CS 135 Fall 2019
Becker, Clarke, Giesbrecht, Holtby, Lanctot, Reetz

Assignment: 10

Due:	Tuesday, December 3, 2019 11:59 pm
Language level:	Intermediate Student with Lambda
Files to submit:	`travel-map.rkt`, `countdown.rkt`
Warmup exercises:	HtDP 28.1.6, 28.2.1, 30.1.1, 31.3.1, 31.3.3, 31.3.6, and 32.3.1
Practice exercises:	HtDP 3.3.2, 28.1.4, 28.2.2, 28.2.3, 28.2.4, 31.3.7, 32.3.2, and 32.3.3

Notes:

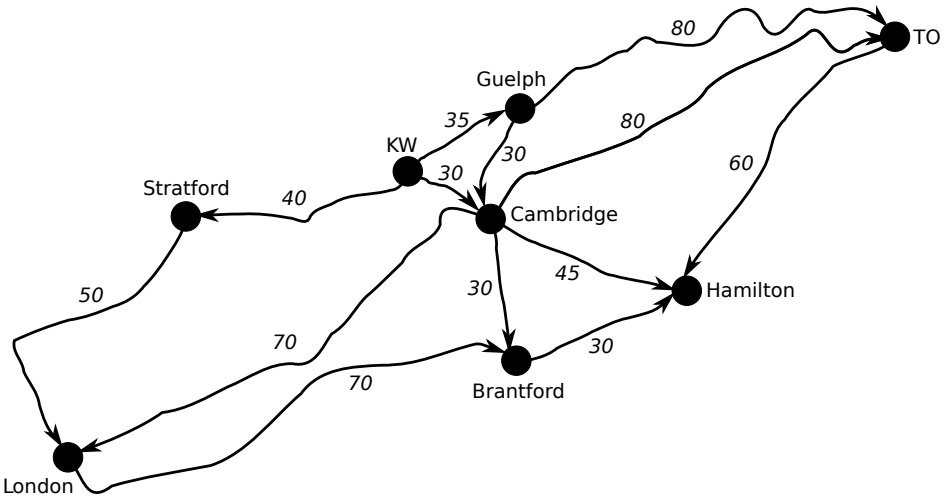
- **Make sure you read the OFFICIAL A10 post on Piazza** for the answers to frequently asked questions.
- You may use any functions and techniques discussed in class, unless otherwise noted.
- You may only define non-local constants and helper functions if they will be used to answer multiple parts of the same question. Otherwise, constants and helper functions should be defined locally. Local helper functions do not require purposes, contracts, examples or tests.
- You may use functions defined in earlier parts of a question to help write or test functions later in the same question.
- Solutions will be marked for both correctness [80%] and style [20%]. Follow the guidelines in the Style Guide, pages 1–15.

Here are the assignment questions you need to submit.

1. (20 %) Whenever we were using graphs so far, we attached values to vertices (vertex names in the form of a `Sym`) but we never gave edges values. By doing so, however, we can increase the usefulness of graphs significantly.

```
;; A Node is a Sym
;; A Map is a (listof (list Node (listof Neighbour)))
;;   requires: Map is directed and acyclic
;; A Neighbour is a (list Node Nat) where the number indicates the
;;   travel time (in minutes) to the neighbour.
```

   (a) For example, consider the map of Southern Ontario shown below. The values on each edge indicate travel times (in minutes) between nodes. With its help you could calculate your estimated travel time between different cities. Write a function `travel-time`. This function consumes an origin and a destination in the form of two `Sym` as well as a `Map`. It then produces the travel time between origin and destination; if no path exists, it produces false. `travel-time` requires that both origin and destination are on the map.



(If there are multiple paths between the origin and destination, any correct travel time is acceptable.)

```
(define southern-ontario
  '((Brantford ((Hamilton 30)))
    (Cambridge ((Brantford 30) (Hamilton 45) (London 70) (TO 80)))
    (Guelph ((Cambridge 30) (TO 80)))
    (Hamilton ())
    (London ((Brantford 70)))
    (KW ((Cambridge 30) (Guelph 35) (Stratford 40)))
    (Stratford ((London 50)))
    (TO ((Hamilton 60)))))

(check-expect travel-time 'Guelph 'Hamilton southern-ontario) 90)
; travel time for '(Guelph Cambridge Brantford Hamilton)
```

