -> a) -> a -> [b] -> [a]
```

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

`scanl1` is a variant of `scanl` that has no starting value argument:

```
scanl1 f [x1, x2, ...] == [x1, x1 `f` x2, ...]
```

`foldr :: (a -> b -> b) -> b -> [a] -> b`

`foldr`, applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left:

```
foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)
```

`foldr1 :: (a -> a -> a) -> [a] -> a`

`foldr1` is a variant of `foldr` that has no starting value argument, and thus must be applied to non- empty lists.

`scanr :: (a -> b -> b) -> b -> [a] -> [b]`

`scanr` is the right-to-left dual of `scanl`. Note that

```
head (scanr f z xs) == foldr f z xs.
```

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

`scanr1` is a variant of `scanr` that has no starting value argument.

`iterate :: (a -> a) -> a -> [a]`

`iterate f x` returns an infinite list of repeated applications of `f` to `x`:

```
iterate f x == [x, f x, f (f x), ...]
```

`repeat :: a -> [a]`

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

`replicate :: Int -> a -> [a]`

`replicate n x` is a list of length `n` with `x` the value of every element. It is an instance of the more general `Data.List.genericReplicate`, in which `n` may be of any integral type.

`cycle :: [a] -> [a]`

`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.

`take :: Int -> [a] -> [a]`

`take n`, applied to a list `xs`, returns the prefix of `xs` of length `n`, or `xs` itself if `n > length xs`:

```
take 5 "Hello World!" == \haddocktt{Hello}
take 3 [1,2,3,4,5] == [1,2,3]
take 3 [1,2] == [1,2]
take 3 [] == []
take (-1) [1,2] == []
take 0 [1,2] == []
```

It is an instance of the more general `genericTake`, in which `n` may be of any integral type.

`drop :: Int -> [a] -> [a]`

`drop n xs` returns the suffix of `xs` after the first `n` elements, or `[]` if `n > length xs`:

```
drop 6 "Hello World!" == "World!"
drop 3 [1,2,3,4,5] == [4,5]
drop 3 [1,2] == []
drop 3 [] == []
drop (-1) [1,2] == [1,2]
drop 0 [1,2] == [1,2]
```

It is an instance of the more general `genericDrop`, in which `n` may be of any integral type.

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

`splitAt n xs` returns a tuple where first element is `xs` prefix of length `n` and second element is the remainder of the list:

```
splitAt 6 "Hello World!" == ("Hello ","World!")
splitAt 3 [1,2,3,4,5] == ([1,2,3],[4,5])
splitAt 1 [1,2,3] == ([1],[2,3])
splitAt 3 [1,2,3] == ([1,2,3],[])
splitAt 4 [1,2,3] == ([1,2,3],[])
splitAt 0 [1,2,3] == ([],[1,2,3])
splitAt (-1) [1,2,3] == ([],[1,2,3])
```

It is equivalent to `(take n xs, drop n xs)`. `splitAt` is an instance of the more general `genericSplitAt`, in which `n` may be of any integral type.

`takeWhile :: (a -> Bool) -> [a] -> [a]`

`takeWhile`, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy `p`:

```
takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
takeWhile (< 9) [1,2,3] == [1,2,3]
takeWhile (< 0) [1,2,3] == []
```

`dropWhile :: (a -> Bool) -> [a] -> [a]`

`dropWhile p xs` returns the suffix remaining after `takeWhile p xs`:

```
dropWhile (< 3) [1,2,3,4,5,1,2,3] == [3,4,5,1,2,3]
dropWhile (< 9) [1,2,3] == []
dropWhile (< 0) [1,2,3] == [1,2,3]
```

`span :: (a -> Bool) -> [a] -> ([a], [a])`

`span`, applied to a predicate `p` and a list `xs`, returns a tuple where first element is longest prefix (possibly empty) of `xs` of elements that satisfy `p` and second element is the remainder of the list:

```
span (< 3) [1,2,3,4,1,2,3,4] == ([1,2],[3,4,1,2,3,4])
span (< 9) [1,2,3] == ([1,2,3],[])
span (< 0) [1,2,3] == ([],[1,2,3])
```

`span p xs` is equivalent to `(takeWhile p xs, dropWhile p xs)`

`break :: (a -> Bool) -> [a] -> ([a], [a])`

`break`, applied to a predicate `p` and a list `xs`, returns a tuple where first element is longest prefix (possibly empty) of `xs` of elements that do not satisfy `p` and second element is the remainder of the list:

