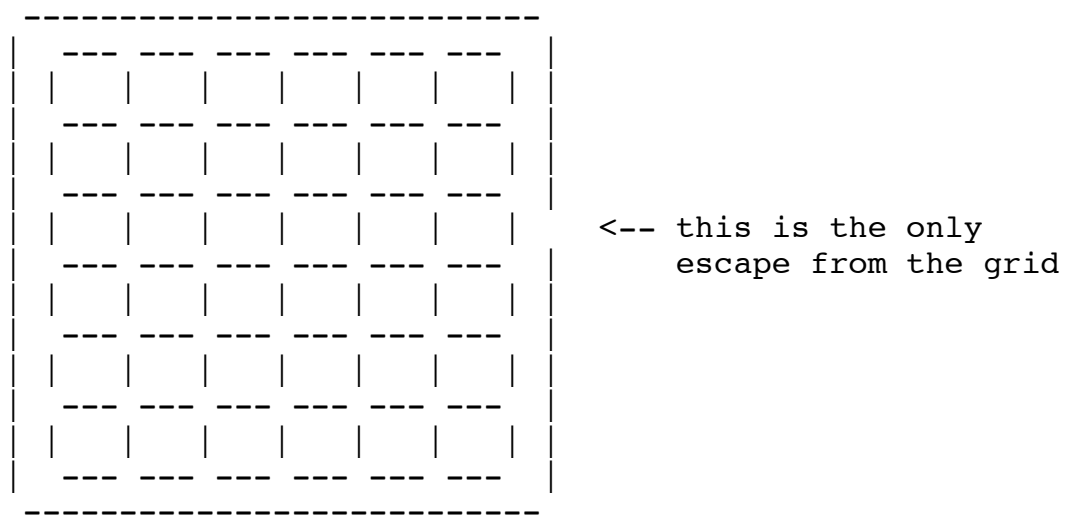
CPSC 312
Homework Assignment #3
Due no later than 6:00pm Sunday, October 27, 2013

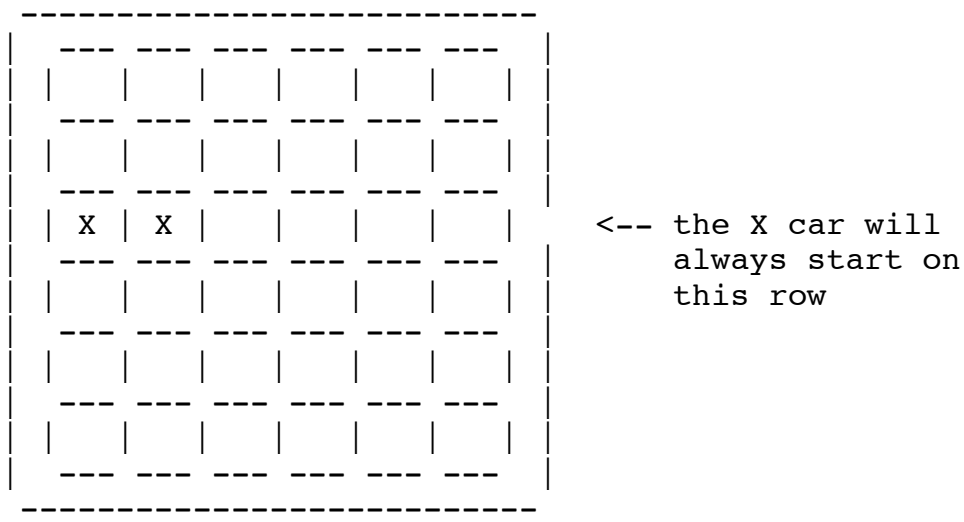
The puzzle called "Rush Hour" (c. ThinkFun, Inc., www.thinkfun.com) begins with a six-by-six grid of squares. Little plastic cars and trucks are distributed across the grid in such a way as to prevent one special car from moving off the grid, thereby escaping the traffic jam. The initial configuration of cars and trucks is preordained by a set of cards, each containing a different starting pattern. A player draws a card from the deck of cards, sets up the cars and trucks according to the pattern, and then tries to move the special car off the grid by sliding the cars and trucks up and down or left and right, depending on the starting orientation of the cars and trucks.

