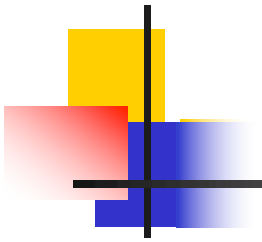
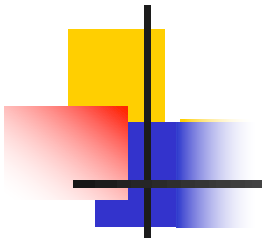


Announcements

- We will release the exam after class

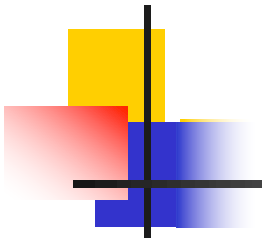
- Haskell homework is posted
  - Restrictions: do NOT include new modules
  - Due December 1st

List Comprehensions, ADTs and Maybe, Higher-Order Functions



So Far

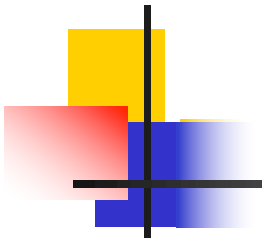
- Functional programming concepts
- Scheme
- Haskell
  - A purely functional language
  - Lazy evaluation
  - Rich syntax
    - Function definitions, comprehensions, sectionals, infinite lists, and many other
  - Static types and type inference



Outline

- *List comprehensions*
- *Type declarations and algebraic data types (ADTs)*
- *Pattern matching and case expressions*
- *Higher-order functions*
- *Useful Prelude higher-order functions*

- *Quiz 7*



Set Comprehensions

In mathematics, the comprehension notation can be used to construct new sets from old sets.

$$\{x^2 \mid x \in \{1..5\}\}$$

The set {1,4,9,16,25} of all numbers x^2 such that x is an element of the set {1…5}.

Lists Comprehensions

In Haskell, a similar comprehension notation can be used to construct new lists from old lists.

[x**2 for x in range(1,6)]

```
[x^2 | x ← [1..5]]
```

The list [1,4,9,16,25] of all numbers x^2 such that x is an element of the list [1..5].

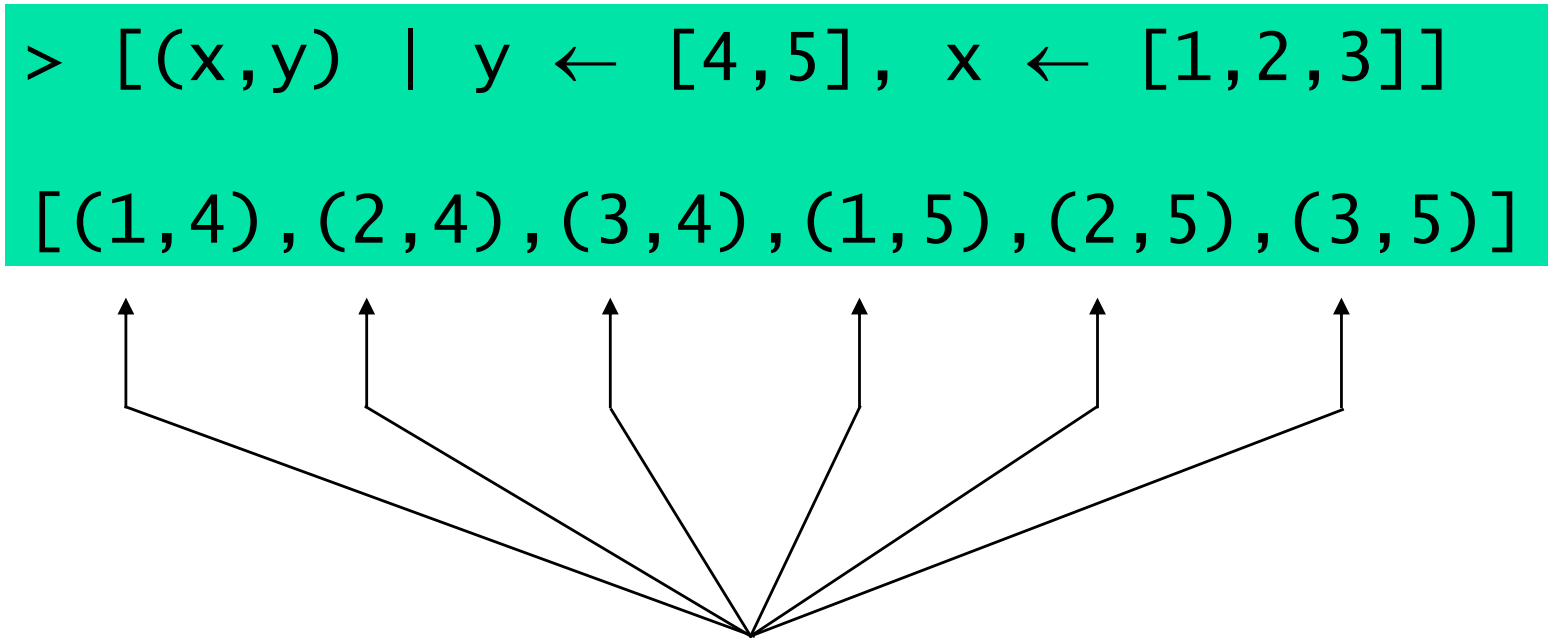
Note:

