CSC 372, Spring 2016
Assignment 3
Due: Friday, February 12 at 23:59:59

Introduction

Some of you may find this to be one of the hardest programming assignments you've ever had. Others may find it to be very easy. I recommend that you start on this assignment as soon as possible, in case you happen to fall into that first group. Remember that as the syllabus states, late work is not accepted and there are no "late days", but I will consider extensions for circumstances beyond your control.

The Haskell slides through 193 show you all the elements of Haskell you need for this assignment, but the two sections that follow in the slides, "Larger Examples" and "Errors" (ending around 230), will broaden your understanding.

I refer to assignments as "`a`*`N`*". This assignment is `a3`. This document is the "`a3` write-up".

My assignment write-ups are a combination of education, specification, and guidance. My goal is to produce write-ups that need no further specification or clarification. That goal is rarely achieved. If you have questions you can either mail to `372s16` or post to Piazza using the appropriate `aN` folder.

This assignment has a long preamble that covers a variety of topics, including the "Tester", a symlink that you need to create, some warnings about old solutions for recycled problems, a very important section on assignment-wide restrictions, and more. You might be inclined to skip all that stuff and get right to the problems, but I recommend you carefully read the preliminary material.

Use the "Tester"!

The syllabus says,

> *For programming problems great emphasis will be placed on the ability to deliver code whose output exactly matches the specification. Failing to achieve that will typically result in large point deductions, sometimes the full value of the problem.*

I'll provide a tester—a testing script—that you can use to confirm that the output of each of your solutions for the programming problems exactly matches the expected output for a number of test cases.

Don't just "eyeball" your output—use the tester! I won't have any sympathy for those who fail tests during grading simply because they didn't want to bother with the tester! However, we'll be happy to help you with using the tester and understanding its output.

The tester is described in the document *Using the Tester*, on the Piazza resources page.

Create a symbolic link to my `a3` directory

For each assignment there will be subdirectory of `/cs/www/classes/cs372/spring16` with assignment-specific files. Those directories will be named `a2`, `a3`, etc.

This write-up assumes your assignment 3 directory on lectura has an `a3` symlink (symbolic link) that references the directory `/cs/www/classes/cs372/spring16/a3`. For example, you'll run the tester with the script `a3/tester` and submit your work with `a3/turnin`.

Some of you may not have worked with symbolic links, so here's a little detail on what you need to do. First, let's imagine that you've made a `372` directory in your home directory, `/home/YOURNETID`, and that in that directory you've made an `a3` directory. (You can do both with "`mkdir -p ~/372/a3`".)

Next, go to your `372/a3` directory and then make a symlink to `spring16/a3`:

```
% cd ~/372/a3
% ln -s /cs/www/classes/cs372/spring16/a3
```

Then check the link with `ls`:

```
% ls -l a3
lrwxrwxrwx 1 whm whm 33 Jan 26 21:12 a3 -> /cw/www/classes/cs372/spring16/a3
```

That lowercase "L" at the start of the line indicates a symbolic link, and `a3 -> ...` shows what the symbolic link `a3` references. If you do "`ls a3`", "`cat a3/delivs`", etc., you'll actually be operating on `/cs/www/classes/cs372/spring16/a3` and files therein.

Let us know if you have trouble with this!

Don't post Piazza questions with pieces of solutions!

Just so there's no doubt about it, IT IS STRICTLY PROHIBITED TO POST PIECES OF SOLUTIONS, whether they work or not.

If you can boil code down to a generic question, like "Why doesn't "`f _ = _`" compile?" or "Is there a function that produces the maximum value of a list of numbers?", that's ok.

If code has any trace of the problem or it conveys or implies anything about a solution, DON'T POST IT. A good rule of thumb is this: If it's apparent what problem some code is related to, that code shouldn't be posted.

If you have even a minor concern that a Piazza post may reveal too much, mail to `372s16` instead!

