Question 4 (20 points + 10pt bonus!)

For each of the following functions, write down a short description of what `foo` does, and some output examples.

(a)

```
foo :: [Int] -> [Int]
foo l = [ x * x | x <- l, x > 0 ]
```

Example answer:
Calculates the squares of all positive numbers in a list.
foo [] = []
foo [1,0,2,-1] = [1, 4]

(b)

```
foo :: [Int] -> [Int]
foo l = [ (x,y) | x <- l, y <- l, x /= y]
```

Answer:
Creates list of tuples of elements of l. Its the cartesian product without tuples in the form (a, a)
foo [1,2,3] = [(1,2), (1,3), (2,1), (2,3), (3,1),(3,2)]

(c)

```
foo :: a -> [a] -> [a]
foo x l = reverse (x : reverse l)
```

Answer:
appends x to end of list
foo 4 [1,2,3] = [1,2,3,4]

(d)

```
bar          :: (a -> Maybe b) -> [a] -> [b]
bar _ []     = []
bar f (x:xs) =
  let rs = bar f xs in
  case f x of
    Nothing -> rs
    Just r  -> r:rs

foo :: [Maybe a] -> [a]
foo = bar id
```

Answer:
Returns a list of the Just values of the parameter.
foo [Just 1, Just 2, Just 4, Nothing] = [1,2,4]

(e)

```
foo :: [Int] -> Int -> (Int, [Int])
foo [] m = (m, [])
foo [x] m = (x, [m])
foo (x : xs) m = (max m' x, m : xs')
  where (m', xs') = foo xs m
```

6 3, 7 : [7]
6 [7] [6] 7

Answer:
returns a tuple with the first element as the biggest element in the first argument (list) and the second element of the tuple is a list where each element is the second argument and has a length that is equal to the length of the first argument
foo [3,6] 7 = (6, [7,7])
foo [3,4,5] 7 = (5, [7,7,7])
foo [6] 7 = (6, [7])

(f) *Bonus!*

```
dropWhileM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
dropWhileM _ []     = return []
dropWhileM p (x:xs) = do
    q <- p x
    if q then dropWhileM p xs else return (x:xs)

foo :: ??
foo = dropWhileM (const [True, False])
```

Answer:

foo :: [a] → [[a]]

foo [1,2,3] =

Question 5 (20 points + 10pt bonus!)

Implement the following Haskell functions:

(a) Implement a function `weave` that given two lists with elements of the same type, returns a list with elements alternating between the two lists. For example:

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

You can assume that the lists have the same length ✓

```
weave :: [a] -> [a] -> [a]
weave [] [] = []
weave (x:xs) (y:ys) = x:y: weave xs ys
```

(b) Implement a function `toMax`, that given a non-empty list of integers, returns a list of the same length, where each element has been replaced by the maximum element of the list. For example:

```
toMax [1,4,2,5,3] = [5,5,5,5,5]
```

You can use the `foo` function of problem (4e) if it helps.

BONUS: Implement `toMax` so that it only traverses a list once!

```
toMax :: [Int] -> [Int]
toMax lst = replicate (maximum lst) (length lst)
```

Typeclass Definitions

```
class Semigroup a where
  (<>)    :: a -> a -> a

class Semigroup m => Monoid m where
  mempty :: m

class  Show a  where
  show    :: a   -> String

class  Eq a  where
  (==)   :: a -> a -> Bool
  (/=)   :: a -> a -> Bool

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

class Functor f where
  fmap   :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure   :: a -> f a
  (<*>)  :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m

class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]

class Num a where
  (+), (-), (*)        :: a -> a -> a
  negate               :: a -> a
  abs                  :: a -> a
  signum               :: a -> a
  fromInteger          :: Integer -> a
```

Susan
Wen

CMSC 488B: Midterm Exam (Spring 2022)

Question 1 (20 points)

Part 1 - 8pts

For each of the following questions, select the appropriate response.

(a) There exist OCaml programs that don't terminate, whose Haskell equivalents do terminate.

- (i) True
- (ii) False

(b) The constraint `Semigroup a => Monoid a` implies that:

- (i) Every type that has a `Semigroup` instance, also has a `Monoid` instance.
- (ii) Every type that has a `Monoid` instance, also has a `Semigroup` instance.
- (iii) Both of the above.
- (iv) None of the above.

(c) Typeclass laws are enforced by the Haskell compiler.

- (i) True
- (ii) False

(d) Given a function `foo :: Int -> Char -> Bool`, consider the function call `quickCheck foo`. Select what will happen if that was inside a `main` function:

- (i) The program will fail to typecheck.
- (ii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for `Int` and `Char` to generate inputs and test `foo`.
- (iii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for () as default to generate inputs and test `foo`.
- (iv) Haskell will require user-provided generators for integers and characters in order to run the tests

Part 2 - 12pts

The standard library defines `foldMap` with the following type:

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

For each of the following `foldMap` calls, select all options that are true:

(a) `foldMap (:[]) [1..10]`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is [] - the instance for lists.

(b) `foldMap show "123456"` = foldMap show ['1', '2', ... '6']

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is [] - the instance for lists.

Question 2 (20 points)

For each of the Haskell expressions below, write their (most general) Haskell type or "ill-typed" if it contains a type error. The type signatures of all functions below are provided in the appendix at the end.

(a) `\x -> x`
Example answer:
`a -> a`

(b) `\x y -> (x,y)`

a → b → (a, b)

(c) `\x y -> if x == y then show x else show (x,y)`

(Show a, Eq a) ⇒ a → a → String

(d) `\x l -> x : l ++ l ++ [x]`

a → [a] → [a]

(e) `getLine >>= putStrLn`

IO ()

(f) `putStrLn 42 >>= putStrLn 43`

ill-typed

(g) `(,) "42"`

a → (String, a)

(h) `reverse . foldMap return`

[a] → [Maybe a]

(i) `\l -> [(x,y) | x <- l, y <- l, x /= y]`

Eq a ⇒ [a] → ~~[(a)]~~ [(a,a)]

(j) `let f x = x in (f 'a', f True)`

(Char, Bool)

(k) `filterM (const [True, False])`

~~ill-typed~~

[a] → [a]

Question 3 (20 points)

For each of the types below, write a Haskell expression that has that type. Don't write trivial expressions (such as [], **Nothing**, or **undefined**) unless there is no other option. You can use any function from the appendix, do syntax, list comprehensions, or any valid Haskell.

(a) `Int -> Int`
Example answers:
`\x -> x + 1`
`(+1)`

(b) `Bool -> [Bool]`

\x → (x && True) : []

(c) `a -> Maybe b`

foo x = undefined

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`

\ f x y → [f x' y' | x' ← x , y' ← y]

(e) `(a -> b -> c) -> IO a -> IO b -> IO c`

(e) \ f x y → do
x' ← x
y' ← y
return f x' y'

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`

\f g xs → filter g (map f xs)

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`

\x f → case x of Nothing → Nothing
Just y → do
z ← f y
return (y, z)

(h) `Eq a => a -> [a] -> [a]`

\ x ls → filter (== x) ls

(i) `Show a => [a] -> IO String`

~~XXXXXX~~ foo _ = getLine

(j) `(a,b) -> (a -> b -> c) -> c`

\(x, y) f → f x y

\f g xs → (map filter (g) f xs)

Question 4 (20 points + 10pt bonus!)

For each of the following functions, write down a short description of what `foo` does, and some output examples.

