Templates and examples

COMP 50

Fall 2013

This homework is due at 11:59PM on **Monday, October 28**. Submit your solutions in a single file using the COMP 50 Handin button on DrRacket; the homework is the `templates` homework.

All the problems should be done using the Beginning Student Language with list abbreviations.

The problems require only templates, test cases, and tables of examples, not code. While you are welcome to write code to see if it "works," *do not submit completed functions*.[1] Submit only templates, test cases, and tables of examples.

This homework has no finger exercises. Except for the tables of examples, it's all stuff you already know how to do. For instructions on the tables and examples, see the lab handout on templates and examples.

Data descriptions

2D-trees

A *town* is a structure:

```
(make-town name latitude longitude),
```

where `name` is a string and `latitude` and `longitude` are numbers in degrees.

A *2D-tree* is a data structure used to locate points in two-dimensional space. A 2D-tree is one of:

- A town
- A structure

  ```
  (make-v-boundary west longitude east),
  ```

  where
  - `longitude` is a number,
  - `west` is a 2D-tree in which every town has a longitude at most `longitude`, and
  - `east` is a 2D-tree in which every town has a longitude at least `longitude`
- A structure

  ```
  (make-h-boundary south latitude north),
  ```

  where
  - `latitude` is a number,
  - `south` is a 2D-tree in which every town has a latitude at most `latitude`, and
  - `north` is a 2D-tree in which every town has a latitude at least `latitude`

[1] Exception: If you do the karma problem, you may submit completed functions for that part only. But don't overwrite your templates!

Natural numbers

A *natural* (aka natural number) is one of:

- `0`
- `(add1 n)` where `n` is a natural

Racket knowledge: These two cases can be identified using the BSL functions `zero?` and `positive?`. The function `sub1` is also useful.

Mobiles

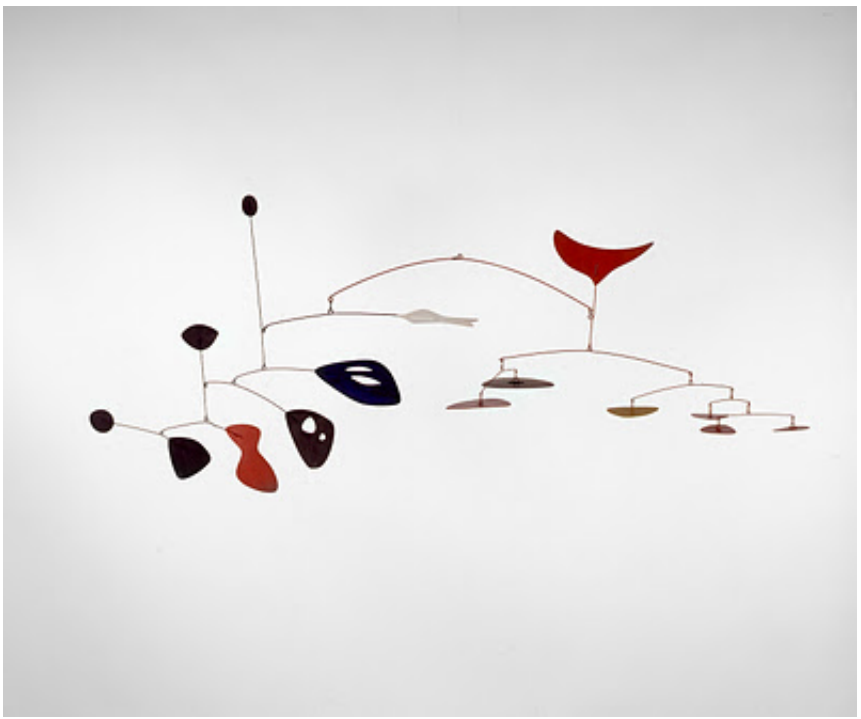
A *mobile* is one of the following:

- A number, which represents a toy with a weight in grams
- A structure

  ```
  (make-arm left right)
  ```

  where `left` and `right` are mobiles.

