Fall, 2005 Name: _____

Com S 541 — Programming Languages 1

Test on Functional Languages and Haskell

Special Directions for this Test

This test has 9 questions and pages numbered 1 through 10.
This test is open book and notes.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
When you write Haskell code on this test, you may use anything in the Haskell Prelude without writing it in your test. You are encouraged to define functions not specifically asked for if they are useful to your programming; however, if they are not in the Prelude, then you must write them into your test.

1. (5 points) What is the difference between type checking and type inference?

2. (5 points) In Haskell, write a function

```
> mapEach :: (a -> b) -> [[a]] -> [[b]]
```

such that `mapEach` f xss returns a list of lists, containing the result of applying f to each element of each sublist of xss. For example,

```
mapEach (+ 3) [] = []
mapEach (+ 4) [[]] = [[]]
mapEach (+ 2) [[5, 4, 1], [7, 6], [8, 12]]
           = [[7, 6, 3], [9, 8], [10,14]]
mapEach (> 3) [[1, 7], [0, 4], [9, 10], []]
           = [[False,True],[False,True],[True,True],[]]
mapEach (\y -> y*y) [[0, 4], [9, 10, 2, 7, 5], [1 .. 3]]
                 = [[0,16], [81,100,4,49,25], [1, 4, 9]]
```

3. (10 points) Write a function

```
> strideIndexes :: Integral a => a -> [b] -> [a]
```

that takes a number, `len`, such that `len` > 0, and a list of elements, `xs`, and returns a list of zero-based indexes, such the i^{th} element of the result is a zero-based index into `xs` such that i is evenly divisible by `len`. (Hint: in your solution, you may want to use Haskell's `mod` operator, which has type `Integral a => a -> a -> a` and computes the mathematical "modulo" operation.)

Note that your code must work for both finite and infinite lists. You may assume that the `len` argument is strictly greater than 0.

The following are examples.

```
strideIndexes 1 [] = []
strideIndexes 2 [0, 1, 2, 3, 4] = [0, 2, 4]
strideIndexes 3 ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'] = [0,3,6]
take 7 (strideIndexes 1 [0 ..]) = [0,1,2,3,4,5,6]
take 9 (strideIndexes 5 [0 ..]) = [0,5,10,15,20,25,30,35,40]
strideIndexes 5 [0 ..] = [0, 5 ..]
```

4. (10 points) Implement the function

```
> movingAverage :: Fractional a => [a] -> [a]
```

that takes an infinite list of numbers, `xs`, and returns an infinite list of the arithmetic means of each adjacent pair of numbers in the list, in order. For example:

```
movingAverage [0, 0 ..] = [0.0, 0.0 ..]
take 5 (movingAverage [1 ..]) = [1.5, 2.5, 3.5, 4.5, 5.5]
take 5 (movingAverage [2, 4 ..]) = [3.0, 5.0, 7.0, 9.0, 11.0]
take 5 (movingAverage ([0] ++ [2, 4 ..])) = [1.0, 3.0, 5.0, 7.0, 9.0]
take 7 (movingAverage ([0, 10, -100, 1000, -10000] ++ [2, 4 ..]))
                    = [5.0, -45.0, 450.0, -4500.0, -4999.0, 3.0, 5.0]
```

Note that your function may assume that the argument, `xs`, is infinite.

5. (10 points) Implement the function

```
> movingMax :: Ord a => [a] -> [a]
```

that takes an infinite list of ordered values, `xs`, and returns an infinite list of the maximums of each adjacent pair of values in the list, in order. For example:

```
movingMax [0, 0 ..] = [0, 0 ..]
movingMax [1 ..] = [2 ..]
take 5 (movingMax [2, 4 ..]) = [4, 6, 8, 10, 12]
take 5 (movingMax [0, -2 ..]) = [0, -2, -4, -6, -8]
take 7 (movingMax ([0, 10, -100, 1000, -10000] ++ [2, 4 ..]))
                = [10, 10, 1000, 1000, 2, 4, 6]
```

Note that your function may assume that the argument, `xs`, is infinite.

