Fall 2022
Becker, Kamal, Nasr Esfahani, Tompkins, Van Leeuwen, Vasiga

Assignment: 10

Due: Tuesday, December 6, 2022 9:00 pm
Coverage: End of Module 19
Language level: Intermediate Student with Lambda
Allowed recursion: Any (see individual questions)
Files to submit: `tubes.rkt, tubes-bonus.rkt`

- Make sure you read the A10 Official Post and FAQ post on Piazza for the answers to frequently asked questions.
- Your solutions for assignments will be graded on both correctness and readability. This assignment will be auto-graded.
- Policies from A09 carry forward.
- Submit examples for all questions by **Friday, December 2 at 8:00AM** (before the due date of A10). See the Example Submissions and Basic Test Results post on Piazza for examples of correct and incorrect example submissions.
- For this assignment, you may use any function learned in any module, as well as any other functions that are valid in Intermediate Student with Lambda. Note, however, that if you use a function or special form that was not covered in any module, you are on your own: course staff will not answer or assist you in any way.

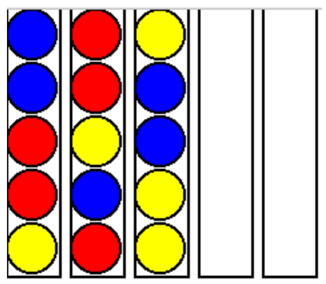
Here are the assignment questions that you need to submit.

1. This entire assignment is related to solving a particular game: Balls in Tubes. There are many variations of this game, including several apps and websites, some which even use fluids rather than balls (though it can be shown those are equivalent in terms of difficulty).[1]

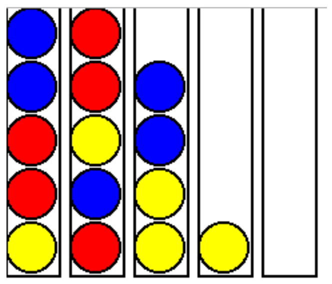
   The actual game involves T tubes, where each tube can hold up to S balls. Scattered across these T tubes are $S \cdot C$ balls, where there are S balls with one of the C colours. As such, $C \leq T$. Note that T, S, and C can be zero, so long as the conditions mentioned in the previous three sentences hold.

   The goal of the game is to arrange the balls so that each non-empty tube has S balls of the same colour. At any given step of the game, the top ball from one tube may be moved to the top of any other non-full tube.

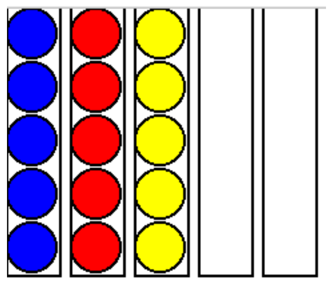
   Here is an example initial configuration with 5 tubes which can hold 5 balls each, and there are 3 different colours for the balls:



   After one possible move, the configuration could be as follows:



   Finally, after a series of moves, the final configuration could be as follows:



[1] https://arxiv.org/pdf/2202.09495.pdf

Your solution will be built with various pieces, but each of these pieces is relatively independent.

The key structure which you will use is a Game, which is defined as follows:

```
;; A Game is (make-game Nat Nat (listof (listof Sym)))
(define-struct game (tubesize maxcolours tubes))
```

As discussed in class, solving puzzles like this game involves searching an implicit graph. The graph searching algorithm (solve) will be similar to the one presented in Module 18 (find-path), but you need to provide two functions: a function to determine if a puzzle is complete, and a function to produce a list of legal "next moves" if it is not complete. You will be solving the game one move at a time. We have broken these two functions into a number of steps to help you along. Note that we will test each of these functions separately, so you can get quite a few part marks on this question by completing some of them, even if your whole program is not complete. This also means that these functions must *not* be local. There are several example Games given in the starter file: use them frequently to test each function, in addition to possibly adding other Games for your own testing. You should begin by downloading tubes-starter.rkt from the course website, renaming it to be tubes.rkt, and place all of your functions for this assignment into that file.

To help you with debugging or to visualize how your program is searching, we have provided you with a module lib-tubes.rkt which gives you the ability to see your code in action. You must perform the following steps to use this module:

- Download the file lib-tubes.rkt to the same directory as your solution file (tubes.rkt).
- You will notice some setup and draw calls already in the starter file. Do not modify those lines of code.
- There are four choices when drawing:
  - 'off disables all drawing (useful for timing purposes)
  - 'norm shows the drawing at normal speed
  - 'slow shows the drawing at a slower speed
  - 'fast shows the drawing at a faster speed

Note that you **must not** define any functions with the same name as those defined in lib-tubes.rkt, or you will fail all automated tests.

