UCC	Title : CS4620 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4620
	Exam Date: Friday 19th December @14.00

Assignments 2013-14

```
--assignment 1
and1 :: [Bool] -> Bool
and1 [] = True
and1 bs =  head bs == True && and1 (tail bs)

or1 :: [Bool] -> Bool
or1 bs =    not(null bs) && head bs == True || or1 (tail bs)

issorted :: [Int] -> Bool
issorted [] = True
issorted ( n : []) = True
issorted (n:ns) = n < (head ns) && issorted ns

range1 :: Int -> Int -> [Int]
range1 lo hi    | lo == hi = hi :[]
             | lo > hi = []
             | otherwise = lo : range1 (lo +1 ) hi

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

```
--assignment 2
applyAll :: [(Int -> Int)] -> Int -> [Int]
applyAll [] x = []
applyAll (f:fs) x = f x : applyAll fs x

remove :: ( Int -> Bool) -> [Int] -> [Int]
remove p [] = []
remove p (x:xs) = if p x then
                   remove p xs
                 else
                   x :remove p xs
--OR

remove1 p = foldr ( \n acc -> if p n then acc else n : acc) []

count:: Eq a => a -> [a] -> Int
count x [] = 0
count x (n:ns) = if x == n then
                 1 + count x ns
                else
                count x ns
--OR
count1 x = foldr ( \n acc -> if x == n then acc + 1 else acc) 0
```

```
--maximum n:[] = n
--maximum n:ns = filter( \x -> x > n) ns
maximums :: [Int] -> Int
maximums ns = maximums' ns 0

maximums' :: [Int] -> Int -> Int
maximums' [] z = z
maximums' (n:ns) z = if n > z then
            maximums' ns n
            else
            maximums' ns z

--OR

maximum2 ns = foldr(  \n acc -> if n > acc then n else acc) (head ns) ns

append :: [Int] -> [Int] -> [Int]
append xs ys = foldr( \x acc -> x : acc ) ys xs
```

```
--assignment 3
partialSums :: [Int] -> [Int]
partialSums [] = []
partialSums ns =   partialSums' ns 0

partialSums' :: [Int] -> Int -> [Int]
partialSums' [] _ = []
partialSums' (n:ns) acc = n + acc : partialSums' ns (n + acc)

powers :: Int -> [Int]
powers n = n : powers' n n

powers' :: Int -> Int -> [Int]
powers' n acc = n*acc : powers' n (n*acc)

--OR
powers1 :: Int -> [Int]
powers1 n = n : map( \x -> x * n) (powers1 n)

factorial :: Int -> Int
factorial 0 = 1
factorial n = factorial(n - 1) * n

factorials :: [Int]
factorials = [factorial n | n <- [1..]]

--OR
factorials1 :: [Int]
factorials1 = 1 : zipWith( \n m -> n*m) factorials1 [2..]
```

```
--assignment 4
approx :: Float -> [Float]
approx x = 1.0 : map( \n -> (n + x/n)/2) (approx x)

squareRoot :: Float -> Float
squareRoot x = squareRoot' (head (approx x)) (tail (approx x))

squareRoot' :: Float -> [Float] -> Float
squareRoot' y (z:zs) = if (abs(z-y)) < 0.0001 then
                z
               else
               squareRoot' z zs

primes :: [Int]
primes = 2: primes' [3,5..]

primes' :: [Int] -> [Int]
primes' (n:ns) = if indivisible n == [] then
               n : primes' ns
              else
               primes' ns
indivisible :: Int -> [Int]
indivisible n = [d | d <- (takeWhile(\x -> x <=
         floor( squareRoot(fromIntegral n) )) primes), mod n d == 0]

--OR

primes1 :: [Int]
primes1 = 2: [p|p <- [3,5..], isPrime p ]

--isprime n : Checks if n has zero factors
isPrime :: Int -> Bool
isPrime n = factors n == []

--factors n : Checks if values from primes less than squareRoot n are factors
-- of n
factors :: Int -> [Int]
factors n = [f|f<- (takeWhile(\x -> x <=
         floor( squareRoot(fromIntegral n) )) primes), mod n f == 0]

