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
116006545

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 ints 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
```