Function Name: `scrabbleBoxes`

Inputs:

1. (*cell*) 1xN cell vector of nested letters

Outputs:

1. (*char*) 1xN word from letters

Topics: (*cell indexing*), (*iteration*)

Background:

Guess what...it's game night!! CS 1371 doesn't mess around when its game night, and to kick things off, we're playing Scrabble! However, you've been given a certain number of Scrabble boxes, each with a number of smaller scrabble boxes inside them. Finally, after opening all the interior boxes inside the first Scrabble box, you find a piece and move on to the next open box. After retrieving each piece, you can finally make a word! Fun!

Function Description:

You are given a 1xN cell array. Each element will be a cell containing nested cells of varying depths. Each position in the array will contain a single char at the deepest level which will compose the final word. Unnest the cells at each position, and concatenate the individual letters to output a word.

Example:

```
boxes = [{{{'M'}}},{'A'},{{'T'}},{'L'},{{{{{'A'}}}}},{{'B'}}];
word = scrabbleBoxes(boxes);
word → 'MATLAB'
```

Notes:

- Each nested cell will always contain either one char or another nested cell.

Function Name: deckCollation

Inputs:

1. (*cell*) A 1xN cell array of attack types
2. (*double*) A 1xN vector of attack strengths
3. (*logical*) A 1xN vector of whether the attack is a critical hit

Outputs:

1. (*cell*) A 1xN cell array of the final cells

Topics: (*cell creation*)

Background:

It's game night, and your friend has decided to try out a new card game he heard about. There are 3 different types of cards that contain different information about attacks on them: One has the strength of the attack; another has the attack type; and the final type has whether or not the attack is a critical hit. The rules are confusing, so while your friend reads the rules, he assigns you the job of collating the cards together, a job you soon realize can be accomplished much faster with MATLAB.

Function Description:

Given a cell array of strings, a vector of doubles, and a vector of logicals, collate them together into a single 1xN cell array, with each cell containing the attack type, the attack strength, and if it was a critical hit, in that order. So the first value of each vector will be combined into the first cell of the output cell array, the second values into the second cell, and so on.

Example:

```
attackType = [{'melee'}, {'projectile'}, {'magic'}];
attackStrength = [8, 3, 6];
criticalHit = [true, true, false];
collated = deckCollation(attackType, attackStrength, criticalHit);

collated → [{{'melee'}, {8}, {true}}, ...
            {{'projectile'}, {3}, {true}}, ...
            {{'magic'}, {6}, {false}}];
```

Function Name: `physicsHW`

Inputs:

1. (*cell*) Nx3 cell array of unknown variables and known parameters

Outputs:

1. (*cell*) Nx2 cell array of the unknown variable and its value

Topics: (*cell indexing*), (*conditionals*)

Background:

You're absolutely killing it at Scrabble! Having 100 points more than the 2nd place player, you're sure your win is inevitable. Suddenly, the Canvas app alerts you that your Physics homework is due in 30 minutes and you haven't even started! Is that even enough time to get everything done? Dr. Greco is sure to roast you in the middle of class in front of everyone and you sure can't have that! You turn to MATLAB in order to help you complete everything fast!

Function Description:

All of the physics homework is based on the following kinetic energy equation:

$$K = \frac{1}{2}mv^2$$

You are given an Nx3 cell array that contains two doubles and one char per row. The columns correspond to K, m, and v respectively. Solve for the unknown variable in each row and return an Nx2 cell array of which variable was calculated and what the value stands for.

Example:

```
hw_i = [{'K'}   {[5]}  {[2]}
        {[50]}  {[4]}  {'v'}
        {[70]}  {'m'}  {[9]}]
hw_f = physicsHW(hw_i);
hf_f → [{'K'}  {[10]}
        {'v'}  {[5]}
        {'m'}  {[1.728]}]
```

Notes:

- Each row is guaranteed to have only one unknown variable
- The char standing for the unknown variable will have the letter of the unknown variable
- Each number calculated must be rounded to the thousandths place (3 decimals)

Hints:

- The class() function may be useful

Function Name: `animalOverload`

Inputs:

1. (*cell*) A Nx5 cell array of animals and their characteristics

Outputs:

1. (*cell*) A Mx6 cell array of eligible animals, sorted and place into teams

Topics: (*subscripted masking*), (*sorting*)

Background:

Oh no! In the midst of game night, your brain has become overloaded with game rules and strategies. At full capacity, your brain simply cannot function anymore, and somehow everyone and everything around you has turned into animals. How preposterous! Curiously enough, your haywire brain was able to come up with a foolproof way to determine which of your hallucinated animals should be on your team in order to guarantee a win for the next game. You decide to use MATLAB to avoid using your brain.

