Assignment: 07
Due: Tuesday, March 9th at 11:45 am Eastern Time
Language level: Beginning Student With List Abbreviations
Files to submit: warmup.rkt checkerboard.rkt screener.rkt mastermind.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 8.
• 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:
  * + - ... / < <= > >= abs add1 and boolean? ceiling char<=? char<? char=? char>=?
  char>? char? check-expect check-within cond cons cons? define eighth else empty?
  equal? even? exp expt fifth first floor fourth integer? length list list->string
  list? max member? min not number->string number? odd? or quotient remainder rest
  second seventh sixth sqr sqrt string->list string-append string-length string<=?
  zero?

• Remember that we have temporarily removed the following functions from our toolbox:
  filter foldl foldr lambda map range

• 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. Warm-up. Submit your solutions for all parts of this question in the file warmup.rkt

....

1.1. Adapting Insert.

Write the function str-insert that consumes a Str and a sorted (listof Str) and produces a new
sorted list which is the result of inserting the consumed Str into the sorted list in its proper place. For
example,
(check-expect (str-insert "g" (list "a" "b" "x" "y")) (list "a" "b" "g" "x" "y"))
(check-expect (str-insert "Z" (list "a" "g" "v")) (list "Z" "a" "g" "v"))
(check-expect (str-insert "Cat" (list "Aardvark" "Tree" "Xylophone")) (list "Aardvark" "
Cat" "Tree" "Xylophone"))

Write the predicate `dictionary=?` that consumes two `(listof (list Str Nat))` which produces `true` if the two dictionaries both contain the same keys with the same values in any order, and `false` otherwise. For example,

```
(define myDict (list (list "A" 1) (list "B" 2) (list "C" 3)))
(define otherDict (list (list "A" 5) (list "B" 2) (list "C" 3)))
(define equalDict (list (list "B" 2) (list "C" 3) (list "A" 1)))
(define anotherDict (list (list "Hello" 6) (list "Boo" 22)))
(define neqDict (list (list "B" 2) (list "C" 3) (list "X" 6) (list "A" 1)))
```

```
(check-expect (dictionary=? myDict equalDict) true)
(check-expect (dictionary=? myDict otherDict) false)
(check-expect (dictionary=? otherDict anotherDict) false)
(check-expect (dictionary=? neqDict myDict) false)
(check-expect (dictionary=? equalDict neqDict) false)
(check-expect (dictionary=? empty empty) true)
```

1.3. Our own foldr.

Write the function `mixed-foldr` that consumes a `(listof (anyof '+ '* '- '/))` which represents the operations to perform and in which order, a `Num` which represents the base value, and a `(listof Num)` which represents the numbers to fold over. Your function should behave as `foldr` does however instead of folding one operation you do each arithmetic operation in the input symbol list in the order presented. The number of symbols and values may not be the same, your function should apply as many operations as it can in order. For example,

```
(check-expect (mixed-foldr (list '+ '* '-') 0 (list 1 3 5)) 4)
(check-expect (mixed-foldr (list '-' '-' '*') 5 (list 5 4 3 2)) -6)
(check-expect (mixed-foldr (list '-' '/') 2 (list 1 3)) -0.5)
(check-expect (mixed-foldr (list '*' '+' '*') 10 empty) 10)
(check-expect (mixed-foldr empty 12 (list 1 2 3 4 5)) 12)

Note: If you’re not sure how to proceed consider the following expanded example

```
(mixed-foldr (list '-' '-' '*') 5 (list 5 4 3 2))
⇒ (- 5 (- 4 (- 3 (* 2 5))))
⇒ -6
```

1.4. List Comparison.

Write the predicate `list<?` that consumes two equal length `(listof Num)` and produces `true` if the first list is “less than” the second list. In this context “less than” means comparing each of the items in the list pairwise and the first list to have a smaller item is the smaller list. For example,

```
(check-expect (list<? (list 1 1 4) (list 1 2 3)) true)
(check-expect (list<? (list 4 2 3 5) (list 3 2 1 7)) false)
(check-expect (list<? (list 1 5 7 4 3) (list 1 5 7 1 3)) false)
(check-expect (list<? (list 1 1 1) (list 1 1 1)) false)
```

```
2. Checkerboards. Submit your solution to this question in the file `checkerboard.rkt`.