--assignment 5
integers :: [Int]
integers = 0 : integers' [1..]

integers' :: [Int] -> [Int]
integers' (n:ns) = n: -n : integers' ns

runs :: Eq a => [a] -> Int
runs [] = 0
```

```
runs ( x: xs) = runs' x xs 1


runs' :: Eq b => b ->  [b] -> Int -> Int
runs' acc (x:[]) result =
       if acc == x then
              result
       else
              (result + 1)

runs' acc ( x : xs ) result =
       if acc == x then
            runs' acc xs result
       else
            runs' x xs (result + 1)

occurences :: Eq a => [a] -> [(a, Int)]
occurences [] = []
occurences (x:xs) = (x,occurs x (x:xs)) : occurences (delete x xs)

occurs :: Eq a => a -> [a] -> Int
occurs x xs = length (filter(\f -> f == x)  xs)

delete :: Eq a => a -> [a] -> [a]
delete x xs = filter(\f -> f /= x ) xs
```

Other

```
-- ***********************************
-- Main Functions, Copybook
-- ***********************************

-- allDifferent xs : are all elements in the list 'xs' different?
-- Here, the equality function, /=, is used over the elements of the list. This needs to be
acknowedged in the type of
-- the function, making it include the constraint that the type a belongs to the equality class, Eq,
allDifferent :: Eq a => [a] -> Bool
allDifferent []  = True
allDifferent [_] = True
allDifferent ( x1:x2:xs ) = x1 /= x2 && allDifferent( x2:xs )


-- countMax xs : the number of times its maximum item occurs in the non-empty list 'xs'
countMax :: [Int] -> Int
countMax xs = length ( filter( \x -> x == maximum xs ) xs )
```

```
-- factorial : takes in a number 'n' and returns its factorial
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)


-- mystery(factorials) : returns list of factorials starting with 1, 1, 2, 6, 24...
-- (1 * 1, 1 * 2, 2 * 3, 6 * 4, 24 * 5...)
mystery :: [Int]
mystery = 1 : zipWith (*) mystery [1..]




square n = n * n

f n = (n * n, n * n * n)

f2 = [n * n | n <- [1 .. 6]]

f3 = [(c, n) | c <- "AB", n <- [1 .. 3]]

-- squares of all even numbers between 1 and 10
f4 = [n * n | n <- [1 .. 10], mod n 2 == 0]

f5 = [c | c <- "AB", n <- [1 .. 3]]

primes = [p | p <- [1..10], isPrime p]
isPrime n = factors n == [1, n]
factors n = [f | f <- [1..n], mod n f == 0]


boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x ]
length' xs = sum [1 | _ <- xs]



-- ***********************************
-- Handout #7 - List Comprehensions
-- ***********************************

-- square n : the square of the number 'n'
square2 :: Num a => a -> a
square2 n = n * n


-- add n1 n2 : the sum of the numbers 'n1' and 'n2'
add :: Num a => a -> a -> a
```

```
add n1 n2 = n1 + n2


-- factorial n : the factorial of the non-negative integer 'n'
factorial2 :: Int -> Int
factorial2 0 = 1
factorial2 n = factorial( n - 1 ) * n


-- sum' ns : the sum of all the elements in the numeric list 'ns'
sum'' :: Num a => [a] -> a
sum'' [] = 0
sum'' ( n:ns ) = n + sum'' ns


-- length xs : the number of elements in the list 'xs'
length'' :: [a] -> Int
length'' [] = 0
length'' ( _:xs ) = 1 + length'' xs


-- allEqual xs : are all the elements in the list 'xs' equal?
-- (Eq a => all values of a must be of equal type)
allEqual :: Eq a => [a] -> Bool
allEqual []  = True
allEqual [_] = True
allEqual ( x1:x2:xs ) = x1 == x2 && allEqual( x2:xs )


-- pimres : the infinite list of prime numbers : 2, 3, 5, 7, 11, 13, 17, ...
primes3 :: [Int]
primes3 = [ p | p <- [ 2.. ], isPrime p ]

-- isPrime n : is the integer 'n' a prime number ?
isPrime2 :: Int -> Bool
isPrime2 n = factors n == [ 1, n ]

