Identifying information will be printed here.

University of Waterloo Final Examination CS 135

Term: Fall Year: 2018

Date:	December 15, 2018
Time:	9:00 am
Duration:	150 minutes
Sections:	001–011
Instructors:	Becker, Clarke, Hackman, Jung, Lanctot, Nijjar, Reetz

Student Signature: ______________________________

UW Student ID Number: ______________________

Number of Exam Pages (including this cover sheet)	21 pages
Additional Material Allowed	Provided reference sheet

Question	Points
Q1	2
Q2	2
Q3	6
Q4	8
Q5	7

Question	Points
Q6	8
Q7	9
Q8	4
Q9	7
Q10	4

Total Points: 57

Instructions:

- There are a total of **10** questions and **57** points on this exam.
- All code is to use the **Intermediate Student with Lambda** language.
- Design recipe components are **not required** for functions we ask you to write, unless explicitly stated in the question. If you write helper functions that are not specifically asked for, they must include a purpose, contract, and function definition, unless stated otherwise.
- If a question or part has specific design recipe requirements or other restrictions, they will be indicated by text in a box.
- Unless otherwise specified, helper functions may be defined locally or separate from your primary function.
- Throughout the exam, you should follow good programming practises as outlined in the course such as appropriate use of constants and meaningful identifier names.
- Your functions do not have to check for invalid arguments.
- Functions you write may use:
  - Any function you have written in another part of the same question.
  - Any function we asked you to write for a previous part of the same question (even if you did not complete that part).
  - Any function on the reference sheet.
  - Any built-in **mathematical** function.
  - Any other built-in function or special form **that appears in lecture slides**, unless specifically noted in the question.
- Unless otherwise specified, for questions where you are required to provide a value, you may use either `cons`, `list`, or quoted list notation. For stepper questions, switching between these notations does **not** count as a "step".
- If you believe there is an error in the exam, notify a proctor. An announcement will be made if a significant error is found.
- It is your responsibility to properly interpret a question.
  - Do not ask questions regarding the interpretation of a question; it will not be answered and you will only disrupt your neighbours.
  - If there is a non-technical term you do not understand you may ask for a definition.
  - If, despite your best efforts, you are still confused about a question, state your assumptions and proceed to the best of your abilities.
- The amount of space allocated to a question does not necessarily reflect the length of the response required.
- If you require more space to answer a question, you may use the blank page(s) at the end of this exam, but you must **clearly indicate** in the provided answer space that you have done so. Marks for that question will be recorded on the initial page for that question and not the additional space.

Points on this page: 4

1. (2 points) History

Pick one of the famous computer scientists/mathematicians we discussed in lecture or in the course notes, describe one of their major contributions to computer science, and explain why that contribution is important today. (Don't write more than a few lines.)

2. (2 points) Producing a Function

Write a function `make-quad-fn` that consumes three `Nums` (called `a`, `b`, and `c`), and produces a function. The produced function should consume a `Num` (called `x`) and produce the value $ax^2 + bx + c$.

For example,

```
(define my-fn (make-quad-fn 1 2 3))
(my-fn 4) =>  (1)(4^2)+(2)(4)+(3)  =>  27
(my-fn 6) =>  (1)(6^2)+(2)(6)+(3)  =>  51
```

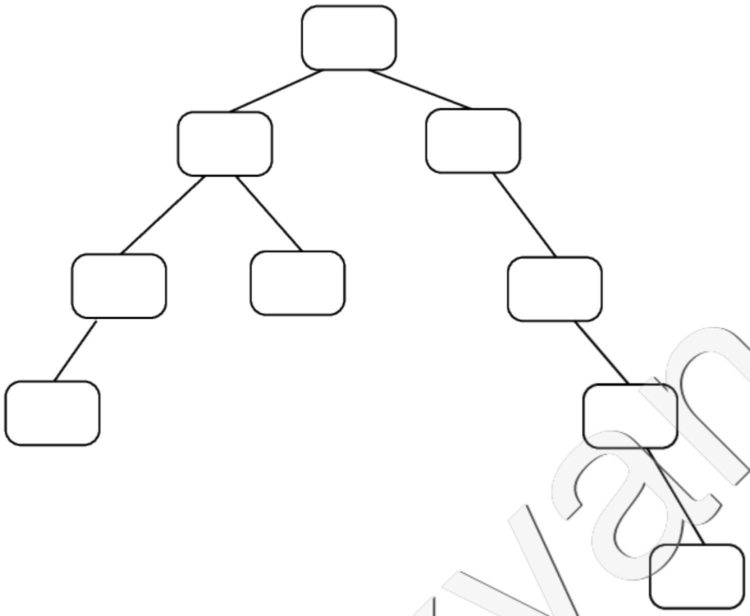
Required design recipe components: function definition, **contract**, and any requirements