6. (15 points) Write a function,

```
> moving :: (a -> a -> a) -> [a] -> [a]
```

such that `moving` is a generalization of the previous two problems. For example, if we define

```
> movingAverage2 = moving (\ x y -> (x+y)/2)
> movingMax2 xs = moving max xs
> movingSum = moving (+)
```

then we have the following

```
take 5 (movingAverage2 [2, 4 ..]) = [3.0, 5.0, 7.0, 9.0, 11.0]
take 7 (movingAverage2 ([0, 10, -100, 1000, -10000] ++ [2, 4 ..]))
                     = [5.0, -45.0, 450.0, -4500.0, -4999.0, 3.0, 5.0]

movingMax2 [1 ..] = [2 ..]
take 5 (movingMax2 [2, 4 ..]) = [4, 6, 8, 10, 12]
take 5 (movingMax2 [0, -2 ..]) = [0, -2, -4, -6, -8]
take 7 (movingMax2 ([0, 10, -100, 1000, -10000] ++ [2, 4 ..]))
                 = [10, 10, 1000, 1000, 2, 4, 6]

movingSum [0 ..] = [1, 3 ..]
movingSum [2, 4 ..] = [6, 10 ..]
movingSum [1, 1 ..] = [2, 2 ..]

moving (\x y -> y) [0 ..] = [1 ..]
```

7. (15 points) In Haskell, consider the following datatype that describes file system entities.

```
> data FSEntity = File Contents | Dir [(EName, FSEntity)]
> type EName = String
> type Contents = String
```

An `FSEntry` is either a *file*, if it has the form `(File` s`)`, or a *directory*, if it has the form `(Dir` m`)`.

Your first task is to write a function

```
> okFSEntity :: FSEntity -> Bool
```

that takes an `FSEntity` value, `fse`, and determines whether it is *ok* in the sense that, if `fse` is a directory of the form `(Dir` m`)`, then all of the following must hold:

- the list m is not empty,
- the *entity names*, i.e., the `EName`s in the first components of each pair of m, are distinct strings (i.e., none are equal to each other),
- one of the entity names of m is the special name "`..`" (which names the directory's parent directory), and
- the value of the `FSEntity` associated with the name "`..`" (i.e., the second component of a pair whose first component is "`..`") is a directory.

There are no other conditions on a file system entity being "ok". Thus, an file is always ok. Since a directory entity may refer to itself (the structure may be circular), it is *not* part of checking that an entity is ok to check that subdirectories contained in the directory are ok (as that could lead to infinite loops). Entity names in an ok directory may be empty and may contain arbitrary characters.

There are examples on the next page. Please write your solution in the space below.

The following are examples for the `okFSEntity` problem on the previous page.

(Recall that the Haskell `let` expression allows mutual recursion among the names being defined and the values used for those names—like Scheme's `letrec`. Hence, in the fifth example below, `root1` is defined to be a directory that maps the name "`..`" to `root1` itself.)

```
okFSEntity (File "my contents")   = True
okFSEntity (File "another one")   = True
okFSEntity (Dir [])               = False
okFSEntity (Dir [("z", File "")]) = False

let root1 = (Dir [("..", root1)])
in okFSEntity root1               = True

let br2 = (Dir [("..", File "")])
in okFSEntity br2                 = False

let root2 = (Dir [("..", root2)])
    root3 = (Dir [("..", root2)])
in okFSEntity root3               = True

let br = (Dir [])
    root4 = (Dir [("..", br)])
in okFSEntity root4               = True

let br5 = (Dir [("..", br5), ("..", br5)])
in okFSEntity br5                 = False

let root6 = (Dir [("..", root6), ("home", File "junk")])
in okFSEntity root6               = True

let r7 = (Dir [("..", r7), ("home", hd)])
    hd = (Dir [("f1", File ""), ("..",  r7), ("junk", File "nada")])
in okFSEntity hd                  = True

let r8 = (Dir [("..", r8), ("home", hd2)])
    hd2 = (Dir [("f1", File ""), ("..",  r8), ("f1", File "")])
in okFSEntity hd2                 = False
```

8. (15 points) In this problem, we will always assume that values of type `FSEntity`, satisfy the predicate `okFSEntry` from problem 7.

