Yusuf Bham

# 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. not an Property or Testable
- (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]`

(Int -> [Int]) -> [Int] -> [Int]

- (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`.
- The expression is equivalent to the identity function.
- The foldable in this call is [] - the instance for lists.

(b) `foldMap show "123456"`

show :: a -> String

foldMap (a -> String)

-> [Char] -> String

- (i) The expression will fail to typecheck.
- (ii) The monoid in this call is `Int`.
- (iii) The monoid in this call is `[Int]`.
- The monoid in this call is `String`.
- The expression is equivalent to the identity function.
- 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]]

## 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]`
pure

(c) `a -> Maybe b`
const Nothing

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`
\f ab -> map (uncurry f) $ zip a b

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

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

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`
\a f -> f (fromJust a) >>= (pure . (a,))

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

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

*Answer:*
Gives every pair of distinct integers in the list (in both orders)

foo [] = []
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:*
It appends at the end of a list

foo 3 [] = [3]
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:*
It gives the just elements of a list
(bar is just mapMaybe)

foo [] = []
foo [Nothing] = []
foo [Just 2, Nothing, Just 4]
= [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
```

*Answer:*
Gives the maximum of the 1st arg and a list of the same length as the first arg full of the second arg

if non-empty use second arg
foo [1,2,3,4] = (3, [4,4,4,4])
foo [7] 1 = (1, [1])
foo [4,5,6] 1 = (6, [1,1,1])

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

*Answer:*

Gives the tails of a list

foo [] = [[]]

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.

```
foo :: [a] -> [a] -> [a]
foo a b = concat . map (\(a, b) -> [a, b]) $ zip a b
```

(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 l = map (const m) l
  where m = maximum l
```

---

Bonus

```
toMax (x:xs) = snd $ go x xs
  where
    go e (x:xs) = (m, m:l)
      where (m, l) = go (max e x) xs
    go e [x] = (m, [m])
      where m = max e x
    go e [] = (e, [e])
```

## 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
```

# CMSC 488B: Midterm Exam (Spring 2022)

Matvey Stepanov
UID: 116684101

## 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]` → `x = [1],[2], .. [10]] -> foldList = List.fold (<>) mempty` ↳ `[]`

- (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"` `show :: a -> String`

- (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. `[char]`

## 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 String → IO ()`

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

ill-typed → should use >>

(g) `(,) "42"`

`a → (String, a)`

(h) `reverse . foldMap return`

`Foldable t => t 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)`

ill-typed

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

`[a] → [[a]]`

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]`
`\b -> [b]`

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

(d) `(Int -> Char -> Bcol) -> [Int] -> [Char] -> [Bool]`
`\f l1 l2 -> fmap (f . uncurry) (zip l1 l2)`

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

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`
`\f v l -> [f a | a <- l, v(f a) == True]`

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

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

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

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

## 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 all combinations (pairs) of distinct elements from given list.

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

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

*Answer:* appends x to end of given list l

Ex: 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:* Creates new list containing non-nothing values from original list. (Any Nothing values are removed and Maybe values are extracted).

Ex: foo [Maybe 4, Maybe 3, Nothing, Maybe 2]
= [4, 3, 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:*

Returns a pair: first element is the max of original list OR the default m passed in if original is empty.
Second element is a list of same size as original containing only the default m repeated — # of times.
↳ length of original list

Ex: foo [] 5 = (5, [])
foo [10] 5 = (10, [5])
foo [1,2,3,4] 20 = (4, [20,20,20,20])

(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)
weave _ _ = undefined
```

↳ should never reach this case if lists are of equal lengths.

(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!

