Fall 2022
Becker, Kamal, Nasr Esfahani, Tompkins, Van Leeuwen, Vasiga

Assignment: 08

Due:	Tuesday, November 22, 2022 9:00 pm
Coverage:	End of Module 15
Language level:	Intermediate Student with lambda
Allowed recursion:	All
Files to submit:	`or-map.rkt, nested.rkt, fizz-buzz-2.rkt, heap.rkt`

- **Make sure you read the OFFICIAL A08 post on Piazza** for the answers to frequently asked questions.
- Your solutions for assignments will be graded on both correctness and on readability. This assignment will be auto-graded.
- Submit examples for all questions except 2(a) and 3(a) by **Friday, November 18 at 8:00AM** (before the due date of assignment A08). See the Example Submissions and Basic Test Results post on Piazza for examples of correct and incorrect example submissions.
- The language level is *Intermediate Student with lambda*, but you may not use **`lambda`** yet.
- This assignment allows for all types of recursion, including *generative recursion*. For most of the questions, the expected solutions use simple, accumulative and/or mutual recursion. However, some functions (*e.g.,* `nested-ref`, `heap-add`, `heap->list`) naturally use generative recursion, which is allowed.
- For this assignment, you **cannot** use the built-in Racket function `filter`.

Here are the assignment questions you need to solve and submit.

1. Place your solution for the following two parts in a file named `or-map.rkt`.

   (a) Write a function `my-ormap` that consumes a predicate function `pred?` and a `(listof X)` (in that order), where the contract for `pred?` is:
   `pred?: X -> Bool`
   `my-ormap` produces `true` if `pred?` is `true` for *any* of the elements in the list (and `false` otherwise). Note that this is equivalent to the built-in function `ormap`, so obviously you cannot use that function.

   ```
   (my-ormap zero? '(8 6 7 5 3 0 9)) => true
   ```

   (b) Write a function `pred?-ormap` that consumes a value (of type `X`) and a list of type `(listof (X -> Bool))` (in that order) and produces `true` if any of the elements in the list are `true` when passed the value (and `false` otherwise).

   ```
   (pred?-ormap 5 (list zero? even? negative? posn? inexact?)) => false
   ```

2. For this question, we have new data definitions:

```
;; A (nested-listof X) is one of:
;; * empty
;; * (cons (nested-listof X) (nested-listof X))
;; * (cons X (nested-listof X))
;; Requires: X itself is not a list type

;; a NotAList is an Any that is not a list type
```

Place your solution for the following parts in a file named `nested.rkt`.

(a) Write the function template for a function named `nested-listof-X-template` that processes a `(nested-listof X)`. The only design recipe components required are the contract and the function definition.

(b) Write a function `nested-count` that consumes a `(nested-listof NotAList)` and produces the number of `NotAList` elements that appear anywhere in the nested list.

(c) Write a function `nested-sum` that consumes a `(nested-listof Num)` and produces the sum of every number that appears anywhere in the nested list.

(d) Write a function `nested-member?` that consumes a `NotAList` value and a `(nested-listof NotAList)` (in that order) and produces `true` if the value appears anywhere in the nested list, and `false` otherwise. In other words, if the nested list were to be "flattened", then `nested-member?` would behave the same as the built-in function `member?`. However, you must not solve this problem by simply flattening the list. The function `equal?` is allowed (and required) to implement this function.

```
(nested-member? 'hot-dog '((pizza) (hamburger) (((hot-dog))))) => true
```

(e) Write a function `nested-ref` that consumes a `(nested-listof NotAList)` and a natural number `k` (in that order) and produces the *k-th* element that appears in the nested list. In other words, if the nested list were to be "flattened", then `nested-ref` would behave the same as the built-in function `list-ref`. However, you must not solve this problem by simply flattening the list. When `k` is zero, it corresponds to the first `NotAList` element, and when `k` is `(nested-count - 1)`, it corresponds to the last `NotAList` element. Accordingly, `k` must be greater or equal to zero and less than the `nested-count` of the list. This also means that the `nested-count` of the list must be greater than zero.

```
(nested-ref '((0) 1 2 (3 4) 5 (6 7 (8)) 9 () (((hot-dog))))  0) => 0
(nested-ref '((0) 1 2 (3 4) 5 (6 7 (8)) 9 () (((hot-dog))))  8) => 8
(nested-ref '((0) 1 2 (3 4) 5 (6 7 (8)) 9 () (((hot-dog)))) 10) =>
   'hot-dog
```

There will be a *hint* for this question listed in the FAQ.

(f) Write a function `nested-filter` that consumes a predicate function (`X -> Bool`) and a (`nested-listof X`) (in that order) and removes every element that appears anywhere in the nested list where the predicate function is false for that element.

