## CSCI 365 Module 8

Due Wednesday 20 April, 2022

Specification

- To receive **Level 2** credit for this module:

  - Complete exercises 1–6.

- Files you should submit: `JoinList.hs` and `Scrabble.hs`, containing modules of the same name.

The First Word Processor

As everyone knows, Charles Dickens was paid by the word.[1] What most people don't know, however, is the story of you, the trusty programming assistant to the great author.
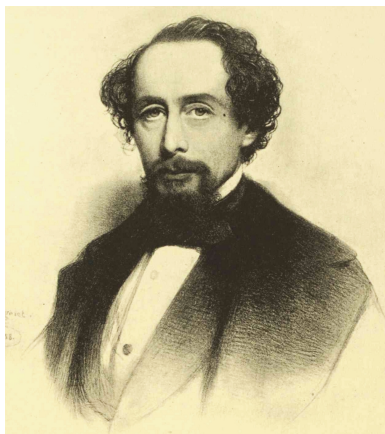
In your capacity as Dickens's assistant, you program and operate the steam-powered Word Processing Engine which was given to him as a thoughtful birthday gift from his friend Charles Babbage.[2] To be helpful, you are developing a primitive word processor the author can use not only to facilitate his craft, but also to ease the accounting.[3] What you have done is build a word processor that keeps track of the total number of words in a document while it is being edited. This is really a great help to Mr. Dickens as the publishers use this score to determine payment, but you are not satisfied with the performance of the system and have decided to improve it.

[2] Unlike the rest of this story, the fact that Dickens and Babbage were friends is 100% true. If you don't believe it, just do a Google search for "Dickens Babbage".

[3] Of course, all of your programming is actually done by assembling steam pipes and valves and shafts and gears into machines which perform the desired computations; but as a mental shortcut you have taken to *thinking* in terms of a higher-order lazy functional programming language and compiling down to steam and gears as necessary. In this assignment we will stick to the purely mental world, but of course you should keep in mind that we could compile everything into Steam-Powered Engines if we wanted to.

Editors and Buffers

You have a working user interface for the word processor implemented in the file `Editor.hs`. The `Editor` module defines functionality for working with documents implementing the `Buffer` type class found in `Buffer.hs`. Take a look at `Buffer.hs` to see the operations that a document representation must support to work with the `Editor` module. The intention of this design is to separate the front-end interface from the back-end representation, with the type class intermediating the two. This allows for the easy swapping of different document representation types without having to change the `Editor` module.

The editor interface is as follows:

- `v` — view the current location in the document

- `n` — move to the next line

- `p` — move to the previous line

- `l` — load a file into the editor

- `e` — edit the current line

- `q` — quit

- `?` — show this list of commands

To move to a specific line, enter the line number you wish to navigate to at the prompt. The display shows you up to two preceding and two following lines in the document surrounding the current line, which is indicated by an asterisk. The prompt itself indicates the current value of the entire document.

The first attempt at a word processor back-end was to use a single `String` to represent the entire document. You can see the `Buffer` instance for `String` in the file `StringBuffer.hs`. Performance isn't great because reporting the document score requires traversing every single character in the document every time the score is shown! Mr. Dickens demonstrates the performance issues with the following (imaginary) editor session:

```
$ runhaskell StringBufEditor.hs
33> n
 0: This buffer is for notes you don't want to save, and for
*1: evaluation of steam valve coefficients.
 2: To load a different file, type the character L followed
 3: by the name of the file.
33> l carol.txt
```

```
31559> 3640
 3638:
 3639: "An intelligent boy!" said Scrooge. "A remarkable boy!
*3640: Do you know whether they've sold the prize Turkey that
 3641: was hanging up there?--Not the little prize Turkey: the
 3642: big one?"
31559> e
Replace line 3640: Do you know whether they've sold the prize Goose that
31559> n
 3639: "An intelligent boy!" said Scrooge. "A remarkable boy!
 3640: Do you know whether they've sold the prize Goose that
*3641: was hanging up there?--Not the little prize Turkey: the
 3642: big one?"
 3643:
31559> e
Replace line 3641: was hanging up there?--Not the little one: the
31558> v
 3639: "An intelligent boy!" said Scrooge. "A remarkable boy!
 3640: Do you know whether they've sold the prize Goose that
*3641: was hanging up there?--Not the little one: the
 3642: big one?"
 3643:
31559> q
```

Sure enough, there is a small delay every time the prompt is shown.