Traverses given list only once to find max and length. Then uses replicate. Hope this counts :)

```
maxl :: [Int] -> (Int, Int)
maxl [] = undefined  -- shouldn't happen
maxl (x:xs) = maxlhelp x xs where
    maxlhelp :: Int -> [Int] -> (Int, Int)
    maxlhelp m [] = (m, 1)
    maxlhelp m (h:t) =
        let (m', l) = maxlhelp (max m h) t in
        (m', l+1)

toMax :: [Int] -> [Int]
toMax l = let (max, len) = (maxl l) in
          replicate len 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
```

Hammaad Khan
11664077

# 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.

show "123456"
show :: a -> String
a = String

String = [Char]

show "123456" = "\"123456\""

## 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 t ⇒ t 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]]

const [True, False])
= (\x → [True, False])

filterM
(a → M Bool) → [a] → M [A]
M = []

reverse . foldMap
[

Foldable t
Monoid M
⇒ (a → M) → t a → M

return: a → M a
[]
M = [a] by reverse

(a → [a]) → t 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`
`\x -> Just []`

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`
`\f xs ys -> (not (f 10 'a')) : (zipWith f xs ys)`

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

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`
`\f1 f2 (x:xs) = if (not (f2 (f1 x))) then [f1 x] else [f1 x]`

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

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

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

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

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

*Answer:* All pairs of elements from l that are not equal

foo [] = []
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 the end of l

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

x: [1, 2, 3, 4]
x: [4, 5, 2, 1, 7
(x, 4, 3, 2, 1, 7
[1, 2, 3, 4, x, 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:* Filters out Nothing from list of Maybe, unwraps Just

foo [] → []
foo [Nothing, Nothing] → []
foo [Just 1, Nothing, Just 3, Nothing] → [1, 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:* Repeats M n times where n is the length of the list
Also gives the maximum value of the list, or M if the list is empty

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

foo :: [a] -> [[a]]

foo gives all tails of the given list

foo [1, 2, 3] -> [[1, 2, 3], [2, 3], [3]]

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

w/ bonus:

```
lengthAndMax :: [Int] -> (Int, Int)
lengthAndMax (x:xs) = (1+l, max x x') where
                      (l, x') = lengthAndMax xs
lengthAndMax [x] = (1, x)
toMax xs = replicate l m where (l, m) = lengthAndMax 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
```

Yishen Zhao

# 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)`

a → b → String

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

ill typed

(e) `getLine >>= putStrLn`

IO ()

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

ill typed

(g) `(,) "42"`

a → (a, String)

(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])`

[[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 && True]
```

(c) `a -> Maybe b`

```
undefined
```

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

```
liftA2
```

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

```
liftM2
```

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

```
\f c a -> if c f a then [f a] else [f a]
```

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

```
\ma f -> do a <- ma
           liftM (a,) (f a)
```

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

```
\a (x:xs) -> if a == x then (x:xs) else (x:xs)
```

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

```
(putStr . show) >>= 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]
foo l = [ (x,y) | x <- l, y <- l, x /= y]
```

*Answer:* all possible pairs of elements, given that elements can't pair with themselves

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

*Answer:* append x to end of the 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:* filter out the "Nothing"s from the list

(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:* produces a tuple where the first is the largest element in the list, and the second is a history of the maxima during iterations

(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:* power set of a 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 (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 l = replicate (length l) (findMax l)
  where findMax []     = Int.MinValue
        findMax [x]    = x
        findMax (x:y:xs) = if x > y then
                             findMax x xs
                           else
                             findMax y 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
```

Philip Wang

# 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)`
a → b → String

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

(e) `getLine >>= putStrLn`
i
a >>= λx

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

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

(h) `reverse . foldMap return`
[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)`
('a', True)

(k) `filterM (const [True, False])`
(Monad m) => 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 -> [x]`

(c) `a -> Maybe b`
undefined

(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 h ->`

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

(h) `Eq a => a -> [a] -> [a]`
```
f _ [] = []
f a (x:xs) = if a == x then xs else []
```

(i) `Show a => [a] -> IO String`
`f s = let x = "all ++ s in 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]
foo l = [ (x,y) | x <- l, y <- l, x /= y]
```

*Answer:* Gets all pairs (x,y) such that $x \neq y$

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

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

*Answer:* Inserts x to end of l

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

(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:* bar returns a list whose elements are those such that f applied to them is not Nothing.

foo is just the identity function.

foo [Just 1, Just 2] = [1,2]
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:*

Foo returns a tuple whose first element is the max of the given list and the second element

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