-- factors n : the list of factors of the positive integer 'n'
factors2 :: Int -> [Int]
factors2 n = [ f | f <- [ 1..n ], mod n f == 0 ]


-- squares : squares using list comprehension
squares3 :: [Int]
squares3 = [ n * n | n <- [1, 2, 3, 4, 5] ]


-- example : list comprehension example
--example :: [ ( Char, a ) ]
```

```
example = [ ( c, n ) | c <- "ABC", n <- [ 1..3 ] ]


-- example2 : list comprehension example
example2 :: [Int]
example2 = [ n * n | n <- [ 1..10 ], mod n 2 == 0 ]






-- **********************************
-- Handout #6 - Accumulators
-- **********************************

-- sum ns : the sum of all items in the numeric list 'ns'
sum3 :: [Int] -> Int
sum3 = sum' 0

-- sum' sumSoFar ns : the sum of the number 'sumSoFar' and all items in the numeric list 'ns'
sum' :: Int -> [Int] -> Int
sum' sumSoFar ns = if null ns then
sumSoFar
else
sum' ( sumSoFar + head ns ) ( tail ns )


-- maxSlow ns : the maximum item in the non-empty numeric list 'ns'
-- ( the running time is exponential in the length of 'ns' )
maxSlow :: [Int] -> Int
maxSlow ns = if null( tail ns ) || head ns > maxSlow( tail ns ) then
head ns
else
maxSlow( tail ns )



-- maxFast ns : the maximum item in the non-empty numeric list 'ns'
-- ( the running time is linear in the length of 'ns' )
maxFast :: [Int] -> Int
maxFast ns = maxFast'( head ns ) ( tail ns )

-- maxFast' maxSoFar ns : the bigger of the number 'maxSoFar' and the maximum number in the numeric list 'ns'
maxFast' :: Int -> [Int] -> Int
maxFast' maxSoFar ns = if null ns then
maxSoFar
else
```

```
maxFast'( max maxSoFar( head ns ) ) ( tail ns )


-- fibs : the infinite list of Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, ...
fibs :: [Int]
fibs = fibs' 0 1

-- fibs' f1 f2 : the infinte list of Fibonacci numbers starting with the consequitive Fibonacci
numbers 'f1' and 'f2'
fibs' :: Int -> Int -> [Int]
fibs' f1 f2 = f1 : fibs' f2( f1 + f2 )






-- ***********************************
-- Handout #5 - Infinite Lists
-- ***********************************

-- fibsSlow : the infinite list of Fibonacci Numbers : 0, 1, 1, 2, 3, 5, 8, ...
fibsSlow :: [Int]
fibsSlow = map fib( [1..] )

-- fib n : the 'n'th Fibonacci number, for any positive integer 'n'
fib :: Int -> Int
fib n = if n == 1 then
0
else if n == 2 then 1
else
fib( n - 1 ) + fib( n - 2 )


-- fibsSlow : the infinite list of Fibonacci Numbers : 0, 1, 1, 2, 3, 5, 8, ...
fibsFast :: [Int]
fibsFast = 0 : 1 : zipWith( \f1 -> \f2 -> f1 + f2 ) fibsFast( tail fibsFast )


-- dropMultiples d ns : the numeric list 'ns' with all multiples of 'd' removed
dropMultiples :: Int -> [Int] -> [Int]
dropMultiples d = filter( \n -> mod n d /= 0 )


-- sieve ns : the result of applyine the sieve of Eratosthenes to the list 'ns'
sieve :: [Int] -> [Int]
sieve [] = []
sieve ns = head ns : sieve( dropMultiples( head ns ) ( tail ns ) )
```

```
-- primes : the infinite list of prime numbers : 2, 3, 5, 7, 11, 13, 17, ...
primes2 :: [Int]
primes2 = sieve( [2..] )


-- primesB40 : the primes below 40
primesB40 :: [Int]
primesB40 = takeWhile( \p -> p <= 40 ) primes2


-- primes100 : the 100th prime
primes100 :: Int
primes100 = head( drop 99 primes2 )


-- primeA1000 : the first prime above 1000
primeA1000 :: Int
primeA1000 = head( dropWhile( \p -> p <= 1000 ) primes2 )





