CISC 360 Assignment 2
due Tuesday, 2023–10–17 at 11:59pm, via onQ

Jana Dunfield

October 3, 2023

Reminder: All work submitted must be your own, or, if you are working with one other student, your teammate's.

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**.

If you choose to work in a group of 2

You **must** use version control (such as GitHub, GitLab, Bitbucket, etc.). This is primarily to help you maintain an equitable distribution of work, because commit logs provide (rough) information about the members' level of contribution.

Your repository **must** be private—otherwise, anyone who has your GitHub (etc.) username can copy your code, which would violate academic integrity. However, upon request from the course staff, you must give us access to your repository. (You do not need to give us access unless we ask.)

We only need *one* submission of the assignment. However, each of you *must* submit a brief statement (.txt preferred; .pdf or .docx are acceptable).

1. Give your names and student ID numbers.
2. Estimate the number of hours you spent on the assignment.
3. Briefly describe your contribution, and your teammate's contribution. (Coding, trying to understand the assignment, testing, etc.)

This is meant to ensure that both group members reflect on their relative contributions.

If you do not submit a statement, you will not receive an assignment mark. This is meant to ensure that each group member is at least involved enough to submit a statement.

Each member must submit a statement.

Add your student ID

Begin by **renaming the file** to `a2-`*studentid*`.hs`. For example, if your student ID number were 87654321, you should rename the file to `a1-87654321.hs`.

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

```
-- Rename this file to include your student ID: a2-studentid.hs
-- Also, add your student ID number after the "=":
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.

If you are working in a group of 2, uncomment the "`second_student_id`" line and add the second student's ID number there.

1 '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 a function that tells you whether a given character is important.

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

```
divisible_by 2
```

then every character whose ASCII code is evenly divisible by 2 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 2) ""                 should evaluate to  ""
rewrite (\x -> x == ' ') "it's a deed"      should evaluate to  "it's  a  deed"
rewrite (divisible_by 2) "CombinatorFest"   should evaluate to  "CombbinnattorrFFestt"
```

2 Comparing lists

2a. Fill in the definition of `listCompare`, which takes two lists of `Int`s, and should return a list of `Bool`s such that:

- if the kth element of the first list is *less than* the kth element of the second list, the kth element of the result should be `True`;
- if the kth element of the first list is *greater than or equal to* the kth element of the second list, the kth element of the result should be `False`;
- 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 [0, 2, 4] [3, 2, 0]  should be  [True,   False,   False ]
                                             ^^^^    ^^^^^    ^^^^^

                                            0 < 3   2 >= 2   4 >= 0

listCompare [5, 4, 3, 2] [2, 9]  should be  [False,  True,   False,      False]
            ^^^^^^^^^^^^ ^^^^^^             ^^^^^   ^^^^    ^^^^^        ^^^^^

            length 4     length 2           5 >= 2  4 < 9   2nd list     2nd list
                                                            has no 3rd   has no 4th
                                                            element      element

                                        (resulting list has length 4)
```


2b. The function `listCompare` only works with integer lists. Fill in the definition of `genCompare`, which takes three arguments.

1. The first argument is a comparison function `cmp`, of type `a -> a -> Bool`, which takes two `a`'s and returns `True` if the first argument should be considered less than the second argument, and `False` otherwise.
2. The second and third arguments are lists, where each list's elements have type `a`.

2c. The following code uses a library function, `zipWith`, that behaves *almost* the same as `listCompare`. In a comment, briefly explain why `almostListCompare` does not fully implement the specification of `listCompare`.

