2018 Exam 1 Review

NAME

CSci 450/503: Organization of Programming Languages
Fall 2017, Examination #1

1. **(27 points)** The following table has columns of Haskell parameter patterns and corresponding arguments. As shown in the example line, indicate whether each match succeeds or fails and, for successful matches, indicate the bindings of values to identifiers (if any).

Pattern	Argument	Succeeds (Yes/No)	Bindings
x	0	yes	x ← 0
`[ ]`	`[ ]`	yes	none
`[x]`	`[1]`	yes	x ← 1
`(x:y)`	`[[ ],[ ]]`	yes	x ← [], y ← [[]]
`x`	`"java"`	yes	x ← "java"
`[x,y]`	`"ruby"`	No	
`(x:y)`	`"python"`	yes	x ← 'p', y ← "ython"
`(w:x@(y:z))`	`[4,3,2,1]`	yes	w ← 4, y ← 3, z ← [2,1], x ← [3,2,1]
`((x,y):z)`	`[(2,4),(1,3)]`	yes	x ← 2, y ← 4, z ← [(1,3)]
`(x:y:_:z)`	`"elixir"`	yes	x ← 'e', y ← 'l', z ← "xir"

2. **(12 points)** Show appropriate polymorphic type declarations (signatures) for the following Haskell functions (e.g., as defined in the standard Prelude library).

(a) Cons `(:)` as a function — `(:) :: a -> [a] -> [a]`

(b) `head` — `head :: [a] -> a`

(c) `++` (append) — `(++) :: [a] -> [a] -> Int`

(d) `length` — `length :: [a] -> Int`

actually in Prelude

↓

`Foldable t => t a -> Int`

```
rad :: Eq b => [b] -> [b]
rad (x : xs@(y:_))
  | x == y = rad xs
  | x /= y = x : rad xs
rad xs = xs
```

3. D (3 points) Which one of the following best describes the recursive style of the Haskell function `rad` defined above.

   A. tail recursive B. nonrecursive C. logarithmic recursive
   D. backward recursive E. stepwise refinement F. leftmost outermost

4. A (3 points) Which one of the following best describes termination of the Haskell function `rad` defined above? assume xs finite

   A. The list argument gets shorter on each recursive call until it becomes empty.
   B. The list result gets longer on each recursive call until it reaches its maximum length.
   C. `x` gets smaller on each recursive call until it reaches zero.
   D. The function does not terminate normally.
   E. The accumulating parameter converges to zero.

5. D (3 points) Which one of the following best describes the execution of the pattern `(x : xs@(y:_))` in the second clause of the Haskell function `rad` defined above (assuming all parts of the argument are finite)?

   A. It matches an empty list.
   B. It matches a one-element list.
   C. It matches any list with exactly two elements.
   D. It matches any list of two or more elements
   E. It matches only lists containing `@` as their second elements

6. C (3 points) Which one of the following best describes the value returned by the Haskell function `rad` defined above?

   A. a copy of the input list
   B. the input list with only one occurrence of each value
   C. the input list with one occurrence of any adjacent duplicate values
   D. the input list rearranged in ascending order
   E. a permutation of the input list

7. E (3 points) Which one of the following best describes the effect of the `Eq b =>` notation in the definition of `rad`?

   A. It is just documentation; it has no effect.
   B. Function `rad` has a higher-order parameter `Eq`.
   C. Function `rad` is in module `Eq`.
   D. Plymorphic type `b` is restricted to types with ordered comparisons (e.g., `<`).
   E. Polymorphic type `b` is restricted to types in type class `Eq`.

```
cav :: [Double] -> Double
cav (x:xs) = cavaux xs (x,1)
    where cavaux [] (s,c)     = s / c
          cavaux (x:xs) (s,c) = cavaux xs (s+x,c+1)
```

8. A (3 points) Which one of the following best describes the recursive style of the Haskell function `cavaux` defined above?

   A. tail recursive B. nonrecursive C. logarithmic recursive
   D. backward recursive E. stepwise refinement F. leftmost outermost

9. C (3 points) Which one of the following best describes the value returned by the Haskell function `cav` defined above?

   A. the sum of the input list of numbers
   B. minimum of the input list of numbers
   C. the average of the input list of numbers
   D. the input list rearranged in ascending order
   E. the number of elements in the list

10. E (3 points) Which one of the following best describes the second parameter of the Haskell function `cavaux` defined above?

