Spring, 1996 Name: ______________________________

Com S 541 — Programming Languages 1

Test on Functional Programming and Core Language Semantics

Special Directions for this Test

This test has 6 questions and pages numbered 1 through 7.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
You can use anything that is built-in to Haskell, **You may use any other function in Davie's book, or from your notes, but to do so you must write it into your answer.**

1. (10 points) In Haskell, write a function

```
> partialSums :: Num a => [a] -> [a]
```

that takes a list of numeric items and returns a list where the i^{th} element is the sum of the i^{th} item and the previous elements in the list. For example:

```
? partialSums ([]::[Integer])
[]
? partialSums [3,4,5]
[3, 7, 12]
? partialSums [9.2,100.0,10.0,1000.0]
[9.2, 109.2, 119.2, 1119.2]
```

2. (15 points) Consider the following Haskell definitions:

```
> data Tree a = Leaf a | Node (Tree a) a (Tree a)

> sumTree :: (Num a) => (Tree a) -> a
> sumTree (Leaf x) = x
> sumTree (Node left root right) = sumTree left + root + sumTree right

> replaceAll :: (Eq a) => a -> a -> (Tree a) -> (Tree a)
> replaceAll new old (Leaf x) = if x == old then (Leaf new) else (Leaf x)
> replaceAll new old (Node left root right) =
>           Node (replaceAll new old left)
>                (if root == old then new else root)
>                (replaceAll new old right)
```

Write a function,

```
> treeIter :: (a -> b) -> ((b,a,b) -> b) -> (Tree a) -> b
```

such that one can define the above functions as follows

```
sumTree :: (Num a) => (Tree a) -> a
sumTree = treeIter (\ x -> x) (\ (left,root,right) -> left + root + right)
replaceAll :: (Eq a) => a -> a -> (Tree a) -> (Tree a)
replaceAll new old =
     treeIter (Leaf . replace1)
              (\ (left,root,right) -> Node left (replace1 root) right)
    where replace1 x = if x == old then new else x
```

3. Consider the following semantics, modeled in Haskell.

```
> type E = Expression       -- Abstract syntax
> type L = Location
> type N = Numeral
>
> data Expression = Deref L | L `Assign` E
>                   | Num N | E `Plus` E | Not E | E `Equals` E | If E E E
> data Location = Loc Integer
> type Numeral = Integer

> type Store = [Integer]     -- the Store domain
> lookup :: Location -> Store -> Integer
> lookup (Loc j) s | (j <= genericLength s) = s !! (j-1)
> update :: Location -> Integer -> Store -> Store
> update (Loc j) n s | (j <= genericLength s) =
>          (take (j-1) s) ++ [n] ++ (drop j s)

> data ExpressibleValue = InInt Integer | InBool Bool

> meaningN :: Numeral -> Integer    -- semantic functions
> meaningN ((n)) = n
> meaningL :: Location -> Location
> meaningL ((Loc i)) = (Loc i)

> meaningE :: Expression -> Store -> (ExpressibleValue,Store)
> meaningE ((Deref al)) s = (InInt (lookup (meaningL((al))) s), s)
> meaningE ((l `Assign` e)) s =
>   (case meaningE((e)) s of
>      (InInt i, s1) -> (InInt 0, update l i s1))
> meaningE ((Num an)) s = (InInt (meaningN ((an))), s)
> meaningE ((e1 `Plus` e2)) s =
>   (case meaningE((e1)) s of
>       (InInt i1, s1) -> (case meaningE((e2)) s1 of
>                           (InInt i2, s2) -> (InInt (i1 + i2), s2)))
> meaningE ((Not e)) s =
>   (case meaningE((e)) s of
>      (InBool b, s1) -> (InBool (not b), s1))
> meaningE ((e1 `Equals` e2)) s =
>   (case meaningE((e1)) s of
>      (InInt i1, s1) -> (case meaningE((e2)) s1 of
>                           (InInt i2, s2) -> (InBool (i1 == i2), s2))
>      (InBool b1, s1) -> (case meaningE((e2)) s1 of
>                           (InBool b2, s2) -> (InBool (b1 == b2), s2)))
> meaningE ((If e1 e2 e3)) s =
>   (case meaningE((e1)) s of
>      (InBool b, s1) -> if b then meaningE((e2)) s1 else meaningE((e3)) s1)
```

Briefly answer the following questions.

(a) (10 points) What is different about the presentation of the semantics compared to the way we have done semantics in class and the way in which semantics are done in Schmidt's book (chapter 1)?

(b) (10 points) What is different about the above language from the core language of Schmidt's chapter 1?

4. (30 points) Give (a) a grammar that defines the type attributes, and (b) a set of typing rules for the language of problem 3. You should use the standard notation for this, not Haskell notation. Your answer will be judged on whether your notation is sensible, and on the reasonableness of the rules you choose. Check that your rules are sound, although you do not need to give a formal proof of that.

Choose one (1) of the following two problems. If your group studied the paper "The Essence of Functional Programming", you should pick the problem on this page. If your group studied the paper "How to make *ad hoc* polymorphism less *ad hoc*" by Wadler and Blott, you should pick the problem on the next page.

5. Consider the following monadic-like thing.

```
> type EV a = (ExpressibleValue, a)
> unitI i = \ s -> (InInt i, s)
> unitB b = \ s -> (InBool b, s)
> m `bind` k = \ s0 -> let (x, s1) = m s0
>                      in k x s1
> m `bindI` k = \ s0 -> let (InInt i, s1) = m s0
>                       in k i s1
> m `bindB` k = \ s0 -> let (InBool b, s1) = m s0
>                       in k b s1
```

(a) (10 points) Which monad or combination of monads in Wadler's paper does the "monadic-like thing" of this problem most resemble? Briefly explain your answer.

(b) (30 points) Using the above monad (and without using `meaningE`), complete the following so that `meaningE2` has the same behavior as `meaningE` in problem 3.

```
> meaningE2 :: Expression -> Store -> EV Store
> meaningE2 ((Deref al)) = \ s -> unitI (lookup (meaningL((al))) s) s
> meaningE2 ((l `Assign` e)) =
>   meaningE2((e)) `bindI` \ i -> \s ->  unitI 0 (update l i s)
```

If your group studied the paper "How to make *ad hoc* polymorphism less *ad hoc*" by Wadler and Blott, you should do the problem on this page. If your group studied the paper "The Essence of Functional Programming", you should do the problem on the previous page.

6. Consider the language and semantics given in problem 3.

   (a) (10 points) Are there any examples of overloaded operators used in the definition of `meaningE`? If so list them, and note where they are used. If not briefly explain how you can tell.

   (b) (10 points) Briefly explain what a Haskell compiler does when it encounters an overloaded operator.

   (c) (20 points) Can you give a declaration in Haskell so that one can use `meaningN`, `meaningL`, and `meaningE` as instances of an overloaded function `meaning`? If so do that; if not, explain why it can't be done.