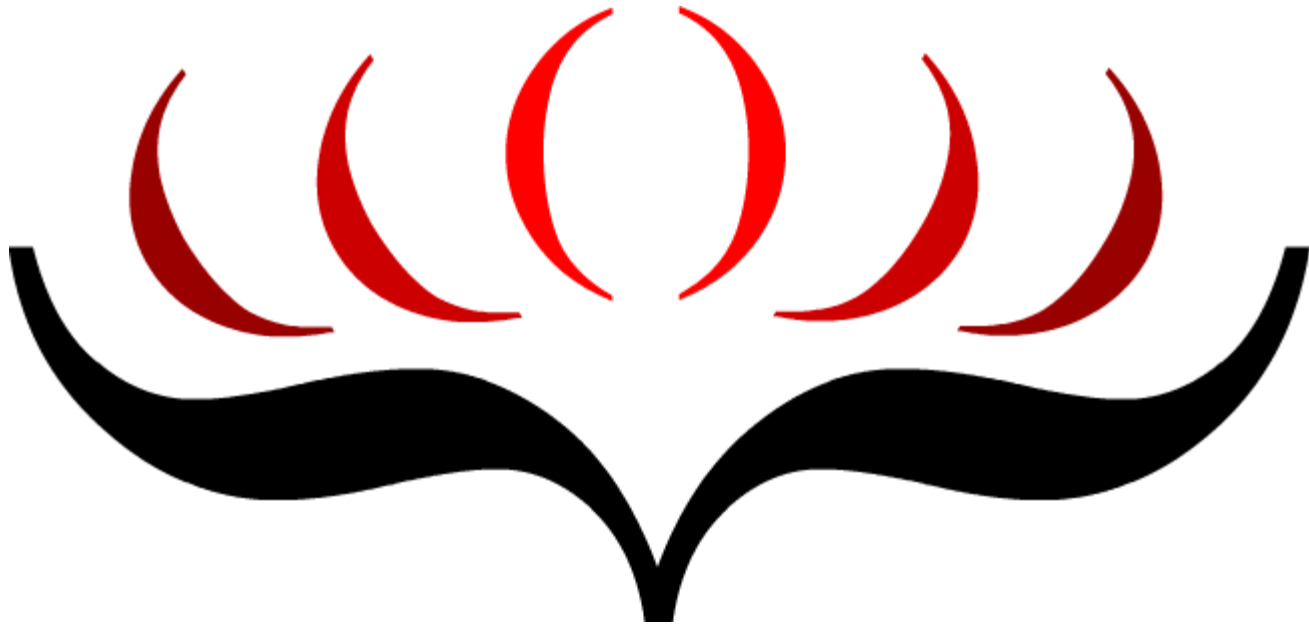


Lazy evaluation and currying in Haskell



Agenda

- Lazy programming
 - Lazy evaluation
 - Infinite lists
 - Infinite loops
- Currying and uncurrying

Lazy evaluation





Lazy evaluation



Questions



Lazy evaluation

- No expression is evaluated until its value is needed.
- No shared expression is evaluated more than **once**; if the expression is ever evaluated then the result is shared between all those places in which it is used.

Non-strict & strict

- Lazy functions are also called **non-strict** and evaluate their arguments **lazily** or **by need**.
- C# and Java methods are **strict** and evaluate their arguments eagerly.

Passing arguments to a function

- By value
- By reference
- By name
- By need

In Haskell

All functions and constructors -
are **non-strict** (lazy) with the
exception of primitive
arithmetic and logic operations.