-- ***********************************
-- Handout #4 - Higher Order Functions
-- ***********************************

-- zipwith f xs ys : the list formed by applying function 'f' to pairs
--                                       of corresponding components in lists 'xs' and 'ys'
--                                              stopping as soon as either list runs out
zipWith2 :: Num a => ( a -> b -> c ) -> [a] -> [b] -> [c]
zipWith2 f xs ys = if null xs || null ys then []
else
f( head xs ) ( head ys ) : zipWith2 f( tail xs ) ( tail ys )


-- take n xs : the list of the first 'n' components of 'xs', or 'xs' itself if 'n' exceeds its length
take2 :: Int -> [a] -> [a]
take2 n xs = if n <= 0 || null xs then [] else head xs : take2( n - 1 ) ( tail xs )


-- drop n xs : the list 'xs' with the first 'n' components removed, or the empty list if 'n' exceeds its
length
drop2 :: Int -> [a] -> [a]
drop2 n xs = if n <= 0 || null xs then xs  else drop( n - 1 ) ( tail xs )
```

```
-- takewhile p xs : the longest prefix of 'xs' whose components all satisfy predicate 'p'
takewhile :: ( a -> Bool ) -> [a] -> [a]
takewhile p xs = if null xs || not( p( head xs ) ) then [] else head xs : takewhile p ( tail xs )


-- dropwhile p xs : the longest suffix of 'xs' whose first component does not satisfy predicate 'p'
dropwhile :: ( a -> Bool ) -> [a] -> [a]
dropwhile p xs = if null xs || not( p( head xs ) ) then xs else dropwhile p ( tail xs )



-- ***********************************
-- Handout #3 - Higher Order Function
-- ***********************************


-- foldr f v xs : the result of appending item 'v' to the right end of the list 'xs'
-- and then cumulatively applying the two-parameter function 'f' from
-- right to left on this augmented list
-- ( A function in Haskell must always return values of the same type )
-- ( ( a -> b -> b ) -> b  ==> Last b  represents value of v )
-- ( [a]  ==> represents value of xs )
-- ( Last b  ==> return type fo foldr( value is whatever v returns ) )

foldr2 :: ( a -> b -> b ) -> b -> [a] -> b
foldr2 f v xs = if null xs then v else f( head xs ) ( foldr2 f v ( tail xs ) )


-- length xs : the number of components in the list 'xs'
-- ( xs not actually needed in computation but makes definition syntactically correct => xs is the
next ( head ) element )
-- ( xs goes in on the far right when the function is being called )
length3 :: Num a => [a] -> a
length3 = foldr( \xs -> \acc -> acc + 1 ) 0


-- map f xs : the list formed by applying function 'f' to each component of list 'xs'
map3 :: ( a -> b ) -> [a] -> [b]
map3 f = foldr( \x -> \acc -> f x : acc ) []


-- filter p xs : the list formed by those components of list 'xs' which satisfy predicate 'p'
filter3 :: ( a -> Bool ) -> [a] -> [a]
filter3 p = foldr( \x -> \acc -> if p x then x : acc else acc ) []


-- sum ns : the sum of all items in the numeric lis 'ns'
sum2 :: [Int] -> Int
sum2 = foldr( \n1 -> \n2 -> n1 + n2 ) 0
```

```
-- product2 ns : the product of all items in the numeric list 'ns'
product2 :: [Int] -> Int
product2 = foldr( \n1 -> \n2 -> n1 * n2 ) 1


-- and bs : do all components of the boolean list 'bs' equal True ?
and2 :: [Bool] -> Bool
and2 = foldr( \b1 -> \b2 -> b1 && b2 ) True


-- or bs : does any component of the list 'bs' equal True ?
or2 :: [Bool] -> Bool
or2 = foldr( \b1 -> \b2 -> b1 || b2 ) False


-- all p xs        : do all componenets of the list 'xs' satisfy predicate 'p'
all2 :: ( a -> Bool ) -> [a] -> Bool
all2 p xs = and2( map p xs )