(a)
```
foo :: [Int] -> [Int]
foo l = [ x * x | x <- l, x > 0 ]
```

Example answer:
Calculates the squares of all positive numbers in a list.
foo [] = []
foo [1,0,2,-1] = [1, 4]

(b)
```
foo :: [Int] -> [Int] [(Int, Int)]
foo l = [ (x,y) | x <- l, y <- l, x /= y]
```

Answer: create list of all pairings of non-equal numbers in list l.

foo [1] = []
foo [1,2,1] = [(1,2), (2,1)]
0 3 2 1

(c)
```
foo :: a -> [a] -> [a]
foo x l = reverse (x : reverse l)
```

Answer:

add x to end of list l
foo 1 [] = [1]
foo 0 [1,2,3] = [1,2,3,0]

(d)
```
bar           :: (a -> Maybe b) -> [a] -> [b]
bar _ []      = []
bar f (x:xs) =
  let rs = bar f xs in
  case f x of
    Nothing -> rs
    Just r  -> r:rs

foo :: [Maybe a] -> [a]
foo = bar id
```

If f x has value, add value to list

Answer: remove all Nothing's and keep values in Just's

foo [Nothing, Just 1] = [1]
foo [ ] = [ ]
foo [Just 1, Just 2] = [1,2]

(e)
```
foo :: [Int] -> Int -> (Int, [Int])
foo []  m = (m, [])
foo [x] m = (x, [m])
foo (x : xs) m = (max m' x, m : xs')
  where (m', xs') = foo xs m
```

foo [3] 0 = (3, [0])
foo [2,3] 0 = (3, [0,0])
foo [1,2,3] 0 = (3, [0,0,0])

Answer:

~~return~~ make a tuple where first is the largest number in the list plus m and second is a list of m's length list of (x:xs)

foo [3] 4 = (3, [4])
foo [2,3] 4 = (3, [4,4])

foo [ ] 0 = (0, [ ])
foo [2,3] 4 = (3, [4,4])

(f) *Bonus!*

```
dropWhileM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
dropWhileM _ []     = return []
dropWhileM p (x:xs) = do
    q <- p x
    if q then dropWhileM p xs else return (x:xs)

foo :: ??
foo = dropWhileM (const [True, False])
```

Answer:

~~returns~~ ~~[illegible]~~ ~~False~~

generates list with False

Question 5 (20 points + 10pt bonus!)

Implement the following Haskell functions:

(a) Implement a function `weave` that given two lists with elements of the same type, returns a list with elements alternating between the two lists. For example:

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

You can assume that the lists have the same length.

```
weave [] [] = []
weave (x: xs) (y: ys) = x: y: (weave xs ys)
```

(b) Implement a function `toMax`, that given a non-empty list of integers, returns a list of the same length, where each element has been replaced by the maximum element of the list. For example:

```
toMax [1,4,2,5,3] = [5,5,5,5,5]
```

You can use the `foo` function of problem (4e) if it helps.

BONUS: Implement `toMax` so that it only traverses a list once!

```
toMax (x: xs) = let max' = foldr (\y a -> if y > a then y else a) x xs
    in snd foo (x : xs) max'

toMax ls @(x: xs) = snd foo ls m
        where m = foldr (\y a -> if y > a then y
                                  else a) x ls
```

Typeclass Definitions

```
class Semigroup a where
  (<>)    :: a -> a -> a

class Semigroup m => Monoid m where
  mempty :: m

class  Show a  where
  show   :: a    -> String

class  Eq a  where
  (==)   :: a -> a -> Bool
  (/=)   :: a -> a -> Bool

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

class Functor f where
  fmap   :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure   :: a -> f a
  (<*>)  :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m

class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]

class Num a where
  (+), (-), (*)         :: a -> a -> a
  negate                :: a -> a
  abs                   :: a -> a
  signum                :: a -> a
  fromInteger           :: Integer -> a
```

Siddharth Taneja

CMSC 488B: Midterm Exam (Spring 2022)

Question 1 (20 points)

Part 1 - 8pts

For each of the following questions, select the appropriate response.

(a) There exist OCaml programs that don't terminate, whose Haskell equivalents do terminate.

- (i) True
- (ii) False

(b) The constraint `Semigroup a => Monoid a` implies that:

- (i) Every type that has a `Semigroup` instance, also has a `Monoid` instance.
- (ii) Every type that has a `Monoid` instance, also has a `Semigroup` instance.
- (iii) Both of the above.
- (iv) None of the above.

(c) Typeclass laws are enforced by the Haskell compiler.

- (i) True
- (ii) False

(d) Given a function `foo :: Int -> Char -> Bool`, consider the function call `quickCheck foo`. Select what will happen if that was inside a `main` function:

- (i) The program will fail to typecheck.
- (ii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for `Int` and `Char` to generate inputs and test `foo`.
- (iii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for () as default to generate inputs and test `foo`.
- (iv) Haskell will require user-provided generators for integers and characters in order to run the tests

Part 2 - 12pts

m = [Int]
t = []

The standard library defines `foldMap` with the following type:

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

For each of the following `foldMap` calls, select all options that are true:

(a) `foldMap (:[]) [1..10]`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is `[]` - the instance for lists.

(b) `foldMap show "123456"`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is `[]` - the instance for lists.

Question 2 (20 points)

For each of the Haskell expressions below, write their (most general) Haskell type or "ill-typed" if it contains a type error. The type signatures of all functions below are provided in the appendix at the end.

(a) `\x -> x`
Example answer:
`a -> a`

(b) `\x y -> (x,y)`

a -> b -> (a,b)

(c) `\x y -> if x == y then show x else show (x,y)`

Eq a, Show a => a -> a -> String

(d) `\x l -> x : l ++ l ++ [x]`

a -> [a] -> [a]

(e) `getLine >>= putStrLn`

IO String -> IO ()

(f) `putStrLn 42 >>= putStrLn 43`

ill-typed

(g) `(,) "42"`

a -> (String, a)

(h) `reverse . foldMap return`

[a] -> [a]

(i) `\l -> [(x,y) | x <- l, y <- l, x /= y]`

Eq a => [a] -> [(a,a)]

(j) `let f x = x in (f 'a', f True)`

(Char, Bool)

(k) `filterM (const [True, False])` m = [] a = a

[a] -> [[a]]

Question 3 (20 points)

For each of the types below, write a Haskell expression that has that type. Don't write trivial expressions (such as [], **Nothing**, or **undefined**) unless there is no other option. You can use any function from the appendix, do syntax, list comprehensions, or any valid Haskell.

(a) `Int -> Int`
Example answers:
`\x -> x + 1`
`(+1)`

(b) `Bool -> [Bool]`
\x → [x && True]

(c) `a -> Maybe b`
\x → Nothing

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`
zipWith

(e) `(a -> b -> c) -> IO a -> IO b -> IO c`
liftA2

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`
\f g → map f

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`
\(Just x) f → do y <- f x ; return (x, y)

(h) `Eq a => a -> [a] -> [a]`
\x xs → [y | y <- xs, x /= y]

(i) `Show a => [a] -> IO String`
\xs → do x <- xs ; putStrLn (show x); getLine

(j) `(a,b) -> (a -> b -> c) -> c`
\(x, y) f → f x y

Question 4 (20 points + 10pt bonus!)

For each of the following functions, write down a short description of what `foo` does, and some output examples.