(g) After applying `nested-filter` function from the previous part, the result may have empty nested lists. Write a function `nested-cleanup` that removes any `empty` list anywhere in the list. For example:

```
(nested-cleanup '(1 () 2 () () 3)) => '(1 2 3)
```

`nested-cleanup` will also remove nested empty lists:

```
(nested-cleanup '(1 (()()) 2 ((3 () (()))) )) => '(1 2 ((3)))
```

And if there are no non-list elements anywhere in the list, it produces `false`:

```
(nested-cleanup '((()(()())(())(()))) => false
```

To implement `nested-cleanup`, you may not define any helper functions. This also excludes local helper functions, but you may define local constants if you wish.

3. For this question, we are going to revisit "Fizz Buzz" from A04. Place your solution for the following two parts in a file named `fizz-buzz-2.rkt`

   (a) Write a function generator named `make-div-pred` that consumes a positive integer `n` and produces a new predicate function. The generated predicate function consumes an integer `m` and produces `true` if `m` is divisible by `n` and `false` otherwise. You do not need to test or provide examples for your `make-div-pred` function. See the following example:

```
(define div5? (make-div-pred 5))

(div5? 21) => false
(div5? 25) => true
```

   (b) Write a function `fizz-buzz-2`, where the first two parameters are integers and behave the same as in `fizz-buzz` from A04 (`start` and `end`). The third parameter is a list of pairs: `(list Any (Int -> Bool))`, where the second element in the pair is a predicate function, and the first element in the pair is the value that should be used if the predicate function is true for the number in question. Seeing two examples of `fizz-buzz-2` will make it easier to understand the parameters:

```
(fizz-buzz-2 8 15 (list (list 'honk (make-div-pred 15))
                        (list 'fizz (make-div-pred 3))
                        (list 'buzz (make-div-pred 5))))
=> '(8 fizz buzz 11 fizz 13 14 honk)


(fizz-buzz-2 -3 3 (list (list "donut" zero?)
                        (list "even" even?)
                        (list 'neg negative?)))
=> '(neg "even" neg "donut" 1 "even" 3)
```

   For each integer in the range `start...end` (inclusive), the list of pairs is processed. Consider that the current number is `n`. If a predicate function is `true` for `n`, then the corresponding value in the pair is used instead of `n` in the produced sequence. If multiple predicate functions are `true`, then the first of those pairs is used. If none of the predicate functions are `true`, then the value of `n` is used in the produced sequence.

   Note that `start` can be greater than `end`, in which case `fizz-buzz-2` produces `empty`.

4. You are familiar with the *Binary Search Tree (BST)*, which is a binary tree that satisfies the *ordering property*.

   Another type of binary tree is a **heap**, which satisfies the (creatively named) *heap property*.

   ***Heap Property**: every key is less than or equal to every key in its subtrees.*[1]

   The following is an example of a heap:

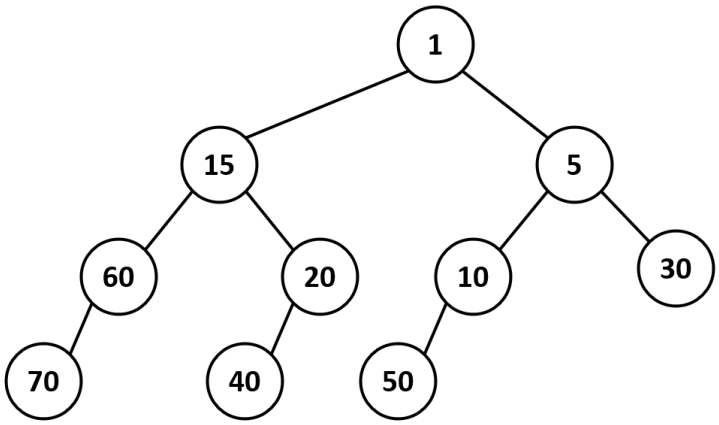


Figure A

   Note that unlike a BST, there is no relationship between the left and right child. There is only a relationship between a parent and its children (descendents).

   To implement a heap in Racket, we will use the following data definition:

```
(define-struct hnode (key left right))

;; A (heapof X) is one of:
;; * empty
;; * (make-hnode X (heapof X) (heapof X))
;; requires: all elements in left are >= key
;;           all elements in right are >= key
```

   Because the type of `X` is unknown, all of the functions you are required to write will consume a `X<=?` comparison function. For example, if your heap is composed of numbers, the comparison function would be `<=`. If your heap is composed of strings, the comparison function would be `string<=?`.

[1] Technically, this is the heap property for a "min" heap. There are also "max" heaps, where every key is greater than or equal to every key in its subtrees, but we will be working with "min" heaps.