You have chosen to address the issue by implementing a light-weight, tree-like structure, both for holding the data and caching the metadata. This data structure is referred to as a *join-list*. A data type definition for such a data structure might look like this:

```
data JoinListBasic a where
  Empty   :: JoinListBasic a
  Single  :: a -> JoinListBasic a
  Append  :: JoinListBasic a -> JoinListBasic a -> JoinListBasic a
```

The intent of this data structure is to directly represent append operations as data constructors. This has the advantage of making append an 0(1) operation: sticking two JoinLists together simply involves applying the Append data constructor. To make this more explicit, consider the function

```
jlbToList :: JoinListBasic a -> [a]
jlbToList Empty         = []
jlbToList (Single a)    = [a]
jlbToList (Append l1 l2) = jlbToList l1 ++ jlbToList l2
```

If `jl` is a `JoinList`, we can think of it as a representation of the list `jlbToList jl` where some append operations have been "deferred". For example, the join-list shown in Figure 1 corresponds to the list `['y', 'e', 'a', 'h']`.

Such a structure makes sense for text editing applications as it provides a way of breaking the document data into pieces that can be processed individually, rather than having to always traverse the entire document. This structure is also what you will be annotating with the metadata you want to track.

Monoidally Annotated Join-Lists

The `JoinList` definition to use for this assignment is

```
data JoinList m a where
  Empty   :: JoinList m a
  Single  :: m -> a -> JoinList m a
  Append  :: m -> JoinList m a -> JoinList m a -> JoinList m a
  deriving (Eq, Show)
```

This definition has been provided for you in a Haskell module named `JoinList.hs`.

The `m` parameter will be used to track monoidal annotations to the structure. The idea is that the annotation at the root of a `JoinList` will always be equal to the combination of all the annotations on the `Single` nodes (according to whatever notion of "combining" is defined for the monoid in question). `Empty` nodes do not explicitly store an annotation, but we consider them to have an annotation of `mempty` (that is, the identity element for the given monoid).

For example,

```
Append (Product 210)
  (Append (Product 30)
    (Single (Product 5) 'y')
    (Append (Product 6)
      (Single (Product 2) 'e')
      (Single (Product 3) 'a')))
  (Single (Product 7) 'h')
```

is a join-list storing four values: the character `'y'` with annotation 5, the character `'e'` with annotation 2, `'a'` with annotation 3, and `'h'` with annotation 7. (See Figure 1 for a graphical representation of the same structure.) Since the multiplicative monoid is being used, each `Append` node stores the product of all the annotations below it. The point of doing this is that all the subcomputations needed to compute the product of all the annotations in the join-list are cached.

If we now change one of the annotations, say, the annotation on 'y', we need only recompute the annotations on nodes above it in the tree. In particular, in this example we don't need to descend into the subtree containing 'e' and 'a', since we have cached the fact that their product is 6. This means that for balanced join-lists, it takes only $O(\log n)$ time to rebuild the annotations after making an edit.
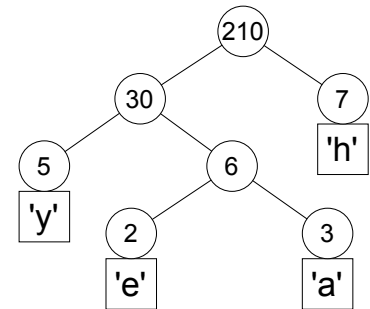

Figure 1: A sample join-list annotated with products

Exercise 1 We first consider how to write some simple operations on these JoinLists. Perhaps the most important operation we will consider is how to append two JoinLists. Previously, we said that the point of JoinLists is to represent append operations as data, but what about the annotations? Write an append function for JoinLists that yields a new JoinList whose monoidal annotation is derived from those of the two arguments.

```
(+++) :: Monoid m => JoinList m a -> JoinList m a -> JoinList m a
```

You may find it helpful to implement a helper function

```
tag :: Monoid m => JoinList m a -> m
```

which gets the annotation at the root of a JoinList.

The first annotation to try out is one for fast indexing into a JoinList. The idea is to cache the *size* (number of data elements) of each subtree. This can then be used at each step to determine if the desired index is in the left or the right branch.

We have provided the Sized module that defines the Size type, which is simply a newtype wrapper around an Int. In order to make Sizes more accessible, we have also defined the Sized type class which provides a method for obtaining a Size from a value.

Use the Sized type class to write the following functions.