```
break (> 3) [1,2,3,4,1,2,3,4] == ([1,2,3],[4,1,2,3,4])
break (< 9) [1,2,3] == ([],[1,2,3])
break (> 9) [1,2,3] == ([1,2,3],[])
```

`break p` is equivalent to `span (not . p)`.

`lines :: String -> [String]`

`lines` breaks a string up into a list of strings at newline characters. The resulting strings do not contain newlines.

`words :: String -> [String]`

`words` breaks a string up into a list of words, which were delimited by white space.

`unlines :: [String] -> String`

`unlines` is an inverse operation to `lines`. It joins lines, after appending a terminating newline to each.

`unwords :: [String] -> String`

`unwords` is an inverse operation to `words`. It joins words with separating spaces.

`reverse :: [a] -> [a]`

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

`and :: [Bool] -> Bool`

`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 :: [Bool] -> Bool`

`or` returns the disjunction of a Boolean list. For the result to be `False`, the list must be finite; `True`, however, results from a `True` value at a finite index of a finite or infinite list.

`any :: (a -> Bool) -> [a] -> Bool`

Applied to a predicate and a list, `any` determines if any element of the list satisfies the predicate. For the result to be `False`, the list must be finite; `True`, however, results from a `True` value for the predicate applied to an element at a finite index of a finite or infinite list.

`all :: (a -> Bool) -> [a] -> Bool`

Applied to a predicate and a list, `all` determines if all elements of the list satisfy the predicate. For the result to be `True`, the list must be finite; `False`, however, results from a `False` value for the predicate applied to an element at a finite index of a finite or infinite list.

`elem :: Eq a => a -> [a] -> Bool`

`elem` is the list membership predicate, usually written in infix form, e.g., `x ‘elem‘ xs`. For the result to be `False`, the list must be finite; `True`, however, results from an element equal to `x` found at a finite index of a finite or infinite list.

`notElem :: Eq a => a -> [a] -> Bool`

`notElem` is the negation of `elem`.

`lookup :: Eq a => a -> [(a, b)] -> Maybe b`

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

`sum :: Num a => [a] -> a`

The `sum` function computes the sum of a finite list of numbers.

`product :: Num a => [a] -> a`

The `product` function computes the product of a finite list of numbers

`maximum :: Ord a => [a] -> a`

`maximum` returns the maximum value from a list, which must be non-empty, finite, and of an ordered type. It is a special case of `maximumBy`, which allows the programmer to supply their own comparison function.

`minimum :: Ord a => [a] -> a`

`minimum` returns the minimum value from a list, which must be non-empty, finite, and of an ordered type. It is a special case of `minimumBy`, which allows the programmer to supply their own comparison function.

`zip :: [a] -> [b] -> [(a, b)]`

`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 :: [a] -> [b] -> [c] -> [(a, b, c)]`

`zip3` takes three lists and returns a list of triples, analogous to `zip`.

`zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

`zipWith` generalises `zip` 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.

`zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]`

The `zipWith3` function takes a function which combines three elements, as well as three lists and returns a list of their point-wise combination, analogous to `zipWith`.

`unzip :: [(a, b)] -> ([a], [b])`

`unzip` transforms a list of pairs into a list of first components and a list of second components.

```
unzip3 ::  [(a, b, c)] -> ([a], [b], [c])
```

The `unzip3` function takes a list of triples and returns three lists, analogous to `unzip`.

35.3 Converting to and from String

35.3.1 Converting to String

```
type ShowS = String -> String
```

```
class Show a where
```

Methods

```
showsPrec ::  Int -> a -> ShowS
```

```
show ::  a -> String
```

```
showList ::  [a] -> ShowS
```

```
instance Show GeneralCategory
instance Show Int
instance Show Int16
instance Show Int32
instance Show Int64
instance Show Int8
instance Show Word
instance Show Word16
instance Show Word32
instance Show Word64
instance Show Word8
instance Show CChar
instance Show CClock
instance Show CDouble
instance Show CFloat
instance Show CInt
instance Show CIntMax
instance Show CIntPtr
instance Show CLLong
instance Show CLong
instance Show CPtrdiff
instance Show CSChar
instance Show CShort
instance Show CSigAtomic
instance Show CSize
instance Show CTime
instance Show CUChar
instance Show CUInt
instance Show CUIntMax
instance Show CUIntPtr
instance Show CULLong
instance Show CULong
instance Show CUShort
instance Show CWchar
instance Show IntPtr
instance Show WordPtr
instance Show Char
instance Show Double
instance Show Float
instance Show Integer
instance Show ExitCode
instance Show BufferMode
instance Show Handle
instance Show HandlePosn
instance Show IOMode
instance Show SeekMode
instance Show IOErrorType
instance Show ()
instance Show a => Show Complex a
instance Show a => Show Maybe a
instance Show ForeignPtr a
instance Show FunPtr a
instance Show Ptr a
instance Integral a => Show Ratio a
instance Show a => Show [a]
instance (Ix a, Show a, Show b) => Show Array a b
instance (Show a, Show b) => Show (a, b)
```

```
shows ::  Show a => a -> ShowS
```

```
showChar ::  Char -> ShowS
```

```
showString ::  String -> ShowS
```

```
showParen ::  Bool -> ShowS -> ShowS
```

35.3.2 Converting from String

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

```
class Read a where
```

Methods

```
    readsPrec ::  Int -> ReadS a
```

```
    readList ::  ReadS [a]
```

```
instance Read GeneralCategory
instance Read Int
instance Read Int16
instance Read Int32
instance Read Int64
instance Read Int8
instance Read Word
instance Read Word16
instance Read Word32
instance Read Word64
instance Read Word8
instance Read CChar
instance Read CClock
instance Read CDouble
instance Read CFloat
instance Read CInt
instance Read CIntMax
instance Read CIntPtr
instance Read CLLong
instance Read CLong
instance Read CPtrdiff
instance Read CSChar
instance Read CShort
instance Read CSigAtomic
instance Read CSize
instance Read CTime
instance Read CUChar
instance Read CUInt
instance Read CUIntMax
instance Read CUIntPtr
instance Read CULLong
instance Read CULong
instance Read CUShort
instance Read CWchar
instance Read IntPtr
instance Read WordPtr
instance Read Char
instance Read Double
instance Read Float
instance Read Integer
instance Read ExitCode
instance Read BufferMode
instance Read IOMode
instance Read SeekMode
instance Read ()
instance Read a => Read Complex a
instance Read a => Read Maybe a
instance (Integral a, Read a) => Read Ratio a
instance Read a => Read [a]
instance (Ix a, Read a, Read b) => Read Array a b
instance (Read a, Read b) => Read (a, b)
```

```
reads :: Read a => ReadS a

readParen :: Bool -> ReadS a -> ReadS a

read :: Read a => String -> a

lex :: ReadS String
```

35.4 Basic Input and output

```
data IO a
```

A value of type `IO a` is a computation which, when performed, does some I/O before returning a value of type `a`.

There is really only one way to "perform" an I/O action: bind it to `Main.main` in your program. When your program is run, the I*O will be performed. It isn't possible to perform I*O from an arbitrary function, unless that function is itself in the `IO` monad and called at some point, directly or indirectly, from `Main.main`.

`IO` is a monad, so `IO` actions can be combined using either the do-notation or the `>>` and `>>=` operations from the `Monad` class.