Recycling of problems from past semesters

When I first started teaching here at The University of Arizona my aim was to come up with a great new set of problems for each assignment each time I taught a class. Maybe I just got old but I eventually found I couldn't keep up with that standard. I noticed that sometimes an old problem was simply better than a new one. I began to reconsider my "all new problems every semester" goal and started thinking about questions like these: Is there an ideal set of problems to teach a set of concepts? How should a teacher's time be balanced between writing new problems and other responsibilities, like working with students and improving lecture material? If creative efforts fail is it better to recycle a good problem or go with a new one that's lesser just so that students won't be tempted to consult an old solution? To make a long story short, I do recycle some problems from past semesters.

If you should come into possession of a solution from a past semester, whether it be from "homework help" site on the net, a friend who took the class, or some other source, let me encourage you to discard it! For one thing, I sometimes put *dunsels* in my solutions. I'd like to distribute solutions with perfectly clean and idiomatic code but I've found that an extra part that sticks out like a sore thumb (and that the lazy don't tend to notice!) helps me eliminate students who find and reuse my solutions. If you notice a silly extra in my code, try removing it and see if things still work. If so, you've perhaps found a dunsel!

While I'm talking about disincentives for cheating I'll also mention that I keep a copy of all former students' submissions for a given problem. We use tools like MOSS to look for suspicious similarities both in the set of this semester's submissions and in that set combined with submissions from any previous semesters when the problem was used.

Remember my cheating policy: one strike and you're out!

Don't forget what you already know about programming!

A key to success in this class is not forgetting everything you've learned about programming just because you're working in a new language. The skills you've learned in other classes for breaking problems down into smaller problems will serve you well in Haskell, too. On the larger problems, particularly `street` and `editstr`, look for small functions to write first that you can then build into larger functions. Test those functions one at a time, as they're written.

If you don't see the cause of a syntax or type error in an expression, whittle the code down until you do. Or, start with something simple and build towards the desired expression until it breaks.

You'll probably have some number of errors due to surprises with precedence. Adding parentheses to match the precedence you're assuming may reveal a problem.

I think of a bug as a divergence between expectation and reality. A key skill for programmers is being able to work backwards to find where that divergence starts. Here's an example of working backwards from an observed divergence to its source: Imagine a program whose expected output is a series of values but instead it produces no output. You then discover that it's producing no output because the count of values to print is zero. You then find that the count is zero because the argument parser is returning a zero for that argument. It then turns out that the argument parser was being passed an incorrect argument.

Once upon a time, I was stumped by a type error in this expression:

```
"#" ++ show lecNum ++ " " ++ [dayOfWeek] ++ " "
: classdays' (lecNum+1) (first+daysToNext) last pairs
```

I reduced it down to the following expression, which works:

```
"#" -- ++ show lecNum ++ " " ++ [dayOfWeek] ++ " "
: []
```

Note the use of that "`--`" (comment to end of line) just after "`#`", **temporarily hiding the rest of the line!**

I then tried advancing that "`--`" past the `show lecNum` call:

```
"#" ++ show lecNum -- ++ " " ++ [dayOfWeek] ++ " "
: []
```

It broke! I then tried a very simple equivalent expression:

```
"a" ++ "b" : []
```

It produced the original type error! I checked the precedence chart. (Slide 84.) I also used `:info` to look at the `++` and "cons" operators:

```
> :info (++)
```

```
(++) :: [a] -> [a] -> [a]
infixr 5 ++

> :info (:)
data [] a = ... | a : [a]
infixr 5 :
```

Since both ++ and **:** are **right** associative operators (`infixR`) with equal precedence (5),

```
"a" ++ "b" : []
```

grouped as

```
"a" ++ ("b" : [])
```

and that was the divergence between expectation and reality! I expected it to group as

```
("a" ++ "b") : []
```

but the reality was the opposite.

If you're puzzled by a syntax or type error, make an effort to chop down the code some before you send it to us. If you get it down as far as I did with `"a" ++ "b" : []`, then you've got a GREAT question! And, something as simple as `"a" ++ "b" : []` can clearly be posted on Piazza for all to see, without any worry about giving away part of a solution (which would cause me to give you a lot of grief!)

