Assignment: 6  
Due: Tuesday, March 6 2018, 9:00pm  
Language level: Beginning Student with List Abbreviations  
Allowed recursion: Pure Structural, unless otherwise specified  
Files to submit: santa-surveillance.rkt, paint-by-numbers.rkt, checksum.rkt, bonus.rkt  
Warmup exercises: HtDP 13.0.3, 13.0.4, 13.0.7, 13.0.8, 14.2.1, 14.2.2, 17.2.1, 17.3.1, 17.6.4, 17.8.3  
Practice exercises: HtDP 12.4.1, 12.4.2, 13.0.5, 13.0.6, 14.2.3, 14.2.4, 14.2.6, 17.1.2, 17.2.2, 17.6.2, 17.8.4, 17.8.8  

- Make sure you read the OFFICIAL A06 post on Piazza for the answers to frequently asked questions.  
- Unless stated otherwise, all policies from Assignment 05 carry forward.  
- Unless otherwise stated, if X is a known type then you may assume that (listof X) is also a known type.  
- You may only use the functions that have been discussed in the lecture slides, unless explicitly allowed or disallowed in the question. In particular, you may NOT use any of the following Racket functions: reverse, make-string, replicate, member, member? or list-ref. You may define your own versions of these functions as long as they are written using pure structural recursion.  
- If your function is a wrapper function, it will require all design recipe components, but the helper function which it calls will require only the purpose, contract, and definition: no examples or tests are needed for the helper function.  
- This is a long assignment. If you start too late you will have problems finishing in time.  
- It will be very helpful to design examples and tests before writing your function implementations, especially for trees.
1. Perform the assignment 6 questions using the online evaluation “Stepping Problems” tool linked to the course web page and available at


The instructions are the same as those in assignment 3; check there for more information if necessary.

2. Santa Claus runs a worldwide surveillance network to spy on children. He knows when they are sleeping, he knows when they are awake, and he tallies up the naughty and nice actions in their lives to determine whether they deserve presents. Each naughty or nice action taken by every child is recorded in its own Action:

;; An Action is a (list Str Int Str)

The first element is a child’s name (which we will assume is unique, even though this is totally unrealistic). The second element is a rating. Positive ratings mean the action is nice and negative ratings mean the action is naughty. Every rating is either positive or negative; Action items with a zero rating are not recorded. The third element is a textual description of the action. For example:

(list "Jean W" 3 "Prepared TA training exercises")

means that Jean prepared TA training exercises, which is worth a nice score of 3. Note that all strings are case-sensitive, so "Jean W" and "jean w" represent two different children.

Children also submit to Santa lists of Wishes:

;; A Wish is a (list Nat Str)
;; requires: Nat > 0

The first element is a niceness score required to receive the gift. The second element is a description of the gift the child wishes to receive. For example:

(list 32 "Hello Kitty Umbrella")

indicates that a gift of a Hello Kitty Umbrella requires a niceness score of 32.

Santa maintains the list of children’s wishes in an association list:

;; A Wishlist is either
;; * empty
;; * (cons (list Str (listof Wish)) Wishlist)

where the keys are child names. The Wishlist is not ordered by child name, but each (listof Wish) is ordered in non-increasing order by niceness score. For example, here is a very small Wishlist:
\begin{verbatim}
(define small-wishlist
  (list (list "Vincent H" (list (list 52 "Commodore 64")
                              (list 17 "Zhu Zhu Pet")
                              (list 13 "Colourful Pencils")))
  (list "Paul N" (list (list 57 "Silver Scooter")
                        (list 43 "Walkman"))
  (list "Jean W" (list (list 37 "Telescope")
                        (list 11 "Slide Rule")))
  (list "Jimmy P" (list (list 54 "Chemistry Set")
                        (list 32 "Hello Kitty Umbrella")
                        (list 32 "Red Wagon")
                        (list 15 "Tamagochi")))))

A child’s niceness score is computed by adding all the niceness points for all Actions for that child. For example, consider the following list of Actions:

\begin{verbatim}
(list (list "Paul N" -4 "Procrastinated on CS 135 assignment")
  (list "Jimmy P" 3 "Helped Vincent troubleshoot test cases")
  (list "Jimmy P" -1 "Stayed up too late on a schoolnight")
  (list "Jimmy P" 3 "Cleaned his room without being nagged")
  (list "Paul N" -7 "Was disobedient at the grocery store"))
\end{verbatim}

Paul N would have a niceness score of -11, and Jimmy P would have a niceness score of 5.

Each year Santa sets a naughtiness threshold. This year the threshold is -20.

Place your solutions in `santa-surveillance.rkt`.