Here is a picture:



A *mobile-state* is one of the following:

- A number representing the mobile's weight in grams
- `false`

A mobile's state is a number only if every arm in the mobile is properly balanced. If any arm is unbalanced, the mobile's state is simply `false`. (We'll oversimplify and consider that an arm is balanced if both sides have equal weight.)

Problems to submit

1. *2D-tree search.* On pages 6 and 7 of this handout, you will find depictions of 20 sample queries trying to locate towns in a 2D-tree.

   - Each query shows a square map with a collection of cities.
   - All the queries operate on the same set of towns. This set is represented by a 2D-tree called `southern-tier`.
     The towns on the map are Apalachin, Binghamton, Chenango, Deposit, Endwell, Fenton, Glen Castle, Horseheads, Ithaca, Johnson City, Kirkwood, Lisle, and Maine. On the map, each town's name is abbreviated to a single letter.
   - Each query has a red X which marks the question point. The goal of the query is to find the town closest to this point.
   - Each map is divided in two by a horizontal or vertical line. The part of the map on the same side as the red X is the *near side*. The part of the map on the opposite side as the red X is the *far side*. On some maps, the far side has been obscured.

   You will use these queries to complete *part* of the design recipe for the following function.

```
;; closest-town : number number 2D-tree -> town
;; *efficiently* find the town that is closest to the given coordinates
(define (closest-town latitude longitude tree)
   ...)
```

   Complete the following parts of the problem:

   (a) *Copy the signature, purpose statement, and header* into your solution.

   (b) *Write three functional examples* for `closest-town`. Each example should be based on a query and should use `check-expect` to check the name of the town. If, for example, you choose Query 7, then please refer to the relevant latitude and longitude as `lat7` and `lon7`, as follows:

```
(check-expect (town-name (closest-town lat7 lon7 southern-tier)) ...)
```

   Please use the single-letter version of the town's name.

   (c) *Complete all 20 rows* of the following table:

Query	Closest Overall	Closest Near Side	Must Cross?	Closest Far Side
1	unknown	B	yes	unknown
...				

   The columns of the table are to be filled out as follows:

   - `Query` is the query number.
   - `Closest overall` one of the following:
     - A capital letter that stands for the name of the town closest to the red X, if that can be determined from the information shown in the query
     - The word "unknown," if the closest town can't be determined without examining a part of the map that has been obscured
   - `Closest Near Side` is the one-letter name of the town *on the near side* which is closest to the red X.
   - `Must Cross?` is either `yes` or `no`:
     - If finding the closest town requires that you cross the boundary and look at the far side of the map, then it is `yes`.
     - If the closest town overall can be found *without* looking at the far side of the map, then it is `no`.

     Looking at the boundary does *not* count as "crossing."
   - `Closest Far Side` is one of the following:
     - If the far side of the map has been obscured, then it is `unknown`.
     - If the far side of the map is visible, then it is the one-letter name of the town *on the far side* which is closest to the red X.

2. *Mobiles*. In this problem you'll carry out most of the steps of the design recipe for the following function:

```
;; mobile-state : mobile -> mobile-state
;; to examine the given mobile for balance and return its state
(define (mobile-state m)
  ...)
```

Assume that an arm weighs nothing.

Please complete the following steps:

(a) *Develop data examples* for mobiles.

(b) *Copy the signature, purpose statement, and header* for `mobile-state`.

(c) *Develop functional examples* for `mobile-state`.