(a)
```
foo :: [Int] -> [Int]
foo l = [ x * x | x <- l, x > 0 ]
```

Example answer:
Calculates the squares of all positive numbers in a list.
foo [] = []
foo [1,0,2,−1] = [1, 4]

(b)
```
foo :: [Int] -> [(Int, Int)]
foo l = [ (x,y) | x <- l, y <- l, x /= y]
```

Answer: Computes all pairs of unequal elements in a list
foo [1] = []
foo [1,2,3] = [(1,2), (1,3), (2,1), (2,3), (3,1), (3,2)]

(c)
```
foo :: a -> [a] -> [a]
foo x l = reverse (x : reverse l)
```

Answer: Puts x at the end of the given list l
foo 1 [2,3,4] = [2, 3, 4, 1]
foo 7 [ ] = [7]

(d)
```
bar           :: (a -> Maybe b) -> [a] -> [b]
bar _ []      = []
bar f (x:xs) =
  let rs = bar f xs in
  case f x of
    Nothing -> rs
    Just r  -> r:rs

foo :: [Maybe a] -> [a]
foo = bar id
```

Answer: Takes a list of Maybe a's and removes all the Nothing's and the Just.
foo [Just 1, Nothing, Just 2] = [Just 1, Just 2]
foo [Nothing, Nothing] = []

(e)
```
foo :: [Int] -> Int -> (Int, [Int])
foo []  m = (m, [])
foo [x] m = (x, [m])
foo (x : xs) m = (max m' x, m : xs')
  where (m', xs') = foo xs m
```

Answer: foo returns the the maximum of the integer list, along with a list with each element is replaced by m. If the list is empty then we return m as the "default" max value.
foo [1, 2, 4, 5] 3 = (5, [3, 3, 3, 3])
foo [1, 2, 3] 7 = (3, [7, 7, 7])
foo [] 4 = (4, [])

m=[ ]

(i) *Bonus!*

```
dropWhileM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
dropWhileM _ []     = return [] → [[]]
dropWhileM p (x:xs) = do
    q <- p x
    if q then dropWhileM p xs else return (x:xs)

foo :: ?? [a] → [[a]]
foo = dropWhileM (const [True, False])
```

Answer: foo takes a list ℓ and returns all suffix sets of ℓ in decreasing size order

foo [1,2,3] = [[1,2,3], [2,3], [3], []]

foo "abcd" = ["abcd", "bcd", "cd", "d", []]

foo [] = [[]]

Question 5 (20 points + 10pt bonus!)

Implement the following Haskell functions:

(a) Implement a function `weave` that given two lists with elements of the same type, returns a list with elements alternating between the two lists. For example:

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

You can assume that the lists have the same length.

```
weave :: [a] -> [a] -> [a]

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

If the lists are of unequal length then we return [].

(b) Implement a function `toMax`, that given a non-empty list of integers, returns a list of the same length, where each element has been replaced by the maximum element of the list. For example:

```
toMax [1,4,2,5,3] = [5,5,5,5,5]
```

You can use the `foo` function of problem (4e) if it helps.

BONUS: Implement `toMax` so that it only traverses a list once!

```
toMax :: [Int] -> [Int]
toMax l = replicate (length l) (maximum l)
```

Bonus:

```
dfold :: (Int -> (Int -> Int) -> (Int -> Int)) -> (Int -> Int) -> [Int] -> [Int]
                                                -> (Int -> Int, [Int])
dfold p fac acc [] = []
dfold p fac acc (x:xs) = do
  (f', l) <- dfold p (p x fac) acc xs
  (p x f', (f' x) : l)

toMap l = snd (dfold (\x f -> \y -> max x (f y)) id [] l)
```

Typeclass Definitions

```
class Semigroup a where
  (<>)   :: a -> a -> a

class Semigroup m => Monoid m where
  mempty :: m

class  Show a  where
  show   :: a   -> String

class  Eq a  where
  (==)   :: a -> a -> Bool
  (/=)   :: a -> a -> Bool

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

class Functor f where
  fmap   :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure   :: a -> f a
  (<*>)  :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m

class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]

class Num a where
  (+), (-), (*)        :: a -> a -> a
  negate               :: a -> a
  abs                  :: a -> a
  signum               :: a -> a
  fromInteger          :: Integer -> a
```

Reid Huntley

CMSC 488B: Midterm Exam (Spring 2022)

Question 1 (20 points)

Part 1 - 8pts

For each of the following questions, select the appropriate response.

(a) There exist OCaml programs that don't terminate, whose Haskell equivalents do terminate.

- (i) True
- (ii) False

(b) The constraint `Semigroup a => Monoid a` implies that:

- (i) Every type that has a `Semigroup` instance, also has a `Monoid` instance.
- (ii) Every type that has a `Monoid` instance, also has a `Semigroup` instance.
- (iii) Both of the above.
- (iv) None of the above.

(c) Typeclass laws are enforced by the Haskell compiler.

- (i) True
- (ii) False

(d) Given a function `foo :: Int -> Char -> Bool`, consider the function call `quickCheck foo`. Select what will happen if that was inside a `main` function:

- (i) The program will fail to typecheck.
- (ii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for `Int` and `Char` to generate inputs and test `foo`.
- (iii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for () as default to generate inputs and test `foo`.
- (iv) Haskell will require user-provided generators for integers and characters in order to run the tests

Part 2 - 12pts

The standard library defines `foldMap` with the following type:

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

For each of the following `foldMap` calls, select all options that are true:

(a) `foldMap (:[]) [1..10]`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is [] - the instance for lists.

(b) `foldMap show "123456"`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is [] - the instance for lists.

Question 2 (20 points)

For each of the Haskell expressions below, write their (most general) Haskell type or "ill-typed" if it contains a type error. The type signatures of all functions below are provided in the appendix at the end.

(a) `\x -> x`
Example answer:
`a -> a`

(b) `\x y -> (x,y)`

a → b → (a, b)

(c) `\x y -> if x == y then show x else show (x,y)`

(Eq a, Show a) ⇒ a → a → String

(d) `\x l -> x : l ++ l ++ [x]`

a → [a] → [a]

(e) `getLine >>= putStrLn`

IO ()

(f) `putStrLn 42 >>= putStrLn 43`

ill-typed

(g) `(,) "42"`

a → (String, a)

(h) `reverse . foldMap return`

[a] → [a]

(i) `\l -> [(x,y) | x <- l, y <- l, x /= y]`

Eq a => [a] → [(a, a)]

(j) `let f x = x in (f 'a', f True)`

(Char, Bool)

(k) `filterM (const [True, False])`

[a] → [[a]]

Question 3 (20 points)

For each of the types below, write a Haskell expression that has that type. Don't write **trivial expressions** (**such** as [], **`Nothing`**, or **`undefined`**) **unless** there is no **other** option. You can use any **function from** the appendix, do **syntax**, list **comprehensions**, or any valid **Haskell**.

(a) `Int -> Int`
Example answers:
`\x -> x + 1`
`(+1)`

(b) `Bool -> [Bool]`

\x -> [x]

(c) `a -> Maybe b`

const Nothing

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`

\f x y -> f <$> x <*> y

(e) `(a -> b -> c) -> IO a -> IO b -> IO c`

\f x y -> f <$> x <*> y

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`

\f p x -> filter p (map f x)

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`

\m f -> return undefined

(h) `Eq a => a -> [a] -> [a]`

\x -> filter (x==)

(i) `Show a => [a] -> IO String`

return . foldMap show

(j) `(a,b) -> (a -> b -> c) -> c`

flip uncurry

Question 4 (20 points + 10pt bonus!)

For each of the following functions, write down a short description of what `foo` does, and some output examples.

(a)
```
foo :: [Int] -> [Int]
foo l = [ x * x | x <- l, x > 0 ]
```

Example answer:
Calculates the squares of all positive numbers in a list.
foo [] = []
foo [1,0,2,−1] = [1, 4]

(b)
```
foo :: [Int] -> [Int]
foo l = [ (x,y) | x <- l, y <- l, x /= y]
```

Answer:
Constructs every pair of distinct Ints that can be made using the elements of a passed-in list

(c)
```
foo :: a -> [a] -> [a]
foo x l = reverse (x : reverse l)
```

Answer:
Inserts an element at the end of a list

(d)
```
bar           :: (a -> Maybe b) -> [a] -> [b]
bar _ []      = []
bar f (x:xs) =
  let rs = bar f xs in
  case f x of
    Nothing -> rs
    Just r  -> r:rs