- The expression x ← [1..5] is called a generator, as it states how to generate values for x
- Comprehensions can have multiple generators, separated by commas. For example:

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

- Multiple generators resemble nested loops, with later generators as more deeply nested loops whose variables change value more frequently



x ← [1,2,3] is the last generator, so the value of the x component of each pair changes most frequently.

Dependant Generators

Later generators can depend on the variables that are introduced by earlier generators.

```
> [(x,y) | x ← [1..3], y ← [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Using a dependent generator can you define the library function that concatenates a list of lists:

For example:

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

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

```
concat :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

concat x = [z | y ← x, z ← y]

Guards

List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | x ← [1..10], even x]
```

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.

Can you define a function that maps a positive integer to its list of factors, e.g.,

```
> factors 15
[1,3,5,15]
```

```
factors :: Int → [Int]
factors n = [x | x ← [1..n], n `mod` x == 0]
```

Using factors, we can define a function that decides if a number is prime? E.g.,

```
> prime 15
False

> prime 7
True
```

```
prime :: Int → Bool
prime n = factors n == [1,n]
```

prime n = length (factors n) == 2

[1,n]

The Zip Function

A useful library function is zip, which maps two lists to a list of pairs of their corresponding elements.

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

For example:

```
> zip ['a','b','c'] [1,2,3,4]

[('a',1),('b',2),('c',3)]
```

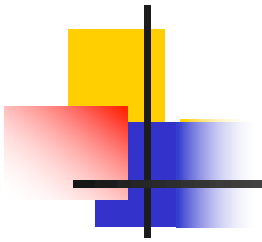
Using zip we can define a function that returns the list of all pairs of adjacent elements from a list:

For example:

```
> pairs [1,2,3,4]

[(1,2),(2,3),(3,4)]
```

[1, 2, 3, 4]
[2, 3, 4]
[(1,2), (2,3), (3,4)]

```
pairs :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)
```



Exercises

(1) A triple (x,y,z) of positive integers is called pythagorean if $x^2 + y^2 = z^2$. Using a list comprehension, define a function

```
pyths :: Int → [(Int,Int,Int)]
```

that maps an integer n to all such triples with components in [1..n]. For example:

```
> pyths 5
[(3,4,5),(4,3,5)]
```

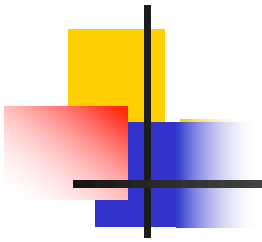
x ← [1..n],
y ← [1..n],
z ← [1..n],
x^2+y^2 == z^2

(2) A positive integer is perfect if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension, define a function

