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 ∈ 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
→ 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 a b -> 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:
2nd 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 map Maybe)

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 2nd second arg
foo [1,2,3] 4 = (3, [4,4,4])
foo [7] 1 = (1, [7])
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]`

(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 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 -> Bool) -> [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 :: b String -> 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 ⇒ (a → M) → t a → M
Monoid 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]

(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
```

Yishan 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

(?'6?

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 → ([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)`
ill-typed

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

filterM :: (a → m Bool) → [a] → m [a]
const [True, False] :: a → [Bool]

m a → (a → m b) → m b

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

(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 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)
```

```
f :: a -> [a] -> Int -> (a, Int)
rep :: a -> Int -> [a]
```

→ (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 "12345€"`

- (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, Char)

(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 element 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 = reverse (weaveHelper l r [])

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

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

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

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