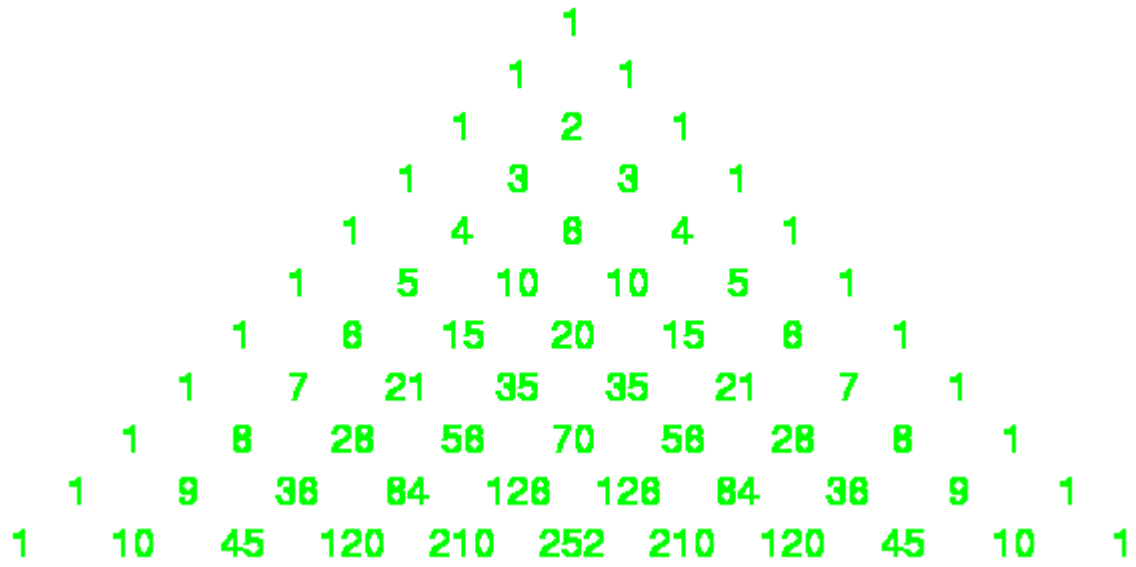
(d) *Develop a template* for `mobile-state`.

(e) For each case in your template that contains a naturally recursive call to `mobile-state`, *create a table* laid out as follows:

- The leftmost column shows what result is wanted, that is, what `mobile-state` is supposed to return.
- There is a column for the result of applying each relevant selector function.
- There is a column for the result of each natural recursion.

(f) *Complete the table* you created in part 2e. Make sure to include a row for each relevant functional example.

3. *Pascal's triangle* Here is an image of a number-theoretic construction called "Pascal's triangle:"



Pascal's triangle is built according to the following rules:

- Row 0 contains the single number 1
- Row `(add1 n)` is obtained by taking row `n` and adding it "pointwise" to a copy of itself that is shifted one position. For example:

		1	4	6	4	1
+	1	4	6	4	1	
=	1	5	10	10	5	1

Any given row of Pascal's triangle can be computed using a function. Here is a partial version, which uses the template for a function consuming a natural number:

```
;; pascal-nums : natural -> (listof number)
;; to produce the numbers that appear in the given row of Pascal's triangle
(define (pascal-nums n)
  (cond [(zero? n) ...]
        [(positive? n) (... (sub1 n) ... (pascal-nums (sub1 n)))]))
```

```
(check-expect (pascal-nums 0) '(1))
(check-expect (pascal-nums 1) '(1 1))
(check-expect (pascal-nums 2) '(1 2 1))
(check-expect (pascal-nums 3) '(1 3 3 1))
(check-expect (pascal-nums 4) '(1 4 6 4 1))
```

Here is a function that can be used for pointwise addition of two lists of numbers:

```
;; zip+ : (listof number) (listof number) -> (listof number)
;; to add the two given lists, which must be of equal lengths, pointwise
(define (zip+ ns ms)
  ...)

(check-expect (zip+ '(1 2 3) '(4 5 6)) '(5 7 9))
(check-error (zip+ '(1 2 3) '(4 5)))
```

Please complete the following parts of this problem:

(a) *Copy the signature, purpose statements, template, and tests* for `pascal-nums`.

(b) For the `positive?` case in `pascal-nums`, *create a table* laid out as follows:

- The leftmost column shows what result is wanted, that is, what `pascal-nums` is supposed to return.
- There is a column showing the value of `(sub1 n)`.
- There is a column showing the value of `(pascal-nums (sub1 n))`.

(c) *Complete the table* you created in part 3b. Make sure to include a row for each relevant functional example.

(d) *Copy the signature, purpose statements, header, and tests* for `zip+`.

(e) *Write two more functional examples* for `zip+`. Use `check-expect`.

(f) *Write a template* for `zip+`.

