Assignment: 09
Due: Tuesday, March 30th at 11:45 am Eastern Time
Language level: Intermediate Student with lambda
Files to submit: localscreener.rkt streams.rkt transpose.rkt aligned.rkt

- Join the discussion for the answers to frequently asked questions.
- Unless stated otherwise, all policies from the previous assignment carry forward.
- This assignment covers material up to the end of Module 13.
- Correct use of `local` is required, `local` should be used for all helper functions and constants whenever appropriate. If a helper function is not `local` there should be good reason for it not to be such as it being used in more than one function. `local` constants should be used whenever they can simplify code and subsequent `local` helper functions.
- The only built-in functions and special forms you may use are listed below. If a built-in function or special form is not in the following list, you may not use it:


- Remember that basic tests are meant as sanity checks only; by design, passing them should not be taken as any indication that your code is correct, only that it has the right form.
- Unless the question specifically says otherwise, you are always permitted to write helper functions to perform any task. You may use any constants or functions from any part of a question in any other part.
- For any inexact tests, use a tolerance of 0.0001.

1. Investing in "local" companies [10%]. Submit your solutions for all parts of this question in the file `localscreener.rkt`

Recall the Racket function `screener` you had to write for A07, for this question you will be re-writing your solution to that question, however you will modify your solution to use `local` constants and helper functions when appropriate! You should have no non-local helper functions, and you should not define a helper function when it is small and could instead be a constant as it is only used to produce one value. If you did not complete this question in A07 you may instead translate the sample solution - or alternatively take this as a chance to re-attempt the question to improve your learning.
Write a function `screener` that consumes a `CompDict` that represents the companies in a market sector you would like to run your stock screener on. Your function should produce a `(listof Str)` that is the stock ticker (a ticker is just a name given to a company’s stock, these are the keys of the `CompDict`) of each company that meets the screener’s requirements as above. You should produce the strings in the same order they appeared in the `CompDict`. You may assume that each `FinDict` will contain at least the financial information on Revenue, Shares Outstanding, Share Price, Tangible Assets, and Liabilities. For example,

```scheme
(define market
  (list
    (list "CMPNY"
      (list "Revenue" 93216000)
      (list "Shares Outstanding" 356347830)
      (list "Share Price" 0.25)
      (list "Tangible Assets" 2128400)
      (list "Liabilities" 16172000))
    (list "CPRTN"
      (list "Tangible Assets" 194180000000)
      (list "Revenue" 3641000000)
      (list "Share Price" 3140.5)
      (list "Liabilities" 163188000000)
      (list "Shares Outstanding" 502456630))
    (list "SMCRP"
      (list "Tangible Assets" 4382000)
      (list "Share Price" 1.23)
      (list "Liabilities" 2345000)
      (list "Shares Outstanding" 3200400)
      (list "Revenue" 53040))))
```

(check-expect (screener market) (list "CPRTN" "SMCRP"))

Note: You must use `local` definitions for relevant helper functions and constants, you may not use non-local helper functions.

---

2. The other kind of streaming [30%]. Submit your solutions for all parts of this question in the file `streams.rkt`

A stream is a datatype that represents a potentially infinite list. The real draw of streams are that they are *lazily-evaluated* which means the subsequent values in the stream are only computed when requested, not before.

We will make our own implementation of streams defined by functions. We will use the following data definition for a `Stream`:

```scheme
;; A Stream is a (list (X -> X) (X -> X) X)
```

Note that this is a `list` not a `struct`. Additionally you may not change this data definition at all, your `Stream` must contain exactly these three values.

The first `(X -> X)` represents the “current” function, which if applied to the provided `X` value will produce the “current” value. The second `(X -> X)` represents the “update” function, which is used to produce the
"next" stream. The final x is the value used to initialize the stream, which represents its initial value.

For example a stream created with `(make-stream (lambda (x) (+ 2 x)) 0)` produces the infinite stream that looks like:

0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, ...

(1)

We will define the following functions, that will need eachother to be tested properly:

- `make-stream` which creates a new `Stream`, as above.
- `stream-get` which consumes a `Stream` and produces its current value.
- `stream-next` which consumes a `Stream` and produces a `Stream` one further along in the data.
- `stream-reset` which consumes a `Stream` and produces a new `Stream` which is equivalent to initial `make-stream` used to produce the stream.
- `stream-gen` which consumes a `Stream` and a `Nat n` and produces a list of the next n values in the `Stream`.

The functions `make-stream`, `stream-get`, `stream-next`, and `stream-reset` all require eachother to test eachother, so the test following test cases are examples of all three:

(define add1-stream (make-stream (lambda (x) (+ 1 x)) 0))
(define collatz-217 (make-stream (lambda (x)
  (cond [(even? x) (/ x 2)]
        [else (+ 1 (* 3 x))])) 217))
(define string-stream (make-stream (lambda (x) (string-append x "O")) "B"))

(check-expect (stream-get add1-stream) 0)
(check-expect (stream-get (stream-next add1-stream)) 1)
(check-expect (stream-get (stream-next collatz-217)) 217)
(check-expect (stream-get (stream-next (stream-next add1-stream))) 2)
(check-expect (stream-get (stream-reset (stream-next (stream-next add1-stream))))) 0)
(check-expect (stream-get (stream-next (stream-reset (stream-next (stream-next add1-stream))))) 1)
(check-expect (stream-get (stream-next collatz-217)) 652)
(check-expect (stream-get (stream-next string-stream)) "BO")


2.1. make-stream.

