CS 135 Winter 2017
Victoria Sakhnini

Assignment: 10

Due:	MONDAY, April 3, 2017 9:00pm
Language level:	Intermediate Student with Lambda
Allowed recursion:	See individual questions
Files to submit:	`kenken.rkt, kenken-bonus.rkt`
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

Here are the assignment questions you need to submit.

1. We saw in class how we can use our graph-search algorithms to solve some kinds of games and puzzles. In this question, you will write a program to solve a KenKen puzzle.

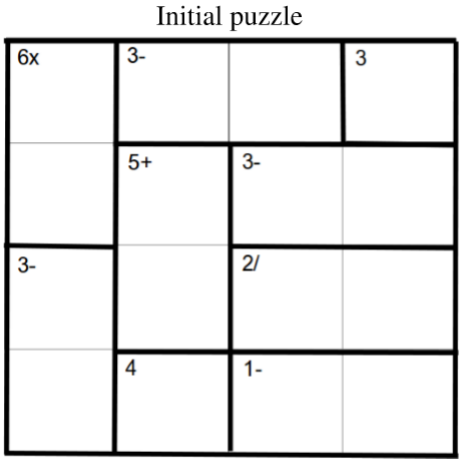
   In case you're not familiar with KenKen, here's how it works: the board is an $n \times n$ grid (with n^2 cells) where $n \geq 1$ that is divided into connected regions called *cages* of various sizes. You are given a *puzzle*, which is a board with each of the cages identified as either:

   - a single number (only used for cages of one cell) or
   - an arithmetic expression of the form NF, where N is a positive integer and F is an arithmetic function ($+$, $-$, $*$ or $/$)

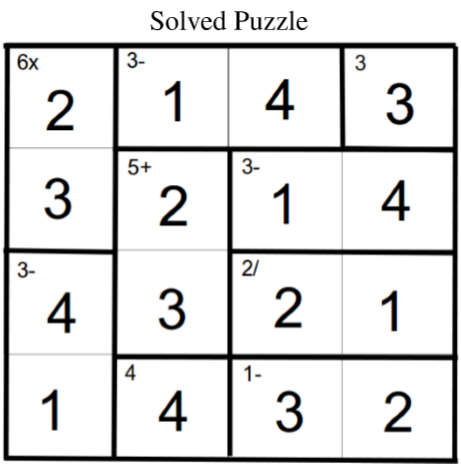
   You are to fill in the blank cells with integers from 1 to n so that each number appears exactly once in every row, once in every column, and the arithmetic expression is satisfied in every cage. We say that an arithmetic expression NF is satisfied if the values in the cage, when combined together using the function F produce the number N. For example, if a cage of three elements had the requirement of $7+$, the cage could (hypothetically) be filled with $1,2,4$ or $1,1,5$ or $1,3,3$ or $2,2,3$. Note that for the operations $/$ and $-$, only cages of exactly two cells are allowed, and that either of the two orderings of operands give the desired number in the cage: for example, if we have $4/$, this may be satisfied by $1,4$ or $4,1$.

   Consider the following KenKen (from `www.kenken.com`):

Initial puzzle



Solved Puzzle



For example, if we are given the puzzle on the left, we would like to fill it in to form the completed puzzle on the right. Note how each number from 1 to 4 appears exactly once in each row, column, and that each cage is satisfied. We will represent the initial puzzle in Racket as:

```
(make-puzzle
  ;; the size of the board
  4
  ;; a list defining the regions of each cage
  (list
    (list 'a 'b 'b 'c)
    (list 'a 'd 'e 'e)
    (list 'f 'd 'g 'g)
    (list 'f 'h 'i 'i))
  ;; a list describing the constraints for each cage
  (list
    (list 'a 6 '*)
    (list 'b 3 '-)
    (list 'c 3 '=)
    (list 'd 5 '+)
    (list 'e 3 '-)
    (list 'f 3 '-)
```

(*list* 'g 2 '/)

(*list* 'h 4 '=)

(*list* 'i 1 '−)))

Notice that the basic configuration is n lists of size n (in this case $n = 4$) and each cage is represented by a unique symbol, with a secondary association list indicating what the specific cage should evaluate to. Also notice that to keep the format of the operations consistent, we use the symbol '= to represent that the cage (of size one) has no operation.

The entries in the board will be one of the following three types:

- a symbol (as shown above in the initial configuration) which indicates that the particular cell is still unknown;
- a number which indicates that the particular cell along with all other cells of the same symbol have been filled in;
- a *Guess*, which contains both the symbol and possible number, indicating that the current cell has a potential answer that has not been verified in terms of the other values in the same cage. Note that all *Guess*es on the board will all have the same symbol (i.e., they are all for the same cage). Moreover, there cannot be two *Guess*es with different symbols on a board.

The provided file `kenken-start.rkt` contains examples highlighting the various entries that a puzzle board can contain.

The file `kenken-start.rkt` also contains the outline of a program to solve KenKen puzzles. Your task is to complete the functions. Follow the instructions below, and make sure the given *check-expect*s work. Be sure to add your own tests! The instructions may tell you how you are to write a given function; for example, some functions instruct you to use abstract list functions (ALFs) and not explicit recursion.