listFrom

```
{- infinity list
--
-- listFrom 10
-- 10 : listFrom 11
-- 10 : 11 : listFrom 12
-- ...
-}
listFrom :: Integer -> [Integer]
listFrom n = n : (listFrom (n + 1))
```

repeat

```
{- repeat value
--
-- repeat 1
-- 1 : repeat 1
-- 1 : 1 : repeat 1
-- ...
-}
repeat    :: a -> [a]
repeat e = ls where ls = e : ls
```

cycle

```
{- cycle value
--
-- cycle "123"
-- "123" : cycle "123"
-- "123" : "123" : cycle "123"
-}
cycle    :: [a] -> [a]
cycle xs = ls where ls = xs ++ ls
```


sumFirst

```
{- sum of list
--
-- sumFirst 0 [1, 2, 3]
-- 0
--
-- sumFirst 3 [1, 2, 3]
-- 1 + sumFirst (3-1) [2, 3]
-- 1 + 2 + sumFirst (3-1-1) [3]
-- 1 + 2 + 3 + sumFirst 0 []
-- 6
-}
sumFirst :: Integer -> [Integer] -> Integer
sumFirst 0 _ = 0
sumFirst n (x:xs) = x + sumFirst (n-1) xs
```

iterate

```
{-  
--  
-- iterate (+1) 2  
-- 2 : iterate (+1) ((+1) 2)  
-- 2 : 3 : iterate (+1) ((+1) 3)  
-- ...  
-}  
iterate :: (a -> a) -> a -> [a]  
iterate f e = e : iterate f (f e)
```

diff

```
-- take 10 (diff [x*x | x <- [1..]])  
-- [3,5,7,9,11,13,15,17,19,21]  
  
-- take 10 (filter (\x -> mod x 5 == 0) (diff [x*x | x <- [1..]]))  
-- [5,15,25,35,45,55,65,75,85,95]  
  
-- take 10 (filter (\x -> mod x 115 == 0) (diff [x*x | x <- [1..]]))  
-- [115,345,575,805,1035,1265,1495,1725,1955,2185]  
diff :: (Num a) => [a] -> [a]  
diff (x:xs@(y:ys)) = y-x : diff xs
```

Sieve of Eratosthenes

	2	3	4	5	6	7	8	9	10	Prime numbers
11	12	13	14	15	16	17	18	19	20	
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	
41	42	43	44	45	46	47	48	49	50	
51	52	53	54	55	56	57	58	59	60	
61	62	63	64	65	66	67	68	69	70	
71	72	73	74	75	76	77	78	79	80	
81	82	83	84	85	86	87	88	89	90	
91	92	93	94	95	96	97	98	99	100	
101	102	103	104	105	106	107	108	109	110	
111	112	113	114	115	116	117	118	119	120	

primes

```
{- prime numbers
--
-- primes
-- sieve [2..]
-- 2 : (sieve [x | x <- [3..], mod x 2 /= 0])
-}
primes :: [Integer]
primes = sieve [2..] where
    sieve (e:ls) = e : (sieve [x | x <- ls, mod x e /= 0])
```


pairs

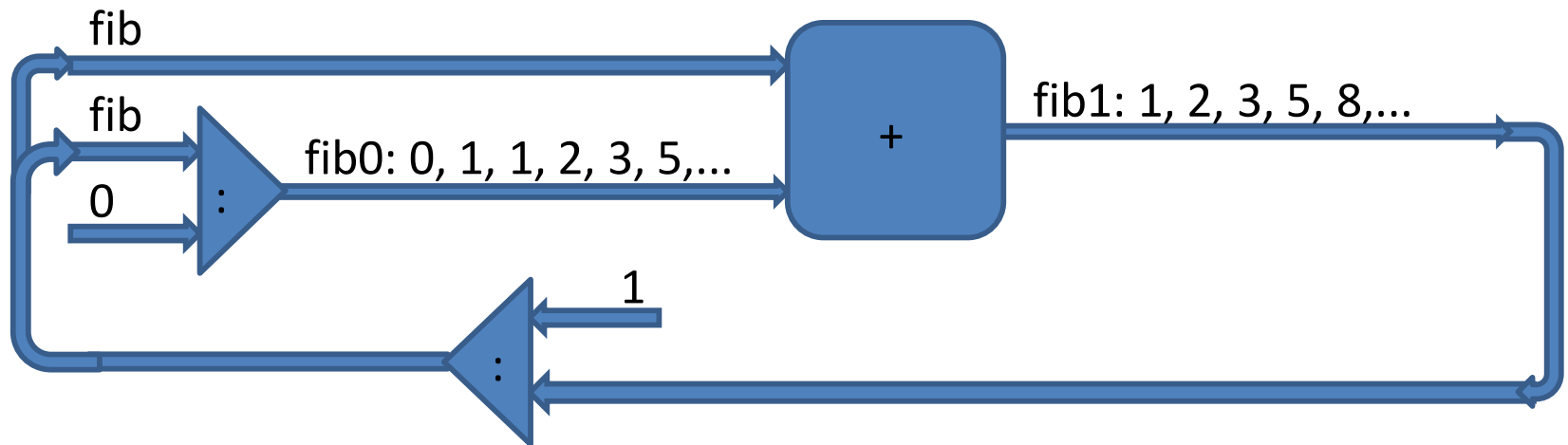
```
{-  
--  
-- take 3 (pairs primes)  
-- [(2,3),(3,5),(5,7)]  
-}  
pairs :: [Integer] -> [(Integer, Integer)]  
pairs (x:xs@(y:ys)) = (x,y):pairs xs
```

