A tour of the Haskell Prelude

Bernie Pope, 2001
bjpop@cs.mu.oz.au

1 Haskell

The Haskell language was conceived during a meeting held at the 1987 Functional Programming and Computer Architecture conference (FPCA 87). At the time of the conference it was believed that the advancement of functional programming was being stifled by the wide variety of languages available. There were more than a dozen lazy, purely functional languages in existence and none had widespread support (except perhaps Miranda*). A committee was formed to design the language. The name *Haskell* was chosen in honour of the mathematician *Haskell Curry*, whose research forms part of the theoretical basis upon which many functional languages are implemented. Haskell is widely used within the functional programming community, and there exists a number of implementations. In 1998 the Haskell community agreed upon a standard definition of the language and supporting libraries. One of the aims of standardisation was to encourage the creation of text books and courses devoted to the language. The resulting language definition is called *Haskell 98*.

Haskell is a lazy functional language with polymorphic higher-order functions, algebraic data types and list comprehensions. It has an extensive module system, and supports ad-hoc polymorphism (via classes). Haskell is *purely functional*, even for I/O. Most Haskell implementations come with a number of libraries supporting arrays, complex numbers, infinite precision integers, operating system interaction, concurrency and mutable data structures. There is a popular interpreter (called Hugs) and many compilers. More information about the Haskell language can be found on following the web-page: `www.haskell.org`.

Hugs† is a freely available interpreter for Haskell, which runs under Unix, Macintosh, and Microsoft Windows. One of the main features of Hugs is that it provides an interactive programming environment which allows the programmer to edit scripts, and evaluate arbitrary Haskell expressions. Hugs is based significantly on Mark Jones' Gofer interpreter. More information about the Hugs interpreter can be found on the following web-page: `www.haskell.org/hugs`.

The following chapter serves as a reference guide to the Haskell language (specifically Haskell 98). In particular it concentrates on the content of the Haskell Prelude, which is a standard library accessible by all Haskell programs. The chapter does not give complete coverage to the whole Prelude, but instead concentrates on those aspects most useful to Haskell beginners (however it should serve as a valuable resource to experienced Haskell programmers as well). The first part of the chapter deals with Prelude functions, the second part of the chapter deals with Prelude operators, and the third part of the deals with Prelude classes.

*Miranda is a trademark of Research Software, Ltd.

†Haskell Users' Gofer System

1.1 Functions from the Haskell Prelude

abs

type: `abs :: Num a => a -> a`

description: returns the absolute value of a number.

definition:

```
abs x
  | x >= 0 = x
  | otherwise = -x
```

usage:

```
Prelude> abs (-3)
3
```

all

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

description: applied to a predicate and a list, returns `True` if all elements of the list satisfy the predicate, and `False` otherwise. Similar to the function `any`.

definition: `all p xs = and (map p xs)`

usage:

```
Prelude> all (<11) [1..10]
True
Prelude> all isDigit "123abc"
False
```

and

type: `and :: [Bool] -> Bool`

description: takes the logical conjunction of a list of boolean values (see also 'or').

definition: `and xs = foldr (&&) True xs`

usage:

```
Prelude> and [True, True, False, True]
False
Prelude> and [True, True, True, True]
True
Prelude> and []
True
```

any

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

description: applied to a predicate and a list, returns `True` if any of the elements of the list satisfy the predicate, and `False` otherwise. Similar to the function `all`.

definition: `any p xs = or (map p xs)`

usage:

```
Prelude> any (<11) [1..10]
True
Prelude> any isDigit "123abc"
True
Prelude> any isDigit "alphabetics"
False
```

atan

type: `atan :: Floating a => a -> a`

description: the trigonometric function inverse tan.

definition: defined internally.

usage:
```
Prelude> atan pi
1.26263
```

break

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

description: given a predicate and a list, breaks the list into two lists (returned as a tuple) at the point where the predicate is first satisfied. If the predicate is never satisfied then the first element of the resulting tuple is the entire list and the second element is the empty list (`[]`).

definition:
```
break p xs
  = span p' xs
    where
    p' x = not (p x)
```

usage:
```
Prelude> break isSpace "hello there fred"
("hello", " there fred")
Prelude> break isDigit "no digits here"
("no digits here","")
```

ceiling

type: `ceiling :: (RealFrac a, Integral b) => a -> b`

description: returns the smallest integer not less than its argument.

usage:
```
Prelude> ceiling 3.8
4
Prelude> ceiling (-3.8)
-3
```

note: the function `floor` has a related use to `ceiling`.

chr

type: `chr :: Int -> Char`

description: applied to an integer in the range 0 – 255, returns the character whose ascii code is that integer. It is the converse of the function `ord`. An error will result if `chr` is applied to an integer outside the correct range.

definition: defined internally.

usage:
```
Prelude> chr 65
'A'
Prelude> (ord (chr 65)) == 65
True
```

concat

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

description: applied to a list of lists, joins them together using the `++` operator.