When you want us to take a look at a problem, send us the whole file, not just an excerpt where you think the error lies. Our first step is to reproduce the problem that you're seeing. We often can't do that without having all of your code.

ASSIGNMENT-WIDE RESTRICTIONS

There are three assignment-wide restrictions:

1. The only module you may import is `Data.Char`. You can use Prelude functions as long as they are not higher-order functions (see third restriction).

2. You may not use list comprehensions. See slide 135 for an example of one. The general form of a list comprehension is `[ expression | qualifier1, ..., qualifierN ]`. That vertical bar after the initial expression is the most obvious characteristic of a list comprehension.

3. You may not use any *higher-order functions*, which are functions that take other functions as arguments. We'll start talking about them in the section "Higher-order functions", probably around slide 250.

   I'd say that higher-order functions are hard to use by accident but I've been surprised before, so let me say a little bit to help you recognize them. A common example of a higher-order function is `map`:

   ```
   > :t map
   map :: (a -> b) -> [a] -> [b]
   ```

   `map` takes two arguments, the first of which has type `(a -> b)` and that's a function type. It's saying that the first argument is a function that takes one argument. Here's a usage of `map`—see if you can understand what it's doing:

```
> map negate [1..5]
[-1,-2,-3,-4,-5]
```

To be clear, the purpose of this whole section on the third restriction is to help you stay away from higher-order functions on this assignment. We'll be learning them about soon, and you'll see that we can use them instead of writing recursive functions in many cases, but **for now I want you writing recursive functions—think of it as getting good at walking before learning how to run!**

Here is, I believe, a full list of all higher-order functions in the Prelude:

```
all any break concatMap curry dropWhile either filter flip
foldl foldl1 foldr foldr1 interact iterate map mapM mapM_
maybe scanl scanl1 scanr scanr1 span takeWhile uncurry
until zipWith zipWith3
```

Try `:t` on some of them and look for argument types with `(... -> ...)`, `(... -> ... ->)`, etc.

Whenever I put restrictions in place we're happy to take a look at your code before it's due to see if there are any violations. Just mail it to `372s16`.

Strong Recommendation: Specify types for functions, but be careful!

Slides 91-93 talk about specifying types for functions but I'm going to say a little more about it here.

The following function has an error.

```
f ((x,y,z):t) index len
    | (index `mod` length) == 0 = ""
```

Here's the type that Haskell infers for that erroneous function:

```
f :: Integral ([a] -> Int) =>
     [(t1, t2, t3)] -> ([a] -> Int) -> t -> [Char]
```

Whoa! Where'd that `([a] -> Int)` for the second argument, `index`, come from?

If you look close, you'll see that instead of using the parameter `len`, the guard inadvertently uses `length`, a Prelude function. Since the type of `mod` is `Integral a => a -> a -> a`, Haskell proceeds to infer that this function needs a second argument that's an `[a] -> Int` function which is a instance of the `Integral` type class. I can't think of any use for such a thing, but **Haskell proceeds with that inferred type and bases subsequent type inferences on the assumption that the inferred type of `f` is correct**. That can produce far-flung false positives for errors in other functions.

If I simply precede the clause for `f` with a specification for the type of `f` that I intend, I get a perfect error message:

```
% cat extype.hs
f::[(Int,Int,Char)] -> Int -> Int -> String  -- The intended type
f ((x,y,z):t) index len
    | (index `mod` length) == 0 = ""
```

```
% ghci extype.hs
...
extype.hs:3:20:
  Couldn't match expected type `Int' with actual type `[a0] -> Int'
  In the second argument of `mod', namely `length'
  In the first argument of `(==)', namely `(index `mod` length)'
  In the expression: (index `mod` length) == 0
```

The downside of specifying a type for a function is that we might inadvertently make a function's type needlessly specific, perhaps by using an `Int` or `Double` when an instance of the `Num` type class would be better.