```
* Not sure if max is defined
  in prelude but if not*
max l@(x:xs) = foldr (\x y -> if y > x then
                        x else y)
                     x l

toMax lst =
    let m = max lst in
    map (\x -> m) lst
```

Bonus! not sure if this counts but could do something like

## 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
```

Zachary
Gurnitz

(?/67)

# 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`
String → IO()

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

(g) `(,) "42"`
a → (a, [Char])

(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]]

filterM :: (a → m Bool) → [a] → m [a]
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)`

`(-1)`

(b) `Bool -> [Bool]`

`\b -> [b]`

(c) `a -> Maybe b`

`\x -> Nothing`

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

`zipWith`

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

`liftM2`

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

`\x y z -> filter y (map x z)`

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

`\x y -> | (Just c) y -> arbitrary | Nothing y -> undefined`

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

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

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

`\x -> return (concatmap show x)`

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

`\(x,y) z -> z 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:*
returns all ordered pairs with unequal terms in a list
foo [] = []
foo [1,2] = [(1,2), (2,1)]

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

*Answer:*
put an element at the end of a list
foo 'a' "def" = "defa"
foo 1 [3,4] = [3,4,1]

(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:*
make a list of values inside maybes in a list. (concat maybes)
foo [Just 1, Nothing] = [1]
foo [Nothing, Nothing] = []
foo [Just "ab", Just "cd"] = ["ab", "cd"]

(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 of a list, and a number repeated the length of the list times. If the list is empty, returns the number as the "max," and the repeat is zero times

foo [1,2] 3 = (max 2 1, 3:[3]) = (2, [3,3])
foo [1] 3 = (2, [3])
foo [] 20 = (20, [])
foo [2,4,1] 3 = (max 4 2, 3:[3,3]) = (4, [3,3,3])
foo [4,1] 3 = (max 4 4, 3:[3]) = (4, [3,3])
foo [1] 3 = (1, [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 :: ?? ~~(a -> m Bool)~~ -> [a] -> [[a]]
foo = dropWhileM (const [True, False])
```

*Answer:*

give a list of lists which are suffixes of the original list.

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

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 [] [] = []
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 x:xs = rep m' n' where
  (m',n') = f x xs 1

f :: a → [a] → Int → (a, Int)
rep :: a → Int → [a]

f m [] n = (m,n)
f m (l:ls) n = f (max m l) ls (n+1)
rep v 0 = []
rep v c = v: repeat v (c-1)
```

→ (rep is replicate with reversed inputs)

## 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
```

Jacob Ginzburg

# 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`
IO String -> (String -> IO()) -> IO()

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

(g) `(,) "42"`
String -> (char, (har)

(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 -> a) -> (Bool -> b) -> (a, b)

(k) `filterM (const [True, False])`
ill-typed

## 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 [not x]`

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

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`
`\f x y -> foldr (if foldr (\x' y' -> f x' y') x) y`

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

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`
`\f g xs -> foldr (\x acc -> if g (f x) then (f x) : acc else acc) xs`

(g) `Maybe a -> (a -> Gen b) -> Gen (a,b)`
`\x f -> if isJust x then arbitrary`

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

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

(j) `(a,b) -> (a -> b -> c) -> c`
`\f g -> let f = (x, y) in g 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:* Creates a list of tuples of all unequal elements in a list
foo [1,1,2,3] = [(1,2),(1,3),(2,3)]

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

*Answer:* appends x to the 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:* Composes a list of elements that aren't Nothing
foo [Just 1, Nothing, 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 maximum element of the given argument and the list and a list of maximum elements at each part of the list

foo [1,2,3] 4 = (4, [4,4,4])
foo [1,2,3] 1 = (3, [1,2,3])
foo [4,5,6] 5 = (6, [5,5,6])

(f) *Bonus!*

```
dropWhileM :: (Monad m) => (a -> m Bcol) -> [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 (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!

```
maxList (x:xs) = case foo x xs of
                   (a,b) -> Just a
                   _     -> nothing
maxList _ = nothing

toMax = foldr (let a = maxList in a) []
```

