Part 4
Functional programming in Haskell (continued)

Call-by-value versus call-by-name

Termination

```
inf :: Int
inf = 1 + inf --diverges with any strategy
Prelude> fst(0,inf)
0
```

`fst` uses call-by-name
if there is a reduction sequence which terminates, then call-by-name also terminates and gives the same result
call-by-name is preferable to call-by-value for the purpose of ensuring termination as often as possible

Call-by-value versus call-by-name

Number of reductions

```
square :: Int -> Int
square x = x*x
square (1+2)
```

with call-by-name `1+2` needs to be evaluated twice
Haskell lazy evaluation = call-by-name + sharing (pointers to arguments)

In other words: functions are non-strict

strict functions need all their arguments to be evaluated

```
Prelude> let bot = bot in (\x -> 0) bot

0

Prelude> let x = 1/0 in (\y -> 15) x

15
```

advantage: computationally expensive values may be passed as arguments
intuition: read declarations as definitions rather than assignments

Infinite data structures

- data constructors are non-strict too
- this allows the definition of infinite data structures

```
ones :: [Int]
ones = 1 : ones
```

evaluation of `ones` diverges, but:

```
*Main> head ones
1
```

property of lazy evaluation: expressions are only evaluated as much as required

```
numFrom n = n : numFrom(n+1)

squaresFrom n = map (^2) (numFrom n)

take _ [] = []
take 0 _ = []
take n (x:xs) = x:take(n-1) xs

*Main> take 5 (squaresFrom 0)
[0,1,4,9,16]
```


Separating control from data

without (or with) lazy evaluation:

```
replicate :: Int -> a -> [a]
replicate 0 _ = []
replicate n x = x:replicate (n-1) x
```

modular solution

```
repeat :: a -> [a]
repeat x = xs where xs = x : xs
replicate n = (take n) . repeat
```

Care is required

Computations which require to examinate an infinite list diverge

```
filter (<=5) [1..]
```

But

```
*Main> takeWhile (<=5) [1..]
[1,2,3,4,5]
```


Sieve of Eratosthenes

```
primes :: [Int]
primes = sieve [2..]
  where
    sieve :: [Int] -> [Int]
    -- filters by all primes starting from the head of the list
    sieve (p:xs) = p : sieve [ x | x <- xs, x `mod` p /= 0]
*Main> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

Another example

Fibonacci numbers

$fibs_0 = 1$
$fibs_1 = 1$
$fibs_{i+2} = fibs_i + fibs_{i+1}$

```
zip (x:xs) (y:ys) = (x,y) :: zip xs ys
zip _ _ = []

fibs = 1: 1: [a+b | (a,b) <- zip fibs (tail fibs)]
```


Variant

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

fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

Two examples you will revisit in coProlog

```
increment [] = []
increment (x : l) = (x + 1) : increment l

allPositive [] = True
allPositive (x : l) = x > 0 && allPositive l
```

`increment` is defined for all (computable) lists, where in coLP it will be defined only for *regular* (finite and infinite) lists

`allPositive` is only defined for finite lists and infinite (computable) lists with (at least) one non positive element, whereas in coLP it will be defined for all regular lists

User-defined types

- `data Bool = False | True`

  `Bool` is a *type constructor*, `True` and `False` are *(data) constructors*

- `data Colour = Red | Green | Blue | Indigo | Violet`

  `data Point a = Point a a`

  (disjoint) union or *sum* types, polymorphic *tuple type*

- the type constructor `Point` has type `a -> a -> Point a`, hence, e.g.:

  ```
  Point 1 2   :: Point Integer
  Point 'a' 'b' :: Point Char
  Point True False :: Point Bool
  ```

Recursive types

```
data BTree a = Empty | Node (a, BTree a, BTree a)
*Main> :type Node
Node :: (a, BTree a, BTree a) -> BTree a

insert :: Ord a => a -> BTree a -> BTree a
insert a Empty = Node(a, Empty, Empty)
insert a n@(Node(b,l,r)) =
  if (a<b) then Node(b, insert a l, r)
  else if (a>b) then Node(b,l, insert a r)
  else n

consBTree :: Ord a => [a] -> BTree a
consBTree = itlist (\t -> \a -> insert a t) Empty

inorder Empty = []
inorder (Node(a,l,r)) = (inorder l)++[a]++(inorder r)
```


Type classes

Overloading

```
*Main> 1+2
3
*Main> 1.0 + 2.0
3.0
```

The idea that `+` can be applied to any numeric type can be made explicit in its type by a class constraint (context) of the form `C a` with `a` type variable

```
(+) :: (Num a) => a -> a -> a
```



```
(+) :: (Num a) => a -> a -> a
```

- for any instantiation of `a` which is an instance of the class `Num` of the numeric types, the function `(+)` has type `a -> a -> a`
- a type which contains class constraints is an overloaded type
- a function with an overloaded type is an overloaded function, e.g., `(-)`, `(*)`, `abs`, ...
- numbers themselves are overloaded:

  ```
  3 :: (Num t) => t
  ```

Classes

Class declaration

```
class MyEq a where
  eq :: a -> a -> Bool --overloaded functions called methods
  neq :: a -> a -> Bool
  neq x y = not (eq x y)
  eq x y = not (neq x y)
```

Instance declaration

```
instance MyEq Bool where
  eq True True = True
  eq False False = True
  eq _ _ = False
```

warning + exception, or even divergence, if we omit some definition

Basic classes: equality types

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x==y)
```

basic types are instances, list and tuple types are instances provided that component types are
it may be derived for any datatype whose component types are also instances
function types are not instances
minimal complete definition: either `==` or `/=`

Example

```
isin :: (Eq a) => a -> [a] -> Bool
isin _ [] = False
isin x (y:ys) = x==y || isin x ys
```

- type of `isin` should be `a -> [a] -> Bool`
- but, we do not expect equality to be defined for all types
- moreover, we expect the definition of equality to be different for each type
- that is, `==` is an overloaded function
- otherwise we should use a different name for every type

Subclasses

Classes can be extended to form new classes

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

class constraint on a class declaration
meaning: we have to make a type instance of `Eq` before we can make it instance of `Ord`
we can assume `==` in function bodies in the class declaration or in an instance declaration

Instance declaration for parametric types

```
instance (Eq a) => Eq (BTree a) where
  Empty == Empty = True
  Node a l1 r1 == Node b l2 r2 = a==b && l1==l2 && r1==r2
  _ == _ = False
```

class constraint on an instance declaration
meaning: requirements on the arguments of the type constructor
`:info MyTypeClass`
works also for types, type constructors, functions

Basic classes: (totally) ordered types

```
class Eq a => Ord a where
  ...
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

basic types are instances, list and tuple types are instances provided that component types are
it may be derived for any datatype whose component types are also instances

Basic classes

Showable and readable types

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

read :: Read a => String -> a
```

to use `read` we may need an explicit type annotation

```
*Main> read "True" :: Bool
True
*Main> not (read "True")
False
```

Enumeration and bounded types

```
succ :: (Enum a) => a -> a
pred :: (Enum a) => a -> a
...
minBound :: (Bounded a) => a
maxBound :: (Bounded a) => a
```


Basic classes

Numeric types

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

Integral types

```
div :: (Integral a) => a -> a -> a
mod :::: (Integral a) => a -> a -> a
```

Derived instances

facility to automatically making new types instances of classes `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, and `Read`

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

in case of constructors with arguments their types must be instances of the derived classes