Introduction

Read and follow the course Haskell Style Guide (https://www.cs.rit.edu/~mtf/teaching/20191/psfp/style.html). Especially, look for opportunities to eliminate unnecessary complexity (use anonymous functions, us partial applications, use sectioning, use functionals). Also, look for opportunities to use standard library functions; see especially


This (and future) assignments will make use of some Haskell libraries for testing. This assignment makes use of the Test.HUnit module, which is installed on the CS Department Linux machines, is automatically installed when using the full installer of the Haskell Platform, and can be manually installed when using the core installer of the Haskell Platform by executing `cabal update ; cabal v2-install --lib HUnit`. 
Text Formatting

Download TextFmt.hs.

In many contexts (e.g., word processing systems, web browsers), text is automatically filled and justified. Filling means to group words into lines such that no line exceeds a target width. Justifying means to add extra inter-word spaces to each line (except the last) such that each line meets a target width. For example, the unformatted text

Alice was beginning
to get very tired of sitting by her sister
on the bank, and of having
nothing to do: once or twice she had peeped into the
book her sister was reading, but it had no pictures
or conversations in it, and what is the use of a book, thought Alice
without pictures or conversations?
(from _Alice's Adventures in Wonderland_ by Lewis Carroll)

would be transformed by filling (with a target width of 40) to:

Alice was beginning to get very tired of sitting by her sister
on the bank, and of having nothing to do: once or
twice she had peeped into the book her sister was reading,
but it had no pictures or conversations in it, and what is
the use of a book, thought Alice without pictures or
conversations? (from _Alice's Adventures in Wonderland_ by
Lewis Carroll)

and would be transformed by justifying (with a target width of 60) to:

Alice was beginning to get very tired of sitting by her sister
on the bank, and of having nothing to do: once or
twice she had peeped into the book her sister was reading,
but it had no pictures or conversations in it, and what is
the use of a book, thought Alice without pictures or
conversations? (from _Alice's Adventures in Wonderland_ by Lewis Carroll)

1. (5pts) Define a function

   • fill :: Int -> [String] -> [[[String]]]

   The function fill takes a target width n and a list of words ws (where each word is a string that includes no space characters) and returns a list of lines (where each line is a list of words) such that the length of each line does not exceed the target width (unless there are words in the input that exceed the target width, in which case each such word will appear alone on a line and those lines may exceed the target width). Note that the length of a line is the sum of the lengths of the words in the line plus one less than the number of words in the line (to account for the inter-word space needed to separate the words of the line).

   In addition to the provided fillTests :: Test, there is a provided runFill :: Int -> String -> IO () that can be used to run fill on an input string (which will be broken into a list of words) and visualize the output. For example:

   *Homework02> runFill 60 text2
   The quick brown fox jumped over the lazy dog. Sphinx of
   black quartz, judge my vow. How vexingly quick daft zebras
   jump! The five boxing wizards jump quickly. Pack my box with
   five dozen liquor jugs!

   Study the provided checkFill :: Int -> String -> Bool to understand the requirements for the fill function.
2. (10pts) Define a function

\[ \text{justify} :: \text{Int} \to [\text{String}] \to [\text{String}] \]

The function \( \text{justify} \) takes a target width \( n \) and a list of words \( ws \) (where each word is a string that includes no space characters) and returns a list of lines (where each line is string that may include space characters) such that each line begins and ends with a non-space character and the length of each line (except the last) meets the target width (unless there are words in the input that exceed the target width, in which case each such word will appear alone on a line and those lines may exceed the target width). In addition, each line (except the last) should have approximately the same number of spaces between each word; the maximum number of spaces between the words of a line must be at most one greater than the minimum number of spaces between the words of that line. The last line should have exactly one space between each word.

In addition to the provided \( \text{justifyTests :: Test} \), there is a provided \( \text{runJustify :: Int} \to \text{String} \to \text{IO ()} \) that can be used to run \( \text{justify} \) on an input string (which will be broken into a list of words) and visualize the output. For example:

*Homework02> runJustify 60 text2
The quick brown fox jumped over the lazy dog. Sphinx of black quartz, judge my vow. How vexingly quick daft zebras jump! The five boxing wizards jump quickly. Pack my box with five dozen liquor jugs!

Note that there maybe multiple justifications of a list of words. For example, the following output also satisfies the requirements for the \( \text{justify} \) function.

*Homework02> runJustify 60 text2
The quick brown fox jumped over the lazy dog. Sphinx of black quartz, judge my vow. How vexingly quick daft zebras jump! The five boxing wizards jump quickly. Pack my box with five dozen liquor jugs!