## 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
```

Aaron Ortwein

# 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"`
b → (String, b)

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

(c) `a -> Maybe b`
`id`

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`
`\f i c -> f (head i) (head c)`

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

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`
`\f g l -> map (\x -> if g (f x) then f x else f x) l`

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

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

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

(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:* Creates pairs out of every permutation of two distinct elements in a list
foo [1,2] = [(1,2),(2,1)]
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:* Moves the head of a non-empty list to be its last element
foo [1] = [1]
foo [1,2,3,4,5] = [2,3,4,5,1]

(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:* Extracts the values from a list of monadic Maybe values
foo [Nothing] = []
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:* Returns a pair consisting of the maximum element of the list (or a default value m for empty lists) and a list of the same length where all elements are m

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

(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 :: [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 [] = []
toMax [x] = [x]
toMax (x:xs) = let (m, _) = foo (x:xs) x in snd (foo (x:xs) 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
```

Hiteesh Nukalapati

# 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]`
ill-typed

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

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

(g) `(,) "42"`
a → (String, a)

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

a → m a

(a → m Bool) → [a] → m [a]

a → b → 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 == True then [x] else [False].`

(c) `a -> Maybe b`

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`
`\f il cl -> if f (head il) (head cl) == True then [True] else [False].`

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

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

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

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

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

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

## 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:*
takes an int list, creates Pairs with unequal elements (from the same list)
foo [1,0] = [(1,0), (0,1)].

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

*Answer:*
foo 3 [1, 2]
Adds an element to the front of a reversed list & then reverses this 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:*

(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 [1, 2] 3
= (2, [1, 3])
or
This function returns a pair, where the first element is the maximum of the current list (list that is passed to it).
& the second element (of the pair) is a list with the rest of the elements and 2nd parameter formed to the list.

Foo [2] 3 = (2, [3])
(2, [1, 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 l r~~
~~weave [] [] =~~
weave l r = reverse (weaveHelper l r [])
l s h

weaveHelper l r acc :: [a] -> [a] -> [a]
weaveHelper [] [] acc = acc
weaveHelper (h1:t1) (h2:t2) acc = ~~h1:h2:acc~~
weaveHelper t1 t2 (h1:h2:acc)

(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 l = let len = length l in
let (max, temp) = foo l 0 in
listbuild max len

listbuild ele size :: Int -> Int -> [Int]
listbuild _ 0 = []
listbuild ele size = ele : (listbuild ele (size - 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 => Crd 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
```

BAHAA HARRAZ

# 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`
Monad m => [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])`
~~ill-typed~~ ill-typed

## 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 b = [b, True]
```

(c) `a -> Maybe b`

```
f a = Just
```

(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 h lst = filter h (map g lst)
~~f g h lst = map g (filter h lst)~~ ~~f g h lst = filter h (map~~
```

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

```
>>=
```

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

```
f a lst = filter (a ==) lst
```

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

```
return
```

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

```
f (a, b) g = g 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]

Examples for 4b:
foo [1, 1, 2] = [(1, 2)]
foo [1, 2, 3] = [(1,2), (2,3), (1,3)]

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

*Answer:* takes a list and creates a new list of all combinations of elements in the original list, so long as they aren't equal.

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

*Answer:* appends the first argument to the reversed second argument

foo 3 [1, 2] = [3, 2, 1]
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:* Takes a list of maybe's, removes all Nothing's and returns a list of all values inside Just's.

foo [Just 2, nothing] = [2]
foo [Just 2, just 3] = [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:* Takes a list of ints and an int and returns tuple of form (max value of list, list of second value repeated (length list) times)

If list is empty, return (value, [])

foo [] 3 = (3, [])
foo [1,2] 3 = (2, [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:* Drops all elements of a 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 (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 lst = let len = length lst in let m = maximum lst in
                replicate len m

Bonus version:  toMax [] = []
toMax lst =
  let aux l = foldl (\acc x -> if x > (fst acc) then
                                 (x, (snd acc) + 1)
                               else
                                 (fst acc, (snd acc) + 1)) (0, 0) in
  let (m, len) = aux lst in replicate len 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
```

Mitchell Fanger

# 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 :: (Moncid m, Foldable t) => (a -> m) -> t a -> m
```

