Assignment: 10

Due:	Friday, December 4, 2015 9:00pm
Language level:	Intermediate Student with lambda
Allowed recursion:	Pure structural, accumulative or generative, except where explicitly restricted.
Files to submit:	`mergesort.rkt`, `puzzle.rkt`, `subsets.rkt`
Warmup exercises:	HtDP 28.1.6, 28.2.1, 30.1.1, 31.3.1, 31.3.3, 31.3.6
Practise exercises:	HtDP 28.1.4, 28.2.2, 28.2.3, 28.2.4, 31.3.7

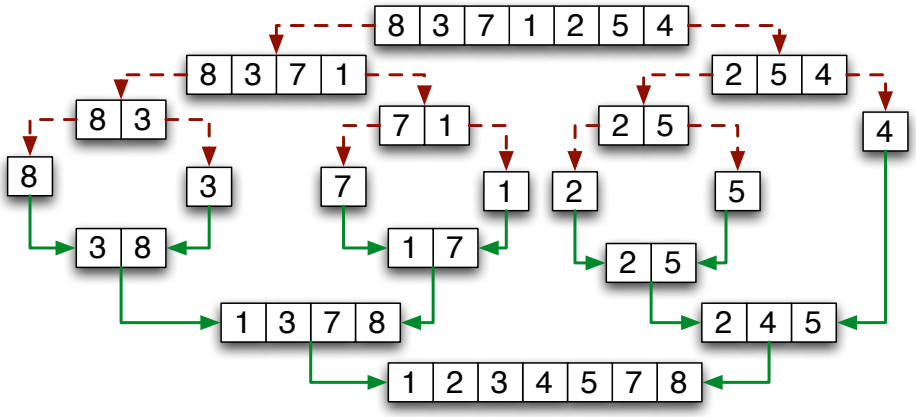
- **Make sure to read the entire assignment before starting to write any code.**
- **Start this assignment as early as possible.**
- For some questions on this assignment we provide a starter file and a file with some helpful functions.
- In this assignment your *Code Complexity/Quality* grade will be determined by both how clear your approach to solving the problem is and how effectively you make use of local constants and helper functions.
- If your solutions are *very* inefficient, it is possible that you may lose correctness marks. For most of the examples provided, your solutions should complete in less than a minute (on any computer built in the last 5 years).
- You may reuse the provided examples, but you should ensure you have an appropriate number of examples and tests.
- Unless restricted by the question, you may use any function in *Intermediate Student with lambda*.
- Your solutions must be entirely your own work.
- Solutions will be marked for both correctness and good style as outlined in the Style Guide
- Good style includes using locally defined constants and helper functions and lambda where appropriate.

Provided code

The file `puzlib.rkt` provides a set of useful helper functions – *Do not modify this file!* You should consult the file to become familiar with the provided functions.

In addition, we have provided `puzzle.rkt` to get you started.

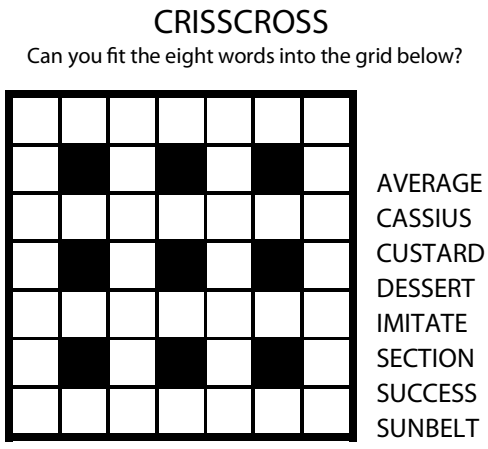
1. [25%] The *mergesort* algorithm uses a divide-and-conquer approach and generative recursion similar to the *quicksort* algorithm described in the notes. In each step of *mergesort*, if the length of the list is greater than one, two sublists are generated. Typically, the two lists correspond to the left (beginning) half of the list and the right (end) half of the list. If there is an odd number of elements, it does not matter which sublist contains the extra element. It is important that every element appears in exactly one of the sublists. Each sublist is then recursively sorted, and then the resulting two sorted lists are **merged** using a *merge* function as described in module 06.



In a file `mergesort.rkt`, write an abstract list function *mergesort* that consumes a list and a *less than* or *greater than* comparator function (*e.g.*, $<$, $>$, *string<?*). Your function must follow the same contract and produce a value equivalent to the result produced by the built-in abstract list function *quicksort*.

Your function MUST implement the *mergesort* algorithm as described above: at each step, you must generate two new sublists which are recursively sorted and then merged. There are other implementations of *mergesort* (described in the textbook) that first generate a list of lists and then continue to merge those lists until only one list remains – we will not accept solutions that use that approach. However, the sublists you generate do not have to correspond to the left half and the right half of the list, as long as they differ in length by at most one. For example, you could generate a list of the elements in the odd positions and a list of the elements in the even positions.

