MIT
6.031: Software Construction
Max Goldman
revised Friday 30th November, 2018, 14:45

Quiz 2 (December 3, 2018)

Your name: _____

Your Kerberos username: _____

You have 50 minutes to complete this quiz. It contains 12 pages (including this page) for a total of 100 points.

The quiz is closed-book and closed-notes, but you are allowed one two-sided page of notes.

Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Before you begin, write your Kerberos username on the top of every page.

Please write neatly. **No credit will be given if we cannot read what you write.**

For questions which require you to choose your answer(s) from a list, do so clearly and unambiguously by circling the letter(s) or entire answer(s). Do not use check marks, underlines, or other annotations – they will not be graded.

Good luck!

Problem	Points
1: Recursive ADTs I	22
2: Recursive ADTs II	22
3: Equality & Streams	20
4: Thread Safety	18
5: Queues & Locks	18
Total	100

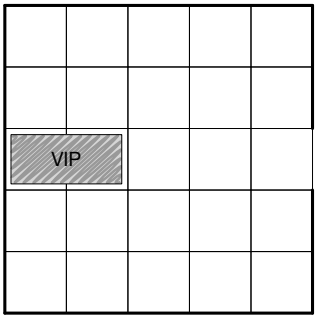
This quiz uses the same abstract data type as Quiz 1.

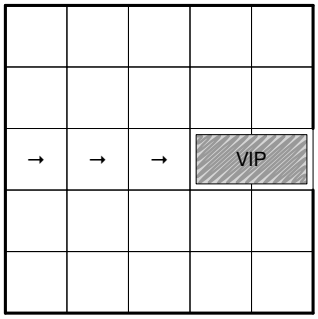
Mini Rush Hour is a sliding block puzzle played on a 5×5 grid of cells numbered 1 to 25:

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

← The board has an *exit* to the right of cell 15.

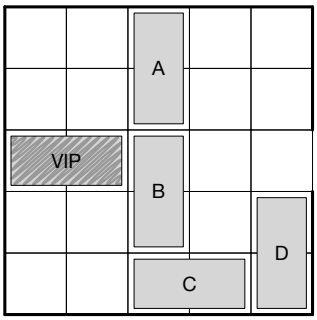
Cars are 1-cell × 2-cells pieces placed horizontally or vertically on the grid. Every Mini Rush Hour puzzle starts with the *VIP car* on cells 11 and 12. The goal is to move the VIP car to the right so it reaches cells 14 and 15, at which point it can exit the board:





← Winning!

Blocking the VIP car's way are other cars, for example:



While they are labeled in this diagram, cars other than the VIP car are not differentiated except by their placement on the board, they have no distinguishing characteristics. All cars in the game move only along their axis of orientation: horizontal cars move left and right, vertical cars move up and down. Cars never overlap. They can only move through and stop on empty cells.

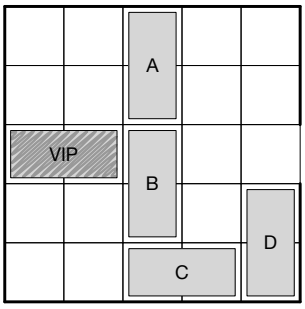
The player wins by making a series of moves that allow the VIP car to reach the exit. The following page shows a solution to this example puzzle, plus another example.

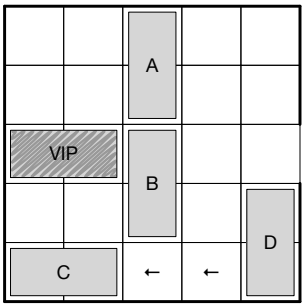
The problems in this quiz use `Direction` and `Car`, defined on page 10; and refer to the code for `RushHourPuzzle` on page 11 and `RushHourGame` on page 12; all of which you may detach.

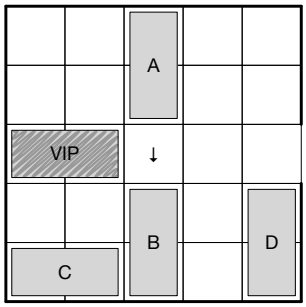
Notes which are not relevant to this quiz:

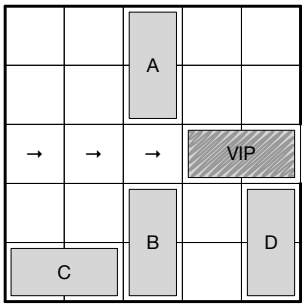
- *Rush Hour*® is a trademark of ThinkFun, Inc. The full game uses varying-size cars on a larger board.
- On arbitrarily large grids, *Rush Hour* and *Size-2 Rush Hour* are PSPACE-complete (Flake & Baum, 2002; Tromp & Cilibrasi, 2005).