twins

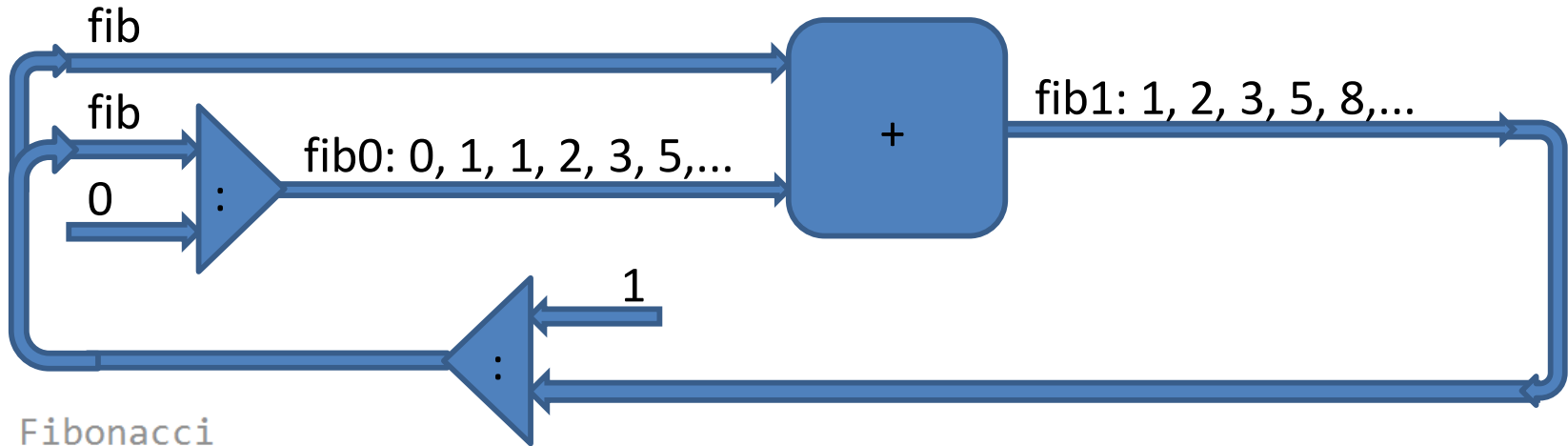
```
{-  
--  
-- take 5 twins  
-- [(3,5),(5,7),(11,13),(17,19),(29,31)]  
-}  
twins :: [(Integer, Integer)]  
twins = [(x,y) | (x,y) <- pairs primes, y - x == 2]
```

Fibonacci

fib: 1, 1, 2, 3, 5, 8,...