    A. tuple B. accumulator C. two-element list
    D. identity element E. both A and B F. both B and C

11. B (3 points) Which one of the following best describes the meaning of `where` as used in the Haskell function `cav` above?

    A. defines `cav` as a private function available in function `cavaux` below
    B. defines `cavaux` as a private function available in function `cav` above
    C. has no Haskell meaning; it is just documentation
    D. defines `cavaux` as a method of class `cav`
    E. defines `cavaux` as a public function available anywhere in the enclosing Haskell `module`

12. B (3 points) Which one of the following best describes the time complexity of function `cav xs`?

    A. O(log m) where m is the length of argument `xs`
    B. O(m) where m is the length of argument `xs`
    C. O(m log m) where m is the length of argument `xs`
    D. O(m^2) where m is the length of argument `xs`
    E. O(m) where m is the value of `head xs`

13. C (3 points) Which one of the following best describes what happens on the call `cav []` (in the function on the previous page)?

A. The function returns the value 0.0.
B. The compiler generates an error message.
C. The function throws an exception (for a non-exhaustive pattern).
D. The function eventually terminates with a runtime stack overflow.
E. The function never returns; you must reboot the system.

14. E (3 points) The imperative programming paradigm is best characterized by which one of the following.

A. The underlying model is the mathematical function.
B. The program has no implicit state; any needed state information must be handled explicitly.
C. The program expresses what is to be computed (rather than how it is to be computed).
D. The underlying model is the mathematical relation.
E. The program has an implicit state that is modified (i.e., side-effected) by a sequence of commands.

15. **(9 points)** Suppose `xsss = [["up"], ["to","no"], ["good"]]`. (Careful: This is a list of lists of lists of characters.) Show the values returned by the following expressions.

(a) `length xsss` 3
(b) `tail xsss` `[["to","no"],["good"]]`
(c) `length (tail (tail xsss))` 1

16. **(2 points bonus)** If the Department were to create a new *advanced* programming language and programming elective for undergraduates, which language would you consider the most helpful and/or interesting? (e.g., Rust, JavaScript, Python, Scala, etc.)

17. **(6 points)** We say that Haskell is *referentially transparent.*

(a) What does referential transparency mean?
(b) Why is it important?

(a) A symbol (e.g. name) always equals the same value in some context — equals can be replaced by equals

(b) Supports ability to reason about programs, substitution-based execution model, parallel execution, efficient transformations

18. **(12 points)** Define Haskell definitions for the following set of text-justification functions. Give both the type signatures and the equations in the function bodies.

    You can use basic features like recursion, pattern-matching, and "cons". You may also use the Prelude functions such as `length`, `take`. `drop`, `++`, `!!`, and `reverse` if appropriate.

    (a) Function `spaces n` returns a string of exactly `n` space characters (i.e., the character `' '`).

    (b) Function `left n xs` returns a string of length `n` in which the string `xs` begins at the head (i.e., left end).

    Examples: `left 3 "ab"` yields `"ab "` and `left 3 "abcd"` yields `"abc"`.

```
{- CSci 450/503 Fall 2017
   Examination #1
   H. Conrad Cunningham

12345678901234567890123456789012345678901234567890123456789012345678901234567890

2017-09-21: First version

To see types, give ghci command :set +t

-}

module Exam01
where

rad :: Eq a => [a] -> [a]
rad (x : xs@(y:_))
    | x == y = rad xs
    | x /= y = x : rad xs
rad xs = xs

cav :: [Double] -> Double
cav (x:xs) = cavaux xs (x,1)
    where cavaux [] (s,c)     = s / c
          cavaux (x:xs) (s,c) = cavaux xs (s+x,c+1)

{- Define the following set of text-justification functions.
   You may want to use Prelude functions like take, drop,
   ++, and `length` as well as pattern matching, cons, etc.
-}

-- spaces n returns a string of length n containing only space
-- characters (i.e., the character ` ' `).

spaces :: Int -> String
spaces n
   | n <= 0    = ""
   | otherwise = ' ' : spaces (n-1)

-- left n xs returns a string of length n in which the string
-- xs begins at the head (i.e., left end).
-- Examples: left 3 "ab" yields "ab "
-- left 3 "abcd" yields "abc"

left :: Int -> String -> String
left n _ | n <= 0  = []
left n []          = spaces n
left n (x:xs)      = x: left (n-1) xs
```

#18

(a)

(b)

```
left n xs
  | n <= l  = take n xs
  | otherwise = xs ++ spaces (n-l)
      where l = length xs
```