(b) Depending on the implementation details for `travel-time`, the function might not have produced the shortest possible travel time between origin and destination. We want to write a function that calculates the travel times for all valid paths. For now, however, we just want to implement a function, `all-paths`, that produces all valid paths—we will worry about travel time later. `all-paths` consumes an origin and a destination in the form of two `Sym` as well as a `Map`. It then produces a list of all possible paths between origin and destination as a `(listof (listof Sym))`. `all-paths` requires that both origin and destination are on the map. (If there are multiple paths between the origin and destination, the order of these paths does not matter.)

```
(check-expect
  (all-paths 'Guelph 'Hamilton southern-ontario)
  '((Guelph Cambridge Brantford Hamilton)
    (Guelph Cambridge Hamilton)
    (Guelph Cambridge London Brantford Hamilton)
    (Guelph Cambridge TO Hamilton)
    (Guelph TO Hamilton)))
```

```
(check-expect (all-paths 'Stratford 'Guelph southern-ontario) empty)
```

(c) Now that we have mastered map-traversal, we can produce more complex data. Write a function `all-travel-times`. It consumes an origin and a destination in the form of two `Sym` as well as a `Map`. It then produces a list of all possible paths between origin and destination as well as their travel time as a `(listof (list Nat (listof Sym)))`. As before, `all-travel-times` requires that both origin and destination are on the map. (If there are multiple paths between the origin and destination, the order of these paths does not matter.)

```
(check-expect
  (all-travel-times 'Guelph 'Hamilton southern-ontario)
  '((90 (Guelph Cambridge Brantford Hamilton))
    (75 (Guelph Cambridge Hamilton))
    (200 (Guelph Cambridge London Brantford Hamilton))
    (170 (Guelph Cambridge TO Hamilton))
    (140 (Guelph TO Hamilton))))

(check-expect (all-travel-times 'Stratford 'Guelph
   southern-ontario) empty)
```

Place your solution in travel-map.rkt.

2. (60 %) *Countdown* is a British TV show, in which two teams compete in three games: a *Number Game*, a *Letter Game*, and a *Conundrum*. In this assignment, we are only interested in the *Number Game*. The rules for this game are as follows:

   - The game-master reveals 6 numbers to the teams. These numbers are natural numbers between (and including) 1 and 100. It is possible that a number appears more than once.
   - The game-master reveals a target number to the teams. Just like the 6 numbers above, the target number is a natural number, this time between (and including) 100 and 999.
   - Each player now has 30 seconds to reach the target (or get as close as possible) using the 6 numbers from above and the four operators (+, -, *, /). Not all of the numbers have to be used, and each number can only be used once (unless, of course, it appears more than once). In addition, intermediate results **must** be natural numbers themselves, i.e., not negative or fractions. Finally, an operator can be used more than once.
   - The team with the player who reached the target (or got closest) wins the *Number Game*.

   For example, the six natural numbers are (1, 3, 7, 10, 25, and 75) and the target number is 385. One solution would be $(3*7+25)*10-75=385$ (note that the solution does not use all six numbers). Another solution (which uses all numbers) would be $((1+3)*75/10+25)*7=385$ (note that in this solution, despite the division, none of the intermediate results are fractions).

The rules for the game are available online. For additional reference (or procrastination), you can find countless videos of *Countdown* on YouTube (a recent episode), as well as its cross-over cousin *8-Out-Of-10-Cats-does-Countdown* (most recent episode).

For our purpose, we will modify the original rules of the *Number Game* slightly.

- Fewer numbers: instead of having six numbers, we will be using five numbers only. We are doing this for better performance.
- All of the numbers have to be used: each valid solution must include every number exactly once. We are doing this to make the assignment easier.

Your task in this question is implementing a function `countdown-numbers`. The purpose of this function is calculating one possible solution to the *Number Game*. The function consumes a list of natural numbers (given by the game-master) and a target number (also given by the game-master). It then uses the modified rules of the *Number Game* to find one possible solution, and produces it in the form of a Binary Expression Tree. If no solution exists, the function produces `false`.

```
;; An Operator (Op) is (anyof '+ '- '* '/)

;; A Binary Expression Tree (BET) is one of:
;; * Nat
;; * (list Op BET BET)

;; (countdown-numbers lon target) produces a BET using the numbers in lon
;;   that evaluates to target, or false if no such BET exists.
;; countdown-numbers: (listof Nat) Nat -> (anyof BET false)
```

Implementing this function is a quite complex task, and there are multiple approaches for the implementation. This means that we have to find a balance between guiding you toward the correct solution without restricting your creativity too much. As a result, we offer you a choice: you can **either** complete sub-question (a) or sub-question (b). Both require you to ultimately implement `countdown-numbers`.