Fibonacci



```
{- Fibonacci
--
-- fib
-- 1 : fib1
-- 1 : zipWith (+) fib fib0
-- 1 : zipWith (+) (1:fib1) (0:fib)
-- 1 : zipWith (+) (1:(zipWith (+) fib fib0)) (0:1:fib1)
-- 1 : zipWith (+) (1:(zipWith (+) fib fib0)) (0:1:(zipWith (+) fib fib0))
-}
fib0 = 0:fib
fib1 = zipWith (+) fib fib0
fib  = 1:fib1
```

Currying

It is the process of transforming a function that takes multiple arguments into a function that takes just a single argument and returns another function if any arguments are still needed.

raiseList

```
{-  
-- raiseList [1, 2, 3]  
-- [2, 3, 4]  
--  
-}  
raiseList :: (Num a) => [a] -> [a]  
raiseList ls = map (+1) ls
```

raiseList

```
{-  
-- raiseList [1, 2, 3]  
-- [2, 3, 4]  
--  
-}  
raiseList :: (Num a) => [a] -> [a]  
raiseList is = map (+1) is
```

raiseList

```
{-  
-- raiseList [1, 2, 3]  
-- [2, 3, 4]  
-- map (+1) is -- \ls -> map (+1) ls  
-}  
raiseList :: (Num a) => [a] -> [a]  
raiseList    = map (+1)
```

plus

```
plus      :: Integer -> Integer -> Integer  
plus x y = x + y
```

```
plus      :: Integer -> Integer -> Integer  
plus x y = \x y -> x + y
```

```
plus      :: Integer -> (Integer -> Integer)  
plus x = \y -> x + y
```

mult & plus1

```
{-  
-- mult 3 5  
-- 15  
-}  
mult :: Integer -> Integer -> Integer  
mult x y = x * y
```

```
{-  
-- plus1 3  
-- 4  
-}  
plus1 :: Integer -> Integer  
plus1 x = 1 + x
```



```
{-  
-- plus1AndMult 3 5  
-- 16  
-}  
plus1AndMult      :: Integer -> Integer -> Integer  
plus1AndMult x y = plus1 (mult x y)  
  
plus1AndMultShort :: Integer -> Integer -> Integer  
plus1AndMultShort x = plus1 . (mult x)  
--plus1AndMultShort x = \y -> plus1 (mult x y)
```

Composition

```
-- | Function composition.  
-- Make sure it has TWO args only on the left, so that it inlines  
-- when applied to two functions, even if there is no final argument  
(.) :: (b -> c) -> (a -> b) -> a -> c  
(.) f g = \x -> f (g x)  
  
-- f . g = \y -> f (g y)  
-- plus1 . (mult x) = \y -> plus1 (mult x y)
```

curried & uncurried

```
{-  
-- is the curried form  
-}  
plusFirst      :: Integer -> Integer -> Integer  
plusFirst x y = x + y  
  
{-  
-- is the uncurried form  
-}  
plusSecond     :: (Integer -> Integer) -> Integer  
plusSecond (x, y) = x + y  
  
curry plusSecond == plusFirst  
uncurry plusFirst == plusSecond
```

Partial application

```
{- Partial application
-}
(+) :: (Num a) => a -> a -> a
(+) 2 3 -> 5
(+) 2 -> \x -> 2 + x
(5 +)

{-
-- (+1) 1 :: map (+1) [2, 3]
-- 2 :: (+1) 2 :: map (+1) [3]
-}
map (+1) [1, 2, 3]
```

checkedList

```
{-  
-- map (== 2) [1, 2]  
-- 1 == 2 : map (==2) [2]  
-- 1 == 2 :: 2 == 2  
-- [False, True]  
-}  
checkedList :: (Eq a) => a -> [a] -> [Bool]  
checkedList e ls = map (== e) ls
```

anyTrue

```
{-  
-- foldr (||) False [False, True]  
-- False || (foldr (||) False [True])  
-- False || (True || False)  
-- False || True  
-- True  
-}  
anyTrue :: [Bool] -> Bool  
anyTrue ls = foldr (||) False ls
```

searchList

```
searchList :: (Eq a) => a -> [a] -> Bool
searchList e ls = anyTrue (checkedList e ls)
```

```
searchList2 :: (Eq a) => a -> [a] -> Bool
searchList2 e ls = foldr (||) False (map (== e) ls)
```

```
{-
-- searchList3 3 [1, 2, 3, 4, 5]
-- (foldr (||) False) . (map (== 3) [1, 2, 3, 4, 5])
-- foldr (||) False [1 == 3, 2 == 3, 3 == 3, 4 == 3, 5 == 3]
-- foldr (||) False [False, False, True, False, False]
-- True
-}
```