The grid looks something like this:

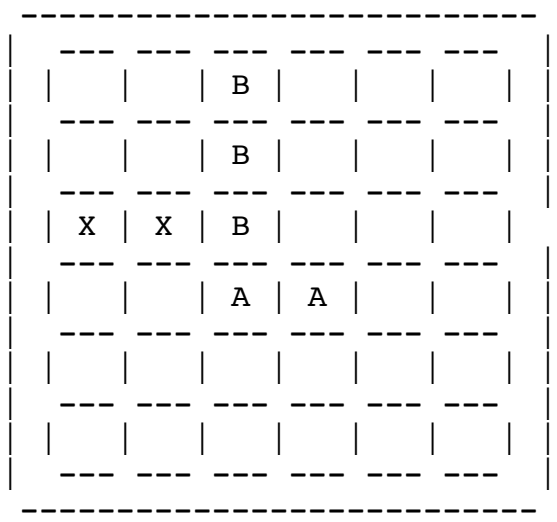


Cars occupy two contiguous squares, while trucks occupy three contiguous squares. All vehicles are oriented horizontally or vertically, never on the diagonal. Vehicles that are oriented horizontally may only move horizontally; vehicles that are oriented vertically may only move vertically. Vehicles may not be lifted from the grid, and there is never any diagonal movement. Vehicles may not be partially on the grid and partially off the grid.

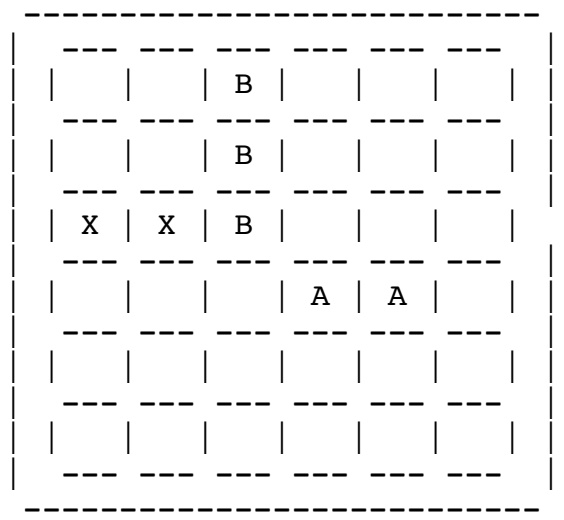
The special car is called the X car. We'll represent it on the grid as two contiguous squares with the letter X in them. (We'll represent all other cars and trucks with different letters of the alphabet, but the special car will always be labeled with the letter X.) The X car is always oriented horizontally on the third row from the top; otherwise, the X car could never escape. If it were the only car on the grid, it might look like this (although it could start anywhere on that third row):



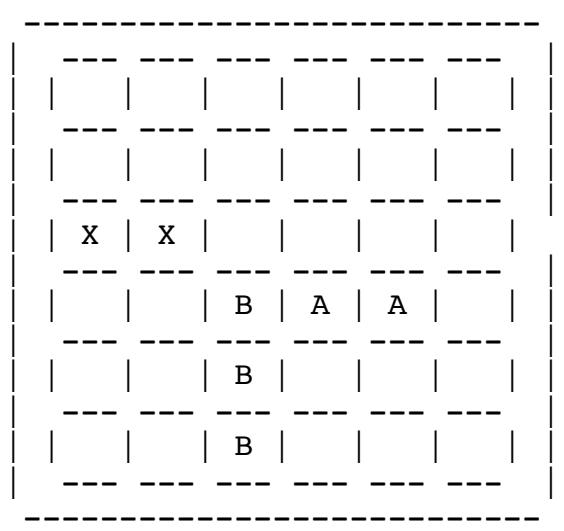
Let's put a blocking car and a blocking truck on the grid now too. We'll label the car as A and the truck as B:



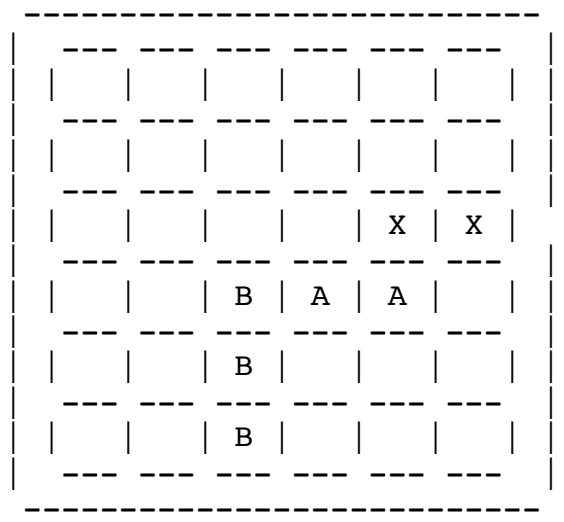
Car A is oriented horizontally on the fourth row, and it can slide only left or right. Truck B is oriented vertically in the third column from the left, and it can slide only up or down. If this configuration were the starting state for a puzzle, it would be pretty simple for us to solve it. To slide the X car all the way to the right and off the grid, we'd have to move the B truck out of the way. But to move the B truck, we'll first have to move the A car out of the B truck's path. So the first solution step might be to move the A car to the right by one square:



(Note that moving the A car further to the right works, as does moving it two squares to the left.) The next step would be to slide the B truck down three squares:

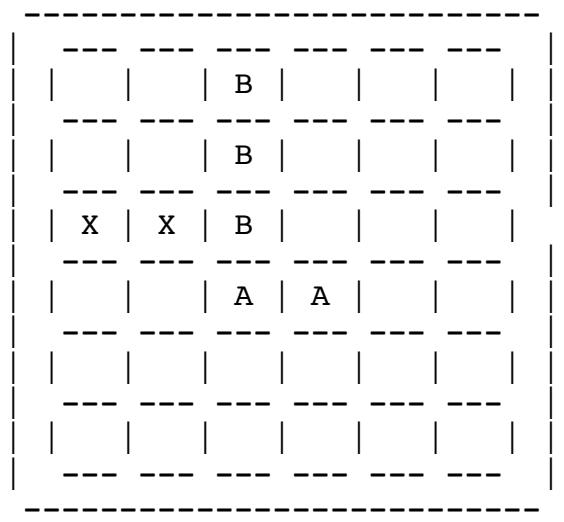


Now it's possible to move the X car all the way to the right and off the grid. It's not necessary to move it off the grid though; it's sufficient to move the X car to cover the rightmost two squares of the third row from the top:



Now we've solved the puzzle. Let's take another look at solving this "Rush Hour" puzzle, but this time from the perspective of a state-space search. The initial state in the state space may be any configuration of cars and trucks such that the X car is always oriented horizontally on the third row from the top. The goal state is any configuration of cars and trucks that can be obtained through correct application of the operators, such that the X car covers the rightmost two squares of the third row from the top. The operators are simply these: move a horizontally-oriented vehicle to the left, move a horizontally-oriented vehicle to the right, move a vertically-oriented vehicle up, and move a vertically-oriented vehicle down.

Your assignment is to use Haskell to write a program called `rush_hour` which employs a depth-first state space search to solve a "Rush Hour" puzzle with any legal initial state. Your `rush_hour` program expects one argument. That argument is a description of the initial-state as a list of six sublists, each list containing six elements. For this initial configuration:



the list of lists passed to `rush_hour` would look like this:

```
["--B---","--B---","XXB---","--AA--","------","------"]
```

The first sublist corresponds to the top row, the second sublist corresponds to the next row down, and so on. If this list were formatted nicely, it would look more like the grid above:

```
 ["--B---",
  "--B---",
  "XXB---",
  "--AA--",
  "------",
  "------"]
```

We don't need to pass the goal state, as it will always be the same regardless of the initial state. The goal state will always be a legally-obtained configuration of cars and trucks with the X car on the rightmost two squares of the third row.

When (if?) the goal has been reached, `rush_hour` should terminate and return a list of successive states that constitutes the path from the initial state to the goal state. So, if one of your TAs invokes `rush_hour` in this way:

```
> rush_hour ["--B---","--B---","XXB---","--AA--","------",
"------"]
```

your program should return something like this (although not necessarily exactly this) if you knew how to get Haskell to format it nicely:

```
[["--B---",
  "--B---",
  "XXB---",
  "--AA--",
  "------",
  "------"],
 ["--B---",
  "--B---",
  "XXB---",
  "---AA-",
  "------",
  "------"],
 ["------",
  "--B---",
  "XXB---",
  "--BAA-",
  "------",
  "------"],
```

```
 ["------",
  "------",
  "XXB---",
  "--BAA-",
  "--B---",
  "------"],
 ["------",
  "------",
  "XX----",
  "--BAA-",
  "--B---",
  "--B---"],
 ["------",
  "------",
  "-XX---",
  "--BAA-",
  "--B---",
  "--B---"],
 ["------",
  "------",
  "--XX--",
  "--BAA-",
  "--B---",
  "--B---"],
 ["------",
  "------",
  "---XX-",
  "--BAA-",
  "--B---",
  "--B---"],
 ["------",
  "------",
  "----XX",
  "--BAA-",
  "--B---",
  "--B---"]]
```