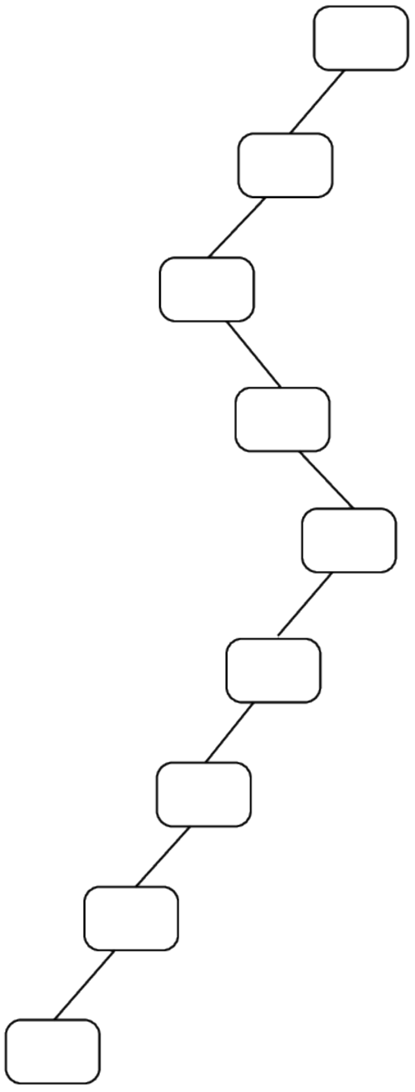
```
;; make-quad-fn:
;;
;;
(define (make-quad-fn a b c)
```

Points on this page: 2

3. (6 points) Binary Search Trees

(a) (2 points) For each tree below, fill in the nodes with keys 1, 2, 3, 4, 5, 6, 7, 8, 9 so the resulting trees satisfy the binary search tree property. You should use each key exactly once per tree.





Points on this page: 4

(b) (4 points) Consider the following definition for a binary search tree (BST):

```
;; A Binary Search Tree (BST) is one of:
;; * empty
;; * a Node

(define-struct node (key left right))
;; A Node is a (make-node Nat BST BST)
;; requires: key >= every key in left BST
;;           key < every key in right BST
```

Notice that this definition allows **duplicate keys** to exist. Write a function `key-count` which consumes a `BST` and a `Nat` called key. It produces the number of occurrences of that key in the tree. For full marks, your solution must make use of the ordering property of a binary search tree.

Required design recipe components: Function definition
Restrictions: Must use explicit recursion

```
(define (key-count bst key)
```

Points on this page: 4

4. (8 points) Stepping Problems

For the Racket expressions below, provide the following three lines:

- The *first* expression that would result from exactly one substitution step, using the substitution rules as defined in lectures;
- The expression that would result from the *second* substitution step; and
- The final value that is produced. If the evaluation would result in an error, describe the error. If the final value is reached in the first or second step, you do not need to write anything.

When one of these lines requires you to write out a list, you can use `cons` notation, the built-in function `list` or quote notation. When renaming **`local`** definitions, append "`_0`" if possible, or else "`_1`", "`_2`", etc. You do not need to repeat lines (e.g. constants) that are already in simplest form and which occur at the top of your code.

(a) (2 points)

```
(filter (lambda (x) (zero? (remainder x 3)))
        (build-list (+ 1 2) sqr))
```

[1st] ⇒

[2nd] ⇒

[final] ⇒

(b) (2 points)

```
(((lambda (x y) (lambda (y z) (list x y z))) 1 2) 3 4)
```

[1st] ⇒

[2nd] ⇒

[final] ⇒

Points on this page: 4

(c) (2 points)

```
(define (mystery x y z)
  (+ (local [(define z (* 2 x))
             (define (y x) (* 3 x))]
       (y z)) z))
```

```
(mystery 1 2 3)
```

[1st] ⇒

[2nd] ⇒

[final] ⇒

(d) (2 points)

```
(define (power x y) (expt x y))
(define (enigma f g a b c)
  (cond [(> (f a b) (g a c)) (map f (list a b c))]
        [else true]))
```

```
(enigma power * 100 2 90)
```

[1st] ⇒

[2nd] ⇒

[final] ⇒

5. (7 points) General Trees and Mutual Recursion

Consider the following mutually recursive data definitions for a set of directories (also known as a folders) and files below:

```
;; A FileDir is one of:
;; * A File
;; * A Directory
;;
;; A File is a Str
;; * where the Str is the name of the file
;;
;; A Directory is a (cons Str (listof FileDir))
;; * where the Str is the name of the directory
```

Write a function `count-name` that consumes a `Directory` called `dir` and a `Str` called `target`, and produces the total number of files and directories that have the name `target`. Hint: Use the data definitions to plan your helper functions.