```
instance Functor IO
instance Monad IO
```

35.4.1 Simple I/O operations

Output functions

```
putChar :: Char -> IO ()
```

Write a character to the standard output device (same as `hPutChar stdout`).

```
putStr :: String -> IO ()
```

Write a string to the standard output device (same as `hPutStr stdout`).

```
putStrLn :: String -> IO ()
```

The same as `putStr`, but adds a newline character.

```
print :: Show a => a -> IO ()
```

The `print` function outputs a value of any printable type to the standard

output device. Printable types are those that are instances of class `Show`; `print` converts values to strings for output using the `show` operation and adds a newline.

For example, a program to print the first 20 integers and their powers of 2 could be written as:

```
main = print ([(n, 2^n) | n <- [0..19]])
```

Input functions

`getChar :: IO Char`

Read a character from the standard input device (same as `hGetChar stdin`).

`getLine :: IO String`

Read a line from the standard input device (same as `hGetLine stdin`).

`getContents :: IO String`

The `getContents` operation returns all user input as a single string, which is read lazily as it is needed (same as `hGetContents stdin`).

`interact :: (String -> String) -> IO ()`

The `interact` function takes a function of type `String->String` as its argument. The entire input from the standard input device is passed to this function as its argument, and the resulting string is output on the standard output device.

Files

`type FilePath = String`

File and directory names are values of type `String`, whose precise meaning is operating system dependent. Files can be opened, yielding a handle which can then be used to operate on the contents of that file.

`readFile :: FilePath -> IO String`

The `readFile` function reads a file and returns the contents of the file as a string. The file is read lazily, on demand, as with `getContents`.

`writeFile :: FilePath -> String -> IO ()`

The computation `writeFile file str` function writes the string `str`, to the file `file`.

`appendFile :: FilePath -> String -> IO ()`

The computation `appendFile file str` function appends the string `str`, to the file `file`. Note that `writeFile` and `appendFile` write a literal string to a file. To write a value of any printable type, as with `print`, use the `show` function to convert the value to a string first.

```
main = appendFile "squares" (show [(x,x*x) | x <- [0,0.1..2]])
```

`readIO :: Read a => String -> IO a`

The `readIO` function is similar to `read` except that it signals parse failure to the `IO` monad instead of terminating the program.

`readLn :: Read a => IO a`

The `readLn` function combines `getLine` and `readIO`.

35.4.2 Exception handling in the I/O monad

`data IOError`

Errors of type `IOError` are used by the `IO` monad. This is an abstract type; the module `System.IO.Error` provides functions to interrogate and construct values of type `IOError`.

`ioError :: IOError -> IO a`

Raise an IOError in the IO monad.

`userError :: String -> IOError`

Construct an `IOError` value with a string describing the error. The `fail` method of the `IO` instance of the `Monad` class raises a `userError`, thus:

```
instance Monad IO where
  ...
  fail s = ioError (userError s)
```

`catch :: IO a -> (IOError -> IO a) -> IO a`

The `catch` function establishes a handler that receives any `IOError` raised in the action protected by `catch`. An `IOError` is caught by the most recent handler established by `catch`. These handlers are not selective: all `IOErrors` are caught. Exception propagation must be explicitly provided in a handler by re-raising any unwanted exceptions. For example, in

```
f = catch g (\e -> if IO.isEOFError e then return [] else ioError e)
```

the function `f` returns `[]` when an end-of-file exception (cf. `isEOFError`) occurs in `g`; otherwise, the exception is propagated to the next outer handler.

When an exception propagates outside the main program, the Haskell system prints the associated `IOError` value and exits the program.

Chapter 36

System.Environment

```
module System.Environment (
    getArgs, getProgName, getEnv
  ) where
```

`getArgs :: IO [String]`

Computation `getArgs` returns a list of the program's command line arguments (not including the program name).

`getProgName :: IO String`

Computation `getProgName` returns the name of the program as it was invoked. However, this is hard-to-impossible to implement on some non-Unix OSes, so instead, for maximum portability, we just return the leaf-name of the program as invoked. Even then there are some differences between platforms: on Windows, for example, a program invoked as foo is probably really `FOO.EXE`, and that is what `getProgName` will return.

`getEnv :: String -> IO String`

Computation `getEnv var` returns the value of the environment variable `var`.

This computation may fail with:

- `isDoesNotExistError` if the environment variable does not exist.

Chapter 37

System.Exit

```
module System.Exit (
    ExitCode(ExitSuccess, ExitFailure), exitWith, exitFailure, exitSuccess
  ) where
```

```
data ExitCode
```

Defines the exit codes that a program can return.

Constructors

=	`ExitSuccess`	indicates successful termination
|	`ExitFailure Int`	indicates program failure with an exit code. The exact interpretation of the code is operating-system dependent. In particular, some values may be prohibited (e.g. 0 on a POSIX-compliant system).

```
instance Eq ExitCode
instance Ord ExitCode
instance Read ExitCode
instance Show ExitCode
```

```
exitWith ::  ExitCode -> IO a
```

Computation `exitWith code` terminates the program, returning `code` to the program's caller. The caller may interpret the return code as it

wishes, but the program should return `ExitSuccess` to mean normal completion, and `ExitFailure n` to mean that the program encountered a problem from which it could not recover. The value `exitFailure` is equal to exitWith `(ExitFailure exitfail)`, where `exitfail` is implementation-dependent. `exitWith` bypasses the error handling in the I/O monad and cannot be intercepted by `catch` from the `Prelude`.