My common practice is to see what type Haskell infers for a newly written function. If it looks reasonable, I then "set it in stone" by adding a specification for that type. If an apparent type problem arises later, I might try temporarily commenting that type specification to help get a handle on the problem.

Helper functions are OK, except in `warmup.hs` and `ftypes.hs`

In general, writing helper functions to break a computation into smaller, simpler pieces is a good practice. However, the functions in `warmup.hs`, the first problem, are simple enough that you shouldn't need a helper to write any of them. The last problem on the assignment, `ftypes.hs`, has a number of problem-specific restrictions, including no helper functions.

Aside from `warmup.hs` and `ftypes.hs`, it's fine to use helper functions.

Haskell version issues

As slide 35 mentions, there are some non-trivial version issues with Haskell. We've yet to see any significant incompatibilities between lectura's 7.4.1 and the version I've suggested for your laptops (7.8.3), but your code will be tested on lectura, so how it behaves on lectura is what matters.

Prelude documentation

Prelude documentation for version 7.4.1 is here: **https://hackage.haskell.org/package/base-4.5.0.0/docs/Prelude.html**. On the far right side of each entry is a link to a file with the source code for the function, but you'll find that many functions are written using elements of Haskell we haven't seen. There's a link for the above on the Piazza resources page, too—search for "Prelude".

You can see a list of all Prelude functions by using `:browse Prelude` at the `gchi` prompt.

At long last I present...

The Problems of `a3`!

Problem 1. (7 points) **`warmup.hs`**

The purpose of this problem is to get you warmed up by writing your own version of several simple functions from the Prelude: `last`, `init`, `replicate`, `drop`, `take`, `elem` and `++`.

The code for these functions is easy to find on the web and in books—a couple are shown as examples of recursion in chapter 4 of LYAH. Whether or not you've seen the code I'd like you first to try to write them from scratch. If you have trouble, go ahead and look for the code. Study it but then put it away and try to write the function from scratch. Repeat as needed. Think of these like practicing scales on a musical instrument.

To avoid conflicts with the Prelude functions of the same name, use these names for your versions:

Your function	Prelude function
`lst`	`last`
`initial`	`init`
`repl`	`replicate`
`drp`	`drop`
`tk`	`take`
`has`	`elem`
`concat2`	`(++)`

You should be able to write these functions using only pattern matching, comparisons in guards, list literals, cons (:), subtraction, and recursive calls to the function itself. If you find yourself about to use `if-else`, think about using a guard instead.

`concat2` is a function that's used exactly like you'd use the ++ operator as a function:

```
> concat2 "abc" "xyz"
"abcxyz"

> (++) "abc" "xyz"    -- see slides 77-78
"abcxyz"
```

You may find that `concat2` is the most difficult of the bunch—it's simple but subtle.

Experiment with the Prelude functions to see how they work. Note that `replicate`, `drop`, and `take` use a numeric count. Be sure to see how the Prelude versions behave with zero and negative values for that count. For testing with negative counts, remember that unary negation typically needs to be enclosed in parentheses:

```
> take (-3) "testing"
""
```

```
> drop (-3) "testing"
"testing"
```

You'll find that `last` and `init` throw an exception if called with an empty list. You can handle that with a clause like this one for `lst`:

```
lst [] = error "emptyList"
```

As I hope you'd assume, you can't use the Prelude function that you are recreating! **Beware that when writing these reproductions it's easy to forget and use the Prelude function by mistake**, like this:

```
drp ... = ... drop ...
```

Here's an `egrep` command you can use to quickly check for accidental use of the Prelude functions in your solution: (`a3/check-warmup`)

```
egrep -w "last|init|replicate|drop|take|elem|\+\+" warmup.hs
```

Testing note: If you run the Tester with `a3/tester warmup` you'll see that it runs tests for all of the expected functions, producing a long stream of failures for any functions you haven't completed. You can test functions one at a time by using a `-t` option following `warmup`:

```
% a3/tester warmup -t lst
...

% a3/tester warmup -t drp
...
```