```
perfects :: Int → [Int]
```

that returns the list of all perfect numbers up to a given limit. For example:

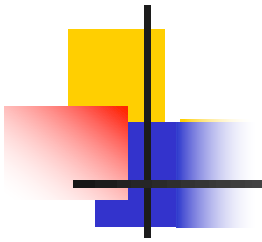
```
> perfects 500

[6,28,496]
```



Outline

- *List comprehensions*
- *Type declarations and algebraic datatypes (ADTs)*
- *Pattern matching and case expressions*
- *Higher-order functions*
- *Useful Prelude higher-order functions*

- *Quiz 7*



Type Declarations

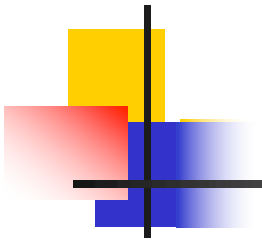
In Haskell, a new name for an existing type can be defined using a type declaration.

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

String is a synonym, also called *type alias*, for the type [Char].

type Name = String

Algebraic Datatypes (ADTs)



A completely new type, an algebraic data type, can be defined by specifying its values using a data declaration.

```
data Bool = False | True
```

Bool is a new type, with two new values False and True.

Note:

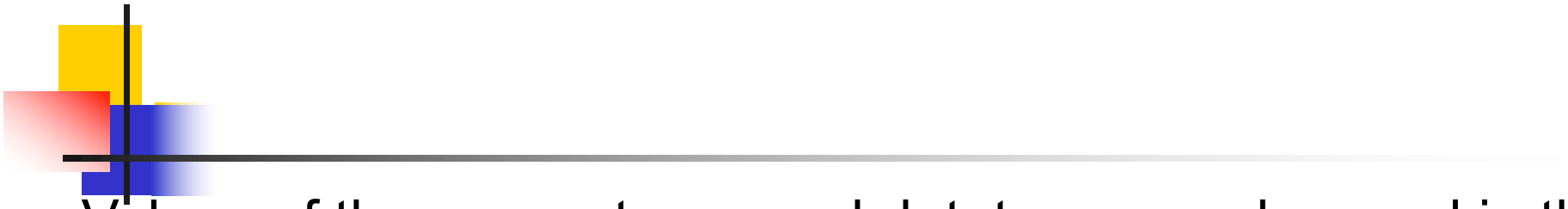
- The two values False and True are called the constructors for the datatype Bool

- Datatype and constructor names must always begin with an upper-case letter

- The Bool datatype is an enumeration type – this is Haskell's way of defining them. Other examples:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

```
data Month = Jan | Feb | … | Nov | Dec
```



Values of those new types and datatypes can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers :: [Answer]
answers = [Yes,No,Unknown]

flip :: Answer → Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

```
data Expr = Var Name
          | App Expr Expr
          | Lambda Name Expr
```

The constructors in a datatype declaration can also have components. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

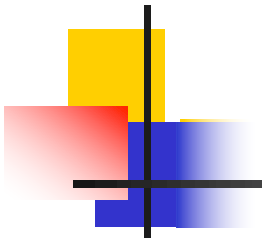
```
square :: Float → Shape
square n = Rect n n

area :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Note:

- Shape has values of the form Circle r where r is a float, and Rect x y where x and y are floats.

- Circle and Rect can be viewed as functions that construct values of type Shape:

```
Circle :: Float → Shape

Rect :: Float → Float → Shape
```



Algebraic data types are tagged unions (aka sums) of products (aka records)

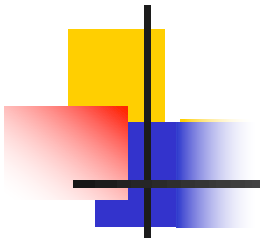
```
data Shape = Circle Float
           | Rect Float Float
```

union

Haskell keyword

the new type

new constructors (a.k.a. tags, disjuncts, summands)
Circle is a unary constructor, Rect is a binary.