In (a), you have to follow our approach. This means that your functions have to produce the exact same values as ours (for exceptions, see individual subquestions); the advantage is that you can receive marks, even if you do not completely finish implementing `countdown-numbers`!

In (b), you are completely free to implement your function as you see fit. The downside is that your marks will entirely depend on the correctness of your implementation for `countdown-numbers`. It is of course possible to receive partial marks as well, depending on how many of our tests your code will pass.

Your mark for Q2 will be $max(Q2(a), Q2(b))$. Independently for your choice, place your solution in countdown.rkt.

We highly suggest testing your functions with small lists of `Nat` and `OP`. The length of the lists that your functions consume should have no influence on your implementation, only on

the time it takes for your code to finish. We will be testing your functions with lists of `Nat` up to length 5, and lists of `Op` up to length 4.

(a) Multiple steps are requires to implement `countdown-numbers`. One way to find a solution to the *Number Game* is by testing all possible combinations of natural numbers and operators. This approach—testing the entire possible input space—is generally referred to as *brute force*. In order to do so, you have to generate all possible combinations for the 5 numbers and all possible combinations for the 4 operators. You then have to combine these two input spaces, i.e., test all possible number combinations with all possible operator combinations.

i. Implement an algorithm that generates all possible number combinations given a list of numbers. Since numbers in the *Number Game* can only be used once, we are actually not interested in all *combinations* but rather all *permutations*. There are several well-established algorithms to generate permutations, but we recommend using *Heap's algorithm*. To implement this algorithm you need a `swap`-function.

```
;; (swap i j lst) produces lst with the elements at positions
;;   i and j swapped
;; swap: Nat Nat (listof X) -> (listof X)
;;   requires: i, j < (length lst)

(check-expect (swap 0 3 '(0 1 2 3 4 5)) '(3 1 2 0 4 5))
```

ii. Now that you have a `swap`-function, implement the rest of *Heap's algorithm*.

```
;; (generate-permutations lst) produces all possible permutations
;;   of lst.
;; generate-permutations: (listof X) -> (listof (listof X))

(check-expect
  (generate-permutations '(2 4 8))
  '((2 4 8) (2 8 4) (4 2 8) (4 8 2) (8 2 4) (8 4 2)))
```

Note: this function can produce a large amount of data; your outer list should contain `(length lst)`! permutations. Make sure to not test your algorithm with unreasonably large lists. Also, in order to receive full marks for this sub-question, the order of values produced by your implementation **does not** have to match ours. All permutations have to be present in your list, though.

iii. Now that we have generated all possible permutations of numbers, we have to combine them into single values using a combination of the operators `'(+ - * /)`. Since operators in *Number Game* can be used more than once, we are actually not interested in just the *combinations* but rather the *tuples*. To be more precise, we are interested in the n-tuples of an m-set, where n is the number of operators to be picked and m is the number of operators to pick from. Implement an algorithm that consumes a list of length `m` and a `Nat n` and produces all possible n-tuples of an m-set of the elements in that list.

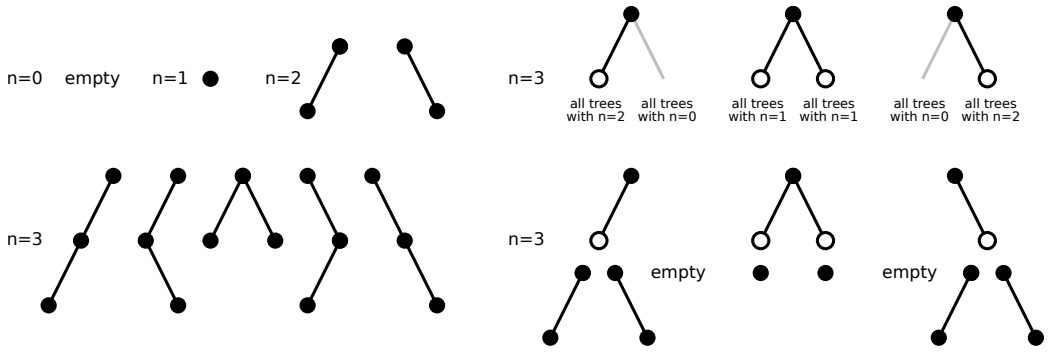
```
;; (generate-tuples lst n) produces all tuples of length n of
;;   elements in lst.
;; generate-tuples: (listof X) Nat -> (listof (listof X))

(check-expect
  (generate-tuples '(+ -) 3)
  '((+ + +) (+ + -) (+ - +) (+ - -)
    (- + +) (- + -) (- - +) (- - -)))

(check-expect
  (generate-tuples '(+ -) 0)
  (list empty))
```

