CSC 372, Spring 2014
Assignment 1
Due: Wednesday, February 12 at 23:00 (not midnight!)

I think this is a tough assignment but I believe you can rise to the challenge. You won't have everything you need for this assignment until we get through Haskell slide 166, which will be on Monday, February 3 but you should go ahead and read through this write-up as soon as you possibly can.

Don't forget 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.

ASSIGNMENT-WIDE RESTRICTIONS

There are two assignment-wide restrictions:

1. The only module you may use is `Data.Char`, which we've used in class on several occasions.
2. You may not use any *higher-order functions*, which are functions that take other functions as arguments. As of January 31 we won't have seen any, but they're coming soon.

Problem 1. (5 points) `warmup.hs`

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

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 GG. 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. Thinks 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`

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.

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, for `lst`:

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

As I'd hope you'd assume, you can't use the Prelude function in your solution! **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" 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 nothing terrible but using an as-pattern saves a cons operation in this case.

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`.

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. 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]]]
```

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

No apostrophes (`'`), double-quotes (`"`) or decimal digits may appear in `ftypes.hs`

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.

You may use the `Data.Char` module (use `import Data.Char` in your `ftypes.hs`) but no other modules may be used.

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])
```

We'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])
```

Problem 3. (3 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. (6 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 source file 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
```

As an example of raising an exception, here's a function that raises an exception if it's called with something other than 1.

```
f 1 = 1
f _ = error "notOne"
```

Problem 6. (8 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. (10 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
```

Examples with failures:

```
> 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. (20 points) **`street.hs`**

In this problem you are to write a function `street` that prints a crude 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`, as inferred by Haskell for my solution:

```
street :: (Num a, Ord a) => [(Int, a, 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 that `streetHW` 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. (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 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 above example 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, reverse, and replication:

```
> 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's it's "reading" from is its string argument. Because `read` 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. (Confession: these operations were added close to my printing deadline; if I'd added them sooner I probably would have switched to lists instead of tuples!)

Let's define some tuple-creating functions and simple value bindings so that we can specify operations with much less punctuation noise:

```
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
```

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 `a1` to submit a zip file named `a1.zip` that contains all your work. If you submit more than one `a1.zip`, we'll grade your final submission. 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.

Have all the deliverables in the uppermost level of the zip. It's ok if your zip includes other files, too.

Miscellaneous

This assignment is based on the material on Haskell slides 1-166.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem on this assignment corresponds 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 us. Give us a chance to speed you up! **Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.**