(a) Write a function \texttt{gift-received} which consumes a child’s name, that child’s niceness score, and a \texttt{Wishlist}. The function produces either a symbol or a string according to the following rules:

- If a child has a high enough niceness score to qualify for a present in the \texttt{Wishlist}, the child receives the best present on their list of wishes they qualify for, where “best” means “highest niceness score requirement”. If there are two or more presents with the same niceness score requirement, the child receives the present latest in the listing.
- If the child has a non-negative niceness score but does not qualify for any of their \texttt{Wish} items, they receive an orange, indicated by \texttt{‘orange}. This is also the case if a child has no \texttt{Wish} items or is not in the wishlist, but has a non-negative niceness score.
- If a child has been naughty and has a negative niceness score, but the niceness score is greater than or equal to Santa’s naughtiness threshold, they receive a lump of coal, indicated by \texttt{‘coal}.
- If a child has been very naughty and has a negative niceness score below Santa’s naughtiness threshold, the child receives a visit from the Krampus, indicated by \texttt{‘krampus}.
\end{verbatim}
Here are some examples:

(check-expect (gift-received "Vincent H" 15 small-wishlist)
"Colourful Pencils")
(check-expect (gift-received "Zainab" 43 small-wishlist)
'orange)
(check-expect (gift-received "Paul N" -37 small-wishlist)
'krampus)
(check-expect (gift-received "Jimmy P" 32 small-wishlist)
"Red Wagon")
(check-expect (gift-received "Jean W" 9 small-wishlist)
'orange)
(check-expect (gift-received "Sana" 0 small-wishlist)
'orange)

(b) Write a function add-wishes which consumes a child’s name, a list of Wish items (sorted in non-increasing order by niceness score requirement), and a Wishlist. If the child does not exist in the Wishlist, a new Wishlist is produced consisting of the child and their wishes added to the end of the Wishlist. If the child does exist, then the resulting Wishlist consists of the child’s new Wishes merged into their existing list of Wishes. If Wishes are added with the same niceness scores as existing wishes, the new Wishes take precedence and occur first.

For example:

(check-expect (add-wishes "Josh J" small-wishlist)
'((31 "Furby") (21 "Playstation 2")))

(list (list "Vincent H" (list (list 52 "Commodore 64")
 (list 17 "Zhu Zhu Pet")
 (list 13 "Colourful Pencils")))
 (list "Paul N" (list (list 57 "Silver Scooter")
 (list 43 "Walkman")))
 (list "Jean W" (list (list 37 "Telescope")
 (list 11 "Slide Rule")))
 (list "Jimmy P" (list (list 54 "Chemistry Set")
 (list 32 "Hello Kitty Umbrella")
 (list 32 "Red Wagon")
 (list 15 "Tamagochi")))
 (list "Josh J" (list (list 31 "Furby")
 (list 21 "Playstation 2"))))

(check-expect (add-wishes "Jimmy P" small-wishlist)
'((200 "World Peace") (54 "ColecoVision")
 (20 "MATH 135 Textbook"))

(list (list "Vincent H" (list (list 52 "Commodore 64"))))
(list 17 "Zhu Zhu Pet")
(list 13 "Colourful Pencils")
(list "Paul N" (list (list 57 "Silver Scooter")
(list 43 "Walkman"))
(list "Jean W" (list (list 37 "Telescope")
(list 11 "Slide Rule"))
(list "Jimmy P" (list (list 200 "World Peace")
(list 54 "ColecoVision")
(list 54 "Chemistry Set")
(list 32 "Hello Kitty Umbrella")
(list 32 "Red Wagon")
(list 20 "MATH 135 Textbook")
(list 15 "Tamagochi")))

(c) Santa is finding the number of Actions to be overwhelming. He would like to store them in a Binary Search Tree indexed by child name.

The Binary Search Tree is made of ActionNodes, defined as follows:

(define-struct actionnode (name score actions left right))
;; An ActionNode is a (make-actionnode Str Int (listof Action)
;; ActionSearchTree
;; ActionSearchTree)
;; requires:
;; name > every name in left ActionNode
;; name < every name in right ActionNode

The score is the niceness score of the child, calculated based upon the Action events for that child in the tree. Those Action events are stored in the actions field, as an unordered list. New Action events are added to the front of the actions list.

An ActionSearchTree (AcST) is defined as follows:

;; An ActionSearchTree (AcST) is one of:
;; * empty
;; * an ActionNode

Write a function add-action which consumes an Action and an ActionSearchTree. It produces an AcST consisting of the original tree with the new Action added, possibly creating an ActionNode or updating the ActionNode’s score.

