CS 340 Spring 2023

Midterm Exam

Question Booklet

Instructions:

- This exam is closed-book, closed-notes. Electronic devices of any kind are not permitted.
- You will write all your responses in the provided answer booklet. Please write legibly in the space provided for each exercise and, if necessary, clearly indicate your final answer.
- You may use the question booklet as scratch paper, but we will only score your answer booklet.
- Turn in both the exam question and answer booklets.

Concepts (24 points):

For each of the following questions, choose the *single best answer* from the choices provided. Completely fill in the bubble corresponding to your choice in the answer booklet. If you make a mistake (e.g., by filling in multiple bubbles), write the letter of your choice next to the question number in the answer booklet.

1. What is *not* possible in a purely functional language?

   a. strict evaluation
   b. function composition
   c. tail-call recursion
   d. side-effects

2. When we say that Haskell is a *statically typed* language, we mean that:

   a. type-errors are caught automatically at run-time
   b. it has a sophisticated type inference system
   c. functions, but not values, must have type signatures
   d. type-checking is performed at compile time

3. We say that a function of many arguments is *curried* when:

   a. it composes its arguments into new functions
   b. it accepts a tuple created from all its arguments
   c. it fails to type-check its arguments correctly
   d. it takes its arguments one at a time, returning a new function each time

4. The type signature of a *polymorphic function* will contain at least one:

   a. type variable
   b. class constraint
   c. higher-order function
   d. list parameter

5. When a *class constraint* is used in a type signature, it means that:

   a. type-checking will be performed by the class
   b. the associated type variable must be an instance of the specific class
   c. methods of that class will be called on the associated argument
   d. the described function can only be called through an instance of the class

6. *Property-based testing* is an approach to testing where:

   a. type-inference is the primary method of ensuring correctness
   b. we manually write input/output specifications for the system
   c. we define properties of a system and then rely on randomly generated test cases to check that the properties hold
   d. full test coverage is achieved by testing every possible input to every function, based on type inference information

7. One of the undesirable consequences of using a *lazy evaluation* strategy in Haskell may be:

   a. it is not possible to write tail-recursive functions
   b. expressions are evaluated even when their results are not needed
   c. unevaluated thunks build up in memory and negatively impact performance
   d. expressions evaluate to the incorrect results because they are performed in the wrong order

8. A *higher-order function* is a function that:

   a. can only be called by other higher-order functions
   b. either accepts or returns a function
   c. makes use of parametric polymorphism
   d. makes use of recursion

9. List comprehension syntax in Haskell is essentially syntactic sugar for:

   a. `map` and `filter`
   b. `foldr` and `foldl`
   c. `zip` and `unzip`
   d. `iterate` and `until`

10. In a *tail-recursive* function:

    a. the result of the recursive call is pattern matched
    b. the recursive call is the last operation performed
    c. an accumulator is always used to "build up" a result
    d. `foldl` is used to process the result of each recursive call

11. A scenario where it is necessary to use a right fold (`foldr`) over a left fold (`foldl`) is when:

    a. the input list is infinite
    b. the combining function is commutative
    c. the combining function is left-associative
    d. it is desirable to reverse the order of the input list

12. A scenario where a strict left fold is likely more efficient than a right fold is when:

    a. the input list is infinite
    b. the combining function is commutative
    c. the left fold is implemented tail-recursively
    d. the combining function is strict in both arguments

Function type matching (16 points):