definition: `concat xs = foldr (++) [] xs`

usage:

```
Prelude> concat [[1,2,3], [4], [], [5,6,7,8]]
[1, 2, 3, 4, 5, 6, 7, 8]
```

cos

type: `cos :: Floating a => a -> a`

description: the trigonometric cosine function, arguments are interpreted to be in radians.

definition: defined internally.

usage:

```
Prelude> cos pi
-1.0
Prelude> cos (pi/2)
-4.37114e-08
```

digitToInt

type: `digitToInt :: Char -> Int`

description: converts a digit character into the corresponding integer value of the digit.

definition:

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

usage:

```
Prelude> digitToInt '3'
3
```

div

type: `div :: Integral a => a -> a -> a`

description: computes the integer division of its integral arguments.

definition: defined internally.

usage:

```
Prelude> 16 `div` 9
1
```

doReadFile

type: `doReadFile :: String -> String`

description: given a filename as a string, returns the contents of the file as a string. Returns an error if the file cannot be opened or found.

definition: defined internally.

usage:

```
Prelude> doReadFile "foo.txt"
"This is a small text file,\ncalled foo.txt.\n"
```

note: This is *not* a standard Haskell function. You must `import` the `MULib.hs` module to use this function.

drop

type:

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

description: applied to a number and a list, returns the list with the specified number of elements removed from the front of the list. If the list has less than the required number of elements then it returns `[]`.

definition:

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

usage:

```
Prelude> drop 3 [1..10]
[4, 5, 6, 7, 8, 9, 10]
Prelude> drop 4 "abc"
""
```

dropWhile

type:

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

description: applied to a predicate and a list, removes elements from the front of the list while the predicate is satisfied.

definition:

```
dropWhile p [] = []
dropWhile p (x:xs)
  | p x = dropWhile p xs
  | otherwise = (x:xs)
```

usage:

```
Prelude> dropWhile (<5) [1..10]
[5, 6, 7, 8, 9, 10]
```

elem

type:

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

description: applied to a value and a list returns `True` if the value is in the list and `False` otherwise. The elements of the list must be of the same type as the value.

definition:

```
elem x xs = any (== x) xs
```

usage:

```
Prelude> elem 5 [1..10]
True
Prelude> elem "rat" ["fat", "cat", "sat", "flat"]
False
```

error

type:

```
error :: String -> a
```

description: applied to a string creates an error value with an associated message. Error values are equivalent to the undefined value (`undefined`), any attempt to access the value causes the program to terminate and print the string as a diagnostic.

definition: defined internally.

usage: `error "this is an error message"`

exp

type: `exp :: Floating a => a -> a`

description: the exponential function (`exp n` is equivalent to e^n).

definition: defined internally.

usage:
```
Prelude> exp 1
2.71828
```

filter

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

description: applied to a predicate and a list, returns a list containing all the elements from the argument list that satisfy the predicate.

definition: `filter p xs = [k | k <- xs, p k]`

usage:
```
Prelude> filter isDigit "fat123cat456"
"123456"
```

flip

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

description: applied to a binary function, returns the same function with the order of the arguments reversed.

definition: `flip f x y = f y x`

usage:
```
Prelude> flip elem [1..10] 5
True
```

floor

type: `floor :: (RealFrac a, Integral b) => a -> b`

description: returns the largest integer not greater than its argument.

usage:
```
Prelude> floor 3.8
3
Prelude> floor (-3.8)
-4
```

note: the function `ceiling` has a related use to `floor`.

foldl

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

description: folds up a list, using a given binary operator and a given start value, in a left associative manner.

foldl op r [a, b, c] → ((r ‘op‘ a) ‘op‘ b) ‘op‘ c

definition:
```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

usage:
```
Prelude> foldl (+) 0 [1..10]
55
Prelude> foldl (flip (:)) [] [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

foldl1

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

description: folds left over non–empty lists.

definition:
```
foldl1 f (x:xs) = foldl f x xs
```

usage:
```
Prelude> foldl1 max [1, 10, 5, 2, -1]
10
```

foldr

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

description: folds up a list, using a given binary operator and a given start value, in a right associative manner.

foldr op r [a, b, c] → a ‘op‘ (b ‘op‘ (c ‘op‘ r))

definition:
```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

usage:
```
Prelude> foldr (++) [] ["con", "cat", "en", "ate"]
"concatenate"
```

foldr1

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

description: folds right over non–empty lists.

definition:
```
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

usage:
```
Prelude> foldr1 (*) [1..10]
3628800
```

fromInt

type: `fromInt :: Num a => Int -> a`

description: Converts from an `Int` to a numeric type which is in the class `Num`.

usage:
```
Prelude> (fromInt 3)::Float
3.0
```

fromInteger

type: `fromInteger :: Num a => Integer -> a`

description: Converts from an `Integer` to a numeric type which is in the class `Num`.