-- any p xs : does any component of the list 'xs' satisfy predicate 'p'
-- ( p takes in a, and returns a Bool )
any2 :: ( a -> Bool ) -> [a] -> Bool
any2 p xs = or2( map p xs )



-- element x xs : does item 'x' occur in list 'xs'
element2 :: Eq a => a -> [a] -> Bool
element2 x xs = any2( \e -> e == x ) xs







-- ***********************************
-- Handout #2 - Higher Order Functions
-- ***********************************

-- map f xs : the list formed by applying function 'f' to each component of the list 'xs'
map2 :: ( a -> b ) -> [a] -> [b]
map2 x xs = if null xs then [] else f( head xs ) : map2 f ( tail xs )


-- filter p xs : the list formed by the components of list 'xs' which satisfy predicate 'p'
```

```
-- ( filter takes in ( a and returns a bool ), and takes in [a] and returns [a] )
filter2 :: ( a -> Bool ) -> [a] -> [a]
filter2 p xs = if null xs then [] else if p( head xs ) then
head xs : filter2 p ( tail xs ) else filter2 p ( tail xs )


-- doublelist xs : the list formed by doubling each number in the list 'xs'
doublelist :: [Int] -> [Int]
doublelist = map( \x -> 2 * x )


-- fliplist xs : the list formed by negating each boolean in the list 'xs'
fliplist :: [Bool] -> [Bool]
fliplist = map not


-- positives xs : the list of positive numbers in the list 'xs'
positives :: [Int] -> [Int]
positives = filter( \x -> x > 0 )


-- multiples d : the list of numbers in the list 'xs' which are divisible by 'd'
multiples :: Int -> [Int] -> [Int]
multiples d = filter( \x -> mod x d == 0 )



-- ***********************************
-- Handout #1 - Simple Recursion
-- ***********************************

-- null ns : is the list 'ns' empty
null2 :: Eq a => [a] -> Bool
null2 xs = xs == []


-- length xs : the number of components
length2 :: [a] -> Int
length2 xs = if null xs then 0 else 1 + length( tail xs )


-- element x xs : does the item 'x' exist in list the 'xs' ?
element :: Eq a => a -> [a] -> Bool
element x xs = not ( null xs )
&& ( ( head xs == x ) || element x ( tail xs ) )


-- count x xs : the number of times that the item 'x' occurs in the list 'xs'
count :: Eq a => a -> [a] -> Int
```

```
count x xs = if null xs then 0
else if head xs == x then 1 + count x (tail xs)
else count x (tail xs)


-- append xs ys : the list formed by joining the lists 'xs' and 'ys'
append :: [a] -> [a] -> [a]
append xs ys = if null xs then
ys else head xs : append( tail xs ) ys


-- ***********************************
-- Assignment 5
-- ***********************************


-- frequencies xs : the list of tuples of distinct items and their individual
--             frequencies in list 'xs'
frequencies :: Eq a => [a] -> [(a, Int)]
frequencies xs = [ (a,b) | a <- rmDuplicates xs, b <- [numOccurences xs a] ]

-- numOccurences xs l : counts the number of occurrences of the letter 'l'
--               in the list 'xs'
numOccurences :: Eq a => [a] -> a -> Int
numOccurences xs l = length (filter (\letter -> letter == l) xs)

-- rmDuplicates xs : the list formed by removing duplicate values from 'xs'
rmDuplicates :: Eq a => [a] -> [a]
rmDuplicates [] = []
rmDuplicates (x:xs) = x : rmDuplicates (filter (\value -> not(x == value)) xs)


-- pytrips : The (infinite) list of Pythagorean Triples of 3-tuple positive
--        integers (x, y, z), and where x and y have no common factor
pytrips :: [(Int, Int, Int)]
pytrips = [(x,y,z) | z<-[1..], y<-[1..z], x<-[1..y], x^2+y^2==z^2, gcd x y==1]


-- as x : Returns an infinite list [ a1, a2, a3, . . . ] of approximations
--  which converge to the square root of 'x' where 'x' > 0
as :: Float -> [Float]
as x = 1.0 : as' 1.0 x