Exercise 2 Implement the function

```
indexJ :: (Sized b, Monoid b) =>
          Int -> JoinList b a -> Maybe a
```

indexJ finds the JoinList element at the specified index. If the index is out of bounds, the function returns Nothing. By an *index* in a JoinList we mean the index in the list that it represents. That is, consider a safe list indexing function

```
(!!?) :: [a] -> Int -> Maybe a
[]     !!? _         = Nothing
_      !!? i | i < 0 = Nothing
(x:xs) !!? 0         = Just x
(x:xs) !!? i         = xs !!? (i-1)
```

which returns `Just` the *i*th element in a list (starting at zero) if such an element exists, or `Nothing` otherwise. We also consider an updated function for converting join-lists into lists, just like `jlbToList` but ignoring the monoidal annotations:

```
jlToList :: JoinList m a -> [a]
jlToList Empty          = []
jlToList (Single _ a)   = [a]
jlToList (Append _ l1 l2) = jlToList l1 ++ jlToList l2
```

We can now specify the desired behavior of `indexJ`. For any index `i` and join-list `jl`, it should be the case that

```
(indexJ i jl) == (jlToList jl !!? i)
```

That is, calling `indexJ` on a join-list is the same as first converting the join-list to a list and then indexing into the list. The point, of course, is that `indexJ` can be more efficient ($O(\log n)$ versus $O(n)$, assuming a balanced join-list), because it gets to use the size annotations to throw away whole parts of the tree at once, whereas the list indexing operation has to walk over every element.

Exercise 3 Implement the function

```
dropJ :: (Sized b, Monoid b) =>
         Int -> JoinList b a -> JoinList b a
```

The `dropJ` function drops the first `n` elements from a `JoinList`. This is analogous to the standard `drop` function on lists. Formally, `dropJ` should behave in such a way that

```
jlToList (dropJ n jl) == drop n (jlToList jl).
```

Exercise 4 Finally, implement the function

```
takeJ :: (Sized b, Monoid b) =>
         Int -> JoinList b a -> JoinList b a
```

The `takeJ` function returns the first `n` elements of a `JoinList`, dropping all other elements. Again, this function works similarly to the standard library `take` function; that is, it should be the case that

```
jlToList (takeJ n jl) == take n (jlToList jl).
```

Ensure that your function definitions use the `size` function from the `Sized` type class to make smart decisions about how to descend into the `JoinList` tree.

Exercise 5 Mr. Dickens's publishing company has changed their minds. Instead of paying him by the word, they have decided to pay

Note: you do not have to include (`!!?`) and `jlToList` in your assignment; they are just to help explain how `indexJ` ought to behave. However, you may certainly use them to help test your implementations if you wish.

him according to the scoring metric used by the immensely popular game of *Scrabble*[TM]. You must therefore update your editor implementation to count Scrabble scores rather than counting words.

Hence, the second annotation you decide to implement is one to cache the *Scrabble*[TM] score for every line in a buffer. Create a `Scrabble` module that defines a `Score` type, a `Monoid` instance for `Score`, and the following functions:

```
score :: Char -> Score
scoreString :: String -> Score
```

The `score` function should implement the tile scoring values as shown at http://www.thepixiepit.co.uk/scrabble/rules.html; any characters not mentioned (punctuation, spaces, *etc.*) should be given zero points.

To test that you have everything working, add the line `import Scrabble` to the import section of your `JoinList` module, and write the following function to test out `JoinLists` annotated with scores:

```
scoreLine :: String -> JoinList Score String
```

Example:

```
*JoinList> scoreLine "yay " +++ scoreLine "haskell!"
Append (Score 23)
       (Single (Score 9) "yay ")
       (Single (Score 14) "haskell!")
```

Exercise 6 Finally, combine these two kinds of annotations. A pair of monoids is itself a monoid:

```
instance (Monoid a, Monoid b) => Monoid (a,b) where
  mempty = (mempty, mempty)
  mappend (a1,b1) (a2,b2) = (mappend a1 a2, mappend b1 b2)
```

(This instance is defined in `Data.Monoid`.) This means that join-lists can track more than one type of annotation at once, in parallel, simply by using a pair type.

Since we want to track both the size and score of a buffer, you should provide a `Buffer` instance for the type

$$\text{JoinList (Score, Size) String.}$$

Due to the use of the `Sized` type class, this type will continue to work with your functions such as `indexJ`.

Finally, make a `main` function to run the editor interface using your join-list backend in place of the slow `String` backend (see

`StringBufEditor.hs` for an example of how to do this). You should create an initial buffer of type `JoinList (Score, Size) String` and pass it as an argument to `runEditor editor`. Verify that the editor demonstration described in the section "Editors and Buffers" does not exhibit delays when showing the prompt.