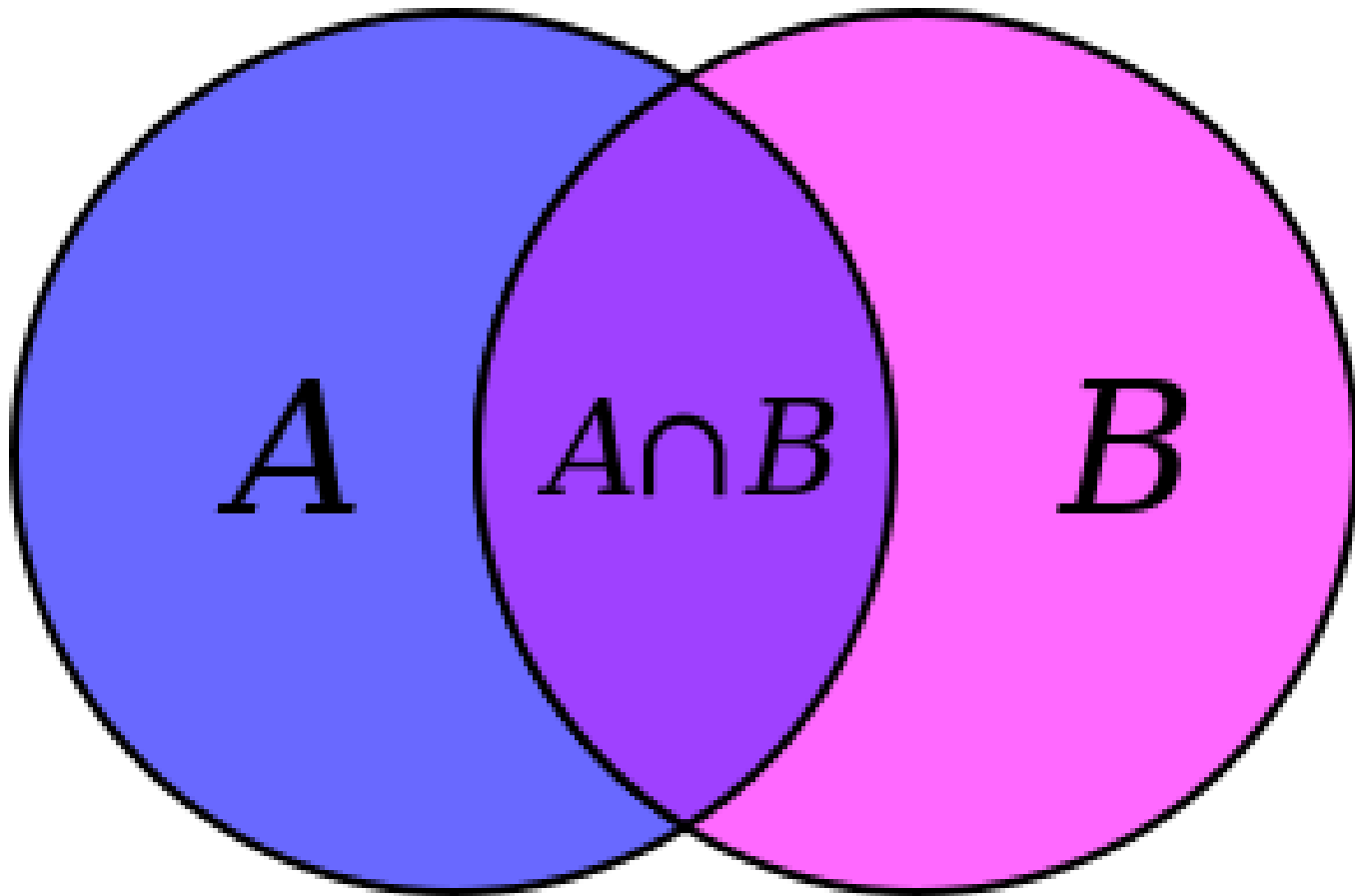
```
searchList3 :: (Eq a) => a -> [a] -> Bool
searchList3 e = (foldr (||) False) . (map (== e))
```