Study the provided \( \text{checkJustify :: Int} \to \text{String} \to \text{Bool} \) to understand the requirements for the \( \text{justify} \) function.
Propositional Logic

Download PropLogic.hs and PropLogicTests.hs; the latter provides a main action that executes tests and displays any failing tests.

Propositional Formula

A propositional formula may be represented by the following Haskell algebraic datatype:

```haskell
type Var = String
data Fmla = FmlaConst Bool | FmlaVar Var
          | FmlaNot Fmla | FmlaOr Fmla Fmla | FmlaAnd Fmla Fmla
  deriving (Eq, Show)
```

The FmlaConst constructor represents a constant logical value (true or false); the FmlaVar constructor represents a propositional variable (identified by a string); the FmlaNot constructor represents the logical negation of a sub-formula; the FmlaOr and FmlaAnd constructors represent the logical conjunction and disjunction of two subformulas.

For example, the propositional formula \( a \land \neg (b \lor \neg (c \lor \top)) \) is represented by the following Haskell expression:

```haskell
FmlaAnd (FmlaVar "a")
  (FmlaNot (FmlaOr (FmlaVar "b")
    (FmlaNot (FmlaOr (FmlaVar "c")
      (FmlaConst True)))))
```

Given a valuation (an assignment of truth values to variables), a propositional formula may be evaluated to a truth value:

```haskell
fmlaEval :: (Var -> Bool) -> Fmla -> Bool
fmlaEval _ (FmlaConst b) = b
fmlaEval v (FmlaVar x) = v x
fmlaEval v (FmlaNot f) = not (fmlaEval v f)
fmlaEval v (FmlaOr f1 f2) = fmlaEval v f1 || fmlaEval v f2
fmlaEval v (FmlaAnd f1 f2) = fmlaEval v f1 && fmlaEval v f2
```

4
Negation Normal Form

In negation normal form (https://en.wikipedia.org/wiki/Negation_normal_form), negation may only be applied to variables and conjunction and disjunction may be applied to an arbitrary number of sub-formulas. The negation normal form may be represented by the following Haskell algebraic datatype:

```haskell
type Lit = (Bool, Var)
data NNF = NNFLit Lit | NNFOr [NNF] | NNFAnd [NNF] deriving (Eq, Show)
```

A literal (Lit) is either a positive or negative propositional variable (i.e., \(a\) is represented as (True, "a") and \(\neg b\) is represented as (False, "b")) and captures the restriction that negation may only be applied to variables.

For example, the propositional formula given above is logically equivalent to the following negation-normal-form Haskell expression:

```haskell
NNFAnd [NNFLit (True,"a"), NNFLit (False,"b"), NNFOr [NNFLit (True,"c"), NNFAnd []]]
```

3. (5pts) Define a function

   ```haskell
   • nnfEval :: (Var -> Bool) -> NNF -> Bool
   ```

   that, given a valuation, evaluates a negation-normal-form formula to a truth value.

   Note: The `nnfEval` must not use the provided `fmlaEval` function.

   Strive for a concise solution.

4. (5pts) Define a function

   ```haskell
   • fmlaToNNF :: Fmla -> NNF
   ```

   that converts a propositional-logic formula to a logically equivalent negation-normal-form formula. To do so, apply De Morgan’s laws and eliminate double negations:

   - \(\neg(p \lor q) \rightarrow \neg p \land \neg q\)
   - \(\neg(p \land q) \rightarrow \neg p \lor \neg q\)
   - \(\neg(\neg p) \rightarrow p\)

   Optionally, further simplify the formula by flattening nested disjunctions and conjunctions.
Conjunctive Normal Form

In conjunctive normal form (https://en.wikipedia.org/wiki/Conjunctive_normal_form), a formula is a conjunction of disjunctions of literals. The conjunctive normal form may be represented by the following Haskell type:

```haskell
type CNF = [[[Lit]]]
```

The “outer” list is the conjunction and the “inner” lists are the disjunctions.

For example, the propositional formula given above is logically equivalent to the following conjunctive-normal-form Haskell expression:

```haskell
([(True,"a"),[(False,"b")]]
```

5. (5pts) Define a function

   ```haskell
   \[ \text{cnfEval} :: (\text{Var} \rightarrow \text{Bool}) \rightarrow \text{CNF} \rightarrow \text{Bool} \]
   ```

   that, given a valuation, evaluates a conjunctive-normal-form formula to a truth value.

   Note: The \text{cnfEval} function must \textbf{not} use the provided \text{fmlaEval} function or the \text{nnfEval} function.