(Note that the pretty-printed list above could be fairly unreadable when it's not displayed nicely:

```
[["--B---","--B---","XXB---","--AA--","------","------"],["--B---","--
B---","XXB---","---AA-","------","------"],["------","--B---","XXB---
","--BAA-","------","------"],["------","------","XXB---","--BAA-","--
B---","------"],["------","------","XX----","--BAA-","--B---","--B---
"],["------","------","-XX---","--BAA-","--B---","--B---"],["------","-
------","--XX--","--BAA-","--B---","--B---"],["------","------","---XX-
","--BAA-","--B---","--B---"],["------","------","----XX","--BAA-","--
B---","--B---"]]
```

Now back to the original discussion.)

The result above reflects the fact that you'll be using operators that move vehicles only one square in any direction at a time. Do not employ additional operators that move a vehicle two, three, or four squares at a time. Keep it simple. Your program may find a different path than the one shown above because your program may apply its operators in a different order than the hypothetical program above does. That's ok with us. If the goal state can't be reached from the initial state, `rush_hour` should just return `[]`.

If you wish, you can convert my representation for the "Rush Hour" puzzle into any other representation of your choice if you think that would make your life easier. I make no claim that the representation I'm using above for passing the start state to your program is necessarily a great representation to use in solving the problem. It just happens to be convenient for representing the start state. So like I said, if you think things might go easier for you if you use a different representation, please feel free to do so. However, you still have to deal with my representation as input to your function as specified above. Furthermore, you'll need to convert your representation to my representation in order to return the path from the initial state to the goal state in the correct (i.e., my) format.

As you work on this assignment, you'll gain lots of experience with the functional programming style in the context of a larger, more complex program as opposed to the little four- and five-line programs you've been writing, and you'll see that you really can use a few simple list processing operations to do interesting things. You'll also gain experience with mapping representations of real-world (but simple) problem domains onto list structures. And you'll also see how with an appropriate representation for state and some corresponding operators, you can use search to solve problems that people find challenging.

In your program we'll want to see lots of well-abstracted, well-named, cohesive, and readable functions. Appropriate comments will make the TAs who mark your program happy, and happy markers are generous markers. Feel free to borrow or adapt the peg-puzzle program that we introduced in class, but it's fine if you don't. If you want to work on this with another student in CPSC 312, that's fine too, as the collaborative learning policy from assignment 1 still applies. In fact, I encourage you to use this opportunity to find someone to work with since you **must** work with another person on the upcoming project.
If you choose to work with another person on the "Rush Hour" puzzle, keep in mind that you may work with only one other person (not two, or three, or more).

Additional Notes:

- The X car (the one to be moved off the grid) will always be indicated by the letter X, and will always be somewhere on the third row.

- The number of other cars and trucks on the grid is limited only by the available space on the grid. Do not assume that there will be only one other car and one other truck just because that's what's shown in the example.

- The other cars and trucks will be labeled with single letters from the alphabet, but don't assume that those letters will always be taken from the beginning of the alphabet.

- Extra credit is possible for this assignment: come up with some heuristic that improves the performance of your "Rush Hour" program, include an explanation for the TAs about what your heuristic is and how it works, and submit two versions of your program (make it clear in your file that you're trying for the extra credit). One version would be the brute-force, depth-first search approach, and the other version would be the heuristic approach. The TAs will compare the performance of your two versions, and if the heuristic version proves to be able to solve more challenging problems in reasonable time, you'll get credit for having done 1.5 homework assignments.

- Googling "rush hour heuristic" does turn up some ideas. Google might also turn up some code. I've seen it. In fact, most of the "Rush Hour" programs ever written in Haskell were written for previous offerings of this course, and we have copies of them. I strongly advise against looking at any code you might find, because if it ends up in your submission, what I've intended to be a fun course for you will turn into something remarkably un-fun for you in a hurry.

- For handin purposes, this is 'assign3' and the course is still 'cs312'.