Part 2 - 12pts

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

```
foldMap :: (Moncid 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])`
[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 && x, x || x]`

(c) `a -> Maybe b`
`\x -> Nothing`

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`
`\f (x:xs) (y:ys) -> replicate x (y + 'c' == 'z')`

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

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`
`\f g xs -> [y | y <- f xs, not (g y)]`

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

(h) `Eq a => a -> [a] -> [a]`
`\x ys -> map (\y -> if x==y then x else y) ys`

(i) `Show a => [a] -> IO String`
`\(x:xs) -> putStrLn (show x)`

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

Answer:

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

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

Answer: adds x to the end of list l

foo 5 [] = [5]

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

(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 in a list of Maybes and removes the Maybe, returning the elements

foo [] = []

foo [Just 5, Just 4, Nothing] = [5, 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
```

Answer: finds the max of a list and returns it along with it removed from the list

foo [] 5 = (5, [])
foo [1] 5 = (1, [5])
foo [1,2,7] 5 = (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:

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 :: Ord a => [a] -> [a]
toMax [] = []
toMax (x:xs) = case findMax x xs 0 of
    (m, l) -> replicate l m
    _ -> []

findMax :: Ord a => a -> [a] -> Int -> (a, Int)
findMax m [] l = (m, l)
findMax m (x:xs) l
  | x > m = findMax x xs (l+1)
  | otherwise = findMax m xs (l+1)
```

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
```

Elvin Liu

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`
[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 -> [not x]`

(c) `a -> Maybe b`
`\x -> Nothing`

(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)`
`\x f -> mapM f (maybeToList x) >>= (\y -> return $ head (maybeToList x), y)`
`-- ?lmao`

(h) `Eq a => a -> [a] -> [a]`
`\x l -> filter (==x) l`

(i) `Show a => [a] -> IO String`
`\x -> putStrLn 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]
foo l = [ (x,y) | x <- l, y <- l, x /= y]
```

Answer: Sequential pairs of unequal elements

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

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

Answer: Appends x to end of l

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: removes Nothings from list of Maybes and retrieves elements in Just.

foo [Just 1, Nothing, 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: tuple of max of list and list of m of length of list

1:3, 5 → m', xs' = (3, [5])
(3, [5:5])
1:3:2, 0 → (3, 4:4:4)
3:2, 0 → (3, 4:0)
2, 0 → (2, [0])

foo [1,2] 5 = (3, [5,5])
foo [1,3,2] 2 = (3, [3,2,2])

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

Answer:

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

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
```

Chris Jose.

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.

>>= m a → (a → m b) → m b.

IO String → St

IO String → (String → IO

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 → b → String.

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

a → [a] → [a].

a [a]

(e) `getLine >>= putStrLn`

IO String → (String → IO ()) → IO ()

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

IO String → (String → IO String) → IO String

(g) `(,) "42"`

(h) `reverse . foldMap return`

Monad m ⇒ [a] → [m 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])`

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]`
`f x = (x || False) : []`

(c) `a -> Maybe b`
`\x -> pure x >> Maybe`

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`
`\f m1 m2 -> (f ((head m1)+1) (head m2) : ['a']) : []`

(e) `(a -> b -> c) -> IO a -> IO b -> IO c`
`liftM2`

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`
`\f g lst -> g (f (head lst)) : []`

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

(h) `Eq a => a -> [a] -> [a]`
`\x m1 -> intersperse x m1`

(i) `Show a => [a] -> IO String`
`\x -> (head x) ++ getLine`

(j) `(a,b) -> (a -> b -> c) -> c`
`\tup f -> f (fst tup) (snd tup)`

`\x -> do`

`\f m1 m2 -> [f (x+1) (y : ['a']) | x <- m1, y <- m2]`

10 [1,2,3]
10 [3,2,1]

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:
Return a list of tuples of ints of pairs of l, where both elements of each tuple are different. foo [1,2,3,1] = [(1,2),(1,3),(2,3)]

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

Answer:
Returns the original list along with element x appended to the end of the list.
foo 10 [1,2,3] = [1,2,3,10] ; foo 10 [] = [10]

(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:

(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 [3,2,1] 6.