Maybe Int
Maybe Expr
[Int]

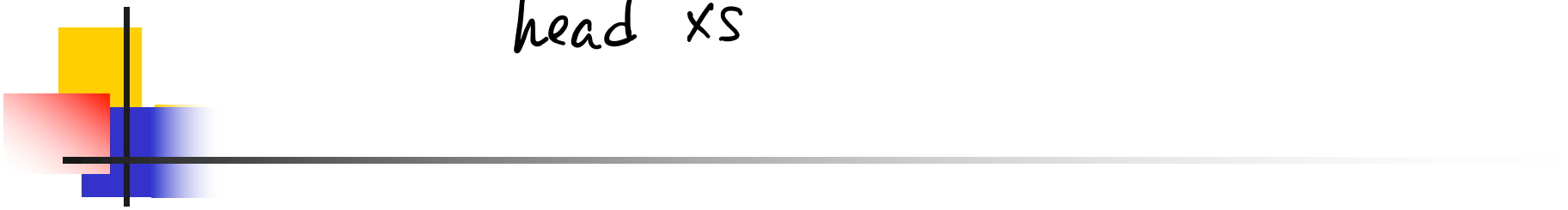
Data declarations can have parameters. For example, given

```
data Maybe a = Nothing | Just a
```

Define `safediv n d` to return `Nothing` if d is 0 and the (Maybe-encapsulated) value otherwise

```
safediv :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv n d = Just (n `div` d)
```

(div n d)



Data declarations can have parameters. For example, given

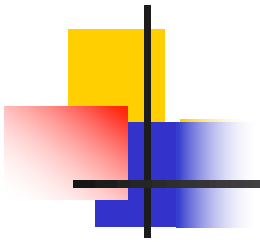
```
data Maybe a = Nothing | Just a
```

Define `safehead lis` to return `Nothing` if `lis` is empty and the (Maybe-encapsulated) head otherwise

```
-- pattern matching
safehead :: [a] → Maybe a
safehead []    = Nothing
safehead (x:_) = Just x
```

xs

Just (head xs)



Recursive Types

In Haskell, algebraic datatypes can be declared in terms of themselves. That is, they can be recursive.

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors Zero :: Nat and Succ :: Nat → Nat.

```
> zero = Zero
> one = Succ zero
> two =
```

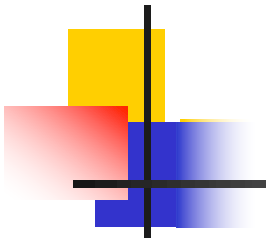
`Expr` is a recursive type to represent arithmetic expressions over addition and multiplication:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

For example, a certain expression is represented as follows:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

1 + 2 * 3



freeVars (Var n) = ...
freeVars (App e1 e2) = ...
freeVars (Lambda n e) = ...

Using recursion, it is now easy to define functions that process expressions. For example:

```
size :: Expr → Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval :: Expr → Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

A data constructor constructs a new data object. For example:

```
Val 1
Val 3
Mul (Val 1) (Val 3)
Add (Val 1) (Mul (Val 2) (Val 3))
```

A type constructor constructs a new type object. For example:

```
safediv :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv n d = Just (n `div` d)
```

Pattern Matching with Case Expressions

Examine values of an algebraic data type. For example:

```
area :: Shape -> Float
area s = case s of
           Circle r -> pi * r^2
           Rect x y -> x * y
```

Two important points on case expressions:

- Test: does the given value match this pattern?
- Binding: if value matches, bind corresponding values of **s** and pattern

Notes:

An underscore _ is the wildcard pattern. It matches anything.

An as-pattern x@pat can be used to match a pattern while retaining the variable referring the object being matched:

```
foo::Expr -> String
foo e@(Add _ _) = e ++ “is an Add expression”
```

Patterns can be nested:

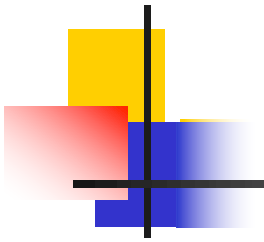
```
bar::Expr -> String
bar e@(Add _ r@(Add _ _)) = e ++ “is …”
```

Notes:

Haskell tries patterns in turn until it finds a match