   Strive for a concise solution; the reference solution has a definition of \text{cnfEval} that is only 7 tokens. The simplicity of the definition of evaluation of conjunctive normal form is one of the reasons that conjunctive normal form is used as the input format for industrial strength SAT solvers.

6. (10pts) Define a function

   ```haskell
   \[ \text{nnfToCNF} :: \text{NNF} \rightarrow \text{CNF} \]
   ```

   that converts a negation-normal-form formula to a logically equivalent conjunctive-normal-form formula. To do so, apply distributivity:

   ```haskell
   \[
   \begin{align*}
   & (p \lor (q \land r)) \rightarrow (p \lor q) \land (q \lor r) \\
   & (p \land q) \lor r \rightarrow (p \lor r) \land (q \lor r)
   \end{align*}
   \]
   ```

   which extends to distribution of arbitrary disjunctions over arbitrary conjunctions as:

   ```haskell
   \[
   \begin{align*}
   & (p_1 \land p_2 \land \cdots \land p_n) \lor (q_1 \land q_2 \land \cdots \land q_m) \lor \cdots \lor (r_1 \land r_2 \land \cdots \land r_l) \\
   \rightarrow
   & (p_1 \lor q_1 \lor \cdots \lor r_1) \land (p_1 \lor q_1 \lor \cdots \lor r_2) \land \cdots \land (p_1 \lor q_1 \lor \cdots \lor r_1) \land \cdots \land \\
   & (p_1 \lor q_2 \lor \cdots \lor r_1) \land (p_1 \lor q_2 \lor \cdots \lor r_2) \land \cdots \land (p_1 \lor q_2 \lor \cdots \lor r_1) \land \cdots \land \\
   & \cdots
   \end{align*}
   \]
   ```

   Strive for a concise solution.

   **Warning:** \text{nnfToCNF} is tricky. For partial credit, write a comment describing how you want to solve the problem.
Cards

Download CardGames.hs and CardGamesTests.hs; the latter provides a main action that executes tests, displays any failing tests, and computes a correctness grade.

The first section of CardGames.hs defines types for ranks, suits, and cards (for the standard, 52-card deck) and functions for converting suits and cards to and from strings.

The Rank type is defined, essentially, as enumeration of the 13 ranks:

```haskell
data Rank = Ace | Two | Three | Four | Five | Six | Seven
    | Eight | Nine | Ten | Jack | Queen | King
  deriving (Read, Show, Eq, Enum, Bounded)
```

and the Suit type is defined, essentially, as an enumeration of the four suits:

```haskell
data Suit = Clubs | Diamonds | Hearts | Spades
  deriving (Read, Show, Eq, Enum, Bounded)
```

Note that neither the Rank type nor the Suit type is made an instance of the Ord type class, because there is no universal ordering of ranks (e.g., some games consider aces low, some games consider aces high, and some games consider aces both low and high) or of suits. You may find it useful to define comparison functions of type Rank -> Rank -> Ordering that are appropriate for different games; the fromEnum :: Enum a => a -> Int function of the Enum type class may help to simplify such game specific comparison functions.

The Card type is defined, essentially, as the pairing of a rank and a suit:

```haskell
data Card = Rank :@: Suit
  deriving (Read, Show, Eq)
```

Note that Card is defined with an infix, symbolic constructor, so that the “ace of spades” is expressed naturally as Ace :@: Spades.

7. (4pts + 1pts design) Define an expression

   • deck :: [Card]

   that is a list (order does not matter) of the 52 cards from the standard 52-card deck. Your expression should be as concise as possible.

   For full credit, your expression for deck should continue to be correct if the declarations for Rank and Suit were to be replaced by alternative declarations:

   ```haskell
data Rank = RK | RQ | RJ | R10 | R09 | R08 | R07
    | R06 | R05 | R04 | R03 | R02 | RA
  deriving (Read, Show, Eq, Enum, Bounded)
data Suit = Spades | Hearts | Diamonds | Clubs
  deriving (Read, Show, Eq, Enum, Bounded)
```

Hints:

   • Review Haskell’s list comprehensions (Programming in Haskell (2nd edition), Chapter 5).
   • Make use of the type class instances that are derived for the Rank and Suit declarations.
Poker


8. (20pts correctness + 5pts design) Define a function

```
• pokerHandCompare :: (Card,Card,Card,Card,Card) ->
   (Card,Card,Card,Card,Card) ->
   Maybe Ordering
```

In the application `pokerHandCompare hand1 hand2`, `hand1` and `hand2` are 5-card poker hands. If either of the hands are illegal (has duplicate cards), then `pokerHandCompare hand1 hand2` returns `Nothing`; otherwise, `pokerHandCompare hand1 hand2` returns `Just GT` if `hand1` wins against `hand2`, `Just EQ` if `hand1` and `hand2` tie, and `Just LT` if `hand1` loses against `hand2`. Assume nothing about the order of cards in the tuples representing the hands.