Answer:
Returns a (Int, [Int]), where the 1st element of tuple is the max element of input list and 2nd element is a list of the 2nd arg with the same length as input list.
foo [3,2,1] 6 = (3, [6,6,6])
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 :: ??
foo = dropWhileM (const [True, False])
```

Answer:

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 [] b = b
weave a [] = a
weave m@(x:xs) n@(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 lst = let max = (fst $ foo lst 0) in
                snd $ foo lst max.
```

```
>> get Gen b.
```

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
```

a -> ma

3 : [2,1] 6 = (max m' 3, 6 : xs')

foo [2,1] 6

(max m' 2, 6 : 6 : xs')

foo [1] 6 = (1, [6])

Samuel Howard 115960176

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`
Foldable f ⇒ f 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])`
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]`
`\X→ if X then [X] else [X,X]`

(c) `a -> Maybe b`
Undefined

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`
`\f li lc → map (\(a,b)→f a b) (zip li lc)`

(e) `(a -> b -> c) -> IO a -> IO b -> IO c`
~~`f a b →`~~ `\f a b →do`
`a'←a`
`b'←b`
`putStrLn (f`

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`
`\fa fb la → filter (\a→ fb (fa a)) la`

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`
~~`m f →`~~ `\m f → do`
`a←m`
`(a, fa)`

(h) `Eq a => a -> [a] -> [a]`
`\a la → filter (\b→a==b) la`

(i) `Show a => [a] -> IO String`
`\L → map putStrLn L`

(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]`

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

Answer:
Returns all pairs of ints in a list that are not equal
foo [1,2,3] = [(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)]
foo [1,1,1,2] = [(1,2),(2,1)]

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

Answer:
Appends x to the end of l
foo 1 [7,8,9] = [7,8,9,1]
foo 'a' [] = ['a']

(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 of 'Maybe's
foo [Nothing, Maybe 1, Maybe 2, Nothing] = [Maybe 1, Maybe 2]
foo [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:
Returns a pair of the max element of a list (First arg) and a list with m (second argument) repeated n times where n is the length of the first argument. OR returns second arg paired with empty list if argument list is empty

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

(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:

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)
-- for invalid cases:
weave _ _ = [] <- Not possible with assumption
```

(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 [] = [] -- Not possible with assumption
toMax (x:xs) = a
  where (_,a) = foo (x:xs) b
    where (b,_) = foo (x:xs) 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
```

Vyoma Jani

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

laziness? Is OCaml lazy?

(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)`
(Show a, Eq a) ⇒ a → a → String
show :: a → String

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

(e) `getLine >>= putStrLn`
(>>=) :: M a → (a → M b) → M b
getLn :: IO String
putStrLn :: String → IO ()
IO String → String → IO () → IO ()

(f) `putStrLn 42 >>= putStrLn 43`
Ill-Typed
(getStrLn 42 putStrLn 43)
IO () → IO ()

(g) `(,) "42"`
a → (String, a)

(h) `reverse . foldMap return`
Foldable t ⇒ t a → [a]
reverse (foldMap return)
[a] → [a]
foldMap return :: (a → M) → t a → M
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]]
filterM :: Monad M ⇒
(a → M Bool) → [a] → M [a]
const [True, False]
b → [Bool]

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 [True]`

(c) `a -> Maybe b`
`\x -> Nothing`

(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]` `\f1 -> \f2 -> \lst -> [f1 a | a <- lst, f2 (f1 a)]`

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`
```
func Nothing _ = undefined
func (Just a) f = do b <- f a
                     return (a, b)
```

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

(i) `Show a => [a] -> IO String`
```
func [] = getLine
func (h:t) = do show h
                getLine
```

(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]

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

Answer: Gets all possible pairs of elements in the list except those where both elements are equal

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

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

Answer: Appends x to the end of the input list!

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

foo 3 [] = [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: Extracts the elements out of the "Just _"s in the list into their own list, ignoring Nothings

foo [Just 3, Nothing, Just 4] = [3, 4]

foo [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
```
All will be m : m : m ...

Answer: Returns a 2-Tuple where the first element is the maximum element of the input list, or the input int if the list is empty, and the second element is a list of the input ints that is of the same length as the original input list

foo [] 3 = (3, [])

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

foo [2, 3, 4] 5 = (4, [5, 5, 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 :: ??
foo = dropWhileM (const [True, False])
```

Answer:

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 (h1:t1) (h2:t2) = h1 : h2 : (weave t1 t2)
```

(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]
```

what if we misunderstood how (4e) foo function operates?

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

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

unnecessary case →

```
toMax [] = []
toMax lst@(h:t) = let mx = getMax lst h in
    lengthl lst mx
    where
        lengthl [] _ = []
        lengthl (h:t) mx = mx : (lengthl t)

getMax [] m = m
getMax (h:t) m = if h > m then getMax t h
                 else getMax t m
```

partial?

```
toMax [] = []
toMax lst = let max = maximum lst in
    replicate (length lst) max
```

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
```

Yuvraj Nayak

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 String → IO String

(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]`

[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 -> [b,b]`

(c) `a -> Maybe b`
`\x -> Nothing`

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> Bool]`
`\f ilst clst -> [f i c | i <- ilst, c <- clst, f i c]`

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

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`
`\f g lst -> [f e | e <- lst, g (f e)]`

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

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

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

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

1 2 3 4

5 4 3 2 1

1 2 3 4 5

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: Finds all pairs of elements in l s.t. no pairs w/ same element

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: adds x to end of list

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

foo 5 [] = [5]

(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 all Justs in a list of maybes and returns a list of elements inside Just

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: finds the max element of a list and returns it in a tuple with a list of ints.