Write the function `checkerboard` that consumes a `Nat n` and produces a `(listof (listof (anyof 'X 'O)))` that represents a $n \times n$ checkerboard pattern. If $n$ is even the top left of the checkerboard should be the symbol 'X, otherwise the top left of the checkerboard should be the symbol 'O. Each 'X in the checkerboard should only be horizontally and vertically adjacent to 'O symbols, and vice-versa. For example,

```
(check-expect (checkerboard 6)
  (list
    (list 'X '0 'X 'O 'X '0)
    (list '0 'X 'O 'X '0 'X)
    (list 'X '0 'X '0 'X '0)
    (list '0 'X '0 'X '0 'X)
    (list 'X '0 'X '0 'X '0)
    (list '0 'X '0 'X '0 'X)))
```

```
(check-expect (checkerboard 3)
  (list (list '0 'X '0)
        (list 'X '0 'X)
        (list '0 'X '0)))
```

3. Stock Screener. Submit your solution to this question in the file `screener.rkt`.

There are many companies that an investor could choose to invest in and the investor likely does not have time to look at every single company. So some investors may choose to use a stock screener which is a program that filters a list of stocks based on some criteria the investor chooses.

Since buying stock in a company is effectively owning a portion of the entirety of that company many investors choose to look at an estimate of the total inherent value of a company by looking at their assets. One metric such investors may like to use is *Net Current Asset Value Per Share (NCAVPS)*. The formula for NCAVPS is simple - it is:

\[
\frac{\text{Tangible Assets} - \text{Liabilities}}{\text{Shares Outstanding}} \tag{1}
\]

which is to say the total current assets the company has minus all liabilities divided by the total number of shares that exist of that company. The reason some investors like to look at this metric is the NCAVPS is effectively how much money would be left to give each shareholder if the company went bankrupt and all assets were liquidated to pay debts.

Some famous investors have suggested in the past one method of only investing in companies whose NCAVPS was lower than the price of an individual share (that means you’re actually purchasing the assets of that company at a discount!). In other words the ratio $\frac{\text{Share Price}}{\text{NCAVPS}}$ should be less than one. Such companies are rare (though not non-existent) in today’s market so you decide to adjust your screener accordingly. You decide to base your screener on the $\frac{\text{Share Price}}{\text{NCAVPS}}$ relative to the size of the company.

For simplicities sake you break companies up into one of three groups: small cap, mid cap, and large cap. These sizes are based on the company’s *market capitalization* which just means how much the stock market values the company at. The formula for a company’s market capitalization is $(\text{Shares Outstanding}) \times (\text{Share Price})$. You decide a company is small cap if its market capitalization is less than $2 \times 10^9$ dollars, mid cap if its market capitalization is in the range of $[2 \times 10^9, 10 \times 10^9]$, and large cap if its market capitalization is greater than $10 \times 10^9$ dollars.
capitalization is larger than $10 \times 10^9$ dollars. You decide that your screener will produce only companies that:

- Have $\frac{\text{Share Price}}{\text{NCAV/PS}} < 5$ if they are small cap
- Have $\frac{\text{Share Price}}{\text{NCAV/PS}} < 20$ if they are mid cap
- Have $\frac{\text{Share Price}}{\text{NCAV/PS}} < 80$ if they are large cap

For this function you should use the following data definitions:

```racket
;; A Financial Dictionary (FinDict) is a (listof (list Str Num))
;; A Company Dictionary (CompDict) is a (listof (list Str FinDict))
```