usage:
```
Prelude> (fromInteger 10000000000)::Float
1.0e+10
```

fst

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

description: returns the first element of a two element tuple.

definition:
```
fst (x, _) = x
```

usage:
```
Prelude> fst ("harry", 3)
"harry"
```

head

type: `head :: [a] -> a`

description: returns the first element of a non–empty list. If applied to an empty list an error results.

definition:
```
head (x:_) = x
```

usage:
```
Prelude> head [1..10]
1
Prelude> head ["this", "and", "that"]
"this"
```

id

type: `id :: a -> a`

description: the identity function, returns the value of its argument.

definition:
```
id x = x
```

usage:
```
Prelude> id 12
12
Prelude> id (id "fred")
"fred"
Prelude> (map id [1..10]) == [1..10]
True
```

init

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

description: returns all but the last element of its argument list. The argument list must have at least one element. If `init` is applied to an empty list an error occurs.

definition:
```
init [x] = []
init (x:xs) = x : init xs
```

usage:
```
Prelude> init [1..10]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

isAlpha

type: `isAlpha :: Char -> Bool`

description: applied to a character argument, returns `True` if the character is alphabetic, and `False` otherwise.

definition:
```
isAlpha c = isUpper c || isLower c
```

usage:
```
Prelude> isAlpha 'a'
True
Prelude> isAlpha '1'
False
```

isDigit

type: `isDigit :: Char -> Bool`

description: applied to a character argument, returns `True` if the character is a numeral, and `False` otherwise.

definition:
```
isDigit c = c >= '0' && c <= '9'
```

usage:
```
Prelude> isDigit '1'
True
Prelude> isDigit 'a'
False
```

isLower

type: `isLower :: Char -> Bool`

description: applied to a character argument, returns `True` if the character is a lower case alphabetic, and `False` otherwise.

definition:
```
isLower c = c >= 'a' && c <= 'z'
```

usage:
```
Prelude> isLower 'a'
True
Prelude> isLower 'A'
False
Prelude> isLower '1'
False
```

isSpace

type: `isSpace :: Char -> Bool`

description: returns `True` if its character argument is a whitespace character and `False` otherwise.

definition:
```
isSpace c  = c == ' '  || c == '\t' || c == '\n' ||
             c == '\r' || c == '\f' || c == '\v'
```

usage:
```
Prelude> dropWhile isSpace "   \nhello  \n"
"hello  \n"
```

isUpper

type: `isUpper :: Char -> Bool`

description: applied to a character argument, returns `True` if the character is an upper case alphabetic, and `False` otherwise.

definition:
```
isDigit c = c >= 'A' && c <= 'Z'
```

usage:
```
Prelude> isUpper 'A'
True
Prelude> isUpper 'a'
False
Prelude> isUpper '1'
False
```

iterate

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

description: `iterate f x` returns the infinite list `[x, f(x), f(f(x)), ...]`.

definition:
```
iterate f x = x : iterate f (f x)
```

usage:
```
Prelude> iterate (+1) 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, .....
```

last

type:
```
last :: [a] -> a
```

description: applied to a non–empty list, returns the last element of the list.

definition:
```
last [x] = x
last (_:xs) = last xs
```

usage:
```
Prelude> last [1..10]
10
```

length

type:
```
length :: [a] -> Int
```

description: returns the number of elements in a finite list.

definition:
```
length [] = 0
length (x:xs) = 1 + length xs
```

usage:
```
Prelude> length [1..10]
10
```

lines

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

description: applied to a list of characters containing newlines, returns a list of lists by breaking the original list into lines using the newline character as a delimiter. The newline characters are removed from the result.

definition:
```
lines [] = []
lines (x:xs)
  = l : ls
  where
  (l, xs') = break (== '\n') (x:xs)
  ls
    | xs' == [] = []
    | otherwise = lines (tail xs')
```

usage:
```
Prelude> lines "hello world\nit's me,\neric\n"
["hello world", "it's me,", "eric"]
```

log

type: `log :: Floating a => a -> a`

description: returns the natural logarithm of its argument.

definition: defined internally.

usage:
```
Prelude> log 1
0.0
Prelude> log 3.2
1.16315
```

map

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

description: given a function, and a list of any type, returns a list where each element is the result of applying the function to the corresponding element in the input list.

definition: `map f xs = [f x | x <- xs]`

usage:
```
Prelude> map sqrt [1..5]
[1.0, 1.41421, 1.73205, 2.0, 2.23607]
```

max

type: `max :: Ord a => a -> a -> a`

description: applied to two values of the same type which have an ordering defined upon them, returns the maximum of the two elements according to the operator `>=`.

definition:
```
max x y
  | x >= y = x
  | otherwise = y
```

usage:
```
Prelude> max 1 2
2
```

maximum

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

description: applied to a non–empty list whose elements have an ordering defined upon them, returns the maximum element of the list.

definition: `maximum xs = foldl1 max xs`

usage:
```
Prelude> maximum [-10, 0 , 5, 22, 13]
22
```

min

type: `min :: Ord a => a -> a -> a`

description: applied to two values of the same type which have an ordering defined upon them, returns the minimum of the two elements according to the operator `<=`.

definition:

```
min x y
  | x <= y = x
  | otherwise = y
```

usage:

```
Prelude> min 1 2
1
```

minimum

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

description: applied to a non–empty list whose elements have an ordering defined upon them, returns the minimum element of the list.

definition: `minimum xs = foldl1 min xs`

usage:

```
Prelude> minimum [-10, 0 , 5, 22, 13]
-10
```

mod

type: `mod :: Integral a => a -> a -> a`

description: returns the modulus of its two arguments.

definition: defined internally.

usage:

```
Prelude> 16 `mod` 9
7
```

not

type: `not :: Bool -> Bool`

description: returns the logical negation of its boolean argument.

definition:

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

usage:

```
Prelude> not (3 == 4)
True
Prelude> not (10 > 2)
False
```

or

type: `or :: [Bool] -> Bool`

description: applied to a list of boolean values, returns their logical disjunction (see also 'and').

definition: `or xs = foldr (||) False xs`

usage:

```
Prelude> or [False, False, True, False]
True
Prelude> or [False, False, False, False]
False
Prelude> or []
False
```

ord

type: `ord :: Char -> Int`

description: applied to a character, returns its ascii code as an integer.

definition: defined internally.

usage:

```
Prelude> ord 'A'
65
Prelude> (chr (ord 'A')) == 'A'
True
```

pi

type: `pi :: Floating a => a`

description: the ratio of the circumference of a circle to its diameter.

definition: defined internally.

usage:

```
Prelude> pi
3.14159
Prelude> cos pi
-1.0
```

putStr

type: `putStr :: String -> IO ()`

description: takes a string as an argument and returns an I/O action as a result. A side-effect of applying `putStr` is that it causes its argument string to be printed to the screen.

definition: defined internally.

usage:

```
Prelude> putStr "Hello World\nI'm here!"
Hello World
I'm here!
```

product

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

description: applied to a list of numbers, returns their product.

definition: `product xs = foldl (*) 1 xs`

usage:

```
Prelude> product [1..10]
3628800
```

repeat

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

description: given a value, returns an infinite list of elements the same as the value.

definition:

```
repeat x
  = xs
  where xs = x:xs
```

usage:

```
Prelude> repeat 12
[12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12 ....
```

replicate

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

description: given an integer (positive or zero) and a value, returns a list containing the specified number of instances of that value.

definition: `replicate n x = take n (repeat x)`

usage:
```
Prelude> replicate 3 "apples"
["apples", "apples", "apples"]
```

reverse

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

description: applied to a finite list of any type, returns a list of the same elements in reverse order.

definition: `reverse = foldl (flip (:)) []`

usage:
```
Prelude> reverse [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

round

type: `round :: (RealFrac a, Integral b) => a -> b`

description: rounds its argument to the nearest integer.

usage:
```
Prelude> round 3.2
3
Prelude> round 3.5
4
Prelude> round (-3.2)
-3
```

show

type: `show :: Show a => a -> String`

description: converts a value (which must be a member of the `Show` class), to its string representation.

definition: defined internally.

usage:
```
Prelude> "six plus two equals " ++ (show (6 + 2))
"six plus two equals 8"
```

sin

type: `sin :: Floating a => a -> a`

description: the trigonometric sine function, arguments are interpreted to be in radians.

definition: defined internally.

usage:
```
Prelude> sin (pi/2)
1.0
Prelude> ((sin pi)^2) + ((cos pi)^2)
1.0
```

snd

type: `snd :: (a, b) -> b`

description: returns the second element of a two element tuple.

definition:

```
snd (_, y) = y
```

usage:

```
Prelude> snd ("harry", 3)
3
```

sort

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

description: sorts its argument list in ascending order. The items in the list must be in the class `Ord`.

usage:

```
List> sort [1, 4, -2, 8, 11, 0]
[-2,0,1,4,8,11]
```

note: This is *not* defined within the Prelude. You must `import` the `List.hs` module to use this function.

span

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

description: given a predicate and a list, splits the list into two lists (returned as a tuple) such that elements in the first list are taken from the head of the list while the predicate is satisfied, and elements in the second list are the remaining elements from the list once the predicate is not satisfied.

definition:

```
span p [] = ([],[])
span p xs@(x:xs')
  | p x = (x:ys, zs)
  | otherwise = ([],xs)
    where (ys,zs) = span p xs'
```

usage:

```
Prelude> span isDigit "123abc456"
("123", "abc456")
```

splitAt

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

description: given an integer (positive or zero) and a list, splits the list into two lists (returned as a tuple) at the position corresponding to the given integer. If the integer is greater than the length of the list, it returns a tuple containing the entire list as its first element and the empty list as its second element.