(g) For each case in your template that contains a naturally recursive call to `zip+`, *create a table* laid out as follows:

- The leftmost column shows what result is wanted, that is, what `zip+` is supposed to return.
- There is a column for the result of applying each relevant selector function.
- There is a column for the result of each natural recursion.

(h) *Complete the table* you created in part 3g. Make sure to include a row for each relevant functional example.

Karma problem

This homework has one karma problem:

A. *Design a function* that produces an image of Pascal's triangle of a given size.

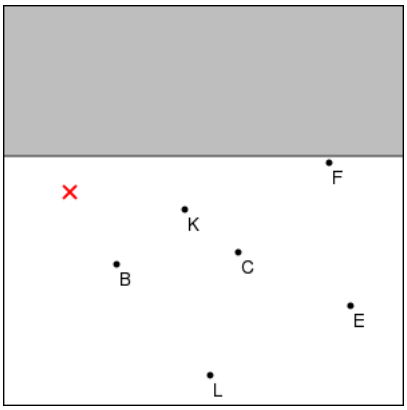
This is the *only* problem for which you are permitted to submit code.

Hint: to get the numbers to align nicely, overlay the image of each number onto a white rectangle of fixed size. Fiddle with the dimensions until you have a good-looking triangle.

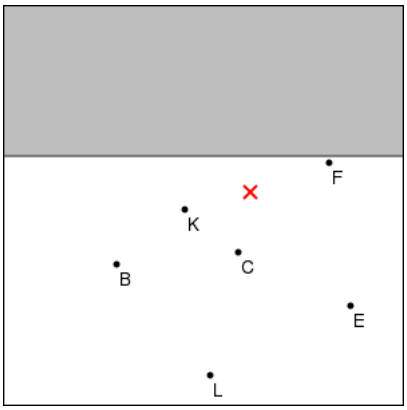
Acknowledgements

Viera Proulx kindly pointed out some possibilities inherent in mobiles.

