CS 135 Winter 2016
Petrick

Assignment: 10

Due: Monday, April 4, 2016 9:00pm
Language level: Intermediate Student with lambda
Allowed recursion: Pure structural, accumulative or generative , except where explicitly restricted.
Files to submit: `polyominoes.rkt`, `subsets.rkt` (for the bonus)
Warmup exercises: HtDP 28.1.6, 28.2.1, 30.1.1, 31.3.1, 31.3.3, 31.3.6, 32.3.1
Practise exercises: HtDP 28.1.4, 28.2.2, 28.2.3, 28.2.4, 31.3.7, 32.3.2, 32.3.3

- **Make sure to read the entire assignment before starting to write any code.**
- **Start this assignment as early as possible.**
- For this assignment we provide a starter file, a file with examples and a file with some helpful functions. There is a description of these files following the specifications of the questions to submit.
- 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 function definitions.
- 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 or technique discussed in the slides for the entire course, including abstract list functions and reverse.
- 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.

An *n-omino* is a connected set of *n* squares, where adjacent squares are connected to each other across complete edges. We typically use terms like *monomino* ($n = 1$) for a single square, *domino* ($n = 2$) where 2 squares are connected but can be oriented horizontally or vertically and *triomino* ($n = 3$) where 3 squares are connected in various configurations. The *tetronimoes* ($n = 4$) are the familiar pieces from the game Tetris; the *pentominoes* ($n = 5$) are a classic puzzle. The *n*-ominoes for all *n* are known collectively as the *polyominoes*. Polyominoes are a recurring theme in recreational mathematics and computer science.

A natural, challenging puzzle is to fill a region (frequently a rectangle) with a given set of polyominoes. Each polyomino is used only once and the total number of squares composing all the polyominoes will exactly match the size of the region so a solution will use all the polyominoes and the region will be completely filled; i.e. the region will not contain any gaps. This is also known as a *packing problem*. For example, there are twelve distinct pentominoes (where we consider rotations and reflections of a given polyomino to be equivalent) consisting of a total of $12 \times 5 = 60$ building squares. They can be used to fill a 6×10 rectangle, a 5×12 rectangle, a 4×15 rectangle, or a 3×20 rectangle. These puzzles can be quite challenging; although there are 2339 solutions in the 6×10 case, there are many, many more ways to get stuck!

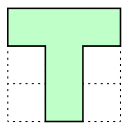
We can use the searching-with-backtracking algorithm presented in Module 12 Graphs to locate polyomino packing solutions when they exist. In fact, if we are careful about how we set up the search, we can use a simple, acyclic version of *find-route* like the one shown on Slide 19 of Module 12.

In order to provide a common framework in which to develop a solution, we begin with the following data definitions, which will be elaborated upon below:

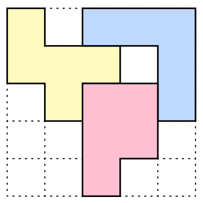
```
(define-struct pos (x y))
;; A Pos is a (make-pos Nat Nat)

;; A Grid is a (listof (listof Char))
;; requires: both inner and outer lists of Grid are non-empty

(define-struct state (puzzle pieces))
;; A State is a (make-state Grid (listof Grid))
```

A *Pos* is essentially the same as a *Posn*, but we define a new type in order to highlight the distinction that a *Pos* should only have natural numbers for coordinates.

A *Grid* is a non-empty 2D array of characters, represented using a list of lists. All of the sublists in the grid are required to have the same length. We reserve the period character #\. to represent an empty space in the grid; all other characters are assumed to be parts of polyominoes. The choice of letters for polyominoes is arbitrary, as long as each piece has its own distinct letter. We can use the *Grid* type to represent both individual polyominoes and a puzzle grid with multiple pieces in place, as shown in Figure 1. When used to represent a single polyomino, we assume that there is no "padding": there are exactly as many rows and columns in the grid as are needed to hold the piece.



```
(list (list #\q #\q #\q)
      (list #\. #\q #\.)
      (list #\. #\q #\.))
```



```
(list (list #\A #\. #\B #\B #\B)
      (list #\A #\A #\A #\. #\B)
      (list #\. #\A #\C #\C #\B)
      (list #\. #\. #\C #\C #\.)
      (list #\. #\. #\C #\. #\.))
```

Figure 1: Examples of an individual piece (left) and a partial puzzle (right) represented using the Grid type.

A *State* holds the current configuration in a polyomino packing problem. It consists of a partially filled-in puzzle, together with a list of individual polyomino pieces left to be placed. Our goal will be to move successively from *State* to *State* via a *neighbours* function, eventually discovering a solution (i.e., a *State* in which the list of pieces to place is empty) or proving by exhaustion that none exists.