```
almostListCompare = zipWith (<)
```

3 Identity

Here is a mysterious `data` declaration:

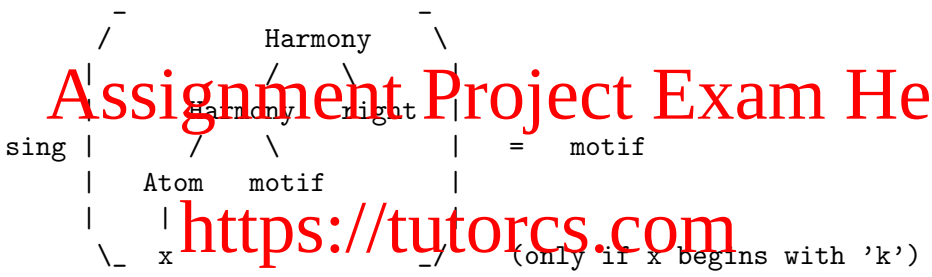
```
data Song = Harmony Song Song
          | Atom String
          deriving (Show, Eq)
```

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

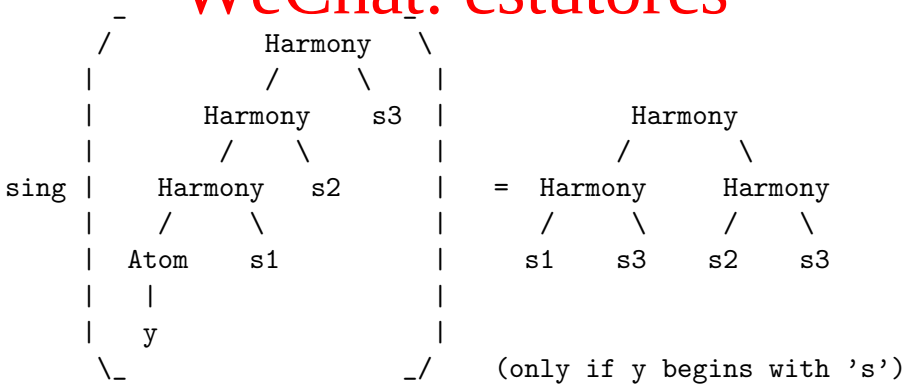
By being sung, a `Song` changes according to the following "rules":

1. If the song is a `Harmony` whose left child is a `Harmony` whose left child is `Atom x` where the first character of `x` is `'k'`, the song becomes the right child of the left child of the root.

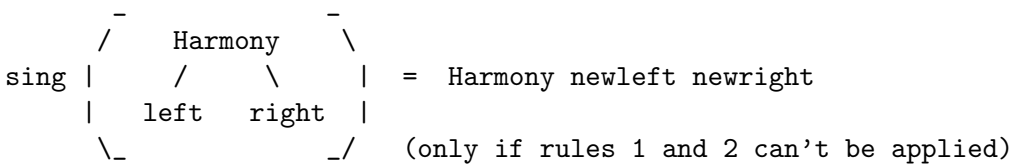
   Viewed as trees:



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



3. If we see a `Harmony` that does *not* match the first two rules, `sing` should recurse on both children:



where `newleft` is the result of calling `sing` on `left`, and `newright` is the result of calling `sing` on `right`.

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

```
sing (Harmony (Harmony (Atom "s") (Atom "y")) (Atom "z"))
```

should return `(Harmony (Harmony (Atom "s") (Atom "y")) (Atom "z"))`:

- The shape of the argument matches the first rule, but `"s"` does not begin with `'k'`.
- The shape of the argument does not match the second rule: it needs three `Harmony` constructors leaning to the left, but there are only two in `(Harmony (Harmony (Atom "s") (Atom "y")) (Atom "z"))`.
- The shape of the argument *does* match the third rule. However, `sing (Harmony (Atom "s") (Atom "y"))` should return `(Harmony (Atom "s") (Atom "y"))`, and `sing (Atom "z")` should return `(Atom "z")`, so `Harmony newleft newright` should be `(Harmony (Harmony (Atom "s") (Atom "y")) (Atom "z"))`.

3a. Implement the function `song` according to the three "rules" above.

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

3b. 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.

3c (BONUS). You can get full marks on the assignment without doing this bonus part; you might get a total assignment grade over 100% by doing this bonus question; but this bonus question is worth no more than 5% of the marks for the assignment.

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

Write a song that is *finite*, yet "diverges": calling `repeat_sing` never returns, because `sing` never returns the same argument.

(We can build an infinite song like this:

```
infinite_song = Harmony infinite_song infinite_song
```

The song you write for this question should be finite, with no recursion or self-reference.)

Hint: Your instructor has (obnoxiously) hidden possible clues to this question throughout the assignment.