As discussed in class, solving puzzles like KenKen involves searching an implicit graph. The graph searching algorithm (*solve-kenken*) will be similar to the one presented in Module 12 (*find-route*), 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 KenKen puzzle one constraint at a time, which may or may not be in consecutive rows or columns. 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 will be more public tests than usual for these functions; use them frequently to test each function (in addition to doing your own testing, of course). **You must add your own tests to the ones given to you.**

To help you with debugging or to visualize how your program is searching, we have provided you with a module `kenken-draw.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 `kenken-draw.rkt` to the same directory as your solution file (`kenken.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 as the same name as those defined in `kenken-draw.rkt`, since you will fail all automated tests.

(a) Write the function *find-blank* which consumes a *Puzzle* and produces one of three possible outcomes:

- the position of the first cell we wish to fill in;
- *false* if there are no more cells to fill in (i.e., the puzzle is complete);
- the symbol 'guess if the symbol in the first constraint has only *Guess*es on the board.

By "first cell to fill in", we mean the cell containing the first symbol in the *constraints* of the given puzzle; if there is more than one occurrence of the first constrained symbol, pick the topmost one, and if there is more than one occurrence of the first bound symbol in the same row, then the leftmost one of those is the first. There are two ways to determine if all cells are filled in:

- all cells on the board will contain numbers, or
- the list of bindings will be empty.

The above two conditions will occur simultaneously for all valid puzzles. The location of the first cell to fill in should be produced as as a *Posn*. The top left corner of the puzzle is (*make-posn* 0 0), the top right is (*make-posn* (*sub1 n*) 0), and the bottom right is (*make-posn* (*sub1 n*) (*sub1 n*)) where *n* is the size of the puzzle. **Note: the function *find-blank* must use accumulative recursion.**

(b) Write the functions *used-in-row* and *used-in-column*, both of which consume a puzzle and a position and produce a list of the numbers used (either as *guess*es or as placed numbers) in the row or column (respectively) of the given position. The produced list should be in increasing order. Note that we will never give you a puzzle that already has two of the same number in the same row or column. Hint: the built-in function *list-ref* will be very helpful here, in addition to your other abstract list functions, including *quicksort*. **Note: these functions must *not* use explicit recursion; they must only use abstract list functions, *list-ref* and *quicksort*.**

(c) Write the function *available-vals* which consumes a puzzle and a position and produces a list of the numbers that could possibly be placed in this cell, ignoring any arithmetic constraints imposed by the cage. This function will use *used-in-row* and *used-in-column* to construct a list of all legal numbers to put in the given position of the given puzzle. The returned list should be in increasing order (that is, '(1 3 4 6), not '(3 6 4 1)) and should have no duplicate elements. **Note: this function must *not* use explicit recursion; it must only use abstract list functions.**

(d) Write the function *place-guess* which will consume the board portion of a *Puzzle*, a position to fill in (as a *Posn*), and a value to place inside the *Guess* structure. The function *place-guess* should produce the board with the symbol at the given position replaced with a *Guess* structure. You can assume the given position contains a symbol (i.e., it does not contain a number or a *Guess*). This function is called by *fill-in-guess*, which has been provided for you: *do not modify fill-in-guess*. You may use any type of recursion for this part.

(e) Write the function *guess-valid?* which consumes a *Puzzle* and produces *true* if the *Guess*es on the board of the given *Puzzle* satisfy the (first) constraint of the given Puzzle, and *false* otherwise. That is, the function should produce true if the values of all the guesses, combined together using the arithmetic operator satisfy the constraint. Be sure to test carefully for division and subtraction. This function will only be called when all occurrences of the symbol of the first constraint are guesses on the board. **Note: this function must *not* use explicit recursion; it must only use abstract list functions.** You may use the function *append* in the function *guess-valid?*.

(f) Write the function *apply-guess*, which consumes a *Puzzle* and produces a *Puzzle* where all of the *Guess*es have been converted into their corresponding natural number and the first constraint in the list of constraints has been removed. Thus, you can assume that all the guesses are valid. **Note: this function must *not* use explicit recursion; it must only use abstract list functions.**

(g) Write the function *neighbours* which consumes a *Puzzle* and produces a (*listof Puzzle*). This is the main function that will be used by *solve-kenken*. It consumes a puzzle, and produces a list of the valid next puzzles obtainable by either filling in the "first" cell for the symbol defined in the first constraint with a guess, or by applying each guess in the puzzle, so long as the guesses are all valid. If the guess is not valid, the guess is not applied and no neighbour is generated from that guess. The neighbours in the produced list should be increasing order of the guess value. **Note: this function must *not* use explicit recursion; it must only use abstract list functions**.

Submit your solution in the file `kenken.rkt` (be sure to rename the start file). If your program is working correctly, it will solve 6x6 KenKen puzzles in less than one second.

For a 10% bonus, modify your program to work on KenKen puzzles of size up to 9x9. It should be able to solve the hardest 9x9 puzzles in a reasonable amount of time (under 30 seconds on the fastest machines on the `linux.student.cs` environment, which are

2.3GHz machines with over 16GB of RAM). Hand in this version as `kenken-bonus.rkt`. We will only test (and time) the *solve-kenken* function for the bonus, but we will of course read your whole program for style. Note that you should not leave any top-level expressions that take a long time to evaluate, since the Basic and Correctness Tests will timeout.

You can find sample puzzles to give to your program here:

http://kenken.com/

http://krazydad.com/inkies/

Google can help you find more, of course.

This concludes the list of questions for which you need to submit solutions. Don't forget to always check your email for the public 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 ?) $\rightarrow$ '(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))

$\Rightarrow$ '((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`