Problem 2. (2 points) **`join.hs`**

Write a function `join separator strings` of type `[Char] -> [[Char]] -> [Char]` that concatenates the strings in `strings` into a single string with `separator` between each string. Examples:

```
> join "." ["a","bc","def"]
"a.bc.def"

> join ", " ["a", "bc"]
"a, bc"

> join "" ["a","bc","def", "g", "h"]
"abcdefgh"

> join "..." ["test"]
"test"

> join "..." []
""

> join ".." ["","","x","",""]
"....x...."

> join "-" (words "just testing this")
"just-testing-this"
```

In Java you might use a counter of some sort to know when to insert the separators but that's not the right approach in Haskell.

Problem 3. (4 points) **`rme.hs`**

Write a function `rme n` of type `Integral a => a -> a` that, **using an arithmetic approach**, removes the even digits from `n`, which is assumed to be greater than zero.

Examples:

```
> rme 3478
37

> rme 100100010010
1111

> rme (17^19)
39735515137153
```

Important: Your solution must be based on arithmetic operations like `*`, `-`, and `div` rather than doing something like using `show` to turn `n` into a string, and then processing that string with list operations.

A obvious difficulty is posed by numbers consisting solely of even digits, like `2468`. For such numbers, `rme` produces zero:

```
> rme 2468
0
```

Problem 4. (4 points) **`splits.hs`**

Consider splitting a list into two non-empty lists and creating a 2-tuple from those lists. For example, the list `[1,2,3,4]` could be split after the first element to produce the tuple `([1],[2,3,4])`. In this problem you are to write a function `splits` of type `[a] -> [([a], [a])]` that produces a list of tuples representing all the possible splits of the given list.

Examples:

```
> :type splits
splits :: [a] -> [([a], [a])]

> splits [1..4]
[([1],[2,3,4]),([1,2],[3,4]),([1,2,3],[4])]

> splits "xyz"
[("x","yz"),("xy","z")]

> splits [True,False]
[([True],[False])]

> length (splits [1..50])
49
```

In order to be split, a list must contain at least two elements. If `splits` is called with a list that has fewer than two elements, raise the exception `shortList`. Example:

```
> splits [1]
*** Exception: shortList
```

In case you missed it, there's an example of raising an exception in problem 1, for `lst`.

Problem 5. (7 points) `cpfx.hs`

Write a function `cpfx`, of type `[[Char]] -> [Char]`, that produces the common prefix, if any, among a list of strings.

If there is no common prefix or the list is empty, return an empty string. If the list has only one string, then that string is the result.

Examples:

```
> cpfx ["abc",  "ab", "abcd"]
"ab"

> cpfx ["abc",  "abcef", "a123"]
"a"

> cpfx ["xabc",  "xabcef", "axbc"]
""

> cpfx ["obscure","obscures","obscured","obscuring"]
"obscur"

> cpfx ["xabc"]
"xabc"

> cpfx []
""
```

Problem 6. (8 points) `paired.hs`

Write a function `paired s` of type `[Char] -> Bool` that returns `True` iff (if and only if) the parentheses in the string `s` are properly paired.

Examples with properly paired parentheses:

```
> paired "()"
True

> paired "(a+b)*(c-d)"
True

> paired "(()()(()))"
True

> paired "((1)(2)((3)))"
True

> paired "((()(()()(((())))((()))))"
True

> paired ""
True
```

Examples with improper pairing:

```
> paired ")"
False

> paired "("
False

> paired "())"
False

> paired "(a+b)*((c-d)"
False

> paired ")("
False
```

Note that you need only pay attention to parentheses:

```
> paired "a+}(/.$#${)[[["
True

> paired"a+}(/.$#$([[)["
False
```

Problem 7. (25 points) `street.hs`

In this problem you are to write a function `street` that prints an ASCII representation of the buildings along a street, as described by a list of `(Int, Int, Char)` tuples, each of which represents a building. The elements of the tuple represent the width, height, and character used to create the building, respectively.