```
exitFailure :: IO a
```

The computation `exitFailure` is equivalent to `exitWith (ExitFailure exitfail)`, where `exitfail` is implementation-dependent.

```
exitSuccess :: IO a
```

The computation `exitSuccess` is equivalent to `exitWith ExitSuccess`. It terminates the program successfully.

Chapter 38

System.IO

```
module System.IO (
    IO, fixIO, FilePath, Handle, stdin, stdout, stderr, withFile, openFile,
    IOMode(ReadMode, WriteMode, AppendMode, ReadWriteMode), hClose, readFile,
    writeFile, appendFile, hFileSize, hSetFileSize, hIsEOF, isEOF,
    BufferMode(NoBuffering, LineBuffering, BlockBuffering), hSetBuffering,
    hGetBuffering, hFlush, hGetPosn, hSetPosn, HandlePosn, hSeek,
    SeekMode(AbsoluteSeek, RelativeSeek, SeekFromEnd), hTell, hIsOpen,
    hIsClosed, hIsReadable, hIsWritable, hIsSeekable, hIsTerminalDevice,
    hSetEcho, hGetEcho, hShow, hWaitForInput, hReady, hGetChar, hGetLine,
    hLookAhead, hGetContents, hPutChar, hPutStr, hPutStrLn, hPrint, interact,
    putChar, putStr, putStrLn, print, getChar, getLine, getContents, readIO,
    readLn
  ) where
```

38.1 The IO monad

```
data IO a
```

A value of type `IO a` is a computation which, when performed, does some I/O before returning a value of type `a`.

There is really only one way to "perform" an I/O action: bind it to `Main.main` in your program. When your program is run, the I*O will be performed. It isn't possible to perform I*O from an arbitrary function,

unless that function is itself in the `IO` monad and called at some point, directly or indirectly, from `Main.main`.

`IO` is a monad, so `IO` actions can be combined using either the do-notation or the `>>` and `>>=` operations from the `Monad` class.

```
instance Functor IO
instance Monad IO
```

```
fixIO ::  (a -> IO a) -> IO a
```

38.2 Files and handles

```
type FilePath = String
```

File and directory names are values of type `String`, whose precise meaning is operating system dependent. Files can be opened, yielding a handle which can then be used to operate on the contents of that file.

```
data Handle
```

Haskell defines operations to read and write characters from and to files, represented by values of type `Handle`. Each value of this type is a *handle*: a record used by the Haskell run-time system to *manage* I/O with file system objects. A handle has at least the following properties:

- whether it manages input or output or both;
- whether it is *open*, *closed* or *semi-closed*;
- whether the object is seekable;
- whether buffering is disabled, or enabled on a line or block basis;
- a buffer (whose length may be zero).

Most handles will also have a current I/O position indicating where the next input or output operation will occur. A handle is *readable* if it manages only input or both input and output; likewise, it is *writable* if it manages only output or both input and output. A handle is *open* when first allocated. Once it is closed it can no longer be used for either input or output, though an implementation cannot re-use its storage while references remain to it. Handles are in the `Show` and `Eq` classes. The string produced by showing a handle is system dependent; it should include enough information to identify the handle for debugging. A handle is equal according to `==` only to itself; no attempt is made to compare the internal state of different handles for equality.

```
instance Eq Handle
instance Show Handle
```

38.3 Standard handles

Three handles are allocated during program initialisation, and are initially open.

`stdin :: Handle`

A handle managing input from the Haskell program's standard input channel.

`stdout :: Handle`

A handle managing output to the Haskell program's standard output channel.

`stderr :: Handle`

A handle managing output to the Haskell program's standard error channel.

38.4 Opening and closing files

38.4.1 Opening files

`withFile :: FilePath -> IOMode -> (Handle -> IO r) -> IO r`

`withFile name mode act` opens a file using `openFile` and passes the resulting handle to the computation `act`. The handle will be closed on exit from `withFile`, whether by normal termination or by raising an exception. If closing the handle raises an exception, then this exception will be raised by `withFile` rather than any exception raised by `act`.

`openFile :: FilePath -> IOMode -> IO Handle`

Computation `openFile file mode` allocates and returns a new, open handle to manage the file `file`. It manages input if `mode` is `ReadMode`, output if `mode` is `WriteMode` or `AppendMode`, and both input and output if `mode` is `ReadWriteMode`.

If the file does not exist and it is opened for output, it should be created as a new file. If `mode` is `WriteMode` and the file already exists, then it

should be truncated to zero length. Some operating systems delete empty files, so there is no guarantee that the file will exist following an `openFile` with mode `WriteMode` unless it is subsequently written to successfully. The handle is positioned at the end of the file if `mode` is `AppendMode`, and otherwise at the beginning (in which case its internal position is 0). The initial buffer mode is implementation-dependent.

This operation may fail with:

- `isAlreadyInUseError` if the file is already open and cannot be reopened;
- `isDoesNotExistError` if the file does not exist; or
- `isPermissionError` if the user does not have permission to open the file

```
data IOMode
```

See `openFile`

Constructors

```
=  ReadMode
|  WriteMode
|  AppendMode
|  ReadWriteMode
```

```
instance Enum IOMode
instance Eq IOMode
instance Ix IOMode
instance Ord IOMode
instance Read IOMode
instance Show IOMode
```

38.4.2 Closing files