Partial application

```
{- Partial application
-}
(+) :: (Num a) => a -> a -> a
(+) 2 3 -> 15
(+) 2 -> \x -> 2 + x
(5 +)

{-
-- (+1) 1 :: map (+1) [2, 3]
-- 2 :: (+1) 2 :: map (+1) [3]
-}
map (+1) [1, 2, 3]
```


The math set



IntSet & empty

```
type IntSet = (Integer -> Bool)
empty      :: IntSet
empty e = False
```

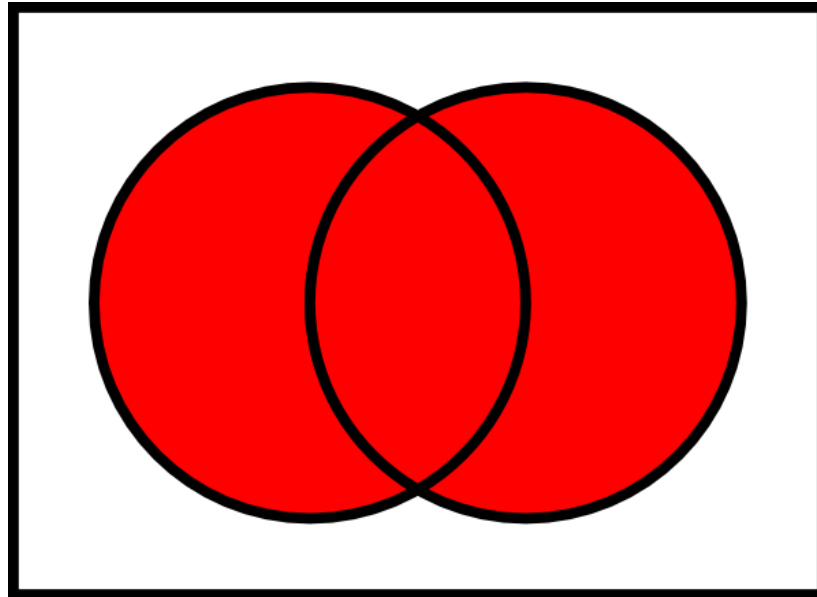
from2to100

```
{- (e > 1) && (e < 101)
--
-- from2to100 1
-- False
--
-- from2to100 100
-- True
-}
from2to100 e = (e >= 2) && (e <= 100)
```

odds

```
{- The set of odd numbers
--
-- odds 39129
-- True
--
-- odds 3290
-- False
-}
odds    :: IntSet
odds e = mod e 2 == 1
```

Unions

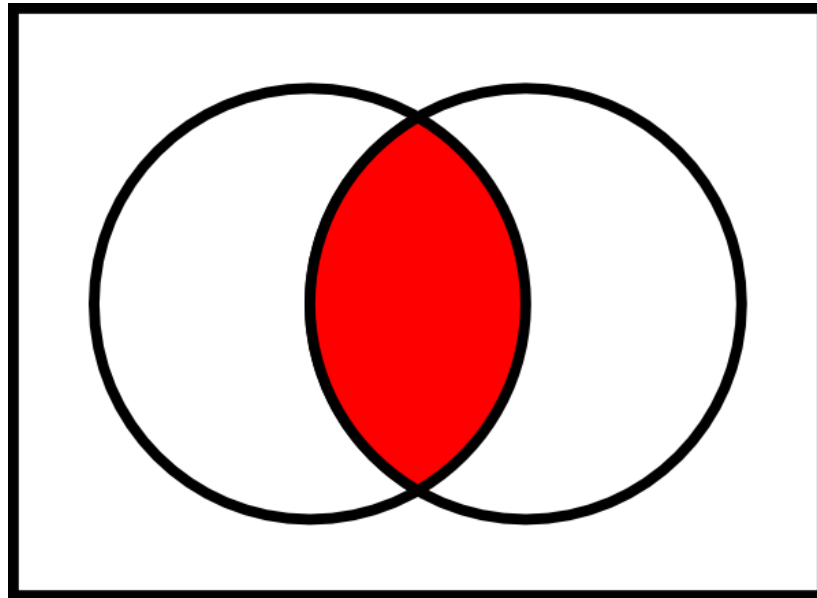


The **union** of A and B ,
denoted $A \cup B$.