Consider this example:

```
> street [(3,2,'x'), (2,6,'y'), (5,4,'z')]

   yy
   yy
   yyzzzzz
   yyzzzzz
xxxyyzzzzz
xxxyyzzzzz
----------
```

The street has three buildings. As specified by the first tuple, the first building has a width of three, a height of two, and is composed of `"x"`s. The second tuple specifies that a width of two, a height of six, and that `"y"`s be used for the second building. The third building has a width of five, a height of four, and is made of `"z"`s. Note that a blank line appears above the buildings and a line of hyphens (minus signs, not underscores) provides a foundation for the buildings.

This function does something we've only touched on lightly in class: it produces output, which being a side-effect, is a big deal in Haskell. Here's the type of `street`:

```
street::[(Int,Int,Char)] -> IO ()
```

What `street` returns is an *IO action*, which when evaluated produces output as a side effect. `putStr` is a Prelude function of type `String -> IO ()` that outputs a string:

```
> :set +t -- just to show us "it" after putStr
> putStr "hello\nworld\n"
hello
world
it :: ()
```

To avoid tangling with the details of I/O in Haskell on this assignment, make your `street` function look like this:

```
    street buildings = putStr result
        where
           ...some number of expressions and helper functions that
              build up result, a string...
```

The string `result` will need to have whatever characters, blanks, and newlines are required, and that's the challenge of this problem—figuring out how to build up that multiline string!

To help, and hopefully not confuse, here's a trivial version of `street` that's hardwired for two buildings, `[(2,1,'a'), (2,2, 'b')]`:

```
    streetHW _ = putStr result
        where
            result = "\n  bb\naabb\n----\n"
```

Execution:

```
> streetHW "foo"

  bb
aabb
----
```

Like I said, I hope this `streetHW` example doesn't confuse! It's intended to show the connection between (1) binding `result` to a string that represents the buildings, (2) calling `putStr` with `result`, and (3) the output being produced.

Open spaces may be placed between buildings by using buildings of zero height:

```
> street [(3,0,'a'), (3,4,'b') ,(1,0,'c'), (5,7,'d'), (2,0,'e')]

       ddddd
       ddddd
       ddddd
   bbb ddddd
   bbb ddddd
   bbb ddddd
   bbb ddddd
```

```
--------------
```

Note that the foundation (the line of hyphens) extends to the left of the `"b"` building and to the right of the `"d"` building because of the zero-height `"a"` and `"e"` buildings.

You may assume that: (1) at least one building is specified (2) a building width is always greater than zero (3) a building height is always greater than or equal to zero.

Additional examples:

```
> street [(2,5,'x')]

xx
xx
xx
xx
xx
--
> street [(5,0,'x')]

-----
```

Problem 8. (25 points) **`editstr.hs`**

For this problem you are to write a function `editstr ops s` that applies a sequence of operations (`ops`) to a string `s` and returns the resulting string. Here is the type of `editstr`:

```
> :type editstr
editstr :: [([Char], [Char], [Char])] -> [Char] -> [Char]
```

Note that `ops` is a list of tuples. One of the available operations is replacement. Here's a tuple that specifies that every blank is to be replaced with an underscore:

```
("rep", " ", "_")
```

Another operation is translation, specified with `"xlt"`.

```
("xlt", "aeiou", "AEIOU")
```

The above tuple specifies that every occurrence of `"a"` should be translated to "A", every `"e"` to `"E"`, etc. A tuple such as `("xlt", "aeiouAEIOU", "**********")` specifies that all vowels should be translated to asterisks.

Here are <u>two cases I won't test with `xlt`</u>:

- A duplicated "from" character, as in `("xlt", "aa", "12")`
- "from" characters appearing in the "to" string, as in `("xlt", "tab", "bats")`

Here is an example of a call that specifies a sequence of two operations, first a replacement and then a translation:

```
> editstr [("rep", " ", "_"),
           ("xlt", "aeiou", "AEIOU")] "just a test"
"jUst_A_tEst"
```

Note that for formatting purposes the example above and some below are broken across lines.