```
hClose ::  Handle -> IO ()
```

Computation `hClose hdl` makes handle `hdl` closed. Before the computation finishes, if `hdl` is writable its buffer is flushed as for `hFlush`. Performing `hClose` on a handle that has already been closed has no effect; doing so is not an error. All other operations on a closed handle will fail. If `hClose` fails for any reason, any further operations (apart from `hClose`) on the handle will still fail as if `hdl` had been successfully closed.

38.4.3 Special cases

These functions are also exported by the `Prelude`.

`readFile :: FilePath -> IO String`

The `readFile` function reads a file and returns the contents of the file as a string. The file is read lazily, on demand, as with `getContents`.

`writeFile :: FilePath -> String -> IO ()`

The computation `writeFile file str` function writes the string `str`, to the file `file`.

`appendFile :: FilePath -> String -> IO ()`

The computation `appendFile file str` function appends the string `str`, to the file `file`. Note that `writeFile` and `appendFile` write a literal string to a file. To write a value of any printable type, as with `print`, use the `show` function to convert the value to a string first.

```
main = appendFile "squares" (show [(x,x*x) | x <- [0,0.1..2]])
```

38.4.4 File locking

Implementations should enforce as far as possible, at least locally to the Haskell process, multiple-reader single-writer locking on files. That is, *there may either be many handles on the same file which manage input, or just one handle on the file which manages output*. If any open or semi-closed handle is managing a file for output, no new handle can be allocated for that file. If any open or semi-closed handle is managing a file for input, new handles can only be allocated if they do not manage output. Whether two files are the same is implementation-dependent, but they should normally be the same if they have the same absolute path name and neither has been renamed, for example.

Warning: the `readFile` operation holds a semi-closed handle on the file until the entire contents of the file have been consumed. It follows that an attempt to write to a file (using `writeFile`, for example) that was earlier opened by `readFile` will usually result in failure with `isAlreadyInUseError`.

38.5 Operations on handles

38.5.1 Determining and changing the size of a file

`hFileSize :: Handle -> IO Integer`

For a handle `hdl` which attached to a physical file, `hFileSize hdl` returns the size of that file in 8-bit bytes.

```
hSetFileSize :: Handle -> Integer -> IO ()
```

`hSetFileSize hdl size` truncates the physical file with handle `hdl` to `size` bytes.

38.5.2 Detecting the end of input

```
hIsEOF :: Handle -> IO Bool
```

For a readable handle `hdl`, `hIsEOF hdl` returns `True` if no further input can be taken from `hdl` or for a physical file, if the current I/O position is equal to the length of the file. Otherwise, it returns `False`.

NOTE: `hIsEOF` may block, because it has to attempt to read from the stream to determine whether there is any more data to be read.

```
isEOF :: IO Bool
```

The computation `isEOF` is identical to `hIsEOF`, except that it works only on `stdin`.

38.5.3 Buffering operations

```
data BufferMode
```

Three kinds of buffering are supported: line-buffering, block-buffering or no-buffering. These modes have the following effects. For output, items are written out, or *flushed*, from the internal buffer according to the buffer mode:

- *line-buffering*: the entire output buffer is flushed whenever a newline is output, the buffer over- flows, a `hFlush` is issued, or the handle is closed.
- *block-buffering*: the entire buffer is written out whenever it overflows, a `hFlush` is issued, or the handle is closed.
- *no-buffering*: output is written immediately, and never stored in the buffer.

An implementation is free to flush the buffer more frequently, but not less frequently, than specified above. The output buffer is emptied as soon as it has been written out.

Similarly, input occurs according to the buffer mode for the handle:

- *line-buffering*: when the buffer for the handle is not empty, the next item is obtained from the buffer; otherwise, when the buffer is empty, characters up to and including the next newline character are read into the buffer. No characters are available until the newline character is available or the buffer is full.
- *block-buffering*: when the buffer for the handle becomes empty, the next block of data is read into the buffer.
- *no-buffering*: the next input item is read and returned. The `hLookAhead` operation implies that even a no-buffered handle may require a one-character buffer.

The default buffering mode when a handle is opened is implementation-dependent and may depend on the file system object which is attached to that handle. For most implementations, physical files will normally be block-buffered and terminals will normally be line-buffered.

Constructors

=	`NoBuffering`	buffering is disabled if possible.
|	`LineBuffering`	line-buffering should be enabled if possible.
|	`BlockBuffering Maybe Int`	block-buffering should be enabled if possible. The size of the buffer is `n` items if the argument is `Just n` and is otherwise implementation-dependent.

```
instance Eq BufferMode
instance Ord BufferMode
instance Read BufferMode
instance Show BufferMode
```

```
hSetBuffering ::  Handle -> BufferMode -> IO ()
```

Computation `hSetBuffering hdl mode` sets the mode of buffering for handle `hdl` on subsequent reads and writes.

If the buffer mode is changed from `BlockBuffering` or `LineBuffering` to `NoBuffering`, then

- if `hdl` is writable, the buffer is flushed as for `hFlush`;
- if `hdl` is not writable, the contents of the buffer is discarded.

This operation may fail with:

- `isPermissionError` if the handle has already been used for reading or writing and the implementation does not allow the buffering mode to be changed.

```
hGetBuffering :: Handle -> IO BufferMode
```

Computation `hGetBuffering hdl` returns the current buffering mode for `hdl`.

```
hFlush :: Handle -> IO ()
```

The action hFlush hdl causes any items buffered for output in handle hdl to be sent immediately to the operating system.

This operation may fail with:

- `isFullError` if the device is full;
- `isPermissionError` if a system resource limit would be exceeded. It is unspecified whether the characters in the buffer are discarded or retained under these circumstances.

38.5.4 Repositioning handles

```
hGetPosn :: Handle -> IO HandlePosn
```

Computation `hGetPosn hdl` returns the current I/O position of `hdl` as a value of the abstract type `HandlePosn`.