-- as' num1 num2 : Returns an infinite list [ a1, a2, a3, . . . ] of
-- approximations which converge to the square root of 'x'
-- where 'num1' & 'num2' are used to calculate the square root
as' :: Float -> Float -> [Float]
as' num1 num2 = (num1 + num2/num1) / 2.0 : as' ((num1 + num2/num1) / 2.0) num2
```

```
-- squareroot x : Returns the square root of 'x' where 'x' > 0
squareroot :: Float -> Float
squareroot x = cmp 0 (sqroot 1.0 x)

-- cmp pos list : compares two items in 'list' at position 'pos' & 'pos+1'
cmp :: Int -> [Float] -> Float
cmp pos list = if abs(list !! pos - list !! (pos+1)) < 0.0001 then
                                                list !! (pos+1)
                                               else
                                                cmp (pos+1) list

-- sqroot x : Returns the square root of 'x' where 'x' > 0 and where
--          'num1' & 'num2' are used to calculate the square root
sqroot :: Float -> Float -> [Float]
sqroot num1 num2 =(num1 + num2/num1)/2.0 : sqroot ((num1 + num2/num1)/2.0) num2



-- ***********************************
-- Assignment 4
-- ***********************************


-- powers n : the list of all positive powers of 'n'
-- ( "two parameters are enough" )
powers3 :: Int -> [Int]
powers3 n = powers' n ( n * n ) n

-- powers' p1 p2 p3 : the infinite list of all positive powers of 'p3'
--                starting with 'p1' and 'p2'
powers' :: Int -> ( Int -> ( Int -> [Int] ) )
powers' p1 p2 p3 = p1 : powers' p2 ( p2 * p3 ) p3




-- factorials : the list of factorials of all positive integers
-- ( "two parameters are enough" )
factorials3 :: [Int]
factorials3 = factorials' 1 2 3

-- factorials' f1 f2 f3 : the infinite list of all positive factorial integers
--                  starting with the consecutive numbers 'f1', 'f2'
--                  and 'f3'
factorials' :: Int -> ( Int -> ( Int -> [Int] ) )
factorials' f1 f2 f3 = f1: factorials' f2 ( f2 * f3 ) ( f3 + 1 )
```

```
-- runs ns : the number of blocks of adjacent equal items in the finite
--         numeric list 'ns'
runs :: [Int] -> Int
runs ns = runs' 1 ns

-- runs' adjSoFar ns : the number of blocks of adjacent equal items in 'adjSoFar'
--                                                  in the finite numeric list 'ns'
runs' :: Int -> ( [Int] -> Int )
runs' adjSoFar ns = if null ns then
                      0
                    else if null( tail ns ) then
                      adjSoFar
                    else
                      if head ns == head( tail ns )
                        then runs' adjSoFar ( tail ns )
                      else
                        runs' ( adjSoFar + 1 ) ( tail ns )



-- **********************************
-- Assignment 3
-- **********************************

-- partialSums ns : the list of partial sums on the numeric list 'ns'
-- ( "theres a way to avoid null ns test, and still handle empty lists - can you find it?" )
partialSums :: [Int] -> [Int]
partialSums ns = if null ns then [] else head ns : zipWith( \n1 -> \n2 -> n1 + n2 ) ( partialSums ns ) (
tail ns )


-- powers n : the list of all positive powers of the number 'n'
powers :: Int -> [Int]
powers n = n : map( \n1 -> n1 * n ) ( powers n )


-- factorials : the list of factorials of all positive integers
-- ( "can be simplified further - its easy!" )
factorials :: [Int]
factorials = 1 : 2 : zipWith( \n1 -> \n2 -> n1 * n2 ) ( tail factorials ) [3..]