(a) Write the Racket function check-colour?, which consumes two natural numbers, size and num, and a list of symbols, los, and produces true if each symbol in the list appears exactly size times and if there are at most num different symbols; otherwise, check-colour? will produce false. You should consider using generative recursion to solve this problem: after checking a particular symbol, remove every occurrence of it, and recurse.

(b) Write the Racket function `valid-game?` which consumes a `Game`, `gm`, and produces `true` if `gm` is a valid game, and `false` otherwise. You can assume that the data definition is met for `gm`. A game is valid if the following conditions hold:

- all tubes have at most `tubesize` symbols in them;
- there are at most `maxcolours` different symbols; and
- there are exactly `tubesize` occurrences of each different symbol.

Note that for all subsequent questions, you can assume that any `Game` is a valid game. As well, you should ensure that any `Game` you create is valid in your testing of subsequent functions. Additionally note that just because a `Game` is valid does not imply that it will be solvable.

(c) Write the Racket function `remove-completed`, which consumes a `Game`, `gm`, and produces a `Game` which is similar to `gm` but has any completed tubes removed. By a "completed tube", we mean any tube that is full, and all the balls in that tube are the same colour. Note that the `maxcolours` should be updated to be the total number of colours that remain after removing completed tubes.

(d) Write the Racket function `finished-game?`, which consumes a `Game`, `gm`, and produces `true` if the game is finished, and `false` otherwise. A game is finished if all tubes are either `empty`, or each tube is full with all balls of the same colour. Note that a game with no tubes is finished, since it satisfies the conditions vacuously. You may want to use `remove-completed` to help you with this question.

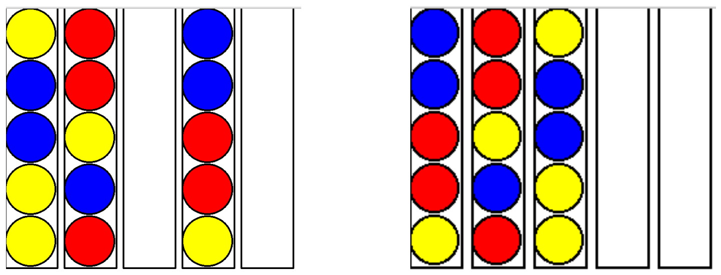
(e) Write the Racket function `num-blocks`, which consumes a list of lists of symbols, `llos`, and produces the number of "blocks" contained in `llos`. A block is a consecutive sequence of identical symbols within one list. For example, there are 4 blocks in `(list empty '(a a a) '(a a b a a))` since:

- `empty` lists have zero blocks.
- A list with identical symbols has one block. That is `'(a a a)` has one block.
- When reading the list from left-to-right, each time the symbol changes, we add one to the total number of blocks. That is, `'(a a b a a)` has three blocks.

(f) Write the Racket function `equiv-game?` which consumes two `Games`, `gm1` and `gm2`, and produces `true` if `gm1` and `gm2` are equivalent, and `false` otherwise. Two games are equivalent if they have:

- the same `maxcolours` value;
- the same `tubesize` value;
- the same number of tubes; and
- the tubes contain identical balls in identical order within the tube.

Note that the tubes themselves may possibly in a different ordering. You may want to use generative recursion in this function: if there is the same tube in both games, remove that tube from both games and recurse. For example, the `Game`s representing the following pictures would be equivalent:



(g) Write the Racket function `all-equiv?`, which consumes two lists of `Game`s, `log1` and `log2`, and produces `true` if every game in `log1` has one equivalent game in `log2`, and every game in `log2` has one equivalent game in `log1`, and otherwise produces `false`. To use a mathematical term, there is a bijection between `log1` and `log2`, up to equivalency in `Game`s.

(h) Write the Racket function `next-games`, which consumes a `Game`, `gm`, and produces a list of `Game`s that can happen by moving one ball from `gm`. You should consider using previously defined functions to help test this function.

Place all your functions for this question in the file `tubes.rkt`. If your solution is working correctly, it should solve all games with at most 3 colours and at most 3 balls per tube in less than 20 seconds. Note that we will test your solution on games which produce `false`, to ensure that you are not just coding `(define (solve ...) true)`: you should also test your program on unsolvable games.

Bonus: For a 10% bonus, modify your program to work on larger games, such as one of the larger examples with 9 non-full tubes of size 4. It should be able to solve games of that size in a reasonable amount of time (under 5 seconds on the fastest machines on the `linux.student.cs` environment, which are 2.3GHz machines with over 16GB of RAM). To attempt the bonus, you should copy your solution from `tubes.rkt` to `tubes-bonus.rkt`, and then you can make any modification you wish to any functions, or add any new functions, so long as the function `solve` has the same contract and purpose as the non-bonus version. We will only test (and time) the `solve` function for the bonus: none of the other functions listed above will be tested in the bonus. Note that you should not leave any top-level expressions that take a long time to evaluate, since the Basic and Correctness Tests may timeout.

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

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 ?)` → `'(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`