CSC 372, Spring 2015
Assignment 2
Due: Thursday, February 12 at 23:00 (not midnight!)

I think this is a tough assignment but I believe you can rise to the challenge. The Haskell slides through 166 show you all the elements of Haskell you need, but the sections that follow in the set of slides, "Larger Examples" and "Errors" (through 208), will broaden your understanding.

This write-up is a combination of assigned problems with lots of examples, hints, tips, advice, encouragement, and additional instruction. I imagine that in terms of a page count, this might be the longest CS assignment write-up you've ever seen. (Let me know if you've ever had a longer one!)

I expect you to read each and every word of this write-up, and the sooner you do that—even if you can't start working on it right away—the better off you'll be. To encourage sooner-rather-later reading I'm offering an on-your-honor 3 points of extra credit for fully reading this write-up before the start of class on Tuesday, February 3. If you do that, submit a plain text file named `readit.txt` via the `a2` D2L dropbox to claim your bonus. The contents of `readit.txt` are immaterial; the presence of the file affirms that you read all the way through this writeup. To be clear, submit `readit.txt` immediately upon completing your read-through.

If you find any bugs, let me know. Please report bugs by mail to keep down clutter on Piazza, especially with possible false positives. Remember that the first person to report a significant bug gets a Bug Bounty point. (I decide what's significant.)

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.

Yesterday I was stumped for a few minutes by a type error in this expression, from slide 184 (slightly simplified, for space):

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

I whittled it down to this, 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] ++ " "
: []
```

and it broke. I then tried a very simple equivalent expression that produced the same type error.

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

I checked the precedence chart on slide 80. 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 me. 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 me to take a look at a problem, send me the whole file, not just an excerpt where you think the error lies. My first step is to reproduce the problem that you're seeing. I often can't do that without having all of your code.

ASSIGNMENT-WIDE RESTRICTIONS

There are two 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 second restriction).

2. 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 230.

   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 "Restriction 2." 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 drive a car!**

   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 I'm happy to take a look at your code before it's due to see if there are any violations. Just mail it to me.

Strong Recommendation: Specify types for functions!

On slide 74 I say, "It is common practice to specify the type of a function along with its definition in a file." I should say "a great practice", because that's what it is, and I should have more examples of that in my examples!

Haskell does a fine job of inferring types when the code is correct but when you make a mistake, having a type specification for a function often makes it much easier to find the problem.

Below is a function with an error. Do you see the 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`, which happens to be 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 moral of the story is simple: Preceding function definitions with a declaration of the intended type often makes it much easier to find a type error.

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 operations 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-then-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"
```

Incidentally, 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 case 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 ...
```

On lectura, Macs, Linux, and Cygwin, here's an `egrep` command you can use to quickly check for accidental use of the Prelude functions in your solution:

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

When writing `drop` you might find an opportunity to use something that's not covered in the slides—an "as-pattern". Here's an example of it, with a function that duplicates the head of a list:

```
duphead all@(x:_) = x:all
```

The portion `all@` causes the name `all` to be bound to the value matched by `(x:_)`. Without using an as-pattern you'd need to do this:

```
duphead (x:xs) = x:x:xs
```

That's no big deal but using an as-pattern saves a cons operation in this case.

The Hoogle (`haskell.org/hoogle`) documentation for functions includes a `Source` link (on the far right) but in some cases the standard implementation uses Haskell features that we haven't yet seen, or won't see at all.

Problem 2. (5 points) **`ftypes.hs`**

We've seen in class that Haskell infers types based on how values are used. For example, assuming we've used `:m Data.Char` at the `ghci` prompt to load the `Data.Char` module, we might do this:

```
> let f x y = ord x == y
> :type f
f :: Char -> Int -> Bool
```

Because `x` is used as the argument for `ord`, whose type is `Char -> Int`, Haskell infers that `x` must be a `Char`. Then, because the result of `ord`, an `Int`, is compared to `y`, Haskell infers that `y` must be an `Int`.

<u>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 type.</u> 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] -> Char -> Bool -> (Bool, Char, [Int])
fb :: (Num t, Num t1) => (t1, t) -> (t, t1)
fc :: (Bool, [Char]) -> Int -> Integer -> [Bool]
fd :: (a, Int) -> (Int, t) -> (t, [a])
fe :: [[[Int]]] -> [[[a]]]
```

<u>To make this problem challenging we need to have some restrictions:</u>

No apostrophes ('), double-quotes (") or decimal digits may appear in `ftypes.hs`. (Yes, this makes character, string, and numeric literals off-limits!)

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

You may not use the `::` ("has type") specification.

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.

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, `fd` is shown above as this,

```
fd :: (a, Int) -> (Int, t) -> (t, [a])
```

but I'd consider the following to be correct, too:

```
fd :: (t1, Int) -> (Int, t) -> (t, [t1])
```

If you look close, you'll see that the only difference is that the latter uses the type variable `t1` instead of `a`.

Similarly, the following two would also be correct:

```
fd :: (t1, Int) -> (Int, t2) -> (t2, [t1])

fd :: (b, Int) -> (Int, a) -> (a, [b])
```

WARNING: Don't fall into the trap of thinking that you must complete this problem, or any following problems, before moving on. The problems generally progress from easier to harder but don't get hung up on a problem. If you stop making progress on a problem, move on or ask me for help.

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

Write a function `tob n` of type `Int -> [Char]` that returns a string of ones and zeroes corresponding to the bit pattern represented by the integer `n`, which is assumed to be greater than zero.

Examples:

```
> tob 5
"101"

> tob 1
"1"

> tob 4
"100"

> tob 127
"1111111"

> tob 1025
"10000000001"
```

You'll find `intToDigit` in the `Data.Char` module handy. Use `import Data.Char` at the top of your `tob.hs` to get it.

Problem 5. (3 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 6. (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 7. (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 8. (23 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 barely talked about 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 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 9. (23 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 modifications, 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"
"tthsmeeeneeess"
```

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 modifications, we're specifying two function calls that create tuples instead.

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. If the modifications list 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
```

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.

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 these:

```
Hours: 10
Hours: 12-15.5
Hours: 15+
```

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 point extra credit) Some description of any time/activity on 372 outside of lectures before starting on the assignment. For some that'll just be, "I didn't do anything but come to lectures." Others might say, "I did all the exercises and all the optional reading." Many will be somewhere in the middle.

(c) (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, "Interesting!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use the D2L Dropbox named `a2` to submit a zip file named `a2.zip` that contains all your work. If you submit more than one `a2.zip`, your final submission will be graded. Here's the full list of deliverables:

```
warmup.hs
ftypes.hs
join.hs
tob.hs
splits.hs
cpfx.hs
paired.hs
street.hs
editstr.hs
```
`observations.txt` (for extra credit)

Note that all characters in the file names are lowercase.

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.

Keep in mind the point value of each problem; don't invest an inordinate amount of time in a problem or become incredibly frustrated before you ask for a hint or help. Remember that the purpose of the assignments is to build understanding of the course material by applying it to solve problems. If you reach the eight-hour mark, regardless of whether you have specific questions, it's probably time to touch base with me. Give me a chance to speed you up! **My goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.**