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 4 questions and pages numbered 1 through 8.
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. The $3x+1$ problem concerns iterates of the following function:

$$T(n) = \begin{cases} (3n+1)/2, & \text{if } n \equiv 1 \pmod{2}, \\ n/2, & \text{if } n \equiv 0 \pmod{2}. \end{cases} \tag{1}$$

which takes odd numbers n to $(3n+1)/2$ and even numbers n to $n/2$. The iterates of T are defined as follows. Let $T^{(0)}(n) = n$, and for all integers $k > 0$, let $T^{(k)}(n) = T(T^{(k-1)}(n))$. The sequence of iterates $(n, T(n), T^{(2)}(n), \ldots)$ is called the trajectory of n. For example, the trajectory of 7 is:

$$7,\ 11,\ 17,\ 26,\ 13,\ 20,\ 10,\ 5,\ 8,\ 4,\ 2,\ 1,\ 2,\ 1,\ 2,\ 1,\ \ldots$$

The $3x+1$ conjecture states that for each $n > 0$, there is some k such that $T^{(k)}(n) = 1$.

(a) (15 points) In Haskell, define a function

```
> trajectory_to_1 :: Integer -> [Integer]
```

such that `trajectory_to_1(n)` returns the first subsequence of the trajectory of `n` that includes 1, provided that `n` is not a counter-example to the conjecture. The following are examples.

```
? trajectory_to_1(1)
[1]
? trajectory_to_1(7)
[7, 11, 17, 26, 13, 20, 10, 5, 8, 4, 2, 1]
```

(b) (10 points) Define a function

```
> max_value :: Integer -> Integer
```

such that `max_value(n)` is the maximum element of `trajectory_to_1(n)`. The following are examples.

```
? max_value 7
26
? max_value 1819
638468
```

2. Consider the abstract syntax for musical compositions (think of it as a scripting language for a MIDI sound card), modeled in Haskell.

```
> type M = Music         -- Abstract syntax
> type P = Phrase
> type N = Note
> type T = Time
> type D = Dynamic
>
> data Music = Now P | Repeat M
> data Phrase = Play N T | P `Sequence` P | Simultaneous P P
>             | Rest T | Loudness D P
> type Note = Integer
> data Time = Beats Integer
> data Dynamic = Silent | Soft | Medium | Loud
```

The following remarks try to give you some intuition about the semantics of the various syntactic categories. For Music:

- `Now` P plays the phrase P (once),
- `Repeat` M plays M twice,

For Phrase:

- `Play` N T plays the note N for time T (you can think of T as some number of, say, seconds),
- P_1 `` `Sequence` `` P_2 plays first P_1, then P_2,
- `Simultaneous` P_1 P_2 plays P_1 and P_2 at the same time, where P_1 and P_2 must take the same length of time to play,
- `Rest` T plays nothing (is silent) for a length of time specified by T,
- `Loudness` D P plays all of P at dynamic (loudness) level D; P must not already have a dynamic level specified.

A phrase with no dynamic level is played at dynamic level `Medium`. Note that P_1 `` `Sequence` `` P_2 is okay even if P_1 and P_2 have different dynamic levels and take different lengths of time.

The semantics of the Music domain will be modeled with the following Haskell type, where `AMusic` is the type of annotated syntax trees of `Music`.

```
> type ChordList = [(Integer, [(Dynamic, [Integer])])]
> meaningM :: AMusic -> ChordList
```

The idea is that a ChordList such as

```
[(1,[(Silent,[])]), (2,[(Medium,[1,3,5]), (Soft, [7])])]
```

means to play noting for 1 beat (time unit), then to play the notes 1, 3, and 5 simulatneously at dynamic level "medium" for 2 beats, while during the same 2 beats the note 7 is played at dynamic level "soft".

Suppose that

```
> annotateM :: Music -> AMusic
> run = meaningM . annotateM
```

Then the following examples.

```
? run (Now (Play 3 (Beats 2)))
[(2,[(Medium,[3])])]
? run (Repeat (Now ((Rest (Beats 1)) `Sequence` (Play 3 (Beats 2)))))
[(1,[(Silent,[])]), (2,[(Medium,[3])]), (1,[(Silent,[])]), (2,[(Medium,[3])])]
? run (Now (Simultaneous (Play 3 (Beats 2)) (Loudness Loud (Play 5 (Beats 2)))))
[(2,[(Medium,[3]), (Loud,[5])])]
```

Your task is to finish defining the semantics of this language. Be sure that you enforce the following rules in the language's type system.

- Time durations must be strictly positive.
- Simultaneous phrases may not have differing lengths of time.
- At most one dynamic level may be specified for a phrase. For example, `(Loudness Soft (Loudness Loud (Play 3 (Beats 2))))` should be an error, although the following is okay.

  ```
  Simultaneous (Loudness Soft (Play 3 (Beats 2)))    -- ok
               (Loudness Loud (Play 5 (Beats 2)))
  ```

  However, the following is not okay.

  ```
  (Loudness Medium                                       -- error
     (Simultaneous (Loudness Soft (Play 3 (Beats 2)))
                   (Loudness Loud (Play 5 (Beats 2)))))
  ```

  For the same reason, neither is the following.

  ```
  (Loudness Medium                                       -- error
     (Loudness Medium (Play 3 (Beats 2))) `Sequence`
                   (Loudness Medium (Play 5 (Beats 2))))
  ```

You do not have to guard against any other error conditions.

Using either Haskell or the standard notation, give the following three parts of the definition of this language. 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.

(a) (20 points) Give a grammar that defines the type attributes for the langauge.

(b) (30 points) Give a set of typing rules for the language.

(c) (30 points) Give the denotational semantics of the language.

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.

3. (a) (20 points) What examples in semantics have we seen in class that would possibly be improved by using a monad? Briefly describe at least two (2) such examples.

   (b) (20 points) What was the main point of Wadler's paper? Did he clearly demonstrate this point? Why or why not?

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.

4. Consider the language and semantics given in problem 2.

   (a) (20 points) Does the mechanism described in Wadler's paper permit *ad hoc* polymorphism of all kinds? Briefly explain your answer.

   (b) (20 points) Write Haskell code to declare a class that would have both the types `[a]` and `Tree a` as instances, and declare a `map'` method like the `map` function on lists and the `treemap` function below. (That is, both would be possible instances. However, you do not have to declare the instances, just the class.)

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

> treemap f (Leaf x)               = (Leaf (f x))
> treemap f (Node left root right) = (Node (treemap f left)
>                                          (f root)
>                                          (treemap f right))
```