definition:

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

usage:
```
Prelude> splitAt 3 [1..10]
([1, 2, 3], [4, 5, 6, 7, 8, 9, 10])
Prelude> splitAt 5 "abc"
("abc", "")
```

sqrt

type: `sqrt :: Floating a => a -> a`

description: returns the square root of a number.

definition: `sqrt x = x ** 0.5`

usage:
```
Prelude> sqrt 16
4.0
```

subtract

type: `subtract :: Num a => a -> a -> a`

description: subtracts its first argument from its second argument.

definition: `subtract = flip (-)`

usage:
```
Prelude> subtract 7 10
3
```

sum

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

description: computes the sum of a finite list of numbers.

definition: `sum xs = foldl (+) 0 xs`

usage:
```
Prelude> sum [1..10]
55
```

tail

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

description: applied to a non–empty list, returns the list without its first element.

definition: `tail (_:xs) = xs`

usage:
```
Prelude> tail [1,2,3]
[2,3]
Prelude> tail "hugs"
"ugs"
```

take

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

description: applied to an integer (positive or zero) and a list, returns the specified number of elements from the front of the list. If the list has less than the required number of elements, `take` returns the entire list.

definition:

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

usage:

```
Prelude> take 4 "goodbye"
"good"
Prelude> take 10 [1,2,3]
[1,2,3]
```

takeWhile

type: `takewhile :: (a -> Bool) -> [a] -> [a]`

description: applied to a predicate and a list, returns a list containing elements from the front of the list while the predicate is satisfied.

definition:

```
takeWhile p [] = []
takeWhile p (x:xs)
  | p x = x : takeWhile p xs
  | otherwise = []
```

usage:

```
Prelude> takeWhile (<5) [1, 2, 3, 10, 4, 2]
[1, 2, 3]
```

tan

type: `tan :: Floating a => a -> a`

description: the trigonometric function tan, arguments are interpreted to be in radians.

definition: defined internally.

usage:

```
Prelude> tan (pi/4)
1.0
```

toLower

type: `toLower :: Char -> Char`

description: converts an uppercase alphabetic character to a lowercase alphabetic character. If this function is applied to an argument which is not uppercase the result will be the same as the argument unchanged.

definition:

```
toLower c
  | isUpper c = toEnum (fromEnum c - fromEnum 'A' + fromEnum 'a')
  | otherwise = c
```

usage:

```
Prelude> toLower 'A'
'a'
Prelude> toLower '3'
'3'
```

toUpper

type: `toUpper :: Char -> Char`

description: converts a lowercase alphabetic character to an uppercase alphabetic character. If this function is applied to an argument which is not lowercase the result will be the same as the argument unchanged.

definition:

```
toUpper c
  | isLower c = toEnum (fromEnum c - fromEnum 'a' + fromEnum 'A')
  | otherwise = c
```

usage:

```
Prelude> toUpper 'a'
'A'
Prelude> toUpper '3'
'3'
```

truncate

type: `truncate :: (RealFrac a, Integral b) => a -> b`

description: drops the fractional part of a floating point number, returning only the integral part.

usage:

```
Prelude> truncate 3.2
3
Prelude> truncate (-3.2)
-3
```

note:

unlines

type: `unlines :: [String] -> String`

description: converts a list of strings into a single string, placing a newline character between each of them. It is the converse of the function `lines`.

definition:

```
unlines xs
  = concat (map addNewLine xs)
  where
  addNewLine l = l ++ "\n"
```

usage:

```
Prelude> unlines ["hello world", "it's me,", "eric"]
"hello world\nit's me,\neric\n"
```

until

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

description: given a predicate, a unary function and a value, it recursively re–applies the function to the value until the predicate is satisfied. If the predicate is never satisfied `until` will not terminate.

definition:

```
until p f x
  | p x = x
  | otheriwise = until p f (f x)
```

usage:

```
Prelude> until (>1000) (*2) 1
1024
```

unwords

type: `unwords :: [String] -> String`

description: concatenates a list of strings into a single string, placing a single space between each of them.

definition:

```
unwords [] = []
unwords ws
  = foldr1 addSpace ws
  where
  addSpace w s = w ++ (' ':s)
```

usage:

```
Prelude> unwords ["the", "quick", "brown", "fox"]
"the quick brown fox"
```

words

type: `words :: String -> [String]`

description: breaks its argument string into a list of words such that each word is delimited by one or more whitespace characters.

definition:

```
words s
  | findSpace == [] = []
  | otherwise = w : words s''
  where
  (w, s'') = break isSpace findSpace
  findSpace = dropWhile isSpace s
```

usage:

```
Prelude> words "the quick brown\n\nfox"
["the", "quick", "brown", "fox"]
```

zip

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

description: applied to two lists, returns a list of pairs which are formed by tupling together corresponding elements of the given lists. If the two lists are of different length, the length of the resulting list is that of the shortest.