Solving the example puzzle in 3 moves:

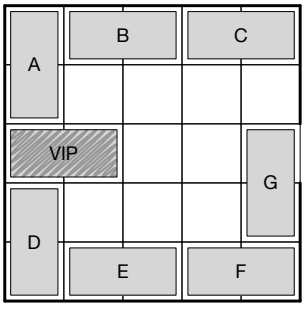


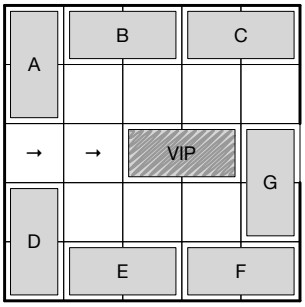
1. 

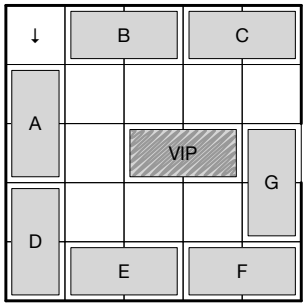
2. 

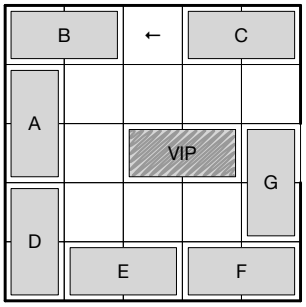
3. 

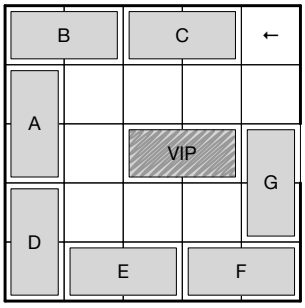
Solving another example in 6 moves:

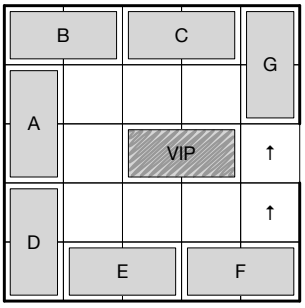


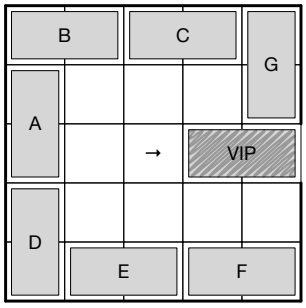
1. 

2. 

3. 

4. 

5. 

6. 

Problem 1 (Recursive ADTs I) (**22 points**).
Immutable `RushHourPuzzle` represents the starting layout of a Mini Rush Hour puzzle.

Let's implement this type as a recursive datatype with the three variants defined on page 11. Remember! Every Mini Rush Hour puzzle has the VIP car on cells 11 and 12.

(a) Write the recursive datatype definition for **`RushHourPuzzle`** using these variants:

(b) Write a complete `checkRep` for **`WithHorizontal`** by writing several "`assert ...;`" statements:

```
private void checkRep() {
    assert
}
```

(c) And write an abstraction function for **`WithHorizontal`** that works with your `checkRep`:

(d) Based on the provided code, is **`WithHorizontal`** safe from rep exposure? Circle YES and write the argument, or circle NO and explain why not:

YES NO

Problem 2 (Recursive ADTs II) (**22 points**).
Complete each method below by **returning a single expression**. Remember, every puzzle has the VIP car.

(a) Complete `occupied(..)` for the **`Minimal`** variant:

```
@Override public boolean occupied(int cell) {
    return
}
```

(b) Complete `occupied(..)` for the **`WithHorizontal`** variant:

```
@Override public boolean occupied(int cell) {
    return
}
```

(c) Using the static `Stream` helper methods on page 10, complete `cars()` for the **`Minimal`** variant:

```
@Override public Stream<Car> cars() {
    return
}
```

(d) Again using the static `Stream` helpers on page 10, complete `cars()` for the **`WithHorizontal`** variant:

```
@Override public Stream<Car> cars() {
    return
}
```

Problem 3 (Equality & Streams) (**20 points**)**.**
Ben is thinking about equality for `RushHourPuzzle` instances. Remember! Every Mini Rush Hour puzzle has the VIP car on cells 11 and 12.

(a) Write two **different** `RushHourPuzzle` reps that represent the **same** Mini Rush Hour puzzle. Use as few `RushHourPuzzle`-type objects as possible. As examples:

- `Minimal()` and `Minimal()` are the same rep, representing the same puzzle
- `WithHorizontal(1, Minimal())` and `WithHorizontal(2, Minimal())` are different reps, representing different puzzles

(b) Alyssa doesn't like having multiple representations for the same puzzle, so she plans to make more reps illegal. Write an additional statement to include in the **`WithHorizontal`** rep invariant to achieve this goal.