Match each of the function definitions on the left with its type signature on the right. Some of the type signatures are not used. In the answer booklet, write the letter (one of A-M) designating your chosen type signature in the blank corresponding to the numbered function definition.

```
-- 1
fn = filter even


-- 2
fn x y = (head y) : show x


-- 3
fn [] _ = []
fn (x:xs) (y,z) = (z,x) : fn xs (z,y)


-- 4
fn f x = map (\(y,z) -> f y z) x


-- 5
fn x y = [show v ++ w | v <- x, w <- y]


-- 6
fn f g h = g . h . f


-- 7
fn [] y = y
fn (x:xs) y | x < y = fn xs x
            | otherwise = fn xs y


-- 8
fn f x = foldl (flip f) (f x [])
```

```
-- A
fn :: Num a => (a -> Bool) -> [a] -> [a]

-- B
fn :: Ord a => [a] -> a -> a

-- C
fn :: Integral a => [a] -> [a]

-- D
fn :: Num a => [a] -> (a -> b) -> b

-- E
fn :: (b -> a) -> [a] -> b -> [a]

-- F
fn :: Show a => [a] -> [String] -> [String]

-- G
fn :: (a -> [b] -> [b]) -> a -> [a] -> [b]

-- H
fn :: String -> String -> String

-- I
fn :: (a -> b -> c) -> [(a,b)] -> [c]

-- J
fn :: ((b,a) -> c) -> [(b,a)] -> [c]

-- K
fn :: (a -> b) -> (c -> d) -> (b -> c) -> a -> d

-- L
fn :: Show a => a -> String -> String

-- M
fn :: [a] -> (a,a) -> [(a,a)]
```

Polymorphic functions (12 points):

For each of the following parametric polymorphic function type signatures, write a valid corresponding function definition in the space provided in the answer booklet. A valid definition is one which, when applied to as many input arguments as possible, will evaluate to a result of the correct type (in the case of a list result, the list should be non-empty whenever possible). Your definitions may use any of the HOFs covered in class.

1. `g1 :: (a -> b -> c) -> a -> [b] -> [c]`
2. `g2 :: (a -> b) -> (c -> d) -> (b -> d -> e) -> a -> c -> e`
3. `g3 :: b -> ((a -> b) -> c) -> c`
4. `g4 :: (a -> (b,c)) -> (b -> c -> d) -> a -> d`

Function evaluation (12 points):

Consider the following function definitions:

```
h1 :: (a -> b) -> (b -> c) -> a -> c
h1 f g x = g (f x)

h2 :: (a -> [a] -> [a]) -> [a] -> [[a]]
h2 f [x] = [[x]]
h2 f (x:xs) = f x xs : h2 f xs

h3 :: (Integer -> Bool) -> (Integer -> Integer) -> Integer -> [Integer]
h3 p f x | p x = [x]
         | otherwise = x : h3 p f (f x)

h4 :: (a -> Bool) -> [a] -> (a -> b) -> [b]
h4 p xs f = map f (filter p xs)
```

In the answer booklet you will find expressions that make use of the function definitions above. For each expression, write the result of evaluating the expression in the space provided.

Code completion (20 points):

For this problem you are to correctly complete the partial function definitions found in the answer booklet, whose behavior (along with rules/restrictions concerning your implementation) are specified below. Do *not* change any of the provided code!

1. `minmax` takes a non-empty list and returns a tuple containing its minimum and maximum elements. E.g., `minmax [3,1,6,2] = (1,6)`. You may use the existing functions `min` and `max` (which are binary functions that return the min/max of their two arguments).

2. `collections` takes an integer $n \geq 0$ and a list of tuples (i, x) where i is an integer ≥ 0 that indicates how many copies of the value x can be used. It returns a list of lists of length n that can be constructed from the available values (where order doesn't matter). E.g., `collections 3 [(3,'A'),(1,'B'),(2,'C')] = ["AAA","AAB","AAC","ABC","ACC","BCC"]`.

3. `isRepetitionsOf` takes two finite lists $l1$ and $l2$ and decides if $l2$ consists of 0 or more complete repetitions of $l1$. E.g., `isRepetitionsOf "hey" "heyheyhey" = True` and `isRepetitionsOf "abc" "abcab" = False`. Your implementation should use a tail-recursive helper function.

4. `bin` takes a list of predicates ps and a list of values xs, and returns a list of lists, where each sublist contains only the values from xs that satisfy the corresponding predicate from ps. E.g., `bin [even,odd,(<3)] [1..5] = [[2,4],[1,3,5],[1,2]]`. Your implementation should use `foldr` instead of explicit recursion.