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,

```racket
(define market (list (list "CMPNY" (list "Revenue" 93216000) (list "Shares Outstanding" 356347830) (list "Share Price" 0.25) (list "Tangible Assets" 21284000) (list "Liabilities" 16172000)))
(list "CPRTN" (list (list "Tangible Assets" 194180000000) (list "Revenue" 36410000000) (list "Share Price" 3140.5) (list "Liabilities" 163188000000) (list "Shares Outstanding" 502456630)))
(list "SMCRP" (list (list "Tangible Assets" 4382000) (list "Share Price" 1.23) (list "Liabilities" 2345000) (list "Shares Outstanding" 3200400) (list "Revenue" 53040))))
```

4. Mastermind. Submit your solution to this question in the file `mastermind.rkt` Mastermind is a code guessing game played between two players the `code master` and the `guesser`. The code master secretly creates a code of four numbers, each between one and six with repetitions allowed. The guesser then
attempts to guess the code in as few guesses as possible. Each time the guesser makes a guess the code master must tell the guesser the total number of pegs the guess is worth. Each guess is awarded pegs in the following fashion:

- The number of black pegs awarded for a guess is equal to the number of numbers the guesser has in the correct position in the code.
- The number of white pegs given for each guess is equal to the number of numbers the guesser has that are within the code but in the wrong position.
- The number of white and black pegs awarded for each individual number between 1 and 6 cannot exceed the number of times that number appears in the code.
- If both a black or white peg could be awarded, the black peg is awarded.

For example, if the code was 1234 and the guess was 4444 the guess would receive a single black peg since the last 4 in the guess is in the correct position. Though there is a 4 in the code the earlier 4’s in the guess do not generate white pegs as the actual 4 in the code is used in generating the black peg. Similarly if the code was 1234 and the guess was 4445 the guess would receive a single white peg, since there is only one 4 in the code so the max number of pegs received for 4’s is 1.


Write the function \texttt{permutations-with-repeats} that consumes a \texttt{Nat} \texttt{n} and a \texttt{Nat} \texttt{k} and produces a \texttt{(listof (listof Nat))} which contains all the possible permutations of size \texttt{k} made out of the numbers from 1 to \texttt{n}. You do not need to worry about the order of your output list, only that the contents contain every permutation. That is all the permutations of two numbers in the range \{1,2\} are \texttt{(list (list 1 1) (list 1 2) (list 2 1) (list 2 2))}, however \texttt{(list (list 2 2) (list 1 1) (list 1 2) (list 2 1))} is also acceptable as a produced value in that case as is any other ordering of those four lists. More examples,

\begin{verbatim}
(check-expect (permutations-with-repeats 3 2) 
  (list 
    (list 1 1) (list 1 2) (list 1 3) (list 2 1) (list 2 2) 
    (list 2 3) (list 3 1) (list 3 2) (list 3 3)))
(check-expect (permutations-with-repeats 2 1) 
  (list (list 1) (list 2)))
(check-expect (permutations-with-repeats 100 0) (list empty))
\end{verbatim}

4.2. Scoring Guesses.

Write the function \texttt{mastermind-peg} that consumes two \texttt{(list Nat Nat Nat Nat)} which represents a Mastermind guess and correct code respectively. Your function should produce as per above a \texttt{(list Nat Nat)} where the first \texttt{Nat} represents the number of black pegs the guess is worth, and the second \texttt{Nat} represents the number of white pegs the guess is worth. For example,

\begin{verbatim}
(check-expect (mastermind-peg (list 1 2 3 4) (list 4 4 4 4)) (list 1 0))
(check-expect (mastermind-peg (list 1 2 3 4) (list 4 3 2 2)) (list 0 3))
(check-expect (mastermind-peg (list 1 2 1 2) (list 5 6 5 6)) (list 0 0))
(check-expect (mastermind-peg (list 3 4 5 1) (list 2 4 1 6)) (list 1 1))
\end{verbatim}