```
foo::Expr -> String
foo e@(Add _ _) = e ++ “is Add expression”
foo e@(Add _ r@(Add _ _)) = e ++ “is …”
…
```

If there is no match, we get an error



Exercises

(1) Write the safediv function using a case expression.

(2) Write the merge function from earlier class using an as-pattern.

An expression language with just division operation and an evaluation function:

```
data Expr = Val Int
          | Div Expr Expr
     deriving (Eq, Show)

eval::Expr -> Int
eval (Val i) = i
eval (Div e1 e2) = (eval e1) `div` (eval e2)
```

A few expressions:

```
t1 = Div (Val 1) (Div (Val 4) (Val 2))
t2 = Div (Val 10) (Div (Val 8) (Val 10))
```

What can go wrong and what can we do to fix?

- Two terms:

```
ok :: Expr
ok = Div (Div (Val 1800) (Val 2)) (Val 21)
42
err :: Expr
err = Div (Val 2) (Div (Val 1) (Div (Val 2) (Val 3)))

> eval ok
 42
> eval err
*** Exception
```

- `safeEval` to safely evaluate an arithmetic expression. Use a case expression

```
safeEval :: Expr -> Maybe Int
safeEval (Val i) = Just i
safeEval (Div x y) =
    case (safeEval y) of
        Just 0 -> Nothing
        Just d -> case (safeEval x) of
                    Just n -> Just (n `div` d)
                    _      -> Nothing
        Nothing -> Nothing
```

`Tree a` is a recursive type to represent binary trees:

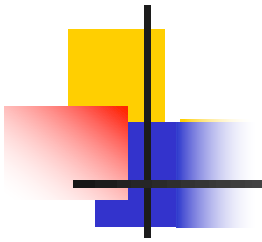
```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
     deriving (Eq, Show)
```

For example, a few trees:

```
t1 = Node (Leaf 1) (Node (Leaf 2)(Leaf 3))
t2 = Node (Leaf 'a') (Node (Node (Leaf 'b') (Leaf 'c')) (Leaf 'd'))
t3 = Node (Leaf 'a') (Node (Leaf 'b')(Leaf 'c'))
```

- `zipTree` to zip two trees. If the two trees are not isomorphic, return Nothing.

```
zipTree :: (Tree a) -> (Tree b) -> Maybe (Tree (a,b))
zipTree (Leaf x) (Leaf y) = return (Leaf (a,b))
zipTree (Node l1 r1) (Node l2 r2) =
        case (zipTree l1 l2) of
          Just l  -> case (zipTree r1 r2) of
                       Just r -> Just (Node l r)
                       _      -> Nothing
          _       -> Nothing
zipTree _ _ = Nothing
```



Outline

- *List comprehensions*
- *Type declarations and algebraic datatypes (ADTs)*
- *Pattern matching and case expressions*
- *Higher-order functions*
- *Useful Prelude higher-order functions*

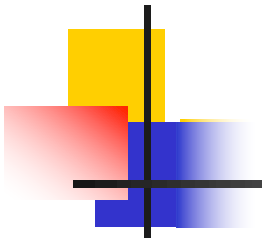
- *Quiz 7*

Higher-Order Functions

A function is called higher-order if it takes a function as an argument or returns a function as a result

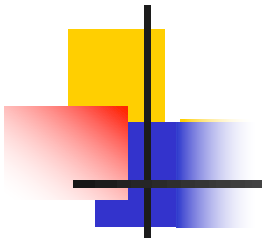
```
twice :: (a → a) → a → a
twice f x = f (f x)
```

twice is higher-order because it takes a function as its first argument.



Things to Do with a List

- Run some function on every element of the list
- “Filter” according to a predicate: keep only certain elements of the list, filter out rest
- “Reduce” a list into a value in some way, e.g., find maximal value, sum, product, etc.
- Other?



The Map Function

The higher-order library function called map applies a function to every element of a list