Function Description:

You are given a Nx5 cell array that has columns with the following information in order: animal name (*string*), most recent game score (*double*), vibe check (*logical*), lucky number (*double*), and brain size (*double*). You want a cell array of each participating animal with their stats and their team. Your foolproof method of determining the players is as follows:

First, you must eliminate the non-eligible animals by deleting the entire row using the following criteria:

1) If the animal doesn't pass the vibe check (false), then it must not be eligible.
2) 6 is your unlucky number. If the animal's lucky number contains the digit 6, then they are definitely not eligible.
3) If the animal's brain size is smaller than 6.2 or larger than 9.7, it is most likely not eligible and cannot participate in the game.

```
originalArr → [{'donkey'}  {[54]}  {true}  {[50]}  {[8.6]}
               {'monkey'}  {[4]}   {false} {[869]} {[12]}
               {'hyena'}   {[156]} {true}  {[78]}  {[7.4]}
               {'piranha'} {[0]}   {true}  {[553]} {[8.9]}
               {'zebra'}   {[-6]}  {true}  {[9]}   {[6.8]}];

potentialPlayers → [{'donkey'}  {[54]}  {true}  {[50]}  {[8.6]}
                    {'hyena'}   {[156]} {true}  {[78]}  {[7.4]}
                    {'piranha'} {[0]}   {true}  {[553]} {[8.9]}
                    {'zebra'}   {[-6]}  {true}  {[9]}   {[6.8]}];
```

(*continued…*)

You must then find the animals that you want on your team. First, rank the remaining animals in *descending* order based on the most recent game score. If you have an odd amount of total players (all the eligible animals in the array plus 1 for yourself), eliminate the animal with the lowest score. The top half of the ranked animals should be on your team, and the bottom half should be on the opposing team. The number of players on your team should always be one less than the number of players on the opposing team in order to have space for you. Finally, add a new column that labels the team of each animal. Animal's on your team should be labeled as `'My Team!'` and the others should be labeled as `'Loser'`.

```
potentialPlayers → [{'donkey'}  {[54]}  {true}  {[50]}  {[8.6]}
                    {'hyena'}   {[156]} {true}  {[78]}  {[7.4]}
                    {'piranha'} {[0]}   {true}  {[553]} {[8.9]}
                    {'zebra'}   {[-6]}  {true}  {[9]}   {[6.8]}];

finalTeams → [{'hyena'}   {[156]} {true}  {[78]}  {[7.4]} {'My Team!'}
              {'donkey'}  {[54]}  {true}  {[50]}  {[8.6]} {'Loser'}
              {'piranha'} {[0]}   {true}  {[553]} {[8.9]} {'Loser'}];
```

Example:

```
      animalArr = [{'donkey'}  {[54]}  {true}  {[50]}  {[8.6]}
                   {'monkey'}  {[4]}   {false} {[869]} {[12]}
                   {'hyena'}   {[156]} {true}  {[78]}  {[7.4]}
                   {'piranha'} {[0]}   {true}  {[553]} {[8.9]}
                   {'zebra'}   {[-6]}  {true}  {[9]}   {[6.8]}];
      animalTeams = animalOverload(animalArr);
      animalTeams  →  [{'hyena'}     {[156]}  {true}   {[78]}   {[7.4]}  {'My
Team!'}
                      {'donkey'}  {[54]}  {true}  {[50]}  {[8.6]} {'Loser'}
                      {'piranha'} {[0]}   {true}  {[553]} {[8.9]} {'Loser'}];
```

Notes:

- You are guaranteed at least two players per team (including yourself) and thus your output array will have at least three rows.
- There will be no ties in most recent game scores.
- The columns will always be animal name (*string*), most recent game score (*double*), vibe check (*logical*), lucky number (*double*), and brain size (*double*) in that order.