Place your solution for the following parts in a file named `heap.rkt`.

IMPORTANT: We have provided a `heap.rkt` file that you must use to get started. We have also provided a file named `heap-support.rkt`, which must be saved in the same folder as your `heap.rkt`. The support file provides the structure definition for a `hnode` and a `heap-print` function to display heaps in the Racket interactions window. You do not need to submit the `heap-support.rkt` file, only your `heap.rkt` file.

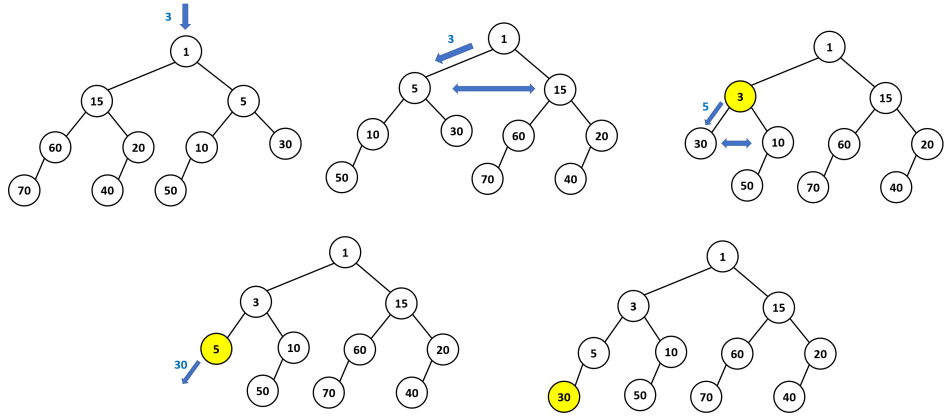
The `heap-print` function has been provided to help you *informally* test your code and to help you "visualize" what your heaps look like. It is not a substitute for testing your code using `check-expect`.

(a) Write a function `heap-add` that adds an element of type `X` to a `(heapof X)`, producing a new heap. The three parameters of `heap-add` are the element to add, the heap, and a `X<=?` comparison function (in that order). There is actually a clever method of adding to a heap to ensure that it remains balanced. You must implement your `heap-add` exactly as we describe in the following, where an element (`item`) is added to a heap:

- if the tree is empty, `item` becomes the root
- if `item` is less than the key at the root of the current subtree, then replace the key with `item` and continue to recursively add the former key as the "new" `item` (see the next step)
- *swap* the two children so that the new left subtree is the original right subtree with the `item` added recursively, and the new right subtree is the original left subtree

In other words, if the tree is empty, produce a new tree with the `item` as the root. Otherwise, produce a new tree where the key is the `min(key, item)`, the left subtree is the original right subtree with the `max(key, item)` added recursively, and the right subtree is the original left subtree. Of course, you cannot use the numerical `min` or `max` functions because the keys may not be numbers.

To visualize a sequence of additions to a heap using this method, we have provided a detailed example (link) that generates the example heap in Figure A above. The following diagram illustrates adding 3 to the heap in Figure A.



(b) Write a function `heap-remove-min` that removes the smallest element from the heap (the root) and produces a new heap. The two parameters are the heap and a `X<=?` comparison function (in that order). Unfortunately, the method to remove an element while maintaining the balanced property of the heap is a bit complicated, so we will use the following method (which could result in an unbalanced heap):

- if the heap is empty or both children are empty, produce empty
- if the left subtree is empty, produce the right subtree (and vice-versa if the right subtree is empty)
- if the root of the left subtree is smaller than or equal to the root of the right subtree, then the new key is the root of the left subtree, and recursively remove the root from the left subtree while keeping the right subtree the same (and vice-versa if the root of the right subtree is smaller)

The detailed example (link) also includes a sequence of removals.

(c) Write a function `list->heap` that generates a heap by recursively calling `heap-add`. The parameters are a `(listof X)` and a `X<=?` comparison function (in that order). Use simple recursion, so the last element of the list is added first, and the first element of the list is added to the heap last.
`(list->heap '(70 1 15 5 60 50 40 30 20 10) <=)` produces the example heap in Figure A above.

(d) Write a function `heap->list` that generates a sorted list of all of the elements in a heap by recursively calling `heap-remove-min`. The parameters are a `(heapof X)` and a `X<=?` comparison function (in that order). The produced list must be in ascending (non-decreasing) order (as determined by the comparison function).

(e) Write a function `heap-sort` that sorts a `(listof X)` by combining the above functions. The parameters are a `(listof X)` and a `X<=?` comparison function (in that order). The produced list must be in ascending (non-decreasing) order (as determined by the comparison function).