#	time

row	col

UNIVERSITY OF ARIZONA

DEPARTMENT OF COMPUTER SCIENCE

CSc372 Comparative Programming Languages

Midterm Exam 2 — Wednesday March 25, 9:30, 2020

(Time allowed: 75 minutes)

①	Sum		30
②	Range		20
③	Strip		8
④	Takeskip		8
⑤	Vecprod		8
⑥	From Java to Haskell		8
⑥	Split		8
⑦	Frequency		10
Total			100

Rules that apply during this exam:

1. **You can use the Haskell compiler (ghci) and a text editor to work the problems. You should edit the file `exam.hs` to enter your answers.**
2. **You cannot use or search the Internet.**
3. **You cannot communicate with anyone over phone, the Internet, email, in person, etc.**
4. **You may have your phone next to you but you can only use it in an emergency.**
5. **You should be alone in your room when you take the exam.**
6. **You should be logged into Zoom throughout the exam, with video turned on, audio turned off. Join one of three rooms, depending on the first letter of your last name:**
   (a) **A-J (Christian):** `https://arizona.zoom.us/j/982017165`
   (b) **K-R (Wenkang):** `https://arizona.zoom.us/j/318806041`
   (c) **S-Z (Claire):** `https://arizona.zoom.us/j/478386720`
7. **This is a closed book exam: no notes, no books, etc.**
8. **You may print out the exam.**
9. **You may have one blank piece of scratch paper and a pen/pencil.**
10. **If you have questions, use Zoom chat to ask them. Keep in mind that everyone can see these chats, so don't write anything that reveals an answer.**
11. **If you get knocked off the Internet during the exam, don't panic and spend 10 minutes figuring out what went wrong! Text me at 520-271-4960 to explain what happened and continue taking the exam.**
12. **When you're done, submit `exam.hs` only.**
    (a) **Primarily, upload `exam.hs` to d2l.**
    (b) **If that doesn't work, email `exam.hs` to me (`collberg@gmail.com`).**
    (c) **If that doesn't work, take a picture of `exam.hs` with your phone and text it to me at 520-271-4960.**

    Don't go over time!
13. **Before you submit the file `exam.hs`, make sure that there are no syntax errors! We will both run your functions and view them manually.**
14. **You *do not* have to turn in this file, `exam.pdf`.**

By submitting the file `exam.hs` you certify that you followed the rules above. Even though it's easier to cheat under these circumstances, you are still bound by the university's and this course's codes of academic integrity.

The function `sum xs` has the signature

```
sum :: [Int] -> Int
```

and computes the sum of a list of integers:

```
> sum []
0
> sum [1..5]
15
```

1. Write a recursive version of `sum`, called `sum1`, using the `if-then-else` syntax. /10
2. Write a recursive version of `sum`, called `sum2`, using the *guard* syntax. /10
3. Write a recursive version of `sum`, called `sum3`, using the *pattern* syntax. /10

The function `range f t` has the signature

```
range :: Int -> Int -> [Int]
```

and computes the list of integers from `f` to `t`, inclusive:

```
> range 1 0
[]
> range 2 2
[2]
> range 2 5
[2,3,4,5]
> range 1 10
[1,2,3,4,5,6,7,8,9,10]
```

1. Write a recursive version of `range`, called `range1`, using the `if-then-else` syntax. /10

2. Write a recursive version of `range`, called `range2`, using the *guard* syntax. /10

The function `strip x xs` has the signature

```
strip :: Char -> String -> String
```

and removes every instance of the first argument in the second. Here are some examples

```
> strip 'a' ""
""
> strip 'a' "a"
""
> strip 'a' "aaaaabbbbaaaa"
"bbbb"
```

1. Give the *function signature* for a version of `strip` that can take *any* type of arguments. For example, `strip` could be called like this: /8

   ```
   > strip 4 [1,2,3,4,5,4,4,7]
   [1,2,3,5,7]
   > strip3 'a' "aaabbbaaa"
   "bbb"
   > strip3 True [True,False,False,True]
   [False,False]
   ```

   You don't need to write the function body.

Write a recursive function `takeskip xs ys` that extracts elements from `ys` depending on the specification given in `xs`.

`xs` is a list of pairs `[...,("take"/"skip",`v`),...]` where the first element of the pair (either `"take"` or `"skip"`) defines whether to take or skip the next v elements from `ys`.

Use *pattern syntax*. You may use `take` and `drop`.

Examples: /8

```
> takeskip [("take",3)] [1..10]
[1,2,3]
> takeskip [("skip",3)] [1..10]
[]
> takeskip [("skip",3),("take",3)] [1..10]
[4,5,6]

> takeskip [("take", 3),("skip",3),("take",2)] [1..4]
[1,2,3]
> takeskip [("take", 3),("skip",3),("take",2)] [1..20]
[1,2,3,7,8]

> takeskip [("take", 3),("skip",3),("take",2)] "abcdefghijklmnopqrstuvwxyz"
"abcgh"
> takeskip [("take", 3),("skip",3),("take",9)] "abcdefghijklmnopqrstuvwxyz"
"abcghijklmno"
```

Write a function `vecprod` a b that takes two lists of numbers as input and computes their vector product, i.e. $a_0 * b_0 + a_1 * b_1 + \ldots$. Use *pattern syntax*. Throw an exception if the lists are not of the same length. $\boxed{\quad\quad /8}$

The signature is:

```
vecprod :: Num a => [a] -> [a] -> a
```

Examples:

```
> vecprod [1,2,3] [1,2,3]
14
> vecprod [1.0,2.0,3.0] [1.0,2.0,3.0]
14.0
> vecprod [111111111111,22222222222222] [333333333333,44444444444444]
987691358024671530864197531
> vecprod [1,2,3] [1,3]
*** Exception: unqual lengths
```