```
hSetPosn :: HandlePosn -> IO ()
```

If a call to `hGetPosn hdl` returns a position `p`, then computation `hSetPosn p` sets the position of `hdl` to the position it held at the time of the call to `hGetPosn`.

This operation may fail with:

- `isPermissionError` if a system resource limit would be exceeded.

```
data HandlePosn
```

```
instance Eq HandlePosn
instance Show HandlePosn
```

```
hSeek :: Handle -> SeekMode -> Integer -> IO ()
```

Computation `hSeek hdl mode i` sets the position of handle `hdl` depending on `mode`. The offset `i` is given in terms of 8-bit bytes.

If `hdl` is block- or line-buffered, then seeking to a position which is not in the current buffer will first cause any items in the output buffer to be

written to the device, and then cause the input buffer to be discarded. Some handles may not be seekable (see `hIsSeekable`), or only support a subset of the possible positioning operations (for instance, it may only be possible to seek to the end of a tape, or to a positive offset from the beginning or current position). It is not possible to set a negative I/O position, or for a physical file, an I/O position beyond the current end-of-file.

This operation may fail with:

- `isIllegalOperationError` if the Handle is not seekable, or does not support the requested seek mode.
- `isPermissionError` if a system resource limit would be exceeded.

```
data SeekMode
```

A mode that determines the effect of `hSeek hdl mode i`.

Constructors

=	`AbsoluteSeek`	the position of `hdl` is set to `i`.
|	`RelativeSeek`	the position of `hdl` is set to offset `i` from the current position.
|	`SeekFromEnd`	the position of `hdl` is set to offset `i` from the end of the file.

```
instance Enum SeekMode
instance Eq SeekMode
instance Ix SeekMode
instance Ord SeekMode
instance Read SeekMode
instance Show SeekMode
```

```
hTell ::  Handle -> IO Integer
```

Computation `hTell hdl` returns the current position of the handle `hdl`, as the number of bytes from the beginning of the file. The value returned may be subsequently passed to `hSeek` to reposition the handle to the current position.

This operation may fail with:

- `isIllegalOperationError` if the Handle is not seekable.

38.5.5 Handle properties

Each of these operations returns `True` if the handle has the the specified property, or `False` otherwise.

```
hIsOpen ::  Handle -> IO Bool

hIsClosed ::  Handle -> IO Bool

hIsReadable ::  Handle -> IO Bool

hIsWritable ::  Handle -> IO Bool

hIsSeekable ::  Handle -> IO Bool
```

38.5.6 Terminal operations

```
hIsTerminalDevice ::  Handle -> IO Bool
```

Is the handle connected to a terminal?

```
hSetEcho ::  Handle -> Bool -> IO ()
```

Set the echoing status of a handle connected to a terminal.

```
hGetEcho ::  Handle -> IO Bool
```

Get the echoing status of a handle connected to a terminal.

38.5.7 Showing handle state

```
hShow ::  Handle -> IO String
```

`hShow` is in the `IO` monad, and gives more comprehensive output than the (pure) instance of `Show` for `Handle`.

38.6 Text input and output

38.6.1 Text input

```
hWaitForInput ::  Handle -> Int -> IO Bool
```

Computation `hWaitForInput hdl t` waits until input is available on handle `hdl`. It returns `True` as soon as input is available on `hdl`, or `False` if no input is available within `t` milliseconds. Note that `hWaitForInput` waits until one or more full characters are available, which means that it needs to do decoding, and hence may fail with a decoding error.

If `t` is less than zero, then `hWaitForInput` waits indefinitely.

This operation may fail with:

- `isEOFError` if the end of file has been reached.
- a decoding error, if the input begins with an invalid byte sequence in this Handle's encoding.

```
hReady ::  Handle -> IO Bool
```

Computation `hReady hdl` indicates whether at least one item is available for input from handle `hdl`.

This operation may fail with:

- `isEOFError` if the end of file has been reached.

```
hGetChar ::  Handle -> IO Char
```

Computation `hGetChar hdl` reads a character from the file or channel managed by `hdl`, blocking until a character is available.

This operation may fail with:

- `isEOFError` if the end of file has been reached.

```
hGetLine ::  Handle -> IO String
```

Computation `hGetLine hdl` reads a line from the file or channel managed by `hdl`.

This operation may fail with:

- `isEOFError` if the end of file is encountered when reading the first character of the line.

If `hGetLine` encounters end-of-file at any other point while reading in a line, it is treated as a line terminator and the (partial) line is returned.

```
hLookAhead ::  Handle -> IO Char
```

Computation `hLookAhead` returns the next character from the handle without removing it from the input buffer, blocking until a character is available.

This operation may fail with:

- `isEOFError` if the end of file has been reached.

```
hGetContents ::  Handle -> IO String
```

Computation `hGetContents hdl` returns the list of characters corresponding to the unread portion of the channel or file managed by `hdl`, which is put into an intermediate state, *semi-closed*. In this state, `hdl` is effectively closed, but items are read from `hdl` on demand and accumulated in a special list returned by `hGetContents hdl`.

Any operation that fails because a handle is closed, also fails if a handle is semi-closed. The only exception is `hClose`. A semi-closed handle becomes closed:

- if `hClose` is applied to it;
- if an I/O error occurs when reading an item from the handle;
- or once the entire contents of the handle has been read.

Once a semi-closed handle becomes closed, the contents of the associated list becomes fixed. The contents of this final list is only partially specified: it will contain at least all the items of the stream that were evaluated prior to the handle becoming closed.

Any I/O errors encountered while a handle is semi-closed are simply discarded.

This operation may fail with:

- `isEOFError` if the end of file has been reached.

38.6.2 Text output

```
hPutChar :: Handle -> Char -> IO ()
```

Computation `hPutChar hdl ch` writes the character `ch` to the file or channel managed by `hdl`. Characters may be buffered if buffering is enabled for `hdl`.

This operation may fail with:

- `isFullError` if the device is full; or
- `isPermissionError` if another system resource limit would be exceeded.