For example:

(check-expect
 (add-action
  (list "Paul N" -7 "Was disobedient at the grocery store")
  empty)
 (make-actionnode
  "Paul N"
  -7
  (list (list "Paul N" -7 "Was disobedient at the grocery store")
       empty
       empty)))
(check-expect
  (add-action
    (list "Josh J" 2 "Proofread CS 135 tutorial")
    (make-actionnode
     "Paul N"
     -7
     (list (list "Paul N" -7 "Was disobedient at the grocery store"))
    (make-actionnode
     "Josh J"
     3
     (list (list "Josh J" 3 "Had good meeting with graduate supervisor")
        empty empty)
    empty)
  empty)

(make-actionnode
  "Paul N"
  -7
  (list (list "Paul N" -7 "Was disobedient at the grocery store"))
(make-actionnode
  "Josh J"
  5
  (list (list "Josh J" 2 "Proofread CS 135 tutorial")
    (list "Josh J" 3 "Had good meeting with graduate supervisor")
  empty empty)
empty)

Testing Hint: Make an additional function which adds a list of Actions to an AcST so that you can build up trees easily. Then make constants for built-up trees, and finally make one function call to add-action on your constants for each test.

(d) Santa now needs to generate a list of children and presents. Write a function gift-list which consumes an ActionSearchTree and a Wishlist, and produces a list of pairs. The first element of each pair is the child’s name, and the second is the gift the child receives (which might be a string representing the received gift, or one of the symbols 'orange, 'coal, or 'krampus). The result should be in increasing order of child name, with all names in the ActionSearchTree represented. (If a name is in the Wishlist but not in the ActionSearchTree it is ignored.)

Here is an example:

(check-expect
  (gift-list
    (make-actionnode
     "Paul N"
     -7
     (list (list "Paul N" -7 "Was disobedient at the grocery store"))
    (make-actionnode
     "Josh J"
     5
     (list (list "Josh J" 2 "Proofread CS 135 tutorial")
       (list "Josh J" 3 "Had good meeting with graduate supervisor")
     empty empty)
  empty)
(list (list "Josh J" 2 "Proofread CS 135 tutorial")
  (list "Josh J" 3
    "Had good meeting with graduate supervisor")))
empty)
(list (list "Vincent H" (list (list 52 "Commodore 64")
  (list 17 "Zhu Zhu Pet")
  (list 13 "Colourful Pencils")))
  (list "Paul N" (list (list 57 "Silver Scooter")
    (list 43 "Walkman")))
  (list "Josh J" (list (list 31 "Playstation 2")
    (list 3 "Furby Sticker")))
  (list (list "Josh J" "Furby Sticker") (list "Paul N" 'coal)))

You may use the built-in function append in your solution. You may also use accumulative recursion.

**Hint:** How can you recursively visit every node in an AcST? How can you do so in increasing order?

3. A paint by numbers puzzle (known as a “nonogram” in mathNEWS) is a puzzle played on a rectangular grid. Each cell of the grid can be coloured black or white. The rows and columns of the grid are labelled with lists of numbers that indicate the sequence of black cells in that row/column.

For example, a row labelled with a sequence of numbers “2 3 2” indicates that this row contains zero or more white squares, followed by 2 adjacent black squares, followed by one or more white squares, followed by exactly 3 black squares, followed by one or more white squares, followed by exactly two black squares, followed by zero or more white squares.

The goal of the puzzle is to fill in black squares so that all of the row and column labels are satisfied.

For example, here is a puzzle, unsolved and solved 1:

![Unsolved and solved paint by numbers puzzle](https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/pattern.html)

---

1These screenshots are from: [https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/pattern.html](https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/pattern.html)
There are many websites on the internet that let you solve paint by numbers puzzles online. One good one is http://webpbn.com.

Surprise! In this question you will not be writing a program to solve paint by numbers puzzles. Instead, this question will solve a different task: given a grid labelled with black and white squares, you will generate the row and column labels.

We can represent a rectangular grid as a nested list, where white squares are represented by the symbol '-' and black squares by 'B:

;; A Grid is a (listof (listof (anyof 'B '-)))
;; requires: all sublists have the same length

For example, the following Grid has 5 rows and 12 columns.

(define fish-grid
  '((B B - - B B B B - - - B)
   (- B B B - - - - B - - -)
   (- B B B - - - - B B - -)
   (- B - - B B B B B - - -)
   (B B - - - - B - - - - -))

Place your solutions to this problem in paint-by-numbers.rkt.

(a) Write a function column which consumes a natural number col and a Grid, and produces the colth column of the Grid, indexed from 0. If the colth column does not exist, the function produces empty. For example:

(check-expect (column 2 fish-grid) '(- B B - - ))

(check-expect (column 32 fish-grid) empty)

(b) Write a function cells->tallies which consumes a (listof (anyof 'B '-)) and produces a list of of tallies of adjacent black cells, where each tally is represented by a Nat.

(check-expect (cells->tallies (first fish-grid)) (list 2 4 1))

You may use accumulative recursion for this question. You may also use the reverse function.

