CMPU-101

CS1: Problem Solving and Abstraction

Fall 2017

Midterm II

Name: _____

Instructions:

1. This is an open book, open notes exam. You may use your CS account to access the book and course notes, including your labs and assignments. You may use DrRacket, but are on your honor not to use the Stepper, Interactions Pane, or Run any programs.
2. BUDGET YOUR TIME. There are 75 points on this exam, so you should spend about 1 minute per point (e.g., 15 minutes on a 15 point question). Don't spend too much time on any one question.
3. Points for each question are indicated below.
4. Please don't hesitate to ask any questions.
5. Good luck!!!

Question	Points	
	received	possible
1. List Expressions		15
2. The Cat in the Hat Comes Back		15
3. Multiple Lists Function		15
4. Binary Tree Function		15
5. Files and Directories		15
Total		75

Problem 1. (15 points)

Here are two examples of lists, expressed using list abbreviations:

```
(define nums (list 1 2 3 4 5))
(define num-names (list "one", "two", "three", "four", "five"))
```

a) (4 points) Rewrite the definitions above for `nums` and `num-names` using `cons` expressions—no list abbreviations.

b) (5 points) Evaluate the following expressions. For expressions that evaluate to a list, you may use list abbreviations or `cons` expressions, whichever you prefer.

```
(first (rest nums))

(rest (rest num-names))

(first (rest (rest (rest (rest num-names)))))

(empty? (rest (rest (rest (rest (rest nums))))))

(cons? '())
```

c) (6 points) Write expressions using `first`, `rest`, `cons`, `'()`, and/or list abbreviations (but no actual numbers or strings), and the two lists defined above, to produce the following:

- list containing `1` from the `nums` list followed by `"two"` from the `num-names` list

- the fourth element from the `num-names` list (not a list, just the element)

- list containing the last element of `nums` followed by the last element of `num-names`

Problem 2. (15 points)

The Cat in the Hat Comes Back is a Dr. Seuss book that your professor loved to read to his children when they were younger. This sequel to the original *Cat in the Hat* reveals a new recursive quality to the Cat in the Hat! When the Cat takes off his hat, Little Cat A is underneath, and under Little Cat A's hat is Little Cat B, and so on until we find Little Cat Z. Under Little Cat Z's hat, however, is something else: Voom! If you don't know what Voom is, I recommend you read the book! (Some of the greatest books for adults are disguised as children's books.) We can represent The Cat with the following Data Definition:

```
(define-struct cat (letter hat))
; A Cat is either:
; - "Voom"
; - (make-cat String Cat)
```

a) (3 points) Define three (3) examples of Cats. These are data examples only. Not functions. The first Cat should just be `"Voom"`. The second Cat, `catB`, should have `"Voom"` in its hat. The third Cat, `catA`, should have `catB` in its hat (i.e., `catB` and `catA` will both be `cat` structs).

b) (5 points) Design the template for a function that consumes a Cat. Your template function should be named `fun-for-cat`. Be sure to include an appropriate signature and purpose statement for your template function. You will use this template to design the function in part **c)**.

c) (7 points) Design the function `contains-letter?`, which consumes a Cat and a single character string (e.g., "A", "B", ...), and determines whether the given Cat contains a hat with the given letter—or if any Cat under its hat does!. Use the template you just designed, `fun-for-cat`, to help you get started. Follow the Design Recipe. Use your three examples of Cats to create `check-expect`s to test your `contains-letter?` function.

Problem 3. (15 points)

Given the now familiar Data Definition for a list of numbers:

```
; A list-of-number (LON) is either
; - '()
; - (cons number LON)
```

and the template we saw in Lecture 9 for a function that consumes two lists of the same size:

```
(define (fun-for-lons lon1 lon2)
  (cond
    [(empty? lon1) ...]
    [(cons? lon1) ...(first lon1)...(first lon2)...
                  ...(fun-for-lons (rest lon1) (rest lon2))...]))
```

Design a function named `any=?` that consumes two LONs of the same size, and determines whether any pair of numbers in corresponding positions in `lon1` and `lon2` are equal. Follow the Design Recipe, though no need to include a separate template, since one is provided above. Your solution should include a signature, purpose statement, header, examples (in the form of at least three `check-expect`s), and function body.

Problem 4. (15 points)

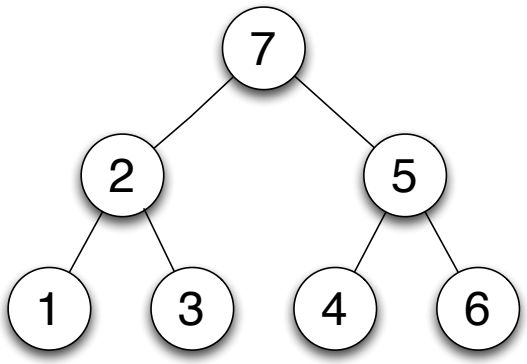
Here is a data definition for a binary tree (which is like our original rumor mill, except this tree contains `node` structs instead of gossiping `person` structs):

```
(define-struct node (value left right))

; A binary-tree (BT) is either
; - '()
; - (make-node Number BT BT)
```

Figure 1 below depicts a binary tree. In Figure 1, the circles represent node structs, and the lines below each circle represent the left and right BTs for each node. (Nodes 4, 5, 6, and 7 have `"null"` left and right fields, which are not shown in the figure.)

Figure 1: A binary tree



a) (5 points) Using the data definition given for a BT, translate Figure 1 into its Racket representation. *Hint: it's easiest to build the BT one level at a time, starting from the bottom. That is, give definitions for nodes 1, 3, 4, and 6 first, then use those names for defining nodes 2 and 5, etc.*

b) (10 points) Develop the function `sum-odds`, which consumes a binary tree (BT), and produces the sum of the odd numbers contained in the tree. **For full credit, be sure to include the following hand-in artifacts from the Design Recipe:** template `fun-for-bt`, signature/purpose/header, examples (you should have at least three, based on the data definition for BT, and the nature of this function), and last but not least, the function body.

Problem 5. (15 points) Recall (from lecture 12) our intertwined data definitions for files and directories:

```
; A directory is
; - (make-dir string LOFD)
(define-struct dir (name content))

; A file is
; - (make-file string number)
(define-struct file (name size))
```

```
; A file-or-directory is either
; - file
; - directory

; A LOFD is either
; - '()
; - (cons file-or-directory LOFD)
```

Design (four versions of) the function `large-file?` which consumes a directory and determines whether the given directory contains any large files (a large file is a file whose size is greater than 100). Collectively, these intertwined functions should determine whether a large file exists among the given files and directories.

To help you get started, I've provided the signatures for each of the four functions. You may find it helpful to use the corresponding templates provided in the Lecture 12 notes. You need only provide purpose statements and the functions for this question. If you're not sure how to write the functions, provide the templates for each function for partial credit.

```
; large-file-for-dir? : directory -> boolean
;
```

```
; large-file-for-file? : file -> boolean
;
```

```
; large-file-for-fod? : file-or-directory -> boolean
;
```

```
; large-file-for-lofd? : LOFD -> boolean
;
```