```
hPutStr :: Handle -> String -> IO ()
```

Computation `hPutStr hdl s` writes the string `s` to the file or channel managed by `hdl`.

This operation may fail with:

- `isFullError` if the device is full; or
- `isPermissionError` if another system resource limit would be exceeded.

```
hPutStrLn :: Handle -> String -> IO ()
```

The same as `hPutStr`, but adds a newline character.

```
hPrint :: Show a => Handle -> a -> IO ()
```

Computation `hPrint hdl t` writes the string representation of `t` given by the `shows` function to the file or channel managed by `hdl` and appends a newline.

This operation may fail with:

- `isFullError` if the device is full; or
- `isPermissionError` if another system resource limit would be exceeded

38.6.3 Special cases for standard input and output

These functions are also exported by the `Prelude`.

```
interact :: (String -> String) -> IO ()
```

The `interact` function takes a function of type `String->String` as its argument. The entire input from the standard input device is passed to this function as its argument, and the resulting string is output on the standard output device.

```
putChar :: Char -> IO ()
```

Write a character to the standard output device (same as `hPutChar stdout`).

```
putStr :: String -> IO ()
```

Write a string to the standard output device (same as `hPutStr stdout`).

```
putStrLn :: String -> IO ()
```

The same as `putStr`, but adds a newline character.

`print :: Show a => a -> IO ()`

The `print` function outputs a value of any printable type to the standard output device. Printable types are those that are instances of class `Show`; `print` converts values to strings for output using the `show` operation and adds a newline.

For example, a program to print the first 20 integers and their powers of 2 could be written as:

```
main = print ([(n, 2^n) | n <- [0..19]])
```

`getChar :: IO Char`

Read a character from the standard input device (same as `hGetChar stdin`).

`getLine :: IO String`

Read a line from the standard input device (same as `hGetLine stdin`).

`getContents :: IO String`

The `getContents` operation returns all user input as a single string, which is read lazily as it is needed (same as `hGetContents stdin`).

`readIO :: Read a => String -> IO a`

The `readIO` function is similar to `read` except that it signals parse failure to the `IO` monad instead of terminating the program.

`readLn :: Read a => IO a`

The `readLn` function combines `getLine` and `readIO`.

Chapter 39

System.IO.Error

```
module System.IO.Error (
    IOError, userError, mkIOError, annotateIOError, isAlreadyExistsError,
    isDoesNotExistError, isAlreadyInUseError, isFullError, isEOFError,
    isIllegalOperation, isPermissionError, isUserError, ioeGetErrorString,
    ioeGetHandle, ioeGetFileName, IOErrorType, alreadyExistsErrorType,
    doesNotExistErrorType, alreadyInUseErrorType, fullErrorType, eofErrorType,
    illegalOperationErrorType, permissionErrorType, userErrorType, ioError,
    catch, try
  ) where
```

39.1 I/O errors

```
data IOError
```

Errors of type `IOError` are used by the `IO` monad. This is an abstract type; the module `System.IO.Error` provides functions to interrogate and construct values of type `IOError`.

```
userError ::  String -> IOError
```

Construct an `IOError` value with a string describing the error. The `fail` method of the `IO` instance of the `Monad` class raises a `userError`, thus:

```
instance Monad IO where
  ...
  fail s = ioError (userError s)
```

```
mkIOError :: IOErrorType -> String -> Maybe Handle -> Maybe FilePath -> IOError
```

Construct an `IOError` of the given type where the second argument describes the error location and the third and fourth argument contain the file handle and file path of the file involved in the error if applicable.

```
annotateIOError :: IOError -> String -> Maybe Handle -> Maybe FilePath -> IOError
```

Adds a location description and maybe a file path and file handle to an `IOError`. If any of the file handle or file path is not given the corresponding value in the `IOError` remains unaltered.

39.2 Classifying I/O errors

```
isAlreadyExistsError :: IOError -> Bool
```

An error indicating that an `IO` operation failed because one of its arguments already exists.

```
isDoesNotExistError :: IOError -> Bool
```

An error indicating that an `IO` operation failed because one of its arguments does not exist.

```
isAlreadyInUseError :: IOError -> Bool
```

An error indicating that an `IO` operation failed because one of its arguments is a single-use resource, which is already being used (for example, opening the same file twice for writing might give this error).

```
isFullError :: IOError -> Bool
```

An error indicating that an `IO` operation failed because the device is full.

```
isEOFError :: IOError -> Bool
```

An error indicating that an `IO` operation failed because the end of file has been reached.

```
isIllegalOperation ::  IOError -> Bool
```

An error indicating that an `IO` operation failed because the operation was not possible. Any computation which returns an `IO` result may fail with `isIllegalOperation`. In some cases, an implementation will not be able to distinguish between the possible error causes. In this case it should fail with `isIllegalOperation`.

```
isPermissionError ::  IOError -> Bool
```

An error indicating that an `IO` operation failed because the user does not have sufficient operating system privilege to perform that operation.

```
isUserError ::  IOError -> Bool
```

A programmer-defined error value constructed using `userError`.

39.3 Attributes of I/O errors

```
ioeGetErrorString ::  IOError -> String
```

```
ioeGetHandle ::  IOError -> Maybe Handle
```

```
ioeGetFileName ::  IOError -> Maybe FilePath
```

39.4 Types of I/O error

```
data IOErrorType
```

An abstract type that contains a value for each variant of `IOError`.