foo :: [Maybe a] -> [a]
foo = bar id
```

Answer:
Takes a list of Maybes and returns a list of the values contained in the Just elements in the same order

(e)
```
foo :: [Int] -> Int -> (Int, [Int])
foo []  m = (m, [])
foo [x] m = (x, [m])
foo (x : xs) m = (max m' x, m : xs')
  where (m', xs') = foo xs m
```

Answer:
Returns the max value amongst a list of Ints, or a passed-in Int if the list is empty, as well as a list that's the same length as the passed-in list and only contains copies of the passed-in Int

(f) *Bonus!*

```
dropWhileM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
dropWhileM _ []     = return []
dropWhileM p (x:xs) = do
    q <- p x
    if q then dropWhileM p xs else return (x:xs)

foo :: ?? [a] -> [[a]]
foo = dropWhileM (const [True, False])
```

Answer:

Takes a list and returns every slice (subsequence) of that list which ends with the last element, as well as the empty list.

Question 5 (20 points + 10pt bonus!)

Implement the following Haskell functions:

(a) Implement a function `weave` that given two lists with elements of the same type, returns a list with elements alternating between the two lists. For example:

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

You can assume that the lists have the same length.

```
weave :: [a] -> [a] -> [a]
weave [] _ = []
weave _ [] = []
weave (x:xs) (y:ys) = x : y : weave xs ys
```

(b) Implement a function `toMax`, that given a non-empty list of integers, returns a list of the same length, where each element has been replaced by the maximum element of the list. For example:

```
toMax [1,4,2,5,3] = [5,5,5,5,5]
```

You can use the `foo` function of problem (4e) if it helps.

BONUS: Implement `toMax` so that it only traverses a list once!

```
toMax :: [Int] -> [Int]

toMax l = replicate (length l) max
    where (max, _) = foo l 0
```

Typeclass Definitions

```
class Semigroup a where
  (<>)   :: a -> a -> a

class Semigroup m => Monoid m where
  mempty :: m

class  Show a  where
  show   :: a   -> String

class  Eq a  where
  (==)   :: a -> a -> Bool
  (/=)   :: a -> a -> Bool

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

class Functor f where
  fmap   :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure   :: a -> f a
  (<*>)  :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m

class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]

class Num a where
  (+), (-), (*)       :: a -> a -> a
  negate              :: a -> a
  abs                 :: a -> a
  signum              :: a -> a
  fromInteger         :: Integer -> a
```

Claude
Zou

CMSC 488B: Midterm Exam (Spring 2022)

Question 1 (20 points)

Part 1 - 8pts

For each of the following questions, select the appropriate response.

(a) There exist OCaml programs that don't terminate, whose Haskell equivalents do terminate.

- (i) True
- (ii) False

(b) The constraint `Semigroup a => Monoid a` implies that:

- (i) Every type that has a `Semigroup` instance, also has a `Monoid` instance.
- (ii) Every type that has a `Monoid` instance, also has a `Semigroup` instance.
- (iii) Both of the above.
- (iv) None of the above.

(c) Typeclass laws are enforced by the Haskell compiler.

- (i) True
- (ii) False

(d) Given a function `foo :: Int -> Char -> Bool`, consider the function call `quickCheck foo`. Select what will happen if that was inside a `main` function:

- (i) The program will fail to typecheck.
- (ii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for `Int` and `Char` to generate inputs and test `foo`.
- (iii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for `()` as default to generate inputs and test `foo`.
- (iv) Haskell will require user-provided generators for integers and characters in order to run the tests

Part 2 - 12pts

The standard library defines `foldMap` with the following type:

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

For each of the following `foldMap` calls, select all options that are true:

(a) `foldMap (:[]) [1..10]`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is `[]` - the instance for lists.

(b) `foldMap show "123456"`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is `[]` - the instance for lists.

Question 2 (20 points)

For each of the Haskell expressions below, write their (most general) Haskell type or "ill-typed" if it contains a type error. The type signatures of all functions below are provided in the appendix at the end.

(a) `\x -> x`
Example answer:
`a -> a`

(b) `\x y -> (x,y)`

a -> b -> (a, b)

(c) `\x y -> if x == y then show x else show (x,y)`

(Eq a) => a -> a -> String

(d) `\x l -> x : l ++ l ++ [x]`

a -> [a] -> [a]

(e) `getLine >>= putStrLn`

IO()

(f) `putStrLn 42 >>= putStrLn 43`

ill typed

(g) `(,) "42"`

a -> (String, a)

(h) `reverse . foldMap return`

ill typed

(i) `\l -> [(x,y) | x <- l, y <- l, x /= y]`

(Eq a) => [a] -> [(a, a)]

(j) `let f x = x in (f 'a', f True)`

(char, Bool)

(k) `filterM (const [True, False])`

[a] -> [[a]]

Question 3 (20 points)

For each of the types below, write a Haskell expression that has that type. Don't write trivial expressions (such as [], `Nothing`, or **`undefined`**) unless there is no other option. You can use any function from the appendix, do syntax, list comprehensions, or any valid Haskell.

(a) `Int -> Int`
Example answers:
`\x -> x + 1`
`(+1)`

(b) `Bool -> [Bool]`

\x-> [not x]

(c) `a -> Maybe b`

\x -> Nothing

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`

zipWith

(e) `(a -> b -> c) -> IO a -> IO b -> IO c`

liftM2

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`

\f1 f2 L -> filter f2 (map f1 L)

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`

\x f -> do a<-x, b <- f a, return (a,b)

(h) `Eq a => a -> [a] -> [a]`

\x L-> filter (\y-> y==x) L

(i) `Show a => [a] -> IO String`

\x -> pure (Show (head x))

(j) `(a,b) -> (a -> b -> c) -> c`

f (a,b) f1 = f1 a b

Question 4 (20 points + 10pt bonus!)

For each of the following functions, write down a short description of what `foo` does, and some output examples.

(a)
```
foo :: [Int] -> [Int]
foo l = [ x * x | x <- l, x > 0 ]
```

Example answer:
Calculates the squares of all positive numbers in a list.
foo [] = []
foo [1,0,2,−1] = [1, 4]

(b)
```
foo :: [Int] -> [Int]
foo l = [ (x,y) | x <- l, y <- l, x /= y]
```

Answer: tuples of all ints in a list with all other non equal ints in the list,
eg. [1, 0, 1] → [(1,0), (0,1)]

(c)
```
foo :: a -> [a] -> [a]
foo x l = reverse (x : reverse l)
```

Answer: append to end of list

(d)
```
bar              :: (a -> Maybe b) -> [a] -> [b]
bar _ []          = []
bar f (x:xs) =
  let rs = bar f xs in
  case f x of
    Nothing -> rs
    Just r  -> r:rs

foo :: [Maybe a] -> [a]
foo = bar id
```

Answer: removes all nothings from a list and "unwraps" the Justs.
eg [Just 1, Nothing] → [1]

(e)
```
foo :: [Int] -> Int -> (Int, [Int])
foo []  m = (m, [])
foo [x] m = (x, [m])
foo (x : xs) m = (max m' x, m : xs')
  where (m', xs') = foo xs m
```

Answer: returns a tuple of the max element of the list and a list of the same size with the second argument as the only element

[1,3,5] 4 → (5, [4,4,4])

(f) *Bonus!*

```
dropWhileM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
dropWhileM _ []     = return []
dropWhileM p (x:xs) = do
    q <- p x
    if q then dropWhileM p xs else return (x:xs)

foo :: ??
foo = dropWhileM (const [True, False])
```

Answer:

[a] → [[a]]

return the powerset of a list

eg. [1, 0] → [[], [1], [0], [1, 0]]

Question 5 (20 points + 10pt bonus!)

Implement the following Haskell functions:

(a) Implement a function `weave` that given two lists with elements of the same type, returns a list with elements alternating between the two lists. For example:

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

You can assume that the lists have the same length.