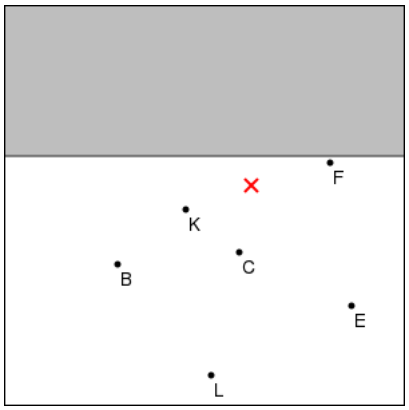
Queries



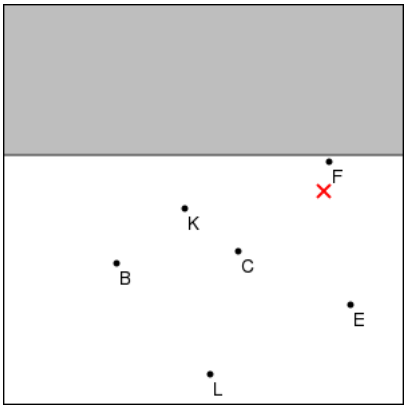
Query 1



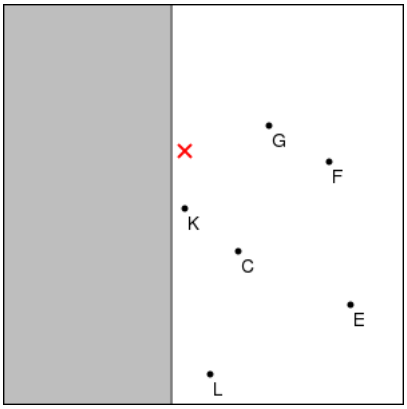
Query 2



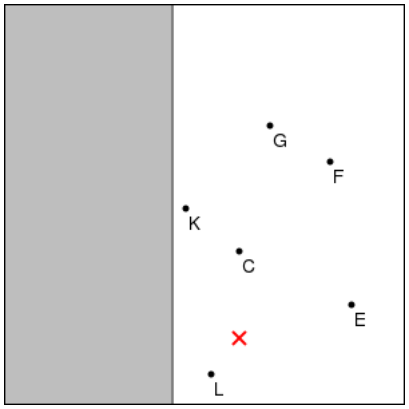
Query 3



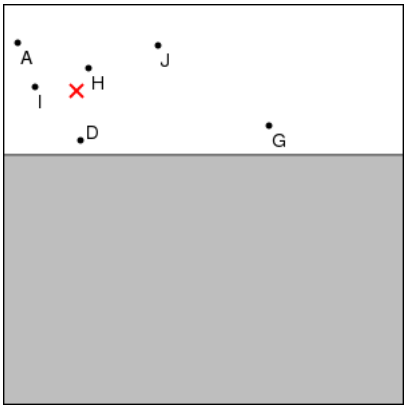
Query 4



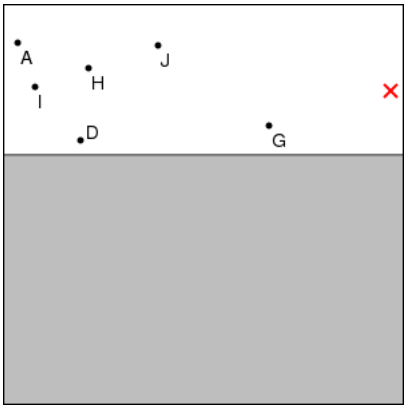
Query 5



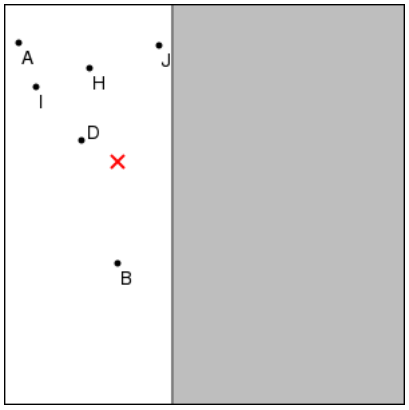
Query 6



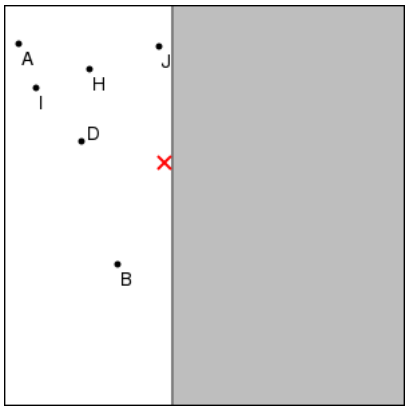
Query 7



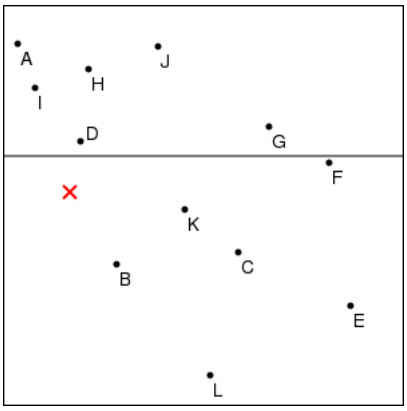
Query 8