For `"rep"` (replace), the second element of the tuple is assumed to be a one-character string. The third element, the replacement, is a string of any length. For example, we can remove `"o"`s and triple `"e"`s like this:

```
> editstr [("rep", "o", ""), ("rep", "e", "eee")] "toothsomeness"
"tthsmeeeneeeess"
```

Another example:

```
> editstr [("xlt", "123456789", "xxxxxxxxx"),
           ("rep", "x", "")] "5203-3100-1230"
"0-00-0"
```

There are three simpler operations, too: length (`len`), reverse (`rev`), and replication (`x`):

```
> editstr [("len", "", "")] "testing"
"7"

> editstr [("rev", "", "")] "testing"
"gnitset"

> editstr [("x", "3", "")] "xy"
"xyxyxy"

> editstr [("x", "0", "")] "the"
""
```

Implementation note: The replication operation (`"x"`) requires conversion of a string to an `Int`. That can be done with the `read` function. Here's an example:

```
> let stringToInt s = read s::Int
> stringToInt "327"
327
```

Note that `read` does not do input! What it is "reading" from is its string argument, like `Integer.parseInt()` in Java. Because `read` is overloaded and can return values of many different types we use `::Int` to specifically request an `Int`.

Because we're using three-tuples of strings, `len`, `rev`, and `repl` leave us with one or two unused elements in the tuples.

Let's define some tuple-creating functions and simple value bindings so that we can specify operations with much less punctuation noise. **Put the following lines in your `editstr.hs`:**

```
rep from to = ("rep", from, to)

xlt from to = ("xlt", from, to)

len = ("len", "", "")

rev = ("rev", "", "")
```

```
x n = ("x", show n, "")      -- Note: show converts a value to a string
```

Recall this example above:

```
editstr [("xlt", "123456789", "xxxxxxxxx"),
           ("rep", "x", "")] "5203-3100-1230"
```

Let's redo it using the `rep` and `xlt` bindings from above.

```
>editstr [xlt "123456789" "xxxxxxxxx", rep "x" ""] "5203-3100-1230"
"0-00-0"
```

Note that instead of specifying two literal tuples as operations, we're specifying two function calls that create tuples instead. Notice what `editstr`'s first argument, a list with two expressions, turns into a list with two operation tuples:

```
> [xlt "123456789" "xxxxxxxxx", rep "x" ""]
[("xlt","123456789","xxxxxxxxx"),("rep","x","")]
```

Here's a more complex sequence of operations:

```
> editstr [x 2, len, x 3, rev, xlt "1" "x"] "testing"
"4x4x4x"
```

Operations are done from left to right. The above specifies the following steps:

1. Replicate the string twice, producing `"testingtesting"`.
2. Get the length of the string, producing `"14"`.
3. Replicate the string three times, producing `"141414"`.
4. Reverse the string, producing `"414141"`.
5. Translate `"1"`s into `"x"`s, producing `"4x4x4x"`.

Any number of modifications can be specified.

Again, it case it helps you understand what the `x`, `len`, `rev`, etc. bindings are all about, let's see what that first argument to `editstr` turns into:

```
> [x 2, len, x 3, rev, xlt "1" "x"]
[("x","2",""),("len","",""),("x","3",""),("rev","",""),("xlt","1","
x")]
```

Incidentally, this is a simple example of an *internal DSL* (Domain Specific Language) in Haskell. An expression like `[x 2, len, x 3, rev, xlt "1" "x"]` is using the facilities of Haskell to specify computation in a new language that's specialized for string manipulation. This write-up is already long enough so I won't say anything about DSLs here but you can Google and learn! What we now call Domain Specific Languages were often called "little languages" years ago.

If the list of operations is empty, the original string is returned.

```
> editstr [] "test"
"test"
```

The exception `badSpec` is raised to indicate any of three error conditions:

- An operation is something other than `"rep"`, `"xlt"`, `"rev"`, `"len"`, or `"x"`.
- For `"rep"`, the length of the string being replaced is not one.
- For `"xlt"`, the two strings are not the same length.