2. [75%] A *Crisscross puzzle* is a logic-based word puzzle in which you are given a crossword-style grid and a list of words that must be placed within the spaces in the grid. The goal is simply to fill the grid with the words so that words that cross use the same letter in the crossing position. All words will read left-to-right or top-to-bottom. Typically the puzzles are constructed to have a unique solution. Occasionally, the solution will not use all the words in the list, though every empty cell in the grid will be filled. Here is a simple example of a Crisscross (by Shawn Kennedy, used with permission):



The black cells are what we will refer to as *unused cells*, and the white cells will initially be *empty cells*.

In this assignment, you will write a program that solves Crisscross puzzles. The core of this algorithm closely resembles the easier of the graph searching algorithms discussed in Module 12 of the lecture notes. Implicitly, a vertex in this "graph" is the current state of a puzzle, with some of the words filled in, and the neighbours of that vertex are new states in which one additional word has been filled in, if possible. We search for a path from an initially empty board to one in which all spaces are filled in.

We must introduce some new data types to manage all this data. First, we define the raw input to the puzzle solving algorithm:

```
;; A Puzzle is a (list (listof Str) (listof Str))
```

The first list of *Str* in the *Puzzle* is a visual description of the grid to be filled in. It is a list of lines, each made out of . (an unused cell) and # (empty cell). The second list is the set of words to be placed in the grid. The puzzle shown above would therefore be described as:

```
'(("#######" "#.#.#.#" "#######" "#.#.#.#" "#######" "#.#.#.#" "#######")
  ("AVERAGE" "CASSIUS" "CUSTARD" "DESSERT"
    "IMITATE" "SECTION" "SUCCESS" "SUNBELT"))
```

Given a *Puzzle*, we immediately construct the initial *State*, the focus of the searching process. A *State* is described as follows:

```
;; A Grid is a (listof (listof Char))

(define-struct wpos (row col horiz? len))
;; A WPos (Word Position) is a (make-wpos Nat Nat Bool Nat)
;; requires: len > 1

(define-struct state (grid positions words))
;; A State is a (make-state Grid (listof WPos) (listof Str))
```

A *Grid* represents the current, *partially finished* puzzle. During the search it will contain the characters . (unused cell), # (empty cells – unfinished parts of the puzzle) and the letters from words that have been filled in the puzzle. Unlike *Puzzle*, which represented the grid portion of the puzzle as a (*listof Str*), the *Grid* is left as a (*listof* (*listof Char*)) for convenience.

A *WPos* (Word Position) keeps track of where a word occurs in the Grid. The *row* and *col* correspond to the starting position of the word. The upper-left position corresponds to row 0 and col 0. The Boolean *horiz?* indicates if it is a horizontal word (true) or vertical word (false). Finally, the *len* is the length of the word.

In the previous example, the list of all *WPos* are:

```
(list (make-wpos 0 0 true 7)
      (make-wpos 2 0 true 7)
      (make-wpos 4 0 true 7)
      (make-wpos 6 0 true 7)
      (make-wpos 0 0 false 7)
      (make-wpos 0 2 false 7)
      (make-wpos 0 4 false 7)
      (make-wpos 0 6 false 7))
```

A *State* stores a (partially solved) Grid, a (*listof WPos*) corresponding to all words that have **not** yet been filled in, and a (*listof Str*) corresponding to all words that have not yet been used.

The fundamental step in the search algorithm is to transform a *State* into a new *State* in which one additional word has been placed in the grid. We do this by replacing the empty (#) cells in the grid with the word's letters, and removing the associated *WPos* and *Word* from the two lists in the *State*.

From a given *State* we must also decide what to consider as its neighbours. We choose a suitable *Wpos* (more on this choice later) and find all words that could be successfully placed in that location in the grid. A word can be successfully placed if all of the grid cells are empty cells (#) or have a matching letter in each corresponding location.

For example, the word *STARWARS* could be successfully placed in cells with values *S##RW##S*, but not if the values were *S##RT##K*.

We construct a list of neighbours consisting of *State*s, one for each of the *Word*s that fits that *WPos*, with the *Word* placed in the grid.

If no words qualify, then the neighbours will be an empty list and search cannot continue from this point: this *State* represents a dead end.

When you have digested this high level description, download the file `a10.zip` from the course website and unzip it. In it you will find a file `puzlib.rkt` containing a few helper functions; a file `puzzle.rkt` containing a partially completed skeleton of the program you must develop; and a set of test puzzles. Your goal is to fill in the missing pieces of `puzzle.rkt`, as described in the file.

While there are numerous ways to solve this problem, we have designed a detailed approach for you to follow.

