CISC 360 Assignment 2
due Friday, 2021–10–08 at 11:59pm, via onQ

Jana Dunfield

October 1, 2021

Reminder: All work submitted must be your own.

Late policy: Assignments submitted up to 24 hours late (that is, by 11:59 pm the following day) will be accepted **without penalty**. Assignments submitted more than 24 hours late will **not** be accepted, except with an accommodation or a consideration granted according to policy.

Document your code

Some of this assignment is a "fill-in-the-blanks" assignment, so you will not need to document much. However, if you need to write a helper function, you need to write a comment that explains what the function does.

Strive for simplicity

You should try to find a simple solution. You do not have to find the simplest solution to get full marks, but you should not have an excessively complicated solution. Marks may be deducted if your solution is too complicated. If you are worried about whether your solution is too complicated, contact the instructor.

Be careful with library functions

Haskell has a rather large built-in library. This assignment is not about how to find library functions, but about how to use some of the core features of Haskell. You will not receive many marks if you just call a library function that solves the whole problem. The point is to solve the problem yourself.

If you are not sure whether you are calling a library function that solves the whole problem, contact the instructor. Note that if we *suggest* a library function, you may certainly use it.

(The only way I know to avoid this issue is to craft problems that are complicated and arbitrary, such that no library function can possibly solve them. I don't like solving complicated and arbitrary problems, and you probably don't either.)

IMPORTANT: Your file must compile

Your file **must** load (`:load` in GHCi) successfully, or we will subtract **30%** from your mark.

If you are halfway through a problem and run out of time, **comment out the code that is causing :load to fail** by surrounding it with {- ... -}, and write a comment describing what you were trying to do. We can often give (partial) marks for evidence of progress towards a solution, but **we need the file to load and compile**.

1 Add your student ID

The file `a2.hs` will not compile until you add your student ID number by writing it after the `=`:

```
-- Your student ID:
student_id :: Integer
student_id =
```

You do not need to write your name. When we download your submission, onQ includes your name in the filename.

2 'rewrite'

Haskell has a built-in function `ord`, with the type

```
ord :: Char -> Int
```

When applied to a `Char`, the ord function returns the ASCII code corresponding to that `Char`. For example, `ord 'A'` returns 65.

Your task is to implement a function named `rewrite`. Given a `String`, `rewrite` returns a copy of that `String` with all "important" `Chars` duplicated.

However, your employer keeps changing their mind about what is important, so the first argument to the function `rewrite` is actually a function that tells you whether a given character is important.

For example, if the first argument passed to `rewrite` is

```
divisible_by 5
```

then every character whose ASCII code is evenly divisible by 5 is "important" and should be duplicated. (The function `divisible_by` is already defined in `a2.hs`.)

If the first argument passed to `rewrite` is

```
(\x -> (x == ' '))
```

then every space character (and only space characters) will be considered important.

Some examples:

```
rewrite (divisible_by 5) ""                 should evaluate to  ""
rewrite (\x -> x == ' ') "it's a deed"      should evaluate to  "it's  a  deed"
rewrite (divisible_by 5) "CombinatorFest"   should evaluate to  "CombiinnatorFFesst"
```

3 Comparing lists

3a. Fill in the definition of `listCompare`, which takes a pair of lists of `Integers`, and should return a list of `Bools` such that:

- if the kth element of the first list is equal to the kth element of the second list, the kth element of the result should be `True` (because the elements are the same);
- if the kth element of the first list is not equal to the kth element of the second list, the kth element of the result should be `False` (because the elements are different);
- if the first and second lists are of different lengths, the result should be "padded" with `False`, so that the result list is as long as the longer input.