```
weave [] [] = []
weave (x:xs) (y:ys) = x : (y : (weave xs ys))
weave _ _ = undefined
```

(b) Implement a function `toMax`, that given a non-empty list of integers, returns a list of the same length, where each element has been replaced by the maximum element of the list. For example:

```
toMax [1,4,2,5,3] = [5,5,5,5,5]
```

You can use the `foo` function of problem (4e) if it helps.

BONUS: Implement `toMax` so that it only traverses a list once!

```
toMax [] = []
toMax L = replicate (length L) (maximum L)
```

Typeclass Definitions

```
class Semigroup a where
  (<>)   :: a -> a -> a

class Semigroup m => Monoid m where
  mempty :: m

class  Show a  where
  show   :: a   -> String

class  Eq a  where
  (==)   :: a -> a -> Bool
  (/=)   :: a -> a -> Bool

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

class Functor f where
  fmap   :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure   :: a -> f a
  (<*>)  :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m

class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]

class Num a where
  (+), (-), (*)        :: a -> a -> a
  negate               :: a -> a
  abs                  :: a -> a
  signum               :: a -> a
  fromInteger          :: Integer -> a
```

GUIDO AMBASZ

CMSC 488B: Midterm Exam (Spring 2022)

Question 1 (20 points)

Part 1 - 8pts

For each of the following questions, select the appropriate response.

(a) There exist OCaml programs that don't terminate, whose Haskell equivalents do terminate.

- (i) True
- (ii) False

(b) The constraint `Semigroup a => Monoid a` implies that:

- (i) Every type that has a `Semigroup` instance, also has a `Monoid` instance.
- (ii) Every type that has a `Monoid` instance, also has a `Semigroup` instance.
- (iii) Both of the above.
- (iv) None of the above.

(c) Typeclass laws are enforced by the Haskell compiler. (Assuming type checker ≠ compiler)

- (i) True
- (ii) False

(d) Given a function `foo :: Int -> Char -> Bool`, consider the function call `quickCheck foo`. Select what will happen if that was inside a `main` function:

- (i) The program will fail to typecheck.
- (ii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for `Int` and `Char` to generate inputs and test `foo`.
- (iii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for () as default to generate inputs and test `foo`.
- (iv) Haskell will require user-provided generators for integers and characters in order to run the tests

Part 2 - 12pts

The standard library defines `foldMap` with the following type:

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

For each of the following `foldMap` calls, select all options that are true:

(a) `foldMap (:[]) [1..10]`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is `[]` - the instance for lists.

(b) `foldMap show "123456"`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is `[]` - the instance for lists.

Question 2 (20 points)

For each of the Haskell expressions below, write their (most general) Haskell type or "ill-typed" if it contains a type error. The type signatures of all functions below are provided in the appendix at the end.

(a) `\x -> x`
Example answer:
`a -> a`

(b) `\x y -> (x,y)`

a -> b -> (a, b)

(c) `\x y -> if x == y then show x else show (x,y)`

a -> a -> String

(d) `\x l -> x : l ++ l ++ [x]`

a -> [a] -> [a]

(e) `getLine >>= putStrLn`

IO ()

(f) `putStrLn 42 >>= putStrLn 43`

ill-typed

(g) `(,) "42"`

a -> (String, a)

(h) `reverse . foldMap return`

(Foldable c, Monad m) => t a -> m

(i) `\l -> [(x,y) | x <- l, y <- l, x /= y]`

Eq a => [a] -> [(a, a)]

(j) `let f x = x in (f 'a', f True)`

(Char, Bool)

(k) `filterM (const [True, False])`

Monad m => [a] -> m [a]

Question 3 (20 points)

For each of the types below, write a Haskell expression that has that type. Don't write trivial expressions (such as [], `Nothing`, or `undefined`) unless there is no other option. You can use any function from the appendix, do syntax, list comprehensions, or any valid Haskell.

(a) `Int -> Int`
Example answers.
`\x -> x + 1`
`(+1)`

(b) `Bool -> [Bool]`

\b -> if b then [b] else [b]

(c) `a -> Maybe b`

\a -> Nothing

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`

liftM2

~~\f il cl -> if f (head il) (head cl) then [True]~~ else ~~[False]~~

(e) `(a -> b -> c) -> IO a -> IO b -> IO c`

liftM2

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`

~~\f g la~~ \f g la -> let b = f (head la) in if g b then [b] else [b]

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`

\ma f -> arbitrary((fromJust ma), f (fromJust ma))

(h) `Eq a => a -> [a] -> [a]`

\a la -> if a == (head la) then la else [a]

(i) `Show a => [a] -> IO String`

\la -> return (show (head la))

(j) `(a,b) -> (a -> b -> c) -> c`

\(a,b) f -> f a b

Question 4 (20 points + 10pt bonus!)

For each of the following functions, write down a short description of what `foo` does, and some output examples.

(a)
```
foo :: [Int] -> [Int]
foo l = [ x * x | x <- l, x > 0 ]
```

Example answer:
Calculates the squares of all positive numbers in a list.
foo [] = []
foo [1,0,2,-1] = [1, 4]

h

(b)
```
foo :: [Int] -> [(Int, Int)]
foo l = [ (x,y) | x <- l, y <- l, x /= y]
```

Answer: Returns a list with all pairs of integers that are not equal

foo [1] = []
foo [1,2,3] = [(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)]

(c)
```
foo :: a -> [a] -> [a]
foo x l = reverse (x : reverse l)
```

Answer: appends an element to the end of the list.

foo 1 [] = [1]
foo 4 [1,2,3] = [1,2,3,4]

(d)
```
bar           :: (a -> Maybe b) -> [a] -> [b]
bar _ []      = []
bar f (x:xs)  =
  let rs = bar f xs in     -- rest
  case f x of
    Nothing -> rs
    Just r  -> r:rs

foo :: [Maybe a] -> [a]
foo = bar id
```

Answer: Takes a list of maybes and returns a list with all values that are not Nothing.

Foo [] = []
~~foo~~ Foo [Just 1, Just 2, Just 3] = [1,2,3]

(e)
```
foo :: [Int] -> Int -> (Int, [Int])
foo []  m = (m, [])
foo [x] m = (x, [m])
foo (x : xs) m = (max m' x, m : xs')
  where (m', xs') = foo xs m
```

Answer: Returns a tuple with the maximum element of the list first, and a list with the maximum between the current head and the parameter m.

foo [1,4,2] 3 = (4, [3,4,3])
foo [] 1 = (1, [])
foo [2] 1 = (2, [1])
~~foo [1,2] 3~~ Foo [1,2] 3 = (2, [3,3])

5

(f) *Bonus!*

```
dropWhileM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
dropWhileM _ []     = return []
dropWhileM p (x:xs) = do
    q <- p x
    if q then dropWhileM p xs else return (x:xs)

foo :: ?? [a] -> m [a]
foo = dropWhileM (const [True, False])
```

Answer: This will just return the empty list. ~~wrapped in a~~ ~~monad~~.

~~dropWhileM~~

foo [ ] = [ ]

foo [1,2,3] = [ ].

foo ["hope", "im", "right"] = [ ].

Question 5 (20 points + 10pt bonus!)

Implement the following Haskell functions:

(a) Implement a function `weave` that given two lists with elements of the same type, returns a list with elements alternating between the two lists. For example:

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

You can assume that the lists have the same length.