union

```
{- Unions
-- Two sets can be "added" together.
-- The union of A and B, denoted by  $A \cup B$ ,
-- is the set of all things that are members of either A or B.
-}
union          :: IntSet -> IntSet -> IntSet
union s0 s1) e = (s0 e) || (s1 e)
```

Intersections

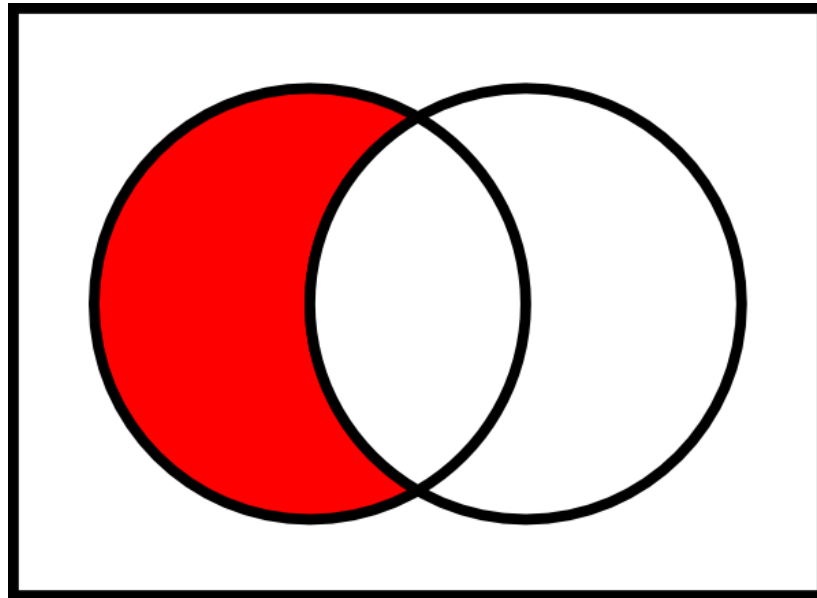


The **intersection** of A
and B , denoted $A \cap B$.

intersec

```
{- Intersections
--
-- A new set can also be constructed by determining
-- The intersection of A and B, denoted by  $A \cap B$ , :
--
-- intersec empty empty is empty
-- intersec odds from2to100 is [3,5..99]
-}
intersec :: IntSet -> IntSet -> IntSet
(intersec s0 s1) e = (s0 e) && (s1 e)
```


Complements



The **relative complement** of B in A .

compl

```
{- Complements
--
-- Two sets can also be "subtracted".
-- The relative complement of B in A
-- denoted by  $A \setminus B$  (or  $A - B$ ),
-- is the set of all elements that are members of A but not members of B.
--
-- compl odds from2to100 is [1, 101, 103, 105 ...]
-}
compl :: IntSet -> IntSet -> IntSet
(compl s0 s1) e = (s0 e) && False == (s1 e)
```

addElem

```
{- add new element
--
-- addElem 1 empty
-- \e -> (1 == e) || (empty e)
-- \e -> (1 == e) || False
-}
addElem :: Integer -> IntSet -> IntSet
addElem new s e = (new == e) || (s e)
```

remElem

```
{- remove element
--
-- remElem 1 empty
-- \e -> (1 /= e) && (empty e)
-- \e -> (1 /= e) && False
--
-- remElem 1 (addElem 1 empty)
-- \e -> (1 /= e) && (addElem 1 empty)
-- \e -> (1 /= e) && ((1 == e) || (empty e))
-- \e -> (1 /= e) && ((1 == e) || False)
--
-}
remElem :: Integer -> IntSet -> IntSet
remElem a s e = (e /= a) && (s e)
```

Resources

- **Real World Haskell** by *Bryan O'Sullivan, Don Stewart, John Goerzen*.
- **Learn You a Haskell for Great Good!** by *Miran Lipovača*.
- **Haskell: The Craft of Functional Programming** by *Simon Thompson*.