-- **********************************
-- Assignment 2
-- **********************************
```

```
-- applyAll fs x : the list formed by applying each function in the function list 'fs' on the item 'x'
--applyAll :: ( [a] -> a ) -> a -> [a]
applyAll fs x = map( \f -> f x ) fs


-- remove p xs : the list formed by those components of list 'xs' which do not satisfy predicate 'p'
-- ( xs taken in at the right side of function when function is called )
remove :: ( a -> Bool ) -> [a] -> [a]
remove p = filter( \n -> not ( p n ) )


-- count x xs : the number of times the item 'x' occurs in the list 'xs'
count2 :: Eq a => a -> [a] -> Int
count2 x = foldr ( \xs -> \acc -> if x == xs then acc +1 else acc ) 0


-- max ns : the maximum number in the non-empty numeric list 'ns'
-- ( must also check for negative number lists eg. max ( -5, -2 ) => 0 ... should be -2? )
max2 :: [Int] -> Int
max2 = foldr ( \ns -> \acc -> if ns > acc then ns else acc ) 0


-- append xs ys : the list formed by joining the list 'xs' and 'ys' in that order
append2 :: [a] -> [a] -> [a]
append2 xs ys = foldr ( \x -> \acc -> x:acc ) ys xs



-- ***********************************
-- Assignment 1
-- ***********************************

-- last xs : takes in a non-empty list 'xs' & returns the rightmost component
last2 :: [a] -> a
last2 xs = if null ( tail xs ) then head xs
                        else last2( tail xs )


-- issorted xs : checks if list 'xs' is sorted
issorted :: Ord a => [a] -> Bool
issorted xs = if xs == [] || tail xs == [] then True
                              else if head xs > head( tail xs ) then False
                              else issorted( tail xs )


-- range lo hi : the list from 'lo' to 'hi' inclusive
range :: Int -> Int -> [Int]
range lo hi = if lo > hi then []
```