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

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

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

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

*Answer:* Append 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])

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

foo [1,3,2] 2 = (3, [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 :: (Mcnoid 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 a`

```
\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)`

```
\x -> do
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)
```

```
\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])

(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`

IO ()

(>>=) :: M a → (a → M b) → M b

getLn : IO String

putStrLn : String → IO ()

IO String → String → IO () → IO ()

(getStrLn 42 putStrLn 43)

IO () → IO ()

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

Ill-Typed

(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 _ []     = []
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 than 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`

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]
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 [5] =
```

## 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`
(IOstring) → (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:*

max 2 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:*

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

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

m' = 2
xs' = [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 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 pair of the maximum element in the list or m default. 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
```

Azhdaha 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 ⇒ t 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`

## 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. It is 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
```

*Answer:*
returns a tuple with the first element as the biggest element in the first argument (list) and as the second element of the tuple as 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"`

(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)]

(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`
`\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 <- fy; return (y, z)`

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

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

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

`\f g xs -> (map f (filter g 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
```

*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 [] 0 = (0, [])

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

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

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

## 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

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 suffixes 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 facc acc [] = []
dfold p facc acc (x:xs) = do
  (f', l) ← dfold p (p x facc) 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 of that list (subsequence) 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
Zon

# 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 <- fa, 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.

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

eg. [1, 0, 1] → [(1,0), (0,1)]

*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 t, 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 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 -> 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]

(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 [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 = (2, [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 :: ?? [a] -> m [a]
foo = dropWhileM (const [True, False])
```

*Answer:* This will just return the empty list.

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 = maximum
    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])
       m  xs'
    (4, [2,2])
     m'  xs'
(4, [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 :: ?? [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 il 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 -> 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~~ x,y pair in l then return the list of all (x,y) pairs
return every possible pair of nonequal values in l
foo [] = []
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:* ~~reverses~~ appends x to l
foo [] = [A]
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:* ~~Takes~~ 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 Mahib

# 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 :: (Mcnoid 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)`
$a \to b \to (a, b)$

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

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

(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`
$[a] \to [a]$

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

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

(k) `filterM (const [True, False])`
$[a] \to [[a]]$

## 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 b -> 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 p2 lst -> map f lst)`

(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
```

# 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
```

Kameron
Hnath
11600654S

# 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)`

Show a ⇒ a → a → String

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

a → [a] → [a]

(e) `getLine >>= putStrLn`

String → IO String → IO ()

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

IO ()

(g) `(,) "42"`

a → (a, String)

(h) `reverse . foldMap return`

Monad m ⇒ [a] → m a

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

Ord 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 -> replicate 5 x`

(c) `a -> Maybe b`
`Just`

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`
`\x y -> x > 0 && y == 'a'`

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

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`
`\f g l -> foldr (g . f) [] l`

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

(h) `Eq a => a -> [a] -> [a]`
`\x l -> case l of`
`[] -> []`
`(h:r) -> if x == h then r else [x]`

(i) `Show a => [a] -> IO String`
`\l -> case l of`
`[] -> do getLine`
`(h:t) -> do putStrLn h`
`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:* Constructs a list of all pairs of non-equal integers in a list.
Foo [1,2] = [(1,2), (2,1)]

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

*Answer:* Adds an element to the end of a list

Foo 3 [1, 2] = [1,2,3]
Foo 'c' "ab" = "abc"

(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 the maybe monad from a list.
Foo [Nothing] = []
Foo [Maybe 2, Nothing, Maybe 4] = [2,4]
Foo [] = []

(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:* On non-empty list: replace all elements of the list with m and return (largest element in list, list of ms)
On empty list: return (m, [])

foo [] 2 = (2, [])
foo [2,4,3] 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:* Drops elements from ~~the~~ start of ~~the~~ list, each with a 50% chance, and stops when an element is not dropped.

Example:
Foo [1,2,3] could be const [1,2,3]
or const [3]
but not const [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 [] [] = []
weave (x:xs) (y:ys) = x : y : (weave xs ys)
weave _ _ = error "lists not same length"
```

(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 xs = replicate (length xs) (maximum 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
```