Hint: Accumulate a list of natural numbers for the tallies.

Hint: for each recursive case, consider the first symbol. Adjacent to that symbol will be either another symbol, or the empty list. What should be done for each case?

(c) Write a function puzzle-labels which consumes a Grid, and produces labels for the puzzle. Each label is represented by a list of natural numbers, corresponding to the adjacent black squares in that row or column. The function produces a list of length 2,
where each element is itself a list of labels. The first element is a list of row labels (from top to bottom in the puzzle) and the second is a list of column labels (from left to right). For example:

```
(check-expect
  (puzzle-labels fish-grid)
  (list (list (list 2 4 1)
              (list 3 1)
              (list 1 5)
              (list 2 1))
       (list (list 1 1)
              (list 5)
              (list 2)
              (list 2)
              (list 1 1)
              (list 1 1)
              (list 1 2)
              (list 1 1)
              (list 3)
              (list 1)
              empty
              (list 1)))
```

4. *Universal Product Codes* (UPCs) are widely used to identify products for sale. Normally, a UPC consists of 12 digits, but we will extend this concept to consider codes of any length greater than or equal to two. In a UPC of length $n$, the $n$th digit is a *checksum*, which is computed based on the values of the other digits. The checksum helps identify common transcription errors.

The checksum digit is computed according to the following algorithm:

- Take the sum of all the odd positions of the UPC (so the first digit, the third, and so on), not including the checksum digit. Multiply this result by 3.
- Take the sum of all the even positions of the UPC, not including the checksum digit.
- Add the two sums together, and take the result modulo 10 (that is, the remainder when divided by 10).
- If the result is 0, the checksum digit is 0. Otherwise, if the result is $x$, the checksum digit is $10 - x$.

For example, a UPC label where the first 11 digits (not including the checksum digit) are 66788809152 should have a checksum digit of 2, because $3 \cdot (6 + 7 + 8 + 0 + 1 + 2) + (6 + 8 + 8 + 9 + 5) = 3 \cdot 24 + 36 = 72 + 36 = 108$. Taking the modulo base 10 gives 8, and $10 - 8 = 2$.

---

2 Technically, this is the UPC-A standard. There are others.
We will represent a UPC as a non-empty \((\text{listof Nat})\), where each element of the list is between 0 and 9. So the above UPC (including the checksum) would be represented as \((\text{list } 6 6 7 8 8 8 0 9 1 5 2 2)\).

Place your answers to this question in \texttt{checksum.rkt}.

(a) Write a function \texttt{valid-checksum-acc?} which consumes the representation of a UPC (including the checksum digit) and produces true if and only if the checksum digit is valid for this UPC. You should use accumulative recursion in your solution.

(b) Write a function \texttt{valid-checksum-pure?} which is the same as \texttt{valid-checksum-acc?} but uses only pure structural recursion.

**Hint**: Recurse on a natural number in addition to recursing on a list, so that you do not need to “toggle” a parameter (which would make your functions generative).

This concludes the list of questions for which you need to submit solutions. As always, check your email for the basic test results after making a submission.

**Bonus (5%)**: Reimplement \texttt{cells->tallies} with two restrictions: you must use pure structural recursion with no accumulators, and you must avoid exponential blowups (such as the exponential blowup for \texttt{max-list} on slide 07-04).

Submit your solution in \texttt{bonus.rkt}. As usual, if you use auxiliary sources for this question (including looking up algorithms) you must cite your sources. Not doing so is an academic integrity violation.

**Enhancements**: Reminder—enhancements are for your interest and are not to be handed in.

The IEEE-754 standard, most recently updated in 2008, outlines how computers store floating-point numbers. A floating-point number is a number of the form

\[ 0.d_1d_2\ldots d_t \times \beta^e \]

where \(\beta\) is the base, \(e\) is the integer exponent, and the mantissa is specified by the \(t\) digits \(d_i \in \{0, \ldots, \beta - 1\}\). For example, the number 3.14 can be written

\[ 0.314 \times 10^1 \]

In this example, the base is 10, the exponent is 1, and the digits are 3, 1 and 4. A computer uses the binary number system; the binary equivalent to the above example is approximately

\[ 0.11001001_2 \times 2^{10_2} \]

where \(10_2\) is the binary integer representing the decimal number 2. Thus, computers represent such numbers by storing the binary digits of the mantissa (“11001001” in the example), and the exponent
(“10” in the example). Putting those together, a ten-bit floating-point representation for 3.14 would be “1100100110”.

However, computers use more binary digits for each number, typically 32 bits, or 64 bits. The IEEE-754 standard outlines how these bits are used to specify the mantissa and exponent. The specification includes special bit-patterns that represent Inf, −Inf, and NaN.