```
        else lo : range( lo + 1 ) hi


-- copies n x : the value of 'x' exactly 'n' times
copies :: Int -> a -> [a]
copies n x = if n == 0 then []
        else x : copies( n - 1 ) x
```

Exam Paper Questions

Static (Compile Time) Dynamic (Real Time)	• The allocation of memory for the specific fixed purposes of a program in a predetermined fashion controlled by the compiler is said to be static memory allocation. • The allocation of memory (and possibly its later de-allocation) during the running of a program and under the control of the program is said to be dynamic memory allocation. Advantages / Disadvantages • Advocates of static typing argue that the advantages of static typing include earlier detection of programming mistakes (e.g. preventing adding an integer to a Boolean), better documentation in the form of type signatures (e.g. incorporating number and types of arguments when resolving names), more opportunities for compiler optimizations (e.g. replacing virtual calls by direct calls when the exact type of the receiver is known statically), increased runtime efficiency (e.g. not all values need to carry a dynamic type), and a better design time developer experience (e.g. knowing the type of the receiver, the IDE can present a drop-down menu of all applicable members). Static typing fanatics try to make us believe that “well-typed programs cannot go wrong”. While this certainly sounds impressive, it is a rather vacuous statement. Static type checking is a compile-time abstraction of the runtime behaviour of your program, and hence it is necessarily only partially sound and incomplete. This means that programs can still go wrong because of properties that are not tracked by the type-checker, and that there are programs that while they cannot go wrong cannot

	be type-checked. The impulse for making static typing less partial and more complete causes type systems to become overly complicated and exotic as witnessed by concepts such as "phantom types" [11] and "wobbly types" [10]. This is like trying to run a marathon with a ball and chain tied to your leg and triumphantly shouting that you nearly made it even though you bailed out after the first mile. • Advocates of dynamically typed languages argue that static typing is too rigid, and that the softness of dynamically languages makes them ideally suited for prototyping systems with changing or unknown requirements, or that interact with other systems that change unpredictably (data and application integration). Of course, dynamically typed languages are indispensable for dealing with truly dynamic program behaviour such as method interception, dynamic loading, mobile code, runtime reflection, etc. In the mother of all papers on scripting [16], John Ousterhout argues that statically typed systems programming languages make code less reusable, more verbose, not safer, and less expressive than dynamically typed scripting languages. This argument is parroted literally by many proponents of dynamically typed scripting languages. We argue that this is a fallacy and falls into the same category as arguing that the essence of declarative programming is eliminating assignment. Or as John Hughes says [8], it is a logical impossibility to make a language more powerful by omitting features. Defending the fact that delaying all type-checking to runtime is a good thing, is playing ostrich tactics with the fact that errors should be caught as early in the development process as possible.
Curried Functions	Currying is when you break down a function that takes multiple arguments into a series of functions that take part of the arguments. Here's an example in Scheme (define (add a b) (+ a b)) (add 3 4) returns 7 This is a function that takes two arguments, a and b, and returns their sum. We will now curry this function: (define (add a) (lambda (b) (+ a b))) This is a function that takes one argument, a, and returns a function that takes another argument, b, and that function returns their sum.

	((add 3) 4) (define add3 (add 3)) (add3 4) The first statement returns 7, like the (add 3 4) statement. The second statement defines a new function called add3 that will add 3 to its argument. This is what some people may call a closure. The third statement uses the add3 operation to add 3 to 4, again producing 7 as a result. Curried functions are easier to read, has clearer intent and provides shorter code. You also get easier reuse of more abstract functions because it lets you specialize/partially apply functions using a lightweight syntax and then pass these partially applied functions around to higher order function such as map or filter. Higher order functions (which take functions as parameters or yield them as results) are the bread and butter of functional programming, and currying and partially applied functions enable higher order functions to be used much more effectively and concisely.
Lazy Evaluation	Lazy evaluation means that expressions are not evaluated when they are bound to variables, but their evaluation is deferred until their results are needed by other computations. In consequence, arguments are not evaluated before they are passed to a function, but only when their values are actually used. Technically, lazy evaluation means Non-strict semantics and Sharing. A kind of opposite is eager evaluation. Non-strict semantics allows one to bypass undefined values (e.g. results of infinite loops) and in this way it also allows one to process formally infinite data. When it comes to machine level and efficiency issues it is important whether or not equal objects share the same memory. A Haskell program cannot know whether 2+2 :: Int and 4 :: Int are different objects in the memory. In many cases it is not necessary to know it, but in some cases the difference between shared and separated objects yields different orders of space or time complexity.
Eager Haskell	Eager Haskell is an implementation of the Haskell programming language which by default uses eager evaluation. The four widely available implementations of Haskell---ghc, nhc, hugs, and hbc---all use lazy evaluation. The design of Eager Haskell permits arbitrary Haskell programs to be compiled and run

	eagerly. This web page highlights some of the techniques we are using to accomplish this.
Prolog computes Relations	In Prolog, program logic is expressed in terms of relations, and a computation is initiated by running a query over these relations. Relations and queries are constructed using Prolog's single data type, the term.[4] Relations are defined by clauses. Given a query, the Prolog engine attempts to find a resolution refutation of the negated query. If the negated query can be refuted, i.e., an instantiation for all free variables is found that makes the union of clauses and the singleton set consisting of the negated query false, it follows that the original query, with the found instantiation applied, is a logical consequence of the program. This makes Prolog (and other logic programming languages) particularly useful for database, symbolic mathematics, and language parsing applications. Because Prolog allows impure predicates, checking the truth value of certain special predicates may have some deliberate side effect, such as printing a value to the screen. Because of this, the programmer is permitted to use some amount of conventional imperative programming when the logical paradigm is inconvenient. It has a purely logical subset, called "pure Prolog", as well as a number of extralogical features. Predicates in Prolog define relations rather than functions.
atLeast	Autumn 2013 AtLeast n xs :: Int -> [a] -> Bool AtLeast n xs = length xs >= n a) Two Problems (10%) • Non-recursive but inefficient because length is required, so it is an additional count. • Checking that it is a specific length when all that is required is that it is a certain length i.e. greater than 0 b) New definition (20%) `atLeast n (_:xs) = atLeast (n-1) xs`
Recursive Function – Integer n and returns the increasing list of all integers from n onwards: From 5 => [5, 6, 7, 8, …]	`from n = n : map (+1) from`
doubleList	doubleList = \xs -> map(\n -> 2 * n) xs
flipList	flipList = \ns -> if null ns then [ ] else (head ns) : flipList (tail ns)

Map	map = \f -> foldr (\x -> \acc -> fx : acc) [ ]
Change Interpreter for Factor from / to : :name definition {definition} name	Use LET to make the assignment factor : : Int -> Int factor 1 = [] factor n = let prime = head $ dropWhile ((/= 0) . mod n) [2 .. n] in (prime :) $ factor $ div n prime