Examples:

```
• pokerHandCompare (Four :@: Hearts, Ace :@: Clubs, Four :@: Clubs,
        Four :@: Diamonds, King :@: Spades)
   (Seven :@: Spades, King :@: Spades, Ace :@: Spades,
        Eight :@: Spades, Nine :@: Spades)
   ~> Just LT

• pokerHandCompare (Four :@: Clubs, King :@: Diamonds, King :@: Spades,
        Four :@: Hearts, King :@: Hearts)
   (Five :@: Diamonds, Eight :@: Spades, Four :@: Hearts,
        Six :@: Diamonds, Seven :@: Spades)
   ~> Just GT

• pokerHandCompare (Nine :@: Clubs, Five :@: Diamonds, Seven :@: Clubs,
        Six :@: Diamonds, Eight :@: Clubs)
   (Eight :@: Spades, Nine :@: Spades, Five :@: Hearts,
        Seven :@: Spades, Six :@: Hearts)
   ~> Just EQ

• pokerHandCompare (Nine :@: Diamonds, Nine :@: Clubs, King :@: Spades,
        Nine :@: Diamonds, Queen :@: Hearts)
   (Six :@: Spades, Ace :@: Hearts, Ace :@: Spades,
        Six :@: Clubs, Six :@: Hearts)
   ~> Nothing
```

Additional poker notes:

- Follow the rules for a “high game” (and ignore the “low rules”).
- In particular, aces are high, but are also low when forming part of a 5-high straight or straight flush.

---

1Note: `pokerHandCompare hand1 hand2` only returns `Nothing` if there are duplicate cards within a hand; `pokerHandCompare hand1 hand2` does not return `Nothing` if there are duplicate cards between the hands. In variations of poker, like Texas hold’em, players may form hands from both hole cards and community cards; thus, it may be necessary to compare poker hands where the same community card appears in both hands.
Hints:

- The function takes two \((\text{Card},\text{Card},\text{Card},\text{Card},\text{Card})\) inputs in order to guarantee that the inputs corresponds to two 5-card hands. However, it may be useful to construct a list of cards in order to analyze hands.

- It may be helpful to define a new type representing categories of poker hands and a function to classify a poker hand as one of the categories. Think about what additional information each category of poker hand needs in order to compare. For example, to compare two two-pair hands, one needs to compare the ranks of the high-pair in each hand; if those are equal, then one needs to compare the ranks of the low-pair in each hand; if those are equal, then one needs to compare the ranks of the kicker card in each hand. In general, each category of poker hand will also need a collection of ranks to compare hands of the same category.

- Consider why it is easier to recognize that the hand
  \[
  \text{Four :@: Hearts, Ace :@: Clubs, Four :@: Clubs, Four :@: Diamonds, King :@: Spades}
  \]
  is a “three of a kind, fours with kickers ace and king” if it is organized as
  \[
  \text{Four :@: Clubs, Four :@: Diamonds, Four :@: Hearts, Ace :@: Clubs, King :@: Spades}
  \]
  and why it is easier to recognize that the hand
  \[
  \text{Four :@: Clubs, King :@: Diamonds, King :@: Spades, Four :@: Hearts, King :@: Hearts}
  \]
  is a “full house, kings over fours” if it is organized as
  \[
  \text{King :@: Diamonds, King :@: Hearts, King :@: Spades, Four :@: Clubs, Four :@: Hearts}
  \]
  and why it is easier to recognize that the hand
  \[
  \text{Five :@: Diamonds, Eight :@: Spades, Four :@: Hearts, Six :@: Diamonds, Seven :@: Spades}
  \]
  is a “eight-high straight” if it is organized as
  \[
  \text{Four :@: Hearts, Five :@: Diamonds, Six :@: Diamonds, Seven :@: Spades, Eight :@: Spades}
  \]
- Identifying flushes should be straightforward.

- Consider how to organize a hand so that identifying straights is straightforward, perhaps with a special case for five-high straights.

- Consider how to organize a hand so that identifying four of a kinds, full houses, three of a kinds, two pairs, one pairs, and high cards are instances of a common representation, which also gives the kicker card(s) for each category. A function from Recitation 01 may be useful.