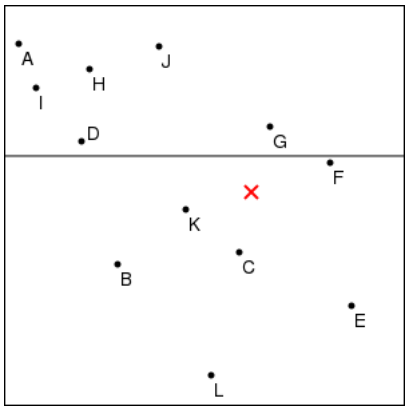
Query 9



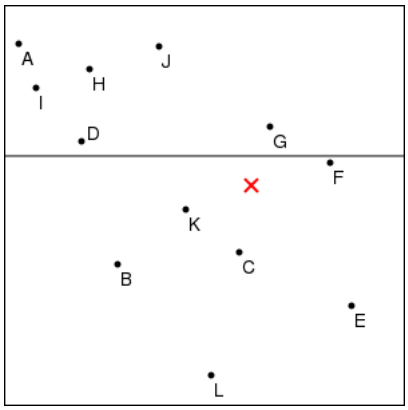
Query 10



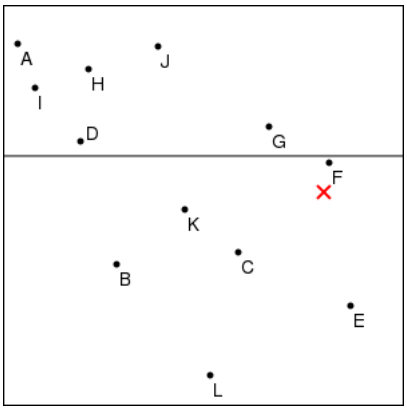
Query 11



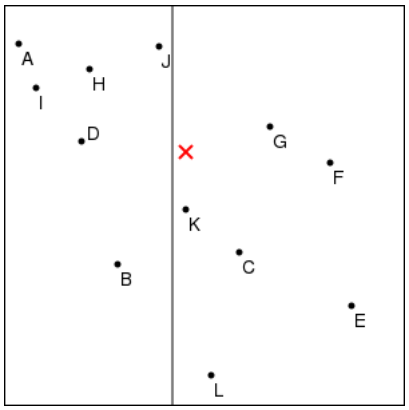
Query 12



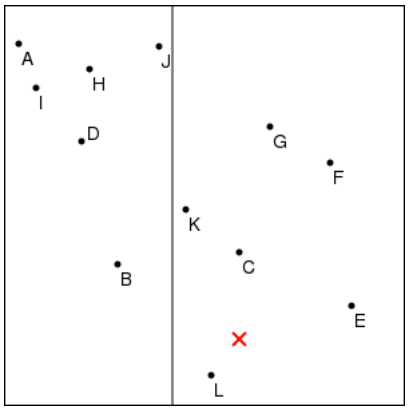
Query 13



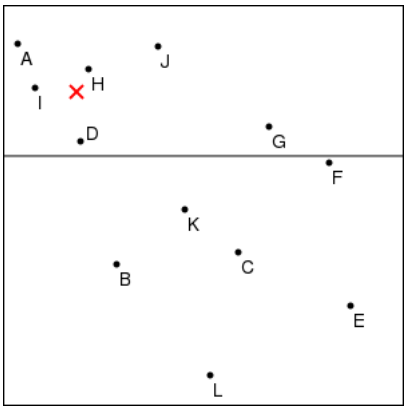
Query 14



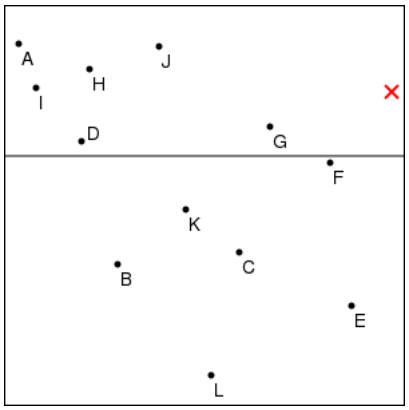
Query 15



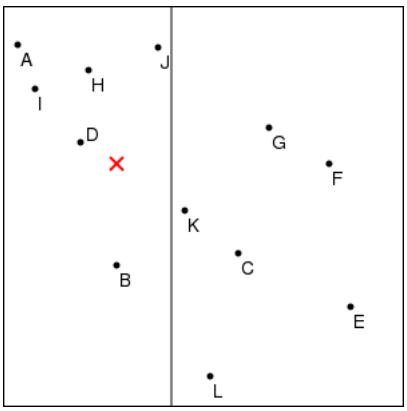
Query 16



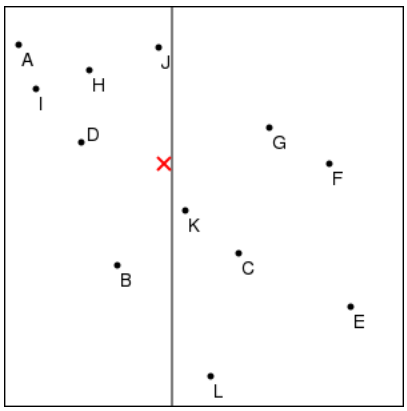
Query 17



Query 18



Query 19



Query 20