```
weave [] [] = []
weave [x] [y] = [x,y]
weave x:xs y:ys = [x,y] ++ (weave xs ys)
```

(b) Implement a function `toMax`, that given a non-empty list of integers, returns a list of the same length, where each element has been replaced by the maximum element of the list. For example:

```
toMax [1,4,2,5,3] = [5,5,5,5,5]
```

You can use the `foo` function of problem (4e) if it helps.

BONUS: Implement `toMax` so that it only traverses a list once!

```
-- Should only traverse once.
toMax lst =
    let m = maximum lst in
    take (length lst) (repeat m)

-- without bonus.
toMax lst =
    let m = maximum lst in
    map (\_ -> m) lst
```

Typeclass Definitions

```
class Semigroup a where
  (<>)    :: a -> a -> a

class Semigroup m => Monoid m where
  mempty :: m

class  Show a  where
  show   :: a    -> String

class  Eq a  where
  (==)   :: a -> a -> Bool
  (/=)   :: a -> a -> Bool

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

class Functor f where
  fmap   :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure   :: a -> f a
  (<*>)  :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m

class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]

class Num a where
  (+), (-), (*)        :: a -> a -> a
  negate               :: a -> a
  abs                  :: a -> a
  signum               :: a -> a
  fromInteger          :: Integer -> a
```

Grayson Wolff

CMSC 488B: Midterm Exam (Spring 2022)

Question 1 (20 points)

Part 1 - 8pts

For each of the following questions, select the appropriate response.

(a) There exist OCaml programs that don't terminate, whose Haskell equivalents do terminate.

- (i) True
- (ii) False

(b) The constraint `Semigroup a => Monoid a` implies that:

- (i) Every type that has a `Semigroup` instance, also has a `Monoid` instance.
- (ii) Every type that has a `Monoid` instance, also has a `Semigroup` instance.
- (iii) Both of the above.
- (iv) None of the above.

(c) Typeclass laws are enforced by the Haskell compiler.

- (i) True
- (ii) False

(d) Given a function `foo :: Int -> Char -> Bool`, consider the function call `quickCheck foo`. Select what will happen if that was inside a `main` function:

- (i) The program will fail to typecheck.
- (ii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for `Int` and `Char` to generate inputs and test `foo`.
- (iii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for () as default to generate inputs and test `foo`.
- (iv) Haskell will require user-provided generators for integers and characters in order to run the tests

Part 2 - 12pts

The standard library defines `foldMap` with the following type:

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

For each of the following `foldMap` calls, select all options that are true:

(a) `foldMap (:[]) [1..10]`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is [] - the instance for lists.

(b) `foldMap show "123456"`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is [] - the instance for lists.

Question 2 (20 points)

For each of the Haskell expressions below, write their (most general) Haskell type or "ill-typed" if it contains a type error. The type signatures of all functions below are provided in the appendix at the end.

(a) `\x -> x`
Example answer:
`a -> a`

(b) `\x y -> (x,y)`

a -> b -> (a, b)

(c) `\x y -> if x == y then show x else show (x,y)`

Eq a, Show a => a -> a -> String

(d) `\x l -> x : l ++ l ++ [x]`

[a] -> a -> [a]

(e) `getLine >>= putStrLn`

IO ()

(f) `putStrLn 42 >>= putStrLn 43`

ill typed

(g) `(,) "42"`

a -> ([Char], a)

(h) `reverse . foldMap return`

[a] -> [a]

(i) `\l -> [(x,y) | x <- l, y <- l, x /= y]`

[a] -> [(a, a)]

(j) `let f x = x in (f 'a', f True)`

([Char], Bool)

(k) `filterM (const [True, False])`

[a] -> [[a]]

Question 3 (20 points)

For each of the types below, write a Haskell expression that has that type. Don't write trivial expressions (such as [], **`Nothing`**, or **`undefined`**) unless there **is no** other option. You can use any function from the appendix, do **syntax**, list **comprehensions**, or any **valid** Haskell.

(a) `Int -> Int`

Example answers:

`\x -> x + 1`

`(+1)`

(b) `Bool -> [Bool]`

return

(c) `a -> Maybe b`

undefined

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`

liftM2

(e) `(a -> b -> c) -> IO a -> IO b -> IO c`

liftM2

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`

\f g l -> filter g $ map f l

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`

undefined

(h) `Eq a => a -> [a] -> [a]`

\x l -> filter (== x) l

(i) `Show a => [a] -> IO String`

return $ foldMap show

(j) `(a,b) -> (a -> b -> c) -> c`

\t g -> uncurry g t

Question 4 (20 points + 10pt bonus!)

For each of the following functions, write down a short description of what `foo` does, and some output examples.

(a)
```
foo :: [Int] -> [Int]
foo l = [ x * x | x <- l, x > 0 ]
```

Example answer:
Calculates the squares of all positive numbers in a list.
`foo [] = []`
`foo [1,0,2,-1] = [1, 4]`

(b)
```
foo :: [Int] -> [Int]
foo l = [ (x,y) | x <- l, y <- l, x /= y]
```

Answer:
returns all pairs of elements that differ from each other
foo [1,1,2,3] = [(1,2), (1,3), (1,2), (1,3), (2,1), (2,1), (2,3), (3,1), (3,1), (3,2)]

(c)
```
foo :: a -> [a] -> [a]
foo x l = reverse (x : reverse l)
```

Answer:
adds x to end of list
foo 3 [1,2] = [1,2,3]

(d)
```
bar           :: (a -> Maybe b) -> [a] -> [b]
bar _ []      = []
bar f (x:xs) =
  let rs = bar f xs in
  case f x of
    Nothing -> rs
    Just r  -> r:rs

foo :: [Maybe a] -> [a]
foo = bar id
```

Answer:
Removes all Nothings and "un-Just"s the remaining elems
foo [Just 1, Just 2, Nothing, Just 3] = [1,2,3]

(e)
```
foo :: [Int] -> Int -> (Int, [Int])
foo []  m = (m, [])
foo [x] m = (x, [m])
foo (x : xs) m = (max m' x, m : xs')
  where (m', xs') = foo xs m
```

Answer:
Extracts the max element, replaces each element w/ m and returns the tuple containing the (max elem, new list)

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

(f) *Bonus!*

```
dropWhileM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
dropWhileM _ []     = return []
dropWhileM p (x:xs) = do
    q <- p x
    if q then dropWhileM p xs else return (x:xs)

foo :: ?? [a] -> [[a]]
foo = dropWhileM (const [True, False])
```

Answer:

equivalent to tails (The "true" branch drops the element and continues, the "false" branch stops the computation)

foo [1,2,3] =

[[1,2,3], [2,3], [3], []]

Question 5 (20 points + 10pt bonus!)

Implement the following Haskell functions:

(a) Implement a function `weave` that given two lists with elements of the same type, returns a list with elements alternating between the two lists. For example:

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

You can assume that the lists have the same length.