Here are examples of each, in turn:

```
> editstr [("foo", "the", "bar")] "test"
"*** Exception: badSpec

> editstr [("rep", "xx", "yy")] "test"
"*** Exception: badSpec

> editstr [("xlt", "abc", "1")] "test"
"*** Exception: badSpec
```

Problem 9. (5 points) **`ftypes.hs`**

Slides 86-89 demonstrate that Haskell infers types based on how values are used. Your task in this problem is to create a sequence of operations on function arguments that cause each of five functions, `fa`, `fb`, `fc`, `fd` and `fe` to have a specific inferred type. The functions will not be run, only loaded, and need not perform any meaningful computation or even terminate.

Here are the types, shown via interaction with `ghci`:

```
% ghci ftypes.hs
...
[1 of 1] Compiling Main             ( ftypes.hs, interpreted )
Ok, modules loaded: Main.
> :browse
fa :: Int -> Bool -> Char -> (Char, Int, Bool)
fb :: [Bool] -> [Int] -> Char -> String
fc :: (Num t1, Num t) => [(t1, t)] -> (t, t1)
fd :: (Integer, [a]) -> Int -> Bool
fe :: [[Char]] -> [[a]]
```

To make this problem challenging we need to have some restrictions:

No apostrophes (`'`), double-quotes (`"`) or decimal digits may appear in `ftypes.hs`, not even in comments. (Yes, this makes character, string, and numeric literals off-limits!)

You may not use `True` or `False`.

You may not use the `::`*`type`* specification, introduced on slide 65.

You may not define any additional functions.

You may not use the `fst` or `snd` functions from the Prelude.

You may not use "as-patterns", the `where` clause, or `let`, `do`, or `case` expressions.

You may not use guards or `if-else`.

Violation of a restriction will result in a score of a zero for that function.

Depending on your code you might end up with a type that's equivalent to a desired type but that has type variables with names that differ from those shown above. For example, `fc` is shown above as this,

```
fc :: (Num t1, Num t) => [(t1, t)] -> (t, t1)
```

but the Tester will consider the following to be correct, too:

```
fc :: (Num a, Num b) => [(a, b)] -> (b, a)
```

If you look close, you'll see that the only difference is that the former uses the type variables `t1` and `t` instead of `a` and `b`, respectively.

Similarly, the following type would also be considered correct:

```
fc :: (Num t, Num a) => [(t, a)] -> (a, t)
```

Problem 10. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with `"Hours:"`. There must be only one "`Hours:`" line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a3/turnin` to submit your work. Each run creates a time-stamped "tar file" in your current directory with a name like `a`*`N.YYYYMMDD.HHMMSS`*`.tz` You can run `a3/turnin` as often as you want. We'll grade your final submission.

Note that each of the `aN.*.tz` files is essentially a backup, too, but perhaps mail to `372s16` if you need to recover a file and aren't familiar with `tar`—it's easy to accidentally overwrite your latest copies with a poorly specified extraction.

`a3/turnin -l` shows your submissions.

To give you an idea about the size of my solutions, here's what I see as of press time:

```
% wc $(grep -v txt < a3/delivs)
   29  115  491 warmup.hs
    3   21   81 join.hs
    8   42  215 rme.hs
    6   34  202 splits.hs
    8   38  171 cpfx.hs
    8   51  278 paired.hs
   25  143  907 street.hs
   49  220 1210 editstr.hs
   12   60  278 ftypes.hs
  148  724 3833 total
```

My code has few comments.

Miscellaneous

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem would correspond to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.) Two minus signs (`--`) is comment to end of line; `{-` and `-}` are used to enclose block comments, like `/*` and `*/` in Java.

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 10 to 15 hours to complete this assignment.

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

If you put ten hours into this assignment and don't seem to be close to completing it, it's probably time to touch base with us. Specifically mention that you've reached ten hours. Give us a chance to speed you up!

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, get a permanent transcript notation stating you cheated, and maybe more. See the syllabus for the details.