Examples:

```
listCompare ([1, 2, 4], [3, 2, 0])  should be  [False,  True,  False]
                                                ^^^^^    ^^^^   ^^^^^
                                                1 /= 3  2 == 2  4 /= 0

listCompare ([1, 2, 1, 1], [1, 2])  should be  [True,  True,   False,      False]
              ^^^^^^^^^^^^ ^^^^^^               ^^^^   ^^^^    ^^^^^       ^^^^^
              length 4     length 2           1 == 1  2 == 2  2nd list    2nd list
                                                              has no 3rd  has no 4th
                                                              element     element

                                               (resulting list is length 4)
```

3b. In the comment "Q3b", briefly explain why `listCompare` *cannot* be implemented by

```
listCompare :: ([Integer], [Integer]) -> [Bool]
listCompare (xs, ys) = zipWith (==) xs ys
```

3c. The function `listCompare` only works with integer lists. Fill in the definition of `polyCompare`, which takes two arguments:

1. a comparison function `eq`, of type `a -> a -> Bool`, which takes two `a`'s and returns `True` if they should be considered equal, and `False` if they should be considered non-equal;
2. a pair of lists, where each list's elements have type `a`.

4 Planet of Visions

Here is a mysterious `data` declaration:

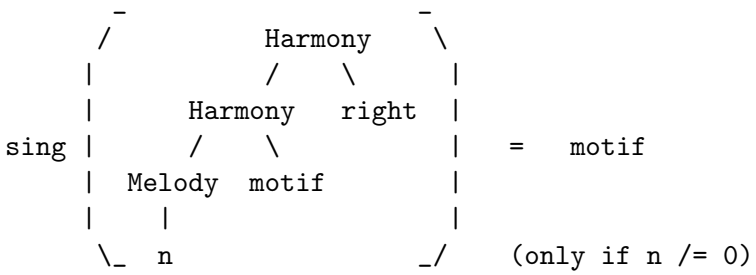
```
data Song = Harmony Song Song
          | Melody Integer
          deriving (Show, Eq)
```

Hint (?): You can think of a `Song` as a tree having branches named `Harmony`, and leaves named `Melody`, where the leaves contain integers.

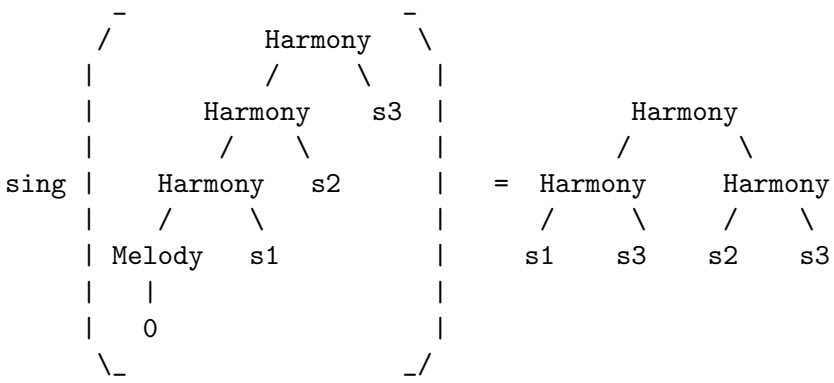
By being sung, a `Song` changes according to the following "rules":

1. If the song is a `Harmony` whose left child is a `Harmony` whose left child is `Melody n` where `n` $\neq 0$, the song becomes the right child of the left child of the root.

   Viewed as trees:



2. I'm not going to try to explain this; it's easier to look at the trees:



If the song does not match either of the above "rules", it remains unchanged. So, for example, `sing (Melody 3)` should return `Melody 3`. Also:

```
sing (Harmony (Harmony (Melody 0) (Melody 1)) (Melody 2))
```

should return `(Harmony (Harmony (Melody 0) (Melody 1)) (Melody 2))`: the shape of the argument matches the first rule, but the condition $0 \neq 0$ is not met.

4a. Implement the function `song` according to the "rules" above.

A "fall-through" clause, matching any song `other`, has already been written for you. Haskell does pattern matching in order, so you need to add your clauses before that one.

4b. Write a function `repeat_sing` that takes a song, and calls `sing` repeatedly until a "fixed point" is reached. That is, if `sing` returns the same song it is given, `repeat_sing` should return that song; otherwise, `repeat_sing` should call itself again with the changed song.

4c (BONUS). You can get full marks on the assignment without doing this question; you might get a total score over 100% by doing this bonus question; but this bonus question is worth no more than 5% of the marks.

This question might not have an answer, so don't attempt it unless you really want to.

Find a song that "diverges": calling `repeat_sing` never returns, because `sing` never returns the same argument.