There are many ways to write this algorithm. In this assignment we will build up a solution in one particular way. You must follow this approach in order to receive full marks, even if you have ideas for alternatives. The benefit is that we can mark the individual parts of the solution, and award part marks even if you don't get everything working. With that in mind, you must solve the individual questions below.

1. (a) Write a function *build-2dlist*, a two-dimensional equivalent of the built-in Racket function *build-list*. The function *build-2dlist* consumes two natural numbers, representing the width and height of a grid, and a function that is applied to all (x,y) positions corresponding to positions in the grid, in that order. The origin, $(0,0)$, is the upper left corner of the grid. For example, (*build-2dlist* 3 2 *f*) should produce (*list* (*list* (*f* 0 0) (*f* 1 0) (*f* 2 0)) (*list* (*f* 0 1) (*f* 1 1) (*f* 2 1))). For full marks, you must solve this question using abstract list functions and **lambda**, without any explicit recursion or helper functions (local or otherwise).

   (b) Write a function *all-positions*. It consumes two natural numbers greater than zero, *w* and *h*, in that order, and produces a (*listof Pos*) containing all possible positions in a grid with width *w* and height *h*. This is a one-line function that uses *build-2dlist* as a helper function. The order of items in the produced list does not matter; you should use the provided function *lists-equiv?* in your *check-expect* statements to verify your result.

2. Write a function *all-orientations* that consumes a *Grid* representing a single polyomino, and produces a list of *Grid*s containing all distinct rotations and reflections of that polyomino. A general polyomino can have as many as eight orientations, but any symmetries will cause some of those orientations to be identical (see Figure 2). The order of the orientations does not matter; however, you must eliminate duplicate orientations; your search algorithm will also be faster if you do. You should use the function *lists-equiv?* that we provide and describe in the provided code section.

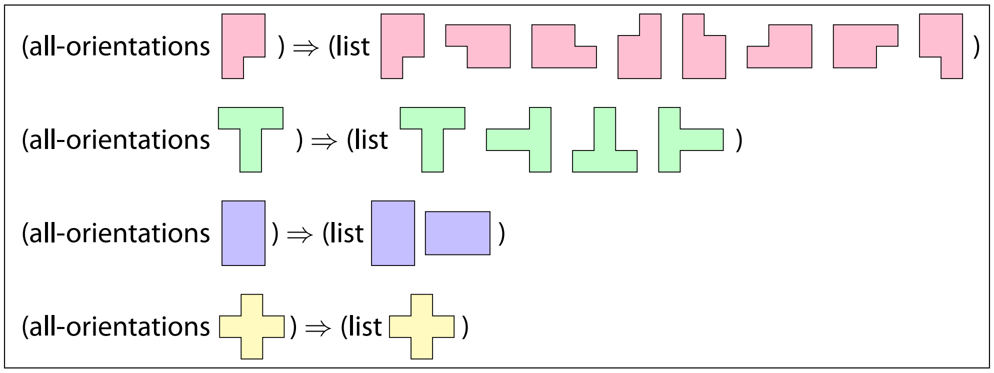

(all-orientations) ⇒ (list)
(all-orientations) ⇒ (list)
(all-orientations) ⇒ (list)
(all-orientations) ⇒ (list)

Figure 2: Demonstrations of computing all the orientations associated with polyominoes with different symmetries.

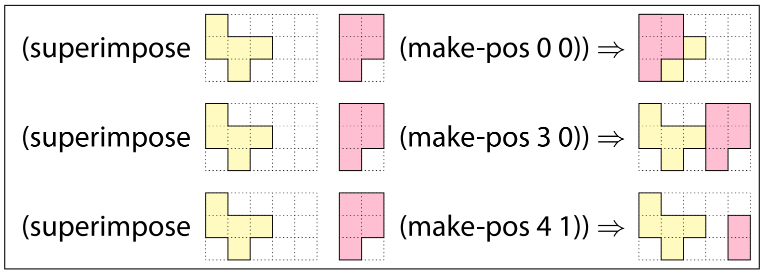

(superimpose (make-pos 0 0)) ⇒
(superimpose (make-pos 3 0)) ⇒
(superimpose (make-pos 4 1)) ⇒

Figure 3: Demonstrations of superimposing a polyomino on top of an underlying grid.

(Hint: this function is best written with the use of several helper functions, which you should consider embedding in a **local**. You can (and will probably need to) use *reverse*.)

3. Write a function *first-empty-pos* that consumes a *Grid* and produces the *Pos* of the first #\. character in the grid. You can assume the origin is the upper left corner of the grid; hence by "first" we mean the leftmost #\. in the uppermost row that has one. If the *Grid* has no empty positions, produce *false*.

   (You may use *all-positions*.)