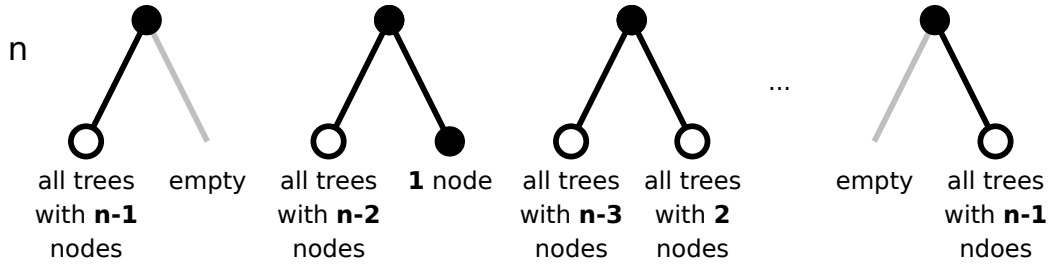
Note: this function can produce a large amount of data; your outer list should contain m^n n-tuples. Make sure to not test your algorithm with unreasonably large lists. In order to receive full marks for this sub-question, the order of values produced by your implementation **does not** have to match ours. All tuples have to be present in your list, though.
Hint: one way to generate tuples are nested calls to `map`.

iv. Now with all possible number and operator combinations, we can (almost) start to calculate values. Before that, we have to put values and operators together into `BET` (see above). One crucial realization here is that the structure of the `BET` influences the outcome of a calculation. If we take, for example, the list of numbers `'(8 6 4 2)` and operators `'(/ + -)`, we get different results depending on the structure of our `BET`:

```
'(/ (+ (- 8 6) 4) 2) ; => 3
'(/ (+ 8 6) (- 4 2)) ; => 7
'(/ 8 (+ 6 (- 4 2))) ; => 1
```

While this order is not important for commutative operators, i.e., $+$ and $*$, it is important for non-commutative ones, i.e., $-$ and $/$.
As a result, we have to create every possible `BET` with `n` internal nodes and `n+1` leaves, where `n` is the number of operators and `n+1` is the number of numbers. Luckily, there is a recursive algorithm for finding all possible trees:





Essentially, to create a tree of n nodes, you have to consider all possible ways to partition $n-1$ nodes between the left and the right subtrees of a node, and then, through recursion, build the left and right subtrees. Be aware that for a given n, you will get more than n possible trees; in fact, there are $\prod_{k=2}^{n} \frac{n+k}{n}$ possible trees (the so-called *Catalan number* C_n).

Implement `create-bets`. The function consumes a list of permutations of a list of numbers and an list of n-tuples of operators. It creates a list of all possible `BET` using the provided permutations and n-tuples, while considering all possible tree structures.

```
;; (create-bets nlon nloop) produces a list of all possible BET
;;   based off of nlon and nloop.
;; create-bets: (listof (listof Num)) (listof (listof Op)) ->
;;                                                  (listof BET)
;;   requires: (length nlon) - (length nloop) = 1

(check-expect
  (create-bets
    '((8 6 4 2))
    '((/ + -)))
  '((/ 8 (+ 6 (- 4 2)))
    (/ 8 (+ (- 6 4) 2))
    (/ (+ 8 6) (+ 4 2))
    (/ (+ 8 (- 6 4)) 2)
    (/ (+ (- 8 6) 4) 2)))
```

Note: this function can produce a large amount of data; for k lists in `nloop` (each of them of length n) and l lists in `nlon` (each of them of length $n+1$), your outer list should contain $k*l*C_n$ `BET`. Make sure to not test your algorithm with unreasonably large lists. In order to receive full marks for this sub-question, the order of values produced by your implementation **does not** have to match ours. All `BET` have to be present in your list, though.

Hint: As before, nested calls to `map` might be helpful.