Exercise

Write a function `make-stream` that consumes a function of type `(X -> X)` and a `X` and produces a `Stream` which is the stream that starts with the provided `X` and is defined by the provided function.

Note: This function cannot be tested in a vacuum, you must design your produced data so that it enables you to solve the subsequent questions. See above test cases.


2.2. stream-get.

Exercise

Write a function `stream-get` that consumes a `Stream` and produces the current value in the `Stream`.

Note: This function cannot be tested in a vacuum, see above test cases.
2.3. stream-next.

**Exercise**

Write a function stream-next that consumes a Stream and produces the Stream that is one further in the Stream.

**Note:** This function cannot be tested in a vacuum, see above test cases.

2.4. stream-reset.

**Exercise**

Write a function stream-reset that consumes a Stream and produces a new Stream which is equivalent to initial make-stream used to produce the consumed Stream.

**Note:** This function cannot be tested in a vacuum, see above test cases.

2.5. stream-gen.

**Exercise**

Write a function stream-gen that consumes a Stream and a Nat n and produces a list of the next n values in the Stream. Given the definitions above for example,

(check-expect (stream-gen add1-stream 5) (list 0 1 2 3 4))
(check-expect (stream-gen collatz-217 6) (list 217 652 326 163 490 245))
(check-expect (stream-gen string-stream 3) (list "B" "BO" "BOO")

3. Transposition [20%].

Submit your solutions for all parts of this question in the file transpose.rkt

We can use 2D nested lists to represent tables of elements, according to the following data definition:

```
;; A Table is a (listof (listof Any))
;; requires: Every sublist of the table has the same length
```

Here is a sample Table containing four rows and five columns:

```
(define mixed-table
  (list (list 3 (make-posn 4 2) "hello" false 4 -3)
        (list 3.2 #\q #\r "baby" -4.2 -6)
        (list '(1 2 3) 4 true false "quack" -9)
        (list "what" "is" "this" "even?" "Oy!" "minus 12")))
```

**Exercise**

Write the function transpose, which consumes a Table and produces a table with its columns and rows switched. For example,

(check-expect
  (transpose mixed-table)
  (list (list 3 3.2 '(1 2 3) "what")
         (list (make-posn 4 2) #\q 4 "is")
         (list "hello" #\r true "this")
         (list false "baby" false "even?")
         (list 4 -4.2 "quack" "Oy!")
         (list -3 -6 -9 "minus 12")))

4. Aligned Text [40%].

Submit your solutions for all parts of this question in the file aligned.rkt
If you’ve worked with a word processor before you’re likely familiar with text alignment. Some options for text alignment are left-aligned, right-aligned, and centered which is text that all starts at the left margin of the page, right margin of the page, or has as equal distance as possible between the left and right margins of the page.

Write the function `align` that consumes a `Symbol` that is either `left`, `right`, or `center` and a `(listof Str)` and produces a new `(listof Str)` that is aligned based on the `Symbol` consumed. If `left` is consumed add enough spaces to the end of each `String` so that all strings are the same length. If `right` is consumed add enough spaces to the beginning of each `String` so that each string is the same length. If `center` is consumed add spaces to both the beginning and end so that each string is the same length, distribute these spaces as evenly as possible adding the leftover to the beginning in the case of an odd number of spaces. For example,

!(define example-list (list "The quick brown fox"
            "had no friends"
            "he simply was too fast"
            "for them to catch"))

(check-expect (align 'left example-list)
  (list "The quick brown fox 
        "had no friends 
        "he simply was too fast"
        "for them to catch "))

(check-expect (align 'right example-list)
  (list " The quick brown fox"
        " had no friends"
        "he simply was too fast"
        " for them to catch "))

(check-expect (align 'center example-list)
  (list " The quick brown fox 
        " had no friends 
        "he simply was too fast"
        " for them to catch "))

4.1. *Timothy Olyphant Text.* Text can also be laid out in a manner called *justified.* Justified text is text where all lines start at the left margin and end at the right margin regardless of how many characters are in the line. Justified text is produced by extending spaces between words to make each sentence the same length.
Write the function **justify** that consumes a \((\text{listof Str})\) and produces a new \((\text{listof Str})\) that is the result of justifying each \text{String} so that it takes up the same amount of space as the largest string. To achieve this add an equivalent number of spaces to each existing space in the original string, if there is any leftover spaces add them to the first space in the original string in addition to the other added spaces. For example,

\[
\begin{align*}
\text{(check-expect} \\
& \quad (\text{justify (list "The quick brown fox" \\
& \quad \quad "had no friends" \\
& \quad \quad "he simply was too fast" \\
& \quad \quad "for them to catch")))} \\
& \quad (\text{list } \\
& \quad \quad "The quick brown fox" \\
& \quad \quad "had no friends" \\
& \quad \quad "he simply was too fast" \\
& \quad \quad "for them to catch"))
\end{align*}
\]

\[
\begin{align*}
\text{(check-expect} \\
& \quad (\text{justify (list "Really Long Sentence here others will have to adjust to" \\
& \quad \quad "shorter sentence with spaces")))} \\
& \quad (\text{list } \\
& \quad \quad "Really Long Sentence here others will have to adjust to" \\
& \quad \quad "shorter sentence with spaces"))
\end{align*}
\]