4. Write a function *superimpose* that consumes three parameters: two *Grid*s which we will refer to as *base* and *top*, and a *Pos*, in that order. Recall that the origin is the upper left corner of the grid. The function produces a new *Grid* in which *top* is laid over *base* such that the consumed *Pos* indicates the location of the upper left corner of *top*. Any #\. characters in *top* do not overwrite the contents of *base*. Any shifted positions that would be outside the bounds of *base* are ignored. See Figure 3 for examples.

5. Now write the function *neighbours* that carries out one search step, (as briefly described in Slide 28 of Module 12). It consumes a single *State* as input and produces a list of *State*s in which one additional polyomino has been placed in the puzzle and removed from the list of pieces yet to be placed. This function is a lot more complicated than all the previous ones. Here are some (non-required) suggestions for how to break it down into parts:

   - Find the first empty position in the *Grid*.
   - Write a helper predicate, similar to *superimpose*, that determines whether a given polyomino can legally be superimposed on top of the current puzzle at a given *Pos*. The superposition is considered legal if each square of the new piece lies in an in-bounds, empty position in the puzzle, and one of those newly occupied positions is the first empty position in the current puzzle grid (from the bullet above).
   - For a given polyomino at a given orientation, find all offsets that lead to legal superpositions in the manner above.
   - Use the techniques above to find *all* legal offsets of *all* orientations of *all* available pieces. For each of these, construct a new *State* in which the superposition has been carried out and the piece has been removed from the available list. This is the list of neighbours of the original input *State*.

   Most of these parts can probably be embedded as local helper functions inside of *neighbours*.

   Although you should experiment with your *neighbours* function to convince yourself of its correctness, you are not required to write explicit examples or tests for it. In any case, this function can produce very long, complicated, and in some sense unpredictable output. The function is better tested in the larger context of puzzle solving.

Provided code

We are providing you with two files to get you started: `a10.rkt` and `polyominoes.rkt`. The file `a10.rkt` provides a set of useful helper functions and data and *must not be modified!*

You will find the following in `a10.rkt`:

- The *search* function behaves like *find-route* in the course notes. This version is a higher-order function: in addition to consuming a *State* to search from, it also consumes helper functions that determine whether the search is complete, and the computation of a state's neighbours. We have also modified the function with a parameter that allows you to visualize the search graphically (see the purpose for *solve-puzzle* in the source code). Watching your search in real time is quite useful as a debugging tool.

- The *draw-grid* function can be used to draw a graphical representation of your grid. This function is very useful during debugging, or to visualize a search while it is underway.

- The *lists-equiv?* function determines whether two lists contain the same elements, regardless of the order of the elements. This function can be useful when testing with *check-expect*, in cases where your function produces a list that is not guaranteed to be in any particular order.
- The file also contains a number of test sets of polyominoes. Open `a10.rkt` in Dr. Racket to see them. We are providing an additional file of tests in `kanoodle.rkt`, adapted from the popular puzzle game Kanoodle. If you want to use the examples from this file, uncomment the line (*require* "kanoodle.rkt") at the top of your *polyominoes.rkt* file and make sure the *kanoodle.rkt* file is in the same directory. It's easy to find more polyomino puzzles online by searching for web pages about polyominoes.

The file `polyominoes.rkt` is a skeleton of the file you will eventually submit. It contains the data definitions for the assignment, a stub for the *neighbours*, and one additional function *solve-puzzle*. **Do not modify the code for** *solve-puzzle*. This function consumes an initial grid (which can be partially filled), a list of pieces that must be added to the grid, and a visualization mode. It uses your *neighbours* function to perform the search, considering a puzzle to be solved when no pieces remain to be placed. The visualization mode is one of three symbols: 'offline, which operates like a normal Racket function, 'at-end, which displays a picture of the filled-in puzzle if a solution is found, and 'interactive, which shows all the intermediate states in the search. The last mode is a good way to convince yourself that your search is working correctly.

Your mission is to answer all the questions above, placing your code in `polyominoes.rkt`.

One final note

Some polyomino puzzles are so large that they overwhelm our solving algorithm. We will test your code mostly on small problems that shouldn't require more than a couple of seconds of computation. Don't be concerned if, for example, your solution takes a really, really long time on the 15×4 pentominoes case. The question of how to search for polyomino packings efficiently is a deep, fascinating problem in computer science. One famous solution that is not easily implemented in Racket is the *Dancing Links* algorithm by the legendary computer scientist Donald Knuth; see the algorithm's Wikipedia entry for details.

---

6. **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.

   (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)) should produce something like (*list* '(1 2) '(1) '(2) '()). The order of subsets in the list may 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 two lines of code, one of which is the function header. Note that if you solve this question, you can also use it as a solution to the previous one—just copy the function and rename the copy *subsets1*. (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**. 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 8 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