```
weave x y = concatMap (uncurry (:) ) $ zip x y
```

(b) Implement a function `toMax`, that given a non-empty list of integers, returns a list of the same length, where each element has been replaced by the maximum element of the list. For example:

```
toMax [1,4,2,5,3] = [5,5,5,5,5]
```

You can use the `foo` function of problem (4e) if it helps.

BONUS: Implement `toMax` so that it only traverses a list once!

```
toMax x = map (const $ maximum x) x
```

BONUS

```
toMax (x:xs) = snd $ f x (x:xs) where
  f a [] = (a, [])
  f a (y:ys) = (q, q:qs) where
     (q, qs) = f (max a y) ys
```

Typeclass Definitions

```
class Semigroup a where
  (<>)   :: a -> a -> a

class Semigroup m => Monoid m where
  mempty :: m

class  Show a  where
  show   :: a    -> String

class  Eq a  where
  (==)   :: a -> a -> Bool
  (/=)   :: a -> a -> Bool

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

class Functor f where
  fmap   :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure   :: a -> f a
  (<*>)  :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m

class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]

class Num a where
  (+), (-), (*)      :: a -> a -> a
  negate             :: a -> a
  abs                :: a -> a
  signum             :: a -> a
  fromInteger        :: Integer -> a
```

Garrett Hill

CMSC 488B: Midterm Exam (Spring 2022)

Question 1 (20 points)

Part 1 - 8pts

For each of the following questions, select the appropriate response.

(a) There exist OCaml programs that don't terminate, whose Haskell equivalents do terminate.

- (i) True lazy :)
- (ii) False

(b) The constraint `Semigroup a => Monoid a` implies that:

- (i) Every type that has a `Semigroup` instance, also has a `Monoid` instance.
- (ii) Every type that has a `Monoid` instance, also has a `Semigroup` instance.
- (iii) Both of the above.
- (iv) None of the above.

(c) Typeclass laws are enforced by the Haskell compiler.

- (i) True
- (ii) False

(d) Given a function `foo :: Int -> Char -> Bool`, consider the function call `quickCheck foo`. Select what will happen if that was inside a `main` function:

- (i) The program will fail to typecheck.
- (ii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for `Int` and `Char` to generate inputs and test `foo`.
- (iii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for () as default to generate inputs and test `foo`.
- (iv) Haskell will require user-provided generators for integers and characters in order to run the tests

Part 2 - 12pts

The standard library defines `foldMap` with the following type:

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

For each of the following `foldMap` calls, select all options that are true:

(a) `foldMap (:[]) [1..10]`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is `[]` - the instance for lists.

(b) `foldMap show "123456"`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is `[]` - the instance for lists.

Question 2 (20 points)

For each of the Haskell expressions below, write their (most general) Haskell type or "ill-typed" if it contains a type error. The type signatures of all functions below are provided in the appendix at the end.

(a) `\x -> x`
Example answer:
`a -> a`

(b) `\x y -> (x,y)`
a -> b -> (a,b)

(c) `\x y -> if x == y then show x else show (x,y)`
Eq a => a -> a -> String

(d) `\x l -> x : l ++ l ++ [x]`
a -> [a] -> [a]

(e) `getLine >>= putStrLn`
IO()

(f) `putStrLn 42 >> putStrLn 43`
ill typed

(g) `(,) "42"`
ill typed

(h) `reverse . foldMap return`
(Foldable t, Monoid m) => t a -> [m a]

(i) `\l -> [(x,y) | x <- l, y <- l, x /= y]`
Eq a => [a] -> [(a,a)]

(j) `let f x = x in (f 'a', f True)`
(Char, Bool)

(k) `filterM (const [True, False])`
Foldable t => t a -> t a

Question 3 (20 points)

For each of the types below, write a Haskell expression that has that type. Don't write trivial expressions (such as [], `Nothing`, or `undefined`) unless there is no other option. You can use any function from the appendix, do syntax, list comprehensions, or any valid Haskell.

(a) `Int -> Int`

Example answers:

`\x -> x + 1`

`(+1)`

(b) `Bool -> [Bool]`

\x -> if x then [x] else []

(c) `a -> Maybe b`

\x -> Nothing

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`

\f i cl -> [f i c | i <- il, c <- cl]

(e) `(a -> b -> c) -> IO a -> IO b -> IO c`

\f a b -> do a' <- a ; b' <- b ; print (f a b)

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`

~~\f1 f2 l -> [(f1 a) `f2` b | a <- l, b <-~~

\f1 f2 l -> map f2 (map f1 l)

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`

foo (Just a) f = return (a, f a)

foo Nothing f = undefined

(h) `Eq a => a -> [a] -> [a]`

foo x [] = [x]

foo x (h:t) = if x == h then x:t else x:h:t

(i) `Show a => [a] -> IO String`

\l -> show (head l)

(j) `(a,b) -> (a -> b -> c) -> c`

\(l,r) f -> f l r

Question 4 (20 points + 10pt bonus!)

For each of the following functions, write down a short description of what `foo` does, and some output examples.

(a)
```
foo :: [Int] -> [Int]
foo l = [ x * x | x <- l, x > 0 ]
```

Example answer:
Calculates the squares of all positive numbers in a list.
`foo [] = []`
`foo [1,0,2,-1] = [1, 4]`

(b)
```
foo :: [Int] -> [Int]
foo l = [ (x,y) | x <- l, y <- l, x /= y]
```

Answer: ~~Calculate x/y for every x,y pair in l then return the list of all (x/y, y) pairs foo [] = []~~
return every possible pair of nonequal values in l

~~foo [1,2,3] = [(1,~~

foo [] = []
foo [1,1,2]
= [(1,2)
(2,1)
(2,1)]

(c)
```
foo :: a -> [a] -> [a]
foo x l = reverse (x : reverse l)
```

Answer: ~~reversed~~ appends x to l

foo 1 [] = [1]
foo 2 [1,3,4] = [1,3,4,2]

(d)
```
bar          :: (a -> Maybe b) -> [a] -> [b]
bar _ []     = []
bar f (x:xs) =
  let rs = bar f xs in
  case f x of
    Nothing -> rs
    Just r  -> r:rs

foo :: [Maybe a] -> [a]
foo = bar id
```

Answer: Prunes a list of maybes to only the values stored in "Just"s

foo [] = []
foo [Nothing] = []
foo [(Just 1), Nothing] = [1]

(e)
```
foo :: [Int] -> Int -> (Int, [Int])
foo []  m = (m, [])
foo [x] m = (x, [m])
foo (x : xs) m = (max m' x, m : xs')
  where (m', xs') = foo xs m
```

Answer: ~~replaces the highest value in the list with m~~
replaces all values lower than m with m

foo [0,1,2,3] 2 = [2,2,2,3]
foo [3,4,5,6] 2 = [3,4,5,6]

(f) *Bonus!*

```
dropWhileM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
dropWhileM _ []      = return []
dropWhileM p (x:xs) = do
    q <- p x [True,False] always!
    if q then dropWhileM p xs else return (x:xs)

foo :: ?? [a] -> m[a]
foo = dropWhileM (const [True, False])
```

Answer: Returns the list of all possible tails of a given list

foo [] = [[]]

foo [1,2] = [[1,2], [2], []]

Question 5 (20 points + 10pt bonus!)

Implement the following Haskell functions:

(a) Implement a function `weave` that given two lists with elements of the same type, returns a list with elements alternating between the two lists. For example:

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

You can assume that the lists have the same length.