We have specified some helper functions that will lead you toward a solution. The helper functions you are required to write are as follows:

- *transpose* consumes a *Grid* and produces the transpose of that *Grid*. In other words, for each element, its row and column are swapped. Because of the nature of lists, many functions are more straightforward when designed to process *rows*. With *transpose*, we can process *columns* as easily as rows, we will just have to be careful to adjust results accordingly. We have provided a *flip* helper function to aid in that adjustment.
- *find-wpos* consumes a row from a *Grid* (*i.e.*, a (*listof Char*)) and the *row* number and produces a list of all horizontal *WPos* that occur in that row. A word must be two or more consecutive #'s. The *col* will indicate where the word begins (remember that the first column is column zero). The *len* is the number of consecutive #'s, and the *horiz?* is always *true*. The order of the produced list does not matter.

  This is one of the hardest helper functions you will have to write, so you should test it thoroughly. Some initial examples have been provided. Be careful not to produce any word positions of length one, as that is not a valid word (it might be easier to process all of the words and then *filter* out words of length one).
- *initial-state* consumes a *Puzzle* and produces the initial *State* to start searching from. The *grid* and *words* will be mostly straightforward; the hardest part will be to construct a list of all *WPos* in the *Puzzle* (both horizontal and vertical).
- *extract-wpos* consumes a *Grid* and a *WPos* and produces the (*listof Char*) corresponding to that word position within the *Grid*. If the *Grid* from the *initial-state* is used, each extracted word will contain only #'s, and at the end of the search, each extracted word will contain only letters. During the search, the extracted word may be a combination of #'s and letters.
- *replace-wpos* consumes a *Grid*, a *WPos* and a (*listof Char*) and produces the *Grid* with the word position replaced by the word represented by the (*listof Char*). In other words, *replace-wpos* is the opposite of *extract-wpos*.

- *fit?* consumes a word as a (*listof Char*) and a (*listof Char*) that represents a word position in the puzzle. *fit?* determines if the word can successfully be placed in the corresponding word position.
- *neighbours* is the primary function required to search and solve the puzzle. *neighbours* consumes a *State* and produces a (*listof State*) that represents valid neighbour *State*s with one additional word placed in the puzzle. To generate the *neighbours*, first a *WPos* must be selected. For small test puzzles, you can simply select the first *WPos* in the *positions* list, but for larger puzzles you will want a better strategy. Once the *WPos* has been selected, each remaining *word* can be tested to see if it can be legitimately placed at the corresponding *WPos*. If a word can be placed there, a new *State* is generated with that word placed, the *WPos* removed, and the corresponding *word* removed from the list of unused words. *neighbours* produces a list of all *State*s, each corresponding to a different word placed in the grid at the selected *WPos*. The order of the list produced by *neighbours* does not matter.

  If you simply select the first *WPos* in your *neighbours* function, you will not be able to solve some of the larger puzzles in time. One strategy you can use is to select the *WPos* that has the most letters already filled in by other criss-crossing words. This will make your search significantly faster.

Each of these steps will probably require you to write additional helper functions, either at the top level or within locals. Top-level helper functions require complete design recipes; local helpers require only contracts and purposes.

The provided `puzzle.rkt` comes with some basic tests that will help you complete the assignment.

Our automated testing will test all of the above functions.

When everything is working, you should be able to solve the problems in the ten `puzzleNN.txt` files. Happy solving!

---

3. **8% Bonus**: Place your solution in the file `subsets.rkt`. You do not need to include the design recipe for any of these bonus questions. However, you must provide a sentence or two that briefly explains your approach and demonstrates that you understand your code and that you are submitting your own work.

   (a) Write the Racket function *subsets1*, which consumes a list of numbers and produces a list of all of its subsets. For example, (*subsets1* '(1 2)) produces something like (*list* '(1 2) '(1) '(2) '()). The order of subsets in the list may vary and the order of the elements within the subsets may also vary – any complete ordering will be accepted. You can assume the input list does not contain any duplicates. Write the function any way you want. (Value: 1%)

   (b) Now write the Racket function *subsets2*, which behaves exactly like *subsets1* but which does not use any explicit recursion or helper functions. You must rely on abstract list functions and **lambda** (and potentially standard list functions like *cons*, *first*, *rest*, *append*, etc.). Your solution must only be one or two lines of code. Note that if you solve this question, you can also use it as a solution to the previous one—just have *subsets1* apply *subsets2*. (Value: 2%)

   (c) For the ultimate challenge, write the Racket function *subsets3*. As always, the function produces the list of subsets of an input list of numbers. Do not write any helper functions, and do not use any explicit recursion (i.e., your function cannot call itself by name). Do not use any abstract list functions. In fact, use only the following list of Racket functions, constants and special forms: *cons*, *first*, *rest*, *empty?*, *empty*, **lambda**, and **cond** (including **else**). You are permitted to use **define** exactly once, to define the function itself. (Value: 5%)