It might be helpful to take two intermediate steps toward solving this problem: implement a function `(create-left-right-pairs node-cnt)`, which produces a list of number-pairs indicating the size of the left and right subtrees for a given overall node-count on a tree, e.g.,

```
(check-expect (create-left-right-pairs 3) '((2 0) (1 1) (0 2)))
```

Next, implement a function `(create-tree-structure node-cnt)`, which produces a list of trees, without putting in the values for internal nodes (operators) and leaves (numbers) yet. Instead, each node value represents the number of nodes , e.g.,

```
(check-expect (create-tree-structure 3)
  '((3 empty (2 empty (1 empty empty)))
    (3 empty (2 (1 empty empty) empty))
    (3 (1 empty empty) (1 empty empty))
    (3 (2 empty (1 empty empty)) empty)
    (3 (2 (1 empty empty) empty) empty)))
```

v. You not only have generated all possible equations, you generated them in a way that they can be evaluated using functions similar to `eval` and `eval-binode` from Module 11! Implement the function `evaluate-bets` that consumes a list of `BET`, and a target number, and produces all `BET` in that list that properly evaluate to the target number. These `BET` must also follow the rule of the game that every intermediate value has to be a `Nat` and larger than 0.

```
;; (evaluate-bets lobet target) produces a list of all BET from
;;   lobet that evaluate to the target value.
;; evaluate-bets: (listof BET) Nat -> (listof BET)

(check-expect
  (evaluate-bets
    (create-bets
      (generate-permutations '(2 4 8))
      (generate-tuples '(+ - *) 2))
    2)
  '((- 8 (+ 4 2))
    (- 8 (+ 2 4))
    (- (- 8 4) 2)
    (- (- 8 2) 4)))
```

Note: this function—like all brute-force algorithms—has to perform a large number of operations due its large input space. Make sure to not test your algorithm with unreasonable parameters. It should be able, however, to execute the command `(evaluate-bets (create-bets (generate-permutations '(1 5 7 10 25)) (generate-tuples '(+ - * /) 4)) 175)` in under 30 seconds. You might have to increase Racket's memory to 512 MB (you will see a window pop up). We will **not** test your submission with values that give impossibly large data; if your program is efficient enough to pass our basic test, it will be able to run our private test as well. In order to receive full marks for this sub-question, the order of values produced by your implementation **does not** have to match ours. All `BET` have to be present in your list, though.

vi. Congratulations! You are almost done. Now simply implement `countdown-numbers` (see above), which should be little more than a wrapper function for `evaluate-bets` at this point.

(b) Implement `countdown-numbers` using your preferred approach. We will test your solution with several number- and target-combinations.

Challenges and Enhancements

Consider the function `(euclid-gcd)` from slide 7-16. Let f_n be the nth Fibonacci number. Show that if $u = f_{n+1}$ and $v = f_n$, then `(euclid-gcd u v)` has depth of recursion n. Conversely, show that if `(euclid-gcd u v)` has depth of recursion n, and $u > v$, then $u \geq f_{n+1}$ and $v \geq f_n$. This shows that in the worst case the Euclidean GCD algorithm has depth of recursion proportional to the logarithm of its smaller input, since f_n is approximately ϕ^n, where ϕ is about 1.618.

You can now write functions which implement the RSA encryption method (since Racket supports unbounded integers). In Math 135 you will see fast modular exponentiation (computing $m^e \mod t$). For primality testing, you can implement the little Fermat test, which rejects numbers for which $a^{n-1} \not\equiv 1 \pmod{n}$, but it lets through some composites. If you want to be sure, you can implement the Solovay–Strassen test. If $n-1 = 2^d m$, where m is odd, then we can compute $a^m \pmod{n}$, $a^{2m} \pmod{n}, \ldots, a^{n-1} \pmod{n}$. If this sequence does not contain 1, or if the number which precedes the first 1 in this sequence is not -1, then n is not prime. If n is not prime, this test is guaranteed to work for at least half the numbers $a \in \{1, \ldots, n-1\}$.

Of course, both these tests are probabilistic; you need to choose random a. If you want to run them for a large modulus n, you will have to generate large random integers, and the built-in function `random` only takes arguments up to 4294967087. So there is a bit more work to be done here.

For a real challenge, use Google to find out about the AKS Primality Test, a deterministic polynomial-time algorithm for primality testing, and implement that.

Continuing with the math theme, you can implement the extended Euclidean algorithm: that is, compute integers a, b such that $am + bn = \gcd(m, n)$, and the algorithm implicit in the proof of the Chinese Remainder Theorem: that is, given a list $(a_1, \ldots, a_n)$ of residues and a list $(m_1, \ldots, m_n)$ of relatively coprime moduli ($\gcd(m_i, m_j) = 1$ for $1 \leq i < j \leq n$), find the unique natural number $x < m_1 \cdots m_n$) (if it exists) such that $x \equiv a_i \pmod{m_i}$ for $i = 1, \ldots, n$.