- It may be helpful to define a \texttt{pokerRankCompare :: Rank -> Rank -> Ordering} function that is appropriate for poker; the \texttt{fromEnum :: Enum a => a -> Int} function of the \texttt{Enum} type class may help to simplify such a comparison function.

- The \texttt{sortBy :: (a -> a -> Ordering) -> [a] -> [a]} function, which sorts a list of elements with a provided comparison function, may be useful.

Note: In addition to correctness, this function will be graded for design, looking for a clear and concise solution making good use of Haskell features. Include comments as necessary to clarify the design.
Cribbage


9. (20pts correctness + 5pts design) Define a function

\[
\text{cribbageScoreTheShow :: Card \to \text{Bool} \to (Card,Card,Card,Card) \to \text{Maybe Integer}}
\]

In the application \text{cribbageScoreTheShow starter crib hand, starter} is the starter (or cut) card, \text{crib} is True if the hand is the crib, False if the hand is not the crib, and \text{hand} is the 4-card cribbage hand. If the combination of the starter card and the hand is illegal (has duplicate cards), then \text{cribbageScoreTheShow starter crib hand} returns \text{Nothing}; otherwise, \text{cribbageScoreTheShow starter crib hand} returns \text{Just scr}, where \text{scr} is the score for the hand during the show phase of a cribbage round. Assume nothing about the order of cards in the tuple representing the hand.

Examples:

- \text{cribbageScoreTheShow (Five :@: Hearts) False} \n  \hspace{1cm} (Six :@: Diamonds, Jack :@: Hearts, Four :@: Hearts, Seven :@: Clubs) \n  \hspace{1cm} \sim \text{Just 9}

- \text{cribbageScoreTheShow (Five :@: Hearts) False} \n  \hspace{1cm} (Five :@: Spades, Four :@: Spades, Two :@: Spades, Six :@: Hearts) \n  \hspace{1cm} \sim \text{Just 12}

- \text{cribbageScoreTheShow (Five :@: Hearts) True} \n  \hspace{1cm} (Ten :@: Spades, Eight :@: Diamonds, King :@: Clubs, Eight :@: Clubs) \n  \hspace{1cm} \sim \text{Just 6}

- \text{cribbageScoreTheShow (Five :@: Hearts) False} \n  \hspace{1cm} (Six :@: Diamonds, Jack :@: Hearts, Five :@: Hearts, Seven :@: Clubs) \n  \hspace{1cm} \sim \text{Nothing}

Additional cribbage notes:

- Aces have a value of 1 for the purposes of counting fifteen-twos. Aces come before 2s (and do not come after Kings) for the purposes of counting runs.

- Jacks, Queens, and Kings all have a value of 10 for the purposes of counting fifteen-twos. Jacks, Queens, and Kings are distinct for the purposes of counting pairs. Jacks come after 10s, Queens come after Jacks, and Kings come after Queens for the purposes of counting runs.

- A run of 5 cards cannot also be counted as two runs of 4 cards and three runs of 3 cards. Similarly, a run of 4 cards cannot also be counted as two runs of 3 cards. A cribbage hand can have either one run of 5 cards, or up to 2 runs of 4 cards (multiple runs due to pairs), or up to 4 runs of 3 cards (multiple runs due to pairs), or no runs.

Hints:

- \text{cribbageScoreTheShow} takes \text{Card} and \text{(Card,Card,Card,Card)} input in order to guarantee that the input corresponds to a starter/cut card and a 4-card hand. However, it may be useful to construct a list of cards in order to count some categories of points.

- Counting \text{runs} is the most difficult category of points to count.

- It may be helpful to define a \text{cribbageRankCompare :: Rank \to \text{Rank} \to \text{Ordering}} function that is appropriate for cribbage; the \text{fromEnum :: Enum a => a \to Int} function of the \text{Enum} type class may help to simplify such a comparison function.

- The \text{sortBy :: (a \to a \to \text{Ordering}) \to [a] \to [a]} function, which sorts a list of elements with a provided comparison function, may be useful.

- The \text{subsequences :: [a] \to [[a]]} function, which returns a list of all subsequences of a list, may be useful.

Note: In addition to correctness, this function will be graded for design, looking for a clear and concise solution making good use of Haskell features. Include comments as necessary to clarify the design.
Requirements and Submission

Your submission must :load into ghci without errors; submissions that have parse errors or type errors will receive no credit. Submissions that violate code style guidelines will lose up to 25%.

Submit PropLogic.hs, TextFmt.hs, and CardGames.hs to the Homework03 Assignment on MyCourses by the due date.