Required design recipe components: For the main function and any helper functions, provide a function definition, **contract**, and any requirements
Restrictions: You must use mutual recursion.

Write your solution on the next page.

Points on this page: 7

```
;; count-name:
;;
;;
(define (count-name dir target)
```

Points on this page: 4

6. (8 points) Functional Abstraction

Recall from class the function template for lock-step processing of two equal-length lists (which we used to implement `dot-product`):

```
(define (lockstep-template lst1 lst2)
  (cond [(empty? lst1) ...]
        [else (... (first lst1) ... (first lst2) ...
                   (lockstep-template (rest lst1) (rest lst2)) ...)]))
```

(a) (4 points) Write `lockstep`, a higher-order function that abstracts `lockstep-template`. It consumes a combining function, a base value, and two equal-length lists. For example, `(lockstep f b (list x1 x2 ... xn) (list y1 y2 ... yn))` produces `(f x1 y1 (f x2 y2  ... (f xn yn b)))`.

Required design recipe components: Function definition, **contract, and any requirements**
Restrictions: Must use explicit recursion

```
;; lockstep:
;;
;;
(define (lockstep f base lst1 lst2)
```

Points on this page: 4

(b) (2 points) Write a function `abs-max` that consumes `lon1` and `lon2`, both of which are `(listof Num)`. It produces a list that contains one element for every pair of items at the same position in `lon1` and `lon2`. This element should be the larger absolute value of the pair. For example,

```
(abs-max '() '()) => '()
(abs-max '(2 1) '(-1 -3)) => '(2 3)
(abs-max '(-1 -2 -3) '(3 2 1)) => '(3 2 3)
```

Complete the following definition of `abs-max` by providing a lambda expression to replace **FUNC** and a value for **BASE**.

`(define (abs-max lon1 lon2) (lockstep` **FUNC BASE** `lon1 lon2))`

FUNC:

BASE:

(c) (2 points) Write a function `factors?` that consumes a list of numerators (`lon`) and a list of denominators (`lod`), both of which are `(listof Nat)`. It produces `true` if all numerators are evenly divisible by their denominators in the same list position, and `false` otherwise. (i.e. Compare each numerator ONLY with the denominator at the same list position). You may assume all values in `lod` are non-zero. For example,

```
(factors? '() '()) => true
(factors? '(4 6 10) '(2 3 5)) => true
(factors? '(4 6 10) '(2 4 5)) => false
```

Complete the following definition of `factors?` by providing a lambda expression to replace **FUNC** and a value for **BASE**.

`(define (factors? lon lod) (lockstep` **FUNC BASE** `lon lod))`

FUNC:

BASE:

Points on this page: 3

7. (9 points) Abstract List Functions

Required design recipe components: For parts that ask for an expression, give only that expression. For parts that ask for a function, give only the function definition.
Restrictions: For all parts of this question, you may not write helper functions and you may not use explicit recursion. **You must use abstract list functions.**

(a) (1 point) `;; A Dataset is a (listof Num) with at least two data points.`

Implement a function to calculate the arithmetic mean ($\overline{x}$) of a `Dataset`. Reminder: the arithmetic mean of a set of numbers is defined as: $\overline{x} = \frac{1}{N} * \sum_{i=1}^{N} x_i$, where x_i is the value of the ith data point, N is the number of data points, and $\sum_{i=1}^{N} x_i$ is the sum of the data points: $x_1 + x_2 + ... + x_N$.

```
;; (dataset-mean ds) calculates the average of ds.
;; dataset-mean: Dataset -> Num
(check-expect (dataset-mean '(2 -1 2 2)) 1.25)
(define (dataset-mean ds)
```

(b) (2 points) Implement a function to calculate the standard deviation (SD) of a `Dataset`.

$$SD = \sqrt{\frac{1}{N-1} * \sum_{i=1}^{N} (x_i - \overline{x})^2}$$

where x_i is the value of the ith data point and N is the number of data points.