definition:

```
zip xs ys
  = zipWith pair xs ys
  where
  pair x y = (x, y)
```

usage:

```
Prelude> zip [1..6] "abcd"
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

zipWith

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

description: applied to a binary function and two lists, returns a list containing elements formed be applying the function to corresponding elements in the lists.

definition:

```
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _ _ = []
```

usage:

```
Prelude> zipWith (+) [1..5] [6..10]
[7, 9, 11, 13, 15]
```

1.2 A description of standard Haskell operators

Operators are simply functions of one or two arguments. Operators usually get written between their arguments (called infix notation), rather than to the left of them. Many operators have symbolic names (like + for plus), however this is out of convention rather than necessity. Others have completely textual names (such as ‘div‘ for integer division).

The following table lists many useful operators defined in the Prelude. Definitions of *associativity* and *binding power* are given after the table.

symbol	*behaviour*	*type*	*assoc–iativity*	*bind–ing power*
!!	list subscript	[a] -> Int -> a	left	9
.	compose	(a -> b) -> (c -> a) -> c -> b	right	9
ˆ	exponentiation	(Integral b, Num a) => a -> b -> a	right	8
**	exponentiation	Floating a => a -> a -> a	right	8
*	multiplication	Num a => a -> a -> a	left	7
/	division	Fractional a => a -> a -> a	left	7
‘div‘	integer division	Integral a => a -> a -> a	left	7
‘mod‘	modulus	Integral a => a -> a -> a	left	7
+	plus	Num a => a -> a -> a	left	6
-	minus	Num a => a -> a -> a	left	6
:	list construct	a -> [a] -> [a]	right	5
++	concatenate	[a] -> [a] -> [a]	right	5
/=	not equal	Eq a => a -> a -> Bool	non	4
==	equal	Eq a => a -> a -> Bool	non	4
<	less than	Ord a => a -> a -> Bool	non	4
<=	less than or equal	Ord a => a -> a -> Bool	non	4
>	greater than	Ord a => a -> a -> Bool	non	4
>=	greater than or equal	Ord a => a -> a -> Bool	non	4
‘elem‘	list contains	Eq a => a -> [a] -> Bool	non	4
‘notElem‘	list not contains	Eq a => a -> [a] -> Bool	non	4
&&	logical and	Bool -> Bool -> Bool	right	3
||	logical or	Bool -> Bool -> Bool	right	3

The higher the binding power the more tightly the operator binds to its arguments.
Function application has a binding power of 10,
and so takes preference over any other operator application.

Associativity: sequences of operator applications are allowed in Haskell for the convenience of the programmer. However, in some circumstances the meaning of such a sequence can be ambiguous. For example, we could interpret the expression `8 - 2 - 1` in two ways, either as `(8 - 2) - 1`, or as `8 - (2 - 1)` (each interpretation having a different value). Associativity tells us whether a sequence of a particular operator should be bracketed to the left or to the right. As it happens, the minus operator (`-`) is left associative, and so Haskell chooses the first of the alternative interpretations as the meaning of the above expression. The choice of associativity for an operator is quite arbitrary, however, they usually follow conventional mathematical notation. Note that some operators are *non-associative*, which means that they cannot be applied in sequence. For example, the equality operator (`==`) is non–associative, and therefore the following expression is not allowed in Haskell: `2 == (1 + 1) == (3 - 1)`.

Binding Power: Haskell expressions may also contain a mixture of operator applications which can lead to ambiguities that the rules of associativity cannot solve. For example, we could interpret the expression `3 - 4 * 2` in two ways, either as `(3 - 4) * 2`, or as `3 - (4 * 2)` (each interpretation having a different value). Binding power tells us which operators take precedence in an expression containing a mixture of operators. The multiplication operator (`*`), has a binding power of 7 (out of a possible 10), and the minus operator (`-`) has a binding power of 6. Therefore the multiplication operator takes precedence over the minus operator, and thus Haskell chooses the second of the alternative interpretations as the meaning of the above expression. All operators must have a binding power assigned to them which ranges from 1 to 10. Function application takes predence over everything else in an expression, and so the expression `reverse [1..10] ++ [0]` is interpreted as `(reverse [1..10]) ++ [0]`, rather than `reverse ([1..10] ++ [0])`.

1.3 Using the standard Haskell operators

`!!`

description: given a list and a number, returns the element of the list whose position is the same as the number.

usage:

```
Prelude> [1..10] !! 0
1
Prelude> "a string" !! 3
't'
```

notes: the valid subscripts for a list l are: $0 \leq subscript \leq ((length\, l) - 1)$. Therefore, negative subscripts are not allowed, nor are subsripts greater than one less than the length of the list argument. Subscripts out of this range will result in a program error.

`.`

description: composes two functions into a single function.

usage:

```
Prelude> (sqrt . sum ) [1,2,3,4,5]
3.87298
```

notes: `(f.g.h) x` is equivalent to `f (g (h x))`.

`**`

description: raises its first argument to the power of its second argument. The arguments must be in the `Floating` numerical type class, and the result will also be in that class.

usage:
```
Prelude> 3.2**pi
38.6345
```

^

description: raises its first argument to the power of its second argument. The first argument must be a member of the `Num` typeclass, and the second argument must be a member of the `Integral` typeclass. The result will be of the same type as the first argument.

usage:
```
Prelude> 3.2^4
104.858
```

%

description: takes two numbers in the `Integral` typeclass and returns the most simple ratio of the two.

usage:
```
Prelude> 20 % 4
5 % 1
Prelude> (5 % 4)^2
25 % 16
```

*

description: returns the multiple of its two arguments.

usage:
```
Prelude> 6 * 2.0
12.0
```

/

description: returns the result of dividing its first argument by its second. Both arguments must in the type class `Fractional`.

usage:
```
Prelude> 12.0 / 2
6.0
```

‘div‘

description: returns the integral division of the first argument by the second argument. Both arguments must be in the type class `Integral`.

usage:
```
Prelude> 10 ‘div‘ 3
3
Prelude> 3 ‘div‘ 10
0
```

notes: ‘div‘ is integer division such that the result is truncated towards negative infinity.

```
Prelude> (-12) ‘div‘ 5
-3
Prelude> 12 ‘div‘ 5
2
```

`mod`

description: returns the integral remainder after dividing the first argument by the second. Both arguments must be in the type class `Integral`.

usage:
```
Prelude> 10 `mod` 3
1
Prelude> 3 `mod` 10
3
```

+

description: returns the addition of its arguments.

usage:
```
Prelude> 3 + 4
7
Prelude> (4 % 5) + (1 % 5)
1 % 1
```

-

description: returns the substraction of its second argument from its first.

usage:
```
Prelude> 4 - 3
1
Prelude> 4 - (-3)
7
```

:

description: prefixes an element onto the front of a list.

usage:
```
Prelude> 1:[2,3]
[1,2,3]
Prelude> True:[]
[True]
Prelude> 'h':"askell"
"haskell"
```

++

description: appends its second list argument onto the end of its first list argument.

usage:
```
Prelude> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
Prelude> "foo " ++ "was" ++ " here"
"foo was here"
```

/=

description: is `True` if its first argument is not equal to its second argument, and `False` otherwise. Equality is defined by the `==` operator. Both of its arguments must be in the `Eq` type class.

usage:
```
Prelude> 3 /= 4
True
Prelude> [1,2,3] /= [1,2,3]
False
```

==

description: is `True` if its first argument is equal to its second argument, and `False` otherwise. Equality is defined by the `==` operator. Both of its arguments must be in the `Eq`

usage:

```
Prelude> 3 == 4
False
Prelude> [1,2,3] == [1,2,3]
True
```

<

description: returns `True` if its first argument is strictly less than its second argument, and `False` otherwise. Both arguments must be in the type class `Ord`.

usage:

```
Prelude> 1 < 2
True
Prelude> 'a' < 'z'
True
Prelude> True < False
False
```

<=

description: returns `True` if its first argument is less than or equal to its second argument, and `False` otherwise. Both arguments must be in the type class `Ord`.

usage:

```
Prelude> 3 <= 4
True
Prelude> 4 <= 4
True
Prelude> 5 <= 4
False
```

>

description:

usage: returns `True` if its first argument is strictly greater than its second argument, and `False` otherwise. Both arguments must be in the type class `Ord`.

```
Prelude> 2 > 1
True
Prelude> 'a' > 'z'
False
Prelude> True > False
True
```

>=

description:

usage: returns `True` if its first argument is greater than or equal to its second argument, and `False` otherwise. Both arguments must be in the type class `Ord`.

```
Prelude> 4 >= 3
True
Prelude> 4 >= 4
True
Prelude> 4 >= 5
False
```

`‘elem‘`

description: returns `True` if its first argument is an element of the list as its second argument, and `False` otherwise.

usage:

```
Prelude> 3 ‘elem‘ [1,2,3]
True
Prelude> 4 ‘elem‘ [1,2,3]
False
```

`‘notElem‘`

description: returns `True` if its first argument is *not* an element of the list as its second argument.

usage:

```
Prelude> 3 ‘notElem‘ [1,2,3]
False
Prelude> 4 ‘notElem‘ [1,2,3]
True
```

`&&`

description: returns the logical conjunction of its two boolean arguments.

usage:

```
Prelude> True && True
True
Prelude> (3 < 4) && (4 < 5) && False
False
```

`||`

description: returns the logical disjunction of its two boolean arguments.

usage:

```
Prelude> True || False
True
Prelude> (3 < 4) || (4 > 5) || False
True
```

1.4 Type Classes from the Haskell Prelude

Eq

description: Types which are instances of this class have equality defined upon them. This means that all elements of such types can be compared for equality.

instances:

- All Prelude types except `IO` and functions.

notes: Functions which use the equality operators (`==`, `/=`) or the functions `elem` or `notElem` will often be subject to the `Eq` type class, thus requiring the constraint `Eq a =>` in the type signature for that function.

Ord

description: Types which are instances of this class have a complete ordering defined upon them.

instances:

- All Prelude types except `IO`, functions, and `IOError`.

notes: Functions which use the comparison operators (`>`, `<`, `>=`, `<=`), or the functions `max`, `min`, `maximum` or `minimum` will often be subject to the `Ord` type class, thus requiring the constraint `Ord a =>` in the type signature for that function.

Enum

description: Types which are instances of this class can be enumerated. This means that all elements of such types have a mapping to a unique integer, thus the elements of the type must be sequentially ordered.

instances:

- `Bool`
- `Char`
- `Int`
- `Integer`
- `Float`
- `Double`

notes: Functions which use dot-dot notation (eg `[1,3 .. y]`) in list comprehensions will often be subject to the `Enum` type class, thus requiring the constraint `Enum a =>` in the type signature for that function.

Show

description: Types which are instances of this class have a printable representation. This means that all elements of such types can be given as arguments to the function `show`.

instances:

- All Prelude types.

notes: Functions which use the function `show` will often be subject to the `Show` type class, thus requiring the constraint `Show a =>` in the type signature for that function.

Read

description: Types which are instances of this class allow a string representation of all elements of the type to be converted to the corresponding element.

instances:
- All Prelude types except `IO` and functions.

notes: Functions which use the function `read` will often be subject to the `Read` type class, thus requiring the constraint `Read a =>` in the type signature for that function.

Num

description: This is the parent class for all the numeric classes. Any type which is an instance of this class must have basic numeric operators (such as plus, minus and multiply) defined on them, and must be able to be converted from an `Int` or `Integer` to an element of the type.

instances:
- `Int`
- `Integer`
- `Float`
- `Double`

notes: Functions which perform operations which are applicable to all numeric types, but not to other non–numeric types will often be subject to the `Num` type class, thus requiring the constraint `Num a =>` in the type signature for that function.

Real

description: This class covers all the numeric types whose elements can be expressed as a ratio.

instances:
- `Int`
- `Integer`
- `Float`
- `Double`

Fractional

description: This class covers all the numeric types whose elements are fractional. All such types must have division defined upon them, they must have a reciprocal, and must be convertible from rational numbers, and double precision floating point numbers.

instances:
- `Float`
- `Double`

notes: Functions which use the division operator (`/`) will often be subject to the `Fractional` type class, thus requiring the constraint `Fractional a =>` in the type signature for that function.

Integral

description: This class covers all the numeric types whose elements are integral.

instances:
- `Int`
- `Integer`

notes: Functions which use the operators `div` or `mod` will often be subject to the `Integral` type class, thus requiring the constraint `Integral a =>` in the type signature for that function.

Floating

description: This class covers all the numeric types whose elements are floating point numbers.

instances:
- `Float`
- `Double`

notes: Functions which use the constant `pi` or the functions `exp`, `log`, `sqrt`, `sin`, `cos` or `tan` will often be subject to the `Floating` type class, thus requiring the constraint `Floating a =>` in the type signature for that function.

1.5 The Haskell Prelude Class hierarchy

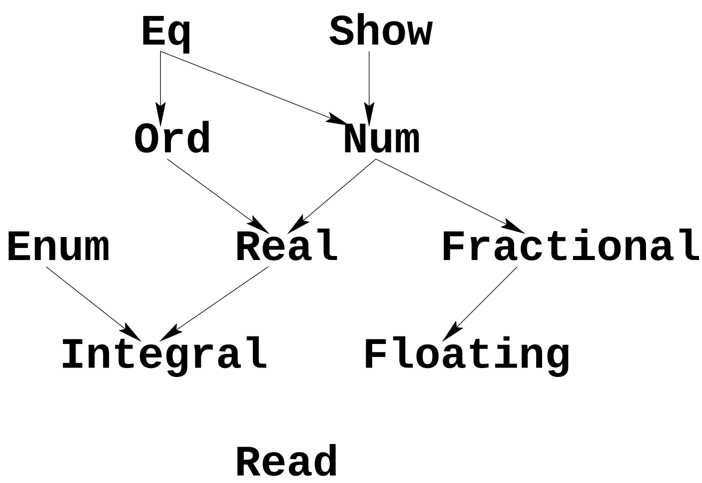

Figure 1: A sample of the class hierarchy from the Haskell Prelude

Figure 1 illustrates a sample of the type class hierarchy from the Haskell Prelude. Arrows in the diagram represent the ordering of classes in the hierarchy. For example, for a type to be in the class `Ord` it must also be in the class `Eq`. Note that the class `Read` is separate from the rest of the hierarchy.