(c) Ben is going to implement `equals` by comparing abstract values, not concrete reps.

He's super jazzed about functional programming with streams, so he wants to write the `sameValue(..)` helper for the **`Minimal`** variant using `map`, `filter`, and `reduce`. Fill in a correct implementation below, returning `true` if and only if `that` is also the minimal Mini Rush Hour puzzle based on its `cars()` stream.

Do **not** use an identity function or a function that returns a constant – those aren't *jazzy* enough.

```
private boolean sameValue(RushHourPuzzle that) {

    return that.cars().map( _____ )

                      .filter ( _____ )

                      .reduce ( _____ ,

                                _____ )

                      == _____ ;
}
```

Use `reduce` : `Stream<T>`$\times$`T`$\times$`(T`$\times$`T`$\rightarrow$`T)`$\rightarrow$`T` by providing an identity value and an accumulator function.
Hint: what cars are in the minimal puzzle, and what cars are in a puzzle that is not the minimal puzzle?

Problem 4 (Thread Safety) (**18 points**).
Mutable `RushHourGame` (on page 12) allows the client to play a Mini Rush Hour game, starting from an initial `RushHourPuzzle` and making a series of moves with the `move(..)` method.

(a) Based on the provided code, circle one or more of CONFINED, IMMUTABLE, and/or THREADSAFE TYPE if they apply; or circle NONE otherwise:

The `vipCar` reference:	CONFINED	IMMUTABLE	THREADSAFE TYPE	NONE
Possible values of `vipCar`:	CONFINED	IMMUTABLE	THREADSAFE TYPE	NONE
The `carsMap` reference:	CONFINED	IMMUTABLE	THREADSAFE TYPE	NONE
The value of `carsMap`:	CONFINED	IMMUTABLE	THREADSAFE TYPE	NONE

(b) Suppose two threads **A** and **B** call `move` on the same `RushHourGame` concurrently, starting from the game state below.

Choose **two different calls**, neither of which moves the VIP car, and explain **how the steps of `move` can interleave** so that both calls return a value, but the rep is invalid and broken after they complete. Say **how the rep is broken**.

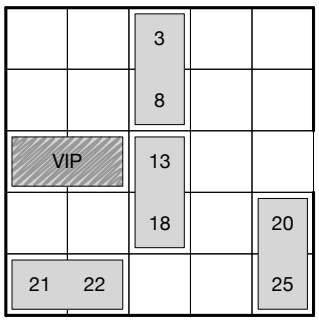
Use line numbers from page 12, and circle all line numbers (not cell numbers!). For example, "thread A runs line (*N*), then thread B runs line (*M*), which causes *problem P*," etc.

Since we don't know its details, treat the computation of `newTopLeft` as a single indivisible step.

Choose the two calls:

A: **B**:

Explain the steps:



How is the rep broken?

Problem 5 (Queues & Locks) (**18 points**).
Charlie wants to implement best-score tracking for Mini Rush Hour using the **producer-consumer pattern**, with many clients submitting scores to a single tracker.

(a) Clients will have access to two threadsafe blocking unbounded-size first-in-first-out queues:

```
BlockingQueue<RushHourPuzzle> puzzleSolved
BlockingQueue<Integer> movesToSolve
```

To record a score (the number of moves `m` to solve a puzzle `p`), a client runs:

```
puzzleSolved.put(p);
movesToSolve.put(m);
```

And the best-score tracker is running:

```
while (true) {
    RushHourPuzzle p = puzzleSolved.take();
    int m = movesToSolve.take();
    // if m is the best score on p, record it
}
```

Explain a problem with this implementation and give an example:

Charlie changes the code to add synchronization on a new object shared by all clients and the tracker:

```
Object movesLock
```

To record a score, a client runs:

```
puzzleSolved.put(p);
synchronized (movesLock) {
    movesToSolve.put(m);
}
```

And the best-score tracker is running:

```
while (true) {
    RushHourPuzzle p = puzzleSolved.take();
    int m;
    synchronized (movesLock) {
        m = movesToSolve.take();
    }
    // if m is the best score on p, record it
}
```

(b) Does this change fix the problem from part **(a)**? Circle: YES NO

(c) What new, different kind of problem does this change introduce? Explain and give an example:

You may detach these pages. Write your username at the top, and hand in all pages when you leave.

SFB
ETU
RFC

`static <T> Stream<T> empty()`
Returns an empty stream.

`static <T> Stream<T> of(T t)`
Returns a stream containing a single element.

`static <T> Stream<T> of(T... values)`
Returns a stream whose elements are the specified values.