You should use `local` to define constants for `N`, `mean` ($\overline{x}$), and `sum` ($\sum_{i=1}^{N} (x_i - \overline{x})^2$). Your solution may use `dataset-mean`, even if you did not implement it.

```
;; (dataset-stdev ds) calculates the standard deviation of ds.
;; dataset-stdev: Dataset -> Num
(check-expect (dataset-stdev '(2 -1 2 2)) 1.5)
(define (dataset-stdev ds)
```

Points on this page: 3

(c) (3 points) When collecting data, you often have to remove outliers, which are the few data points that do not seem to fit the rest of your data. A common practise is removing a data point x_i if its value is more than $3 * SD$ larger or smaller than $\bar{x}$. In other words:

- If $x_i > \bar{x} + 3 * SD$: remove data point x_i
- If $x_i < \bar{x} - 3 * SD$: remove data point x_i
- Otherwise: keep data point x_i

Implement a function that consumes a `Dataset` and produces a new `Dataset` with all outliers removed. (You may assume that the above algorithm will always produce a `Dataset` with at least 2 elements.) For full marks, avoid explicitly calling helpers more than once by using **`local`**.
Your solution may use `dataset-mean` and `dataset-stdev`, even if you did not implement them.

```
;; (remove-outliers ds) removes all outliers from ds using the
;; 3-SD rule (see above) and produces a clean Dataset.
;; remove-outliers: Dataset -> Dataset
(check-expect (remove-outliers '(1 -1 45 2 1 -2 0 0 -2 1 1))
              '(1 -1 2 1 -2 0 0 -2 1 1))

(define (remove-outliers ds)
```

Points on this page: 3

(d) (3 points) Write the following Racket function `destruction-sort`. Hint: The body can be written using only `foldr`, and a single **`lambda`**.

```
;; (destruction-sort lon) produces the list of numbers sorted in strictly
;;   increasing order (meaning no duplicates are allowed) which is the
;;   result of working backwards from the last number in lon and
;;   removing any elements that are out of order.
;; destruction-sort: (listof Num) -> (listof Num)
;; Examples:
(check-expect (destruction-sort '(20 5)) '(5))
(check-expect (destruction-sort '(-10 40 50 5)) '(-10 5))
(check-expect (destruction-sort '(1 3 5 9 4 10 16)) '(1 3 4 10 16))

(define (destruction-sort lon)
```

Points on this page: 4

8. (4 points) Short Answer

Write the **final values** that result from evaluating each of the following expressions. When the results are lists, use either `list` or quoted list notation (not `cons` notation).

i. `(* 6 7)`

ii. `(rest (first (rest (list (list 1 2) (list 3 4)))))`

iii. `((lambda (x) (* x x)) 3)`

iv. `((lambda (y) (lambda (x) (* x y))) 2)`

v. `(filter empty? '((1) () (2 3 4) (())))`

vi. `(map (lambda (x) (* (add1 x) x)) (list 1 2 3))`

vii. `(build-list (foldr + -4 '(1 2 3)) even?)`

viii. `(foldr append '(pass) '((This) (2 "shall")))`

Points on this page: 4

9. (7 points) Generative Recursion

Required design recipe components: For parts that ask for an expression, give only that expression. For parts that ask for a function, give only the function definition.

For this question, we define the concept of "shifting" a list. When a list is shifted to the **left** one time, its **first** element is removed and appended to its end.

e.g. `'(1 2 3 4 5 6) => '(2 3 4 5 6 1)`

When a list is shifted to the **right** one time, its **last** element is removed and appended to its front.

e.g. `'(1 2 3 4 5 6) => '(6 1 2 3 4 5)`

To shift a list `n` times, we can apply `n` individual shifts.

(a) (2 points) Write a function `shift-left` that consumes a `Nat` called `n`, and a **non-empty** list called `lst`. It should produce `lst` shifted **left** `n` times.

```
(define (shift-left n lst)
```

(b) (2 points) Write a function `shift-right` that consumes a `Nat` called `n`, and a **non-empty** list called `lst`. It should produce `lst` shifted **right** `n` times. Hint: You may find `reverse` useful.

```
(define (shift-right n lst)
```

Points on this page: 3

(c) (2 points) Write a function `shift-list` that consumes a **non-empty** `(listof Int)` called `loi`. It produces a new list in the following way:

- If `(first loi)` is negative, shift the list **left** `(abs (first loi))` times, then call `shift-list` again.
- If `(first loi)` is positive, shift the list **right** `(first loi)` times, then call `