Function Name: `checkers`

Inputs:

1. (*cell*) An 8x8 cell array describing the state of the checkerboard

Outputs:

1. (*cell*) An Nx1 column vector containing all possible moves.

Topics: (*string formatting*), (*consecutive conditionals*)

Background:

Game night is heating up! Your friend challenged you to a game of checkers, but he has changed the rules in his favour. In order to play a fair game, he has allowed you to use MATLAB to help you calculate all possible moves according to the new rules.

Function Description:

Write a function that takes in a single input, an 8x8 cell-array that contains information about all squares on the checkerboard, and gives out all possible moves as an Nx1 cell array.

Each square is a cell containing one of the three possible character vectors:

1. `'blue'` for a blue piece
2. `'red'` for a red piece
3. `'avail'` if the square is not occupied by any piece.

Each position on the board is represented by its row and column number. The topmost row is Row 1, while the bottommost row is Row 8. Similarly, the leftmost column is Column 1 and the rightmost column is Column 8.

You are playing with the blue pieces which means that forward means moving up in the array. The rules of the game are as follows:

1. Pieces can only advance diagonally forward, in forward-left or forward-right directions.
2. A piece can 'move' diagonally one step to occupy an empty square or diagonally ‘jump' over an opponent's piece to capture it.
3. A piece can only capture pieces of the opposite color and can only capture one piece in a 'jump'. (It cannot jump over 2 or more pieces)
4. In each turn, a player can either make a 'move' to occupy an empty square or 'jump' over the opponent’s piece. (Players are NOT forced to jump)
5. In each turn, a player can either move or jump once. (Multiple jumps or moves are not allowed).

(*continued…*)

It is your turn in the game. Your function should find all the possible 'moves' or 'jumps'. And make a string for each in the following format:

- If the checker makes a 'move':

```
'checker at (<initial_row>, <initial_column>) moves to (<final_row>, <final_column>)'
```

- If the checker makes a 'jump':

```
'checker at (<initial_row>, <initial_column>) jumps to (<final_row>, <final_column>)'
```

Store these strings in a Nx1 cell array and sort them. Output the sorted array as the final output of the function.

Example:

```
board =

{'avail'}  {'avail'}  {'avail'}  {'avail'}  {'avail'}  {'avail'}  {'red'  }  {'avail'}
{'avail'}  {'blue' }  {'avail'}  {'blue' }  {'avail'}  {'avail'}  {'avail'}  {'avail'}
{'avail'}  {'avail'}  {'avail'}  {'avail'}  {'avail'}  {'avail'}  {'red'  }  {'avail'}
{'avail'}  {'red'  }  {'avail'}  {'avail'}  {'avail'}  {'blue' }  {'avail'}  {'blue' }
{'blue' }  {'avail'}  {'red'  }  {'avail'}  {'red'  }  {'avail'}  {'avail'}  {'avail'}
{'avail'}  {'blue' }  {'avail'}  {'avail'}  {'avail'}  {'blue' }  {'avail'}  {'red'  }
{'avail'}  {'avail'}  {'avail'}  {'avail'}  {'avail'}  {'avail'}  {'avail'}  {'avail'}
{'avail'}  {'avail'}  {'avail'}  {'avail'}  {'avail'}  {'avail'}  {'avail'}  {'avail'}

moves = checkers(board)

moves → {'checker at (2, 2) moves to (1, 1)'}
        {'checker at (2, 2) moves to (1, 3)'}
        {'checker at (2, 4) moves to (1, 3)'}
        {'checker at (2, 4) moves to (1, 5)'}
        {'checker at (4, 6) jumps to (2, 8)'}
        {'checker at (4, 6) moves to (3, 5)'}
        {'checker at (4, 8) jumps to (2, 6)'}
        {'checker at (5, 1) jumps to (3, 3)'}
        {'checker at (6, 2) jumps to (4, 4)'}
        {'checker at (6, 6) jumps to (4, 4)'}
        {'checker at (6, 6) moves to (5, 7)'}
```

Notes:

- There are no "king pieces". (i.e no piece can move backwards)
- There is guaranteed to be at least one valid move.