`static <T> Stream<T> iterate(T seed, Function<T,T> f)`
Returns an infinite stream produced by iterative application of a function `f` to an initial element `seed`, producing a `Stream` consisting of `seed`, `f(seed)`, `f(f(seed))`, etc.

`static <T> Stream<T> generate(Supplier<T> s)`
Returns an infinite stream where each element is generated by the provided `Supplier`.

`static <T> Stream<T> concat(Stream<T> a, Stream<T> b)`
Creates a concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.

```
public enum Direction { LEFT, RIGHT, UP, DOWN }
```

```
/** Immutable car on a Mini Rush Hour board. */
public class Car {

    private final int topLeft;
    private final Direction direction;

    /**
     * Make a Mini Rush Hour car whose top (for a vertical car) or left (for horizontal)
     *   half is in cell topLeft.
     * @param topLeft  must be a valid top (for vertical) or left (for horizontal) cell
     * @param direction  must be DOWN for a vertical car or RIGHT for a horizontal car
     */
    public Car(int topLeft, Direction direction) {
        this.topLeft = topLeft;
        this.direction = direction;
    }

    /** @return the top/left cell of this car */
    public int topLeft() { return topLeft; }

    /** @return the bottom/right cell of this car */
    public int bottomRight() { return topLeft + (direction == Direction.RIGHT ? 1 : 5); }

    /** @return DOWN if this is a vertical car or RIGHT if this is a horizontal car */
    public Direction direction() { return direction; }

    ... other observers ...
}
```

```
/** Immutable starting layout of a Mini Rush Hour puzzle. */
public interface RushHourPuzzle {
    /**
     * @param cell  requires 1 <= cell <= 25
     * @return true iff that cell of this puzzle is occupied by a car
     */
    public boolean occupied(int cell);

    /**
     * @return a stream of all the unique cars, including the VIP, in this puzzle
     */
    public Stream<Car> cars();
}

class Minimal implements RushHourPuzzle {
    public Minimal() {
    }
    @Override public boolean occupied(int cell) { ... }
    @Override public Stream<Car> cars() { ... }
    ... other observers ...
}

class WithHorizontal implements RushHourPuzzle {

    private final int left;
    private final RushHourPuzzle rest;

    public WithHorizontal(int left, RushHourPuzzle rest) {
        this.left = left;
        this.rest = rest;
        checkRep();
    }
    private void checkRep() { ... }
    @Override public boolean occupied(int cell) { ... }
    @Override public Stream<Car> cars() { ... }
    ... other observers ...
}

class WithVertical implements RushHourPuzzle {

    private final int top;
    private final RushHourPuzzle rest;

    public WithVertical(int top, RushHourPuzzle rest) { ... }
    private void checkRep() { ... }
    @Override public boolean occupied(int cell) { ... }
    @Override public Stream<Car> cars() { ... }
    ... other observers ...
}
```

```
/** Mutable Mini Rush Hour game. */
public class RushHourGame {
    // the VIP car in its current position
    private Car vipCar;
    // all cars on the game board, indexed by both cells they occupy
    private final Map<Integer,Car> carsMap;

    /**
     * Make a new Mini Rush Hour game starting from the given puzzle layout.
     * @param starting  puzzle to play
     */
    public RushHourGame(RushHourPuzzle starting) {
        // find the VIP car
        this.vipCar = starting.cars().filter(car -> car.topLeft() == 11)
                                     .findFirst().orElseThrow();
        // populate the cars map
        this.carsMap = Collections.synchronizedMap(new HashMap<>());
        starting.cars().forEach(car -> carsMap.put(car.topLeft(), car));
        starting.cars().forEach(car -> carsMap.put(car.bottomRight(), car));
    }

    /**
     * Move the car that occupies cellNum as far as it can move in the given direction.
     * @param cellNum  indicates car to move, a car must currently occupy that cell
     * @param direction  direction to move
     * @return true if-and-only-if the VIP car is in the game-winning position
     */
    public boolean move(int cellNum, Direction direction) {
        // find the old Car in carsMap
        final Car oldCar = carsMap.get(cellNum);

        // determine how far it can move
        final int newTopLeft = ...

        // create a new Car in the destination position
        final Car newCar = new Car(newTopLeft, oldCar.direction());

        // remove old Car from carsMap, add new Car
        carsMap.remove(oldCar.topLeft());
        carsMap.remove(oldCar.bottomRight());
        carsMap.put(newCar.topLeft(), newCar);
        carsMap.put(newCar.bottomRight(), newCar);

        // if we moved the VIP car, reassign vipCar
        if (oldCar == vipCar) { vipCar = newCar; }

        // return true iff the game is won
        return vipCar.topLeft() == 14;
    }
}
```