```
map ::
```

For example:

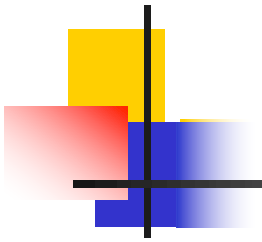
```
> map (+1) [1,3,5,7]

[2,4,6,8]
```

The map function can be defined in a simple manner using a list comprehension:

```
map f xs = [f x | x ← xs]
```

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

```
map f []     = []
map f (x:xs) = f x : map f xs
```



The Filter Function

The higher-order library function filter selects every element from a list that satisfies a predicate

```
filter ::
```

For example:

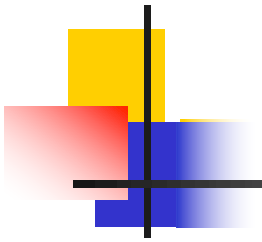
```
> filter even [1..10]

[2,4,6,8,10]
```

Filter can be defined using a list comprehension:

```
filter p xs =
```

Alternatively, it can be defined using recursion:

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



The Foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f []     = v
f (x:xs) = x ⊕ f xs
```

f maps the empty list to some value v, and any non-empty list to some function ⊕ applied to its head and f of its tail.

The higher-order library function foldr (fold right) encapsulates this simple pattern of recursion, with the function $\oplus$ and the value v as arguments.

For example:

```
sum xs =

product xs =

or xs = foldr (||) False xs

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

The higher-order library function foldr (fold right) encapsulates this simple pattern of recursion, with the function $\oplus$ and the value v as arguments.

For example:

```
length xs =

reverse xs =
```

Foldr itself can be defined using recursion:

```
foldr ::

foldr f v []     = v

foldr f v (x:xs) = f x (foldr f v xs)
```

However, it is best to think of foldr non-recursively, as simultaneously replacing each (:) in a list by a given function, and [] by a given value.

foldl

```
foldl ⊕ v [x1,x2,x3] = ((v ⊕ x1) ⊕ x2) ⊕ x3
```

v [x_1 , x_2 , x_3 , … x_n]

v_1 [x_2 , x_3 , … x_n]

v_2 [x_3 , … x_n]

…

v_{n-1} [x_n]

v_n [ ]

For example:

```
  sum [1,2,3]
=
  foldl (+) 0 [1,2,3]
=
  foldl (+) 0 (1:(2:(3:[])))
=
  (((0+1)+2)+3)
=
  6
```

We cannot replace each (:) by (+) as we did with foldr.

For example:

```
  foldl (-) 0 [1,2,3]
=
  foldl (-) 0 (1:(2:(3:[])))
=
  (((0-1)-2)-3)
=
  -6
```

Other Foldl Examples

```
length' :: [a] → Int
length' xs = foldl  ?  ?  xs
```

```
length' xs = foldr (\_ n → 1+n) 0 xs
```

versus:

```
length' xs = foldl
```

Other Foldl Examples

```
reverse' :: [a] → Int
reverse' (x:xs) = reverse' xs ++ [x]
```

```
reverse' xs = foldr (\el v → v++[el]) [] xs
```

versus

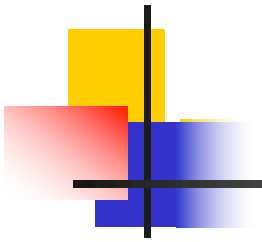
```
reverse' xs = foldl
```

Note: use foldl' from Data.List.

foldl vs. foldr vs. foldl'

```
Prelude> import Data.List
Prelude Data.List> :set +s
Prelude Data.List> foldl (+) 0 [0..1000000]
500000500000
(0.23 secs, 161,298,112 bytes)

Prelude Data.List> foldr (+) 0 [0..1000000]
500000500000
(0.17 secs, 161,588,432 bytes)

Prelude Data.List> foldl' (+) 0 [0..1000000]
500000500000
(0.05 secs, 88,071,320 bytes)
```

Why?



Outline

- *List comprehensions*
- *Type declarations and algebraic datatypes (ADTs)*
- *Pattern matching and case expressions*
- *Higher-order functions*
- *Useful Prelude higher-order functions*

- *Quiz 7*

The library function all decides if every element of a list satisfies a given predicate.