(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:

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 (s) =
```

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
```

Maverick Kieu

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 → [b] → [b]

(e) **`getLine >>= putStrLn`**
(IO string) → (string → IO ()) → IO ()

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

(g) `(,) "42"`
(a → b → (a,b)) → [char] → [char] → ([char], [char])

(h) **`reverse . foldMap return`**
[a] → [m a]

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

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

(k) `filterM (const [True, False])`
a → [Bool]

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 -> [True && x]
```

(c) `a -> Maybe b`

```
\x -> Just x
```

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

```
foo a b = if (head a) > 0 then [head b == 'a'] else [head b == 'b']
```

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

```
undefined
```

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

```
foo f g lst -> let b = head (map f lst) in if (f b) then [b] else [b]
```

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

```
undefined
```

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

```
foo a b = if a > head b then [a] else [a]
```

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

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

```
foo (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]

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

Answer:
Returns a list of numbers if it is not equal to itself.

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

[1,3,2,1]

Answer:
Appends to the end of list with extra steps
f 1 [] = [1]
f 1 [1,2,3] = [1,2,3,1]

(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:

(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:

max 2 1

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

m' = 2
xs' = [3]

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

(:) *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:

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 :: [Int] -> [Int] -> [Int]
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!

```
toMaxHelper :: Int -> [Int] -> Int
toMaxHelper a [] = a
toMaxHelper a (x:xs) = if a<x then toMaxHelper x xs else
                                   toMaxHelper a x

toMax (x:xs)= fmap (\y -> maxHelper x xs) (x:xs)
```

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
```

Segev Elazar Mittelman

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, (putStrLn 43) not a function type

(g) `(,) "42"`
a -> (String, a)

(h) `reverse . foldMap return`
ill-typed, return doesn't return a monoid, it returns a monad

(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]`
`\b -> [(b&&b)]`

(c) `a -> Maybe b`
`\a -> 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]`
`\f p as -> filter p (fmap f as)`

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`
`undefined` (can't handle the Nothing case)

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

(i) `Show a => [a] -> IO String`
`\as -> return (foldMap show as)`

(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: calculates list of all pairs of nonequal elements of the input list of ints
foo [] = [], foo [1,2] = [(1,2),(2,1)]
foo [1] = []

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

Answer: appends the element x to the end of the list l.
foo 1 [] = [1]
foo 1 [2] = [2,1]

(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: foo takes a list of Maybe a values, and returns a list of all the values in Just constructors in the input list, removing the Nothings.
foo [] = [], foo [Nothing] = [], foo [Just 1, Just 2, Nothing, Just 5] = [1,2,5]

(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 pair of the maximum element in the list or m default if the list is empty and a list repeating m a number of times equal to the length of the input list
foo [1,2,3] 0 = (3,[0,0,0])
foo [2,3] 0 = (3,[0,0])
foo [3] 0 = (3,[0])
foo [5,5,5] 1 = (5,[1,1,1])
foo [] 1 = (1,[])

(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: returns the list of all tails of a list including the original list

foo [] = [[]]

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

foo "lol" = ["lol","ol","l",""]

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 (a:as) (b:bs) = a : b : weave as bs
weave _ _ = error "lists diff size"
```

(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 [] = []
toMax (x:xs) = let (m, as) = foo (x:xs) x
  in fmap (const m) as
```

Bonus:

```
toMax [] = []
toMax (x:xs) = let (m,l) = toMax' x 1 xs in replicate l m
  where
  toMax' m l [] = (m,l)
  toMax' m l (a:as)
    | a > m = toMax' a (l+1) as
    | otherwise = toMax' m (l+1) as
```

```
replicate n m | n <= 0 = []
otherwise = m : replicate (n-1) m
```

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
```

Ardacha Fayaz

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 → (a, String)

(h) `reverse . foldMap return`
Monoid m ⇒ [a] → m a

(i) `\l -> [(x,y) | x <- l, y <- l, x /= y]`
(Ord a, Eq a) ⇒ [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]`
`\a → if a then(replicate a 4)else [a]`

(c) `a -> Maybe b`
`\a → Just 4`

(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]`
`\f g → map f`

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`
`fun ma f = do a ← ma`
`b ← (f a)`
`return (a,b)`

(h) `Eq a => a -> [a] -> [a]`
`\a lst → filter (==a) lst`

(i) `Show a => [a] -> IO String`
`\lst → do map show lst`
`getLine`

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