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

It is perhaps stretching to call what we are about to look at "an artificial intelligence application". It is a program which mimics a human psychologist in an extremely simple way. The idea comes from Joseph Weizenbaum, who wrote a similar program and reported on it in the January 1966 issue of Communications of the ACM. A typical interaction might look like this:

```
eliza> Hello, how are you?
human> I feel really tired today.
eliza> Why do you feel really tired today?
human> I stayed up all night doing CS 135.
eliza> You stayed up all night doing CS 135?
human> They make us work too hard.
eliza> Can you be more specific?
human> Assignment 7 was a killer.
eliza> Tell me more about Assignment 8.
```

You can see how it wouldn't take a lot of processing to do this. In fact, we are going to do even less processing than Weizenbaum's program, which remembered past interactions in a certain way and could bring up phrases from them. Our program will treat every input from the human in isolation. To avoid dealing with the complexities of parsing strings, we will assume the input and output are lists of symbols:

```
> (eliza '(I feel really tired today))
'(Why do you feel really tired today)
```

We're not going to bother with punctuation, either. Since this is an enhancement, you can put the ability to handle strings with punctuation instead of lists of symbols into your implementation if you wish. (To get output that uses quote notation, select Details in the Choose Language dialog, and choose quasiquote.)

The key to writing an *eliza* procedure lies in patterns. The patterns we use in *eliza* allow the use of the question mark *?* to indicate a match for any one symbol, and the asterisk $*$ to indicate a match for zero or more symbols. For example:

'(I ? you) matches '(I love you) and '(I hate you), but not '(I really hate you).

'(I $*$ your ?) matches '(I like your style) and '(I really like your style), but not '(I really like your coding style).

We can talk about the parts of the text matched by the pattern; the asterisk in '(I $*$ your ?) matches '(really like) in the second example in the previous paragraph. Note that there are two different uses of the word "match": a pattern can match a text, and an asterisk or question mark (these are called "wildcards") in a pattern can match a sublist of a text.

What to do with these matches? We can create rules that specify an output that depends on matches. For instance, we could create the rule

'(I $*$ your ?) $\rightarrow$ '(Why do you 1 my 2)

which, when applied to the text '(I really like your style), produces the text '(Why do you really like my style).

So *eliza* is a program which tries a bunch of patterns, each of which is the left-hand side of a rule, to find a match; for the first match it finds, it applies the right-hand side (which we can call a "response") to create a text. Note that we can't use numbers in a response (because they refer to matches with the text) but we can use an asterisk or question mark; we can't use an asterisk or question mark in a pattern except as a wildcard. So we could have added the question mark at the end of the response in the example above.

A text is a list of symbols, as is a pattern and a response; a rule is a list of pairs, each pair containing a pattern and a response.

Here's how we suggest you start writing *eliza*.

First, write a function that compares two lists of symbols for equality. (This is basic review.) Then write the function *match-quest* which compares a pattern that might contain question marks (but no asterisks) to a text, and returns *true* if and only if there is a match.

Next, write the function *extract-quest*, which consumes a pattern without asterisks and a text, and produces a list of the matches. For example,

```
(extract-quest '(CS ? is ? fun) '(CS 135 is really fun))
 => '((135) (really))
```

You are going to have to decide whether *extract-quest* returns *false* if the pattern does not match the text, or if it is only called in cases where there is a match. This decision affects not only how *extract-quest* is written, but other code developed after it.

Next, write *match-star* and *extract-star*, which work like *match-quest* and *extract-quest*, but on patterns with no question marks. Test these thoroughly to make sure you understand. Finally, write the functions *match* and *extract*, which handle general patterns.

Next we must start dealing with rules. Write a function *find-match* that consumes a text and a list of rules, and produces the first rule that matches the text. Then write the function *apply-rule* that consumes a text and a rule that matches, and produces the text as transformed by the right-hand side of that rule.

Now you have all the pieces you need to write *eliza*. We've provided a sample set of starter rules for you, but you should feel free to augment them (they don't include any uses of question marks in patterns, for example).

Prabhakar Ragde adds a personal note: a version was available on the computer system that he used as an undergraduate, and he knew fellow students who would occasionally "talk" to it about things they didn't want to discuss with their friends. Weizenbaum reports that his secretary, who knew perfectly well who created the program and how simplistic it was, did the same thing. It's not a substitute for advisors, counsellors, and other sources of help, but you can try it out at the following URL:

http://www-ai.ijs.si/eliza/eliza.html

The URL below discusses Eliza-like programs, including a classic dialogue between Eliza and another program simulating a paranoid.

http://www.stanford.edu/group/SHR/4-2/text/dialogues.html

Here is a more recent example of modern, graphical chatbots conversing with each other, produced at Cornell:

http://www.youtube.com/watch?v=WnzlbyTZsQY