```
instance Eq IOErrorType
instance Show IOErrorType
```

```
alreadyExistsErrorType ::  IOErrorType
```

I/O error where the operation failed because one of its arguments already exists.

```
doesNotExistErrorType ::  IOErrorType
```

I/O error where the operation failed because one of its arguments does not exist.

`alreadyInUseErrorType :: IOErrorType`

I/O error where the operation failed because one of its arguments is a single-use resource, which is already being used.

`fullErrorType :: IOErrorType`

I/O error where the operation failed because the device is full.

`eofErrorType :: IOErrorType`

I/O error where the operation failed because the end of file has been reached.

`illegalOperationErrorType :: IOErrorType`

I/O error where the operation is not possible.

`permissionErrorType :: IOErrorType`

I/O error where the operation failed because the user does not have sufficient operating system privilege to perform that operation.

`userErrorType :: IOErrorType`

I/O error that is programmer-defined.

39.5 Throwing and catching I/O errors

`ioError :: IOError -> IO a`

Raise an IOError in the IO monad.

`catch :: IO a -> (IOError -> IO a) -> IO a`

The `catch` function establishes a handler that receives any `IOError` raised in the action protected by `catch`. An `IOError` is caught by the most recent handler established by `catch`. These handlers are not selective: all `IOErrors` are caught. Exception propagation must be explicitly provided in a handler by re-raising any unwanted exceptions. For example, in

```
f = catch g (\e -> if IO.isEOFError e then return [] else ioError e)
```

the function `f` returns `[]` when an end-of-file exception (cf. `isEOFError`) occurs in `g`; otherwise, the exception is propagated to the next outer handler.

When an exception propagates outside the main program, the Haskell system prints the associated `IOError` value and exits the program.

`try :: IO a -> IO (Either IOError a)`

The construct `try comp` exposes IO errors which occur within a computation, and which are not fully handled.

Chapter 40

Text.Read

```
module Text.Read (
    ReadS, Read(readList, readsPrec), read, reads, readParen, lex
  ) where
```

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

```
class Read a where
```

Methods

```
readsPrec ::  Int -> ReadS a
```

```
readList ::  ReadS [a]
```

```
instance Read GeneralCategory
instance Read Int
instance Read Int16
instance Read Int32
instance Read Int64
instance Read Int8
instance Read Word
instance Read Word16
instance Read Word32
instance Read Word64
instance Read Word8
instance Read CChar
instance Read CClock
instance Read CDouble
instance Read CFloat
instance Read CInt
instance Read CIntMax
instance Read CIntPtr
instance Read CLLong
instance Read CLong
instance Read CPtrdiff
instance Read CSChar
instance Read CShort
instance Read CSigAtomic
instance Read CSize
instance Read CTime
instance Read CUChar
instance Read CUInt
instance Read CUIntMax
instance Read CUIntPtr
instance Read CULLong
instance Read CULong
instance Read CUShort
instance Read CWchar
instance Read IntPtr
instance Read WordPtr
instance Read Char
instance Read Double
instance Read Float
instance Read Integer
instance Read ExitCode
instance Read BufferMode
instance Read IOMode
instance Read SeekMode
instance Read ()
instance Read a => Read Complex a
instance Read a => Read Maybe a
instance (Integral a, Read a) => Read Ratio a
instance Read a => Read [a]
instance (Ix a, Read a, Read b) => Read Array a b
instance (Read a, Read b) => Read (a, b)
```

```
read ::  Read a => String -> a

reads ::  Read a => ReadS a

readParen ::  Bool -> ReadS a -> ReadS a

lex ::  ReadS String
```

Chapter 41

Text.Show

```
module Text.Show (
    ShowS, Show(show, showList, showsPrec), shows, showChar, showString,
    showParen
  ) where
```

```
type ShowS = String -> String
```

```
class Show a where
```

Methods

```
showsPrec ::  Int -> a -> ShowS
```

```
show ::  a -> String
```

```
showList ::  [a] -> ShowS
```

```
instance Show GeneralCategory
instance Show Int
instance Show Int16
instance Show Int32
instance Show Int64
instance Show Int8
instance Show Word
instance Show Word16
instance Show Word32
instance Show Word64
instance Show Word8
instance Show CChar
instance Show CClock
instance Show CDouble
instance Show CFloat
instance Show CInt
instance Show CIntMax
instance Show CIntPtr
instance Show CLLong
instance Show CLong
instance Show CPtrdiff
instance Show CSChar
instance Show CShort
instance Show CSigAtomic
instance Show CSize
instance Show CTime
instance Show CUChar
instance Show CUInt
instance Show CUIntMax
instance Show CUIntPtr
instance Show CULLong
instance Show CULong
instance Show CUShort
instance Show CWchar
instance Show IntPtr
instance Show WordPtr
instance Show Char
instance Show Double
instance Show Float
instance Show Integer
instance Show ExitCode
instance Show BufferMode
instance Show Handle
instance Show HandlePosn
instance Show IOMode
instance Show SeekMode
instance Show IOErrorType
instance Show ()
instance Show a => Show Complex a
instance Show a => Show Maybe a
instance Show ForeignPtr a
instance Show FunPtr a
instance Show Ptr a
instance Integral a => Show Ratio a
instance Show a => Show [a]
instance (Ix a, Show a, Show b) => Show Array a b
instance (Show a, Show b) => Show (a, b)
```

```
shows ::  Show a => a -> ShowS

showChar ::  Char -> ShowS

showString ::  String -> ShowS

showParen ::  Bool -> ShowS -> ShowS
```