   Your task in this problem is to write a function,

```
> fetchFromPath :: FSEntity -> Path -> Maybe FSEntity
> type Path = [EName]
```

   that takes a file system entity, `curdir`, that represents the current directory, and a Path (i.e., a list of strings), `path`, and returns a `Maybe` value; the result which either is `(Just` fse`)`, if fse is found by following `path` starting at `curdir`, or is `Nothing` if there is no such file system entity. You can assume that the `curdir` argument is a directory (i.e., has the form `Dir` m), for some finite list of pairs, m. You can also assume that the list `path` is non-empty.

   Recall that the data type `Maybe a` is defined in the Haskell Prelude as follows.

```
data Maybe a = Nothing | Just a   deriving (Eq, Ord, Read, Show)
```

   There are examples on the next page. Please write your answer in the space below.

For purposes of giving examples of the problem on the previous page, we will make the following definitions.

```
> r0 = Dir [("..", r0)]
> root1 = Dir [("..", root1), ("home", hd)]
> hd = Dir [("f1", File ""), ("..",  root1), ("junk", File "nada"),
>           ("rhe", rushome), ("gtl", gtlhome)]
> rushome = Dir [("cs541", r541), ("..",  hd), (".login", File "!/bin/sh...")]
> r541 = Dir [("..", rushome), ("fib.hs", File "fib n = ...")]
> gtlhome = Dir [("cs541", g541), ("..",  hd), ("cs342", g342), ("root", root1)]
> g541 = Dir [("..", gtlhome), ("hw1.tex", File "Write ...")]
> g342 = Dir [("hw0.tex", File "Hand in ..."), ("..", gtlhome)]
```

Using the above definitions, the following are examples of the `fetchFromPath` function you are to implement.

```
(fetchFromPath r0 [".."]) = Just r0
(fetchFromPath r0 ["unk"]) = Nothing
(fetchFromPath r0 ["..", "unk"]) = Nothing
(fetchFromPath r0 ["unk", "..", "x"]) = Nothing
(fetchFromPath root1 [".."]) = Just root1
(fetchFromPath root1 ["home"]) = Just hd
(fetchFromPath root1 ["..", "home"]) = Just hd
(fetchFromPath root1 ["..", "home", ".."]) = Just root1
(fetchFromPath hd ["rhe"]) = Just rushome
(fetchFromPath root1 ["home", "rhe"]) = Just rushome
(fetchFromPath hd ["..", "home", "rhe"]) = Just rushome
(fetchFromPath hd ["..", "home", "..", "home", "rhe"]) = Just rushome
(fetchFromPath root1 ["home", "gtl", "cs541"]) = Just g541
(fetchFromPath root1 ["home", "gtl", "cs541", "hw1.tex"]) = Just (File "Write ...")
```

9. (15 points) Consider the following simplified (higher-order) abstract syntax for a logic over one Integer variable.

```
> data Formula =
>                Pred (Integer -> Bool)   -- predicates over the variable
>              | Formula `And` Formula    -- conjunction of formulas
>              | Formula `Or` Formula     -- disjunction of formulas
```

Notice that negation is not a kind of formula in the logic. Your task is to write a function,

```
> neg :: Formula -> Formula
```

that takes a formula, `f`, and returns a formula that is true just when `f` is false. The following are examples.

```
neg (Pred odd) = Pred (not . odd)

neg (Pred (\ i -> 0 <= i && i < 10)) = Pred (\ i -> not (0 <= i && i < 10))

neg (Pred (== 3)) = Pred (/= 3)

neg (Pred (/= 3)) = Pred (== 3)

neg ((Pred (== 3)) `Or` (Pred (== 4))) = (Pred (/= 3)) `And` (Pred (/= 4))

neg ((Pred (>= 3)) `And` (Pred (<= 10))) = (Pred (< 3)) `Or` (Pred (> 10))

neg (((Pred (< 2)) `And` (Pred (> -5))) `Or` ((Pred (>= 3)) `And` (Pred (<= 10))))
   = (((Pred (>= 2)) `Or` (Pred (<= -5))) `And` ((Pred (< 3)) `Or` (Pred (> 10))))
```