Here's a Java program which prints out the numbers in the *Collatz* sequence (also know as *hailstones sequence*): /8

```
public class Collatz {
    public static void collatz(int n) {
        StdOut.print(n + " ");
        if (n == 1) return;
        else if (n % 2 == 0) collatz(n / 2);
        else collatz(3*n + 1);
    }

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        collatz(n);
        StdOut.println();
    }

}
```

Write the corresponding function in Haskell, but, instead of printing out the number sequence you should return a list of the numbers:

```
> collatz 6
[6,3,10,5,16,8,4,2,1]
> collatz 35
[35,106,53,160,80,40,20,10,5,16,8,4,2,1]
```

- Don't forget to provide the function signature.
- Don't use `if-then-else` syntax, use *patterns* or *guards* or both.

1. As part of a QuickSort routine, we want a function `split x xs` which splits a list `xs` in three parts: one list with all the elements less than `x`, one list with the elements equal to `x`, and one with the elements greater than `x`. `split` should have the following signature: /8

```
split:: (Eq a,Ord a) => a -> [a] -> ([a],[a],[a])
```

Examples:

```
> split 5 [1..10]
 ([1,2,3,4],[5],[6,7,8,9,10])
> split 5 [4,7,5,5,9,5,1,10,45,2,5]
([4,1,2],[5,5,5,5],[7,9,10,45])
> split 5 []
([],[],[])
> split "cc" ["ab","a","cc","b","ddd","ee"]
(["ab","a","b"],["cc"],["ddd","ee"])
```

You can write `split` any way you want, but in order to get full marks you can only traverse the input list *once.* (HINT: to do this, you may want to add a helper function with an extra argument.)

Write a function `freq xs` that takes a list of `Integer`s `xs` as input and returns a list of pairs (v, f) where v is a value from `xs` and f is the number of times it occurs in `xs`. The signature of `freq` should be: /10

```
freq :: [Integer] -> [(Integer,Int)]
```

Here are some examples:

```
> freq []
[]
> freq [1,2,3,4]
[(1,1),(2,1),(3,1),(4,1)]
> freq [1,2,3,4,1,2,3,4,1,1,1]
[(1,5),(2,2),(3,2),(4,2)]
> freq [1,2,332423423423,4,332423423423,332423423423]
[(1,1),(2,1),(332423423423,3),(4,1)]
```

Note that the list elements can be arbitrarily large. Your function must finish within a reasonable amount of time even for large values!

This function is best written by starting with a few helper functions. For full points, you should include these using a `where` clause.

DONE!

Useful Functions from the Haskell Standard Prelude

```
fst,snd           :: (a,b) -> a
fst (x,_)         = x
snd (_,y)         = y

even :: Integral a => a -> Bool

id               :: a -> a
id     x          = x

(.)              :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x         = f (g x)

head,last        :: [a] -> a
head (x:_)          = x

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

tail,init        :: [a] -> [a]
tail (_:xs)         = xs

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

null               :: [a] -> Bool
null []             = True
null (_:_)          = False

(++)               :: [a] -> [a] -> [a]
[]      ++ ys       = ys
(x:xs) ++ ys        = x : (xs ++ ys)

map                :: (a -> b) -> [a] -> [b]
map f [ ]           = [ ]
map f (x:xs)        = f x : map f xs

filter             :: (a -> Bool) -> [a] -> [a]
filter _ []         = []
filter p (x:xs)
        | p x          = x : filter p xs
        | otherwise    = filter p xs

concat             :: [[a]] -> [a]
concat              = foldr (++) []

length             :: [a] -> Int
length              = foldl (\x _ ->x+1) 0

(!!)               :: [a] -> Int -> a
(x:_)  !! 0         = x
```

```
(_:xs) !! n | n>0 = xs !! (n-1)
(_:_)  !! _       = error "Prelude.!!: negative index"
[]     !! _       = error "Prelude.!!: index too large"

foldl             :: (a -> b -> a) -> a -> [b] -> a
foldl f z []       = z
foldl f z (x:xs)   = foldl f (f z x) xs

foldr             :: (a -> b -> b) -> b -> [a] -> b
foldr f z []       = z
foldr f z (x:xs)   = f x (foldr f z xs)

iterate           :: (a -> a) -> a -> [a]
iterate f x        = x : iterate f (f x)

take                 :: Int -> [a] -> [a]
take n _   | n <= 0   = []
take _ []             = []
take n (x:xs)         = x : take (n-1) xs

drop                 :: Int -> [a] -> [a]
drop n xs | n <= 0    = xs
drop _ []             = []
drop n (_:xs)         = drop (n-1) xs

zip               :: [a] -> [b] -> [(a,b)]
zip                = zipWith  (\a b -> (a,b))

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

takeWhile :: (a->Bool) -> [a] -> [a]
takeWhile p [ ] = [ ]
takeWhile p (x:xs)
     | p x          = x : takeWhile p xs
     | otherwise    = [ ]

dropWhile :: (a->Bool) -> [a] -> [a]
dropWhile p [ ] = [ ]
dropWhile p (x:xs)
     | p x          = dropWhile p xs
     | otherwise    = x:xs

flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x

until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x      = if p x then x else until p f (f x)

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

```
and,or :: [Bool] -> Bool

ord :: Char -> Int
chr :: Int -> Char
toUpper, toLower :: Char -> Char
isAscii,isDigit  :: Char -> Bool
isUpper,isLower  :: Char -> Bool

sum, product :: Num a => [a] -> a
```