```
all :: (a → Bool) → [a] → Bool
all p xs = and [p x | x ← xs]
```

For example:

```
> all even [2,4,6,8,10]

?
```

Dually, the library function any decides if at least one element of a list satisfies a predicate.

```
any :: (a → Bool) → [a] → Bool
any p xs = or [p x | x ← xs]
```

For example:

```
> any (== ' ') "abc def"

?
```

The library function takeWhile selects elements from a list while a predicate holds of all the elements.

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

For example:

```
> takeWhile (/= ' ') "abc def"

?
```

Dually, the function dropWhile removes elements while a predicate holds of all the elements.

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

For example:

```
> dropWhile (== ' ') "   abc"

?
```

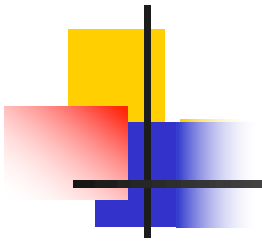
The function iterate generates an infinite list by applying a function f repeatedly starting from an initial value x:

```
iterate :: (a → a) → a → [a]
iterate f x = x : iterate f (f x)
```

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

For example:

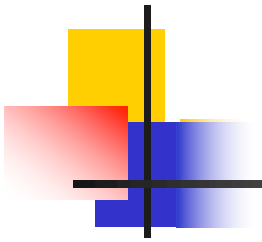
```
> take 10 $ iterate (+1) 0
?
```



Exercise

Newton's method for finding positive `sqrt(x)`:

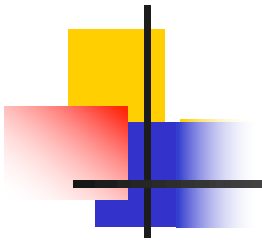
```
root :: Float
root x = rootiter x 1

rootiter :: Float -> Float -> Float
rootiter x y
   | satisfactory x y = y
   | otherwise = rootiter x (improve x y)

satisfactory x y = abs (y * y - x) < 0.01

improve x y = (y + x/y) / 2
```



Exercises

(1) Express the comprehension [f x | x ← xs, p x] using the functions map and filter.

(2) Redefine map f and filter p using foldr.

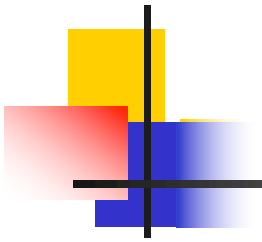


Exercises

(3) Voting. Write a program in that determines the winner of a vote. E.g., in the list of votes below 5 wins:

```
votes = [Int]
votes = [1,2,5,3,1,2,5,6,6,5,5,5,5,4]
```

```
count :: Eq a => a -> [a] -> Int
…
> count 5 votes
6
```

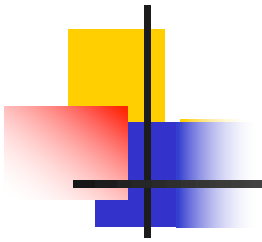


Exercises

```
rdups :: [a] -> [a]
…
> rdups votes
[1,2,5,3,6,4]
```

```
summarize :: Ord b => [b] -> [(Int, b)]
…
> summarize votes
[(1,3),(1,4),(2,1),(2,2),(2,6),(6,5)]
```

Moving On To Higher Grounds



In Haskell we recognize and generalize patterns. We already saw patterns on list structures. How about trees?

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

`treeSize` and `treeSum`:

```
treeSize :: Tree a → Int
treeSize (Leaf n)   = 1
treeSize (Node x y) = treeSize x + treeSize y

treeSum :: Tree Int → Int
treeSum (Leaf n)   = n
treeSum (Node x y) = treeSum x + treeSum y
```

`treeFlatten` and `treeDepth`:

```
flatten :: Tree a → [a]
flatten (Leaf n)   = [n]
flatten (Node x y) = flatten x ++ flatten y

depth :: Tree a → Int
depth (Leaf n)   = 0
depth (Node x y) = 1 + max (depth x) (depth y)
```

The End