```
weave (x:xs) (y:ys) = x : y:(weave xs ys)
weave  []     [] =  []
```

(b) Implement a function `toMax`, that given a non-empty list of integers, returns a list of the same length, where each element has been replaced by the maximum element of the list. For example:

```
toMax [1,4,2,5,3] = [5,5,5,5,5]
```

You can use the `foo` function of problem (4e) if it helps.

BONUS: Implement `toMax` so that it only traverses a list once!

```
toMax [] = []
toMax l = foo l (~~max~~ l)
                 maximum
```

Typeclass Definitions

```
class Semigroup a where
  (<>)   :: a -> a -> a

class Semigroup m => Monoid m where
  mempty :: m

class  Show a  where
  show   :: a   -> String

class  Eq a  where
  (==)   :: a -> a -> Bool
  (/=)   :: a -> a -> Bool

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

class Functor f where
  fmap   :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure   :: a -> f a
  (<*>)  :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m

class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]

class Num a where
  (+), (-), (*)       :: a -> a -> a
  negate              :: a -> a
  abs                 :: a -> a
  signum              :: a -> a
  fromInteger         :: Integer -> a
```

Talha Muhib

CMSC 488B: Midterm Exam (Spring 2022)

Question 1 (20 points)

Part 1 - 8pts

For each of the following questions, select the appropriate response.

(a) There exist OCaml programs that don't terminate, whose Haskell equivalents do terminate.

- (i) True
- (ii) False

(b) The constraint `Semigroup a => Monoid a` implies that:

- (i) Every type that has a `Semigroup` instance, also has a `Monoid` instance.
- (ii) Every type that has a `Monoid` instance, also has a `Semigroup` instance.
- (iii) Both of the above.
- (iv) None of the above.

(c) Typeclass laws are enforced by the Haskell compiler.

- (i) True
- (ii) False

(d) Given a function `foo :: Int -> Char -> Bool`, consider the function call `quickCheck foo`. Select what will happen if that was inside a `main` function:

- (i) The program will fail to typecheck.
- (ii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for `Int` and `Char` to generate inputs and test `foo`.
- (iii) After typeclass resolution, the resulting program will use the `Arbitrary` instance for `()` as default to generate inputs and test `foo`.
- (iv) Haskell will require user-provided generators for integers and characters in order to run the tests

Part 2 - 12pts

The standard library defines `foldMap` with the following type:

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

For each of the following `foldMap` calls, select all options that are true:

(a) `foldMap (:[]) [1..10]`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is `[]` - the instance for lists.

(b) `foldMap show "123456"`

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- (iv) The monoid in this call is `String`.
- (v) The expression is equivalent to the identity function.
- (vi) The foldable in this call is `[]` - the instance for lists.

Question 2 (20 points)

For each of the Haskell expressions below, write their (most general) Haskell type or "ill-typed" if it contains a type error. The type signatures of all functions below are provided in the appendix at the end.

(a) `\x -> x`
Example answer:
`a -> a`

(b) `\x y -> (x,y)`

a -> b -> (a, b)

(c) `\x y -> if x == y then show x else show (x,y)`

ill-typed

(d) `\x l -> x : l ++ l ++ [x]`

a -> [a] -> [a]

(e) `getLine >>= putStrLn`

Applicative IO => Monad IO

(f) `putStrLn 42 >>= putStrLn 43`

ill-typed

(g) `(,) "42"`

((Char -> Char -> ((Char, Char)) -> String

(h) `reverse . foldMap return`

[a] -> [a]

(i) `\l -> [(x,y) | x <- l, y <- l, x /= y]`

[a] -> [(a, a)]

(j) `let f x = x in (f 'a', f True)`

ill-typed

(k) `filterM (const [True, False])`

[a] -> [[a]]

Question 3 (20 points)

For each of the types below, write a Haskell expression that has that type. Don't write trivial expressions (such as [], `Nothing`, or `undefined`) unless there is no other option. You can use any function from the appendix, do syntax, list comprehensions, or any valid Haskell.

(a) `Int -> Int`
Example answers:
`\x -> x + 1`
`(+1)`

(b) `Bool -> [Bool]`
\x -> if x then [True] else [False]

(c) `a -> Maybe b`
Monad maybe => (>>=)

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`
liftM2 (\a h -> True) list1 list2

(e) `(a -> b -> c) -> IO a -> IO b -> IO c`
liftM2 f i1 i2 = undefined

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`
(\f f2 list -> map f list)

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`
(\a f -> undefined)

(h) `Eq a => a -> [a] -> [a]`
(\a (h:t) -> if h==a then a:h:t else a:t)

(i) `Show a => [a] -> IO String`
(\l -> getLine)

(j) `(a,b) -> (a -> b -> c) -> c`
(\t f -> uncurry f t)

Question 4 (20 points + 10pt bonus!)

For each of the following functions, write down a short description of what `foo` does, and some output examples.

(a)
```
foo :: [Int] -> [Int]
foo l = [ x * x | x <- l, x > 0 ]
```

Example answer:
Calculates the squares of all positive numbers in a list.
foo [] = []
foo [1,0,2,−1] = [1, 4]

(b)
```
foo :: [Int] -> [Int]
foo l = [ (x,y) | x <- l, y <- l, x /= y]
```

Answer:
Makes a list of tuples where no tuple has duplicates

(c)
```
foo :: a -> [a] -> [a]
foo x l = reverse (x : reverse l)
```

Answer:
Reverses l, then prepends x, then reverses again

(d)
```
bar           :: (a -> Maybe b) -> [a] -> [b]
bar _ []      = []
bar f (x:xs) =
  let rs = bar f xs in
  case f x of
    Nothing -> rs
    Just r  -> r:rs

foo :: [Maybe a] -> [a]
foo = bar id
```

Answer:
Filters out elements that result to Nothing when passed into f

(e)
```
foo :: [Int] -> Int -> (Int, [Int])
foo []  m = (m, [])
foo [x] m = (x, [m])
foo (x : xs) m = (max m' x, m : xs')
  where (m', xs') = foo xs m
```

Answer:
Returns a tuple where the first element is the max of the list, and the second element is a list of m

(f) *Bonus!*

```
dropWhileM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
dropWhileM _ []     = return []
dropWhileM p (x:xs) = do
    q <- p x
    if q then dropWhileM p xs else return (x:xs)

foo :: ?? [a] -> [[a]]
foo = dropWhileM (const [True, False])
```

Answer:

Powerset of list

Question 5 (20 points + 10pt bonus!)

Implement the following Haskell functions:

(a) Implement a function `weave` that given two lists with elements of the same type, returns a list with elements alternating between the two lists. For example:

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

You can assume that the lists have the same length.

```
weave [] [] = []
weave (h:t) (h':t') = h:(h':(weave t t'))
```

(b) Implement a function `toMax`, that given a non-empty list of integers, returns a list of the same length, where each element has been replaced by the maximum element of the list. For example:

```
toMax [1,4,2,5,3] = [5,5,5,5,5]
```

You can use the `foo` function of problem (4e) if it helps.

BONUS: Implement `toMax` so that it only traverses a list once!

```
toMax list = let m = maximum list in map (\h -> m) list
```

Bonus:

```
toMaxAux :: [a] -> [a] -> [a]
toMaxAux [] t = [] ++ [t]
toMaxAux (h:t) (h':t') = let m = if h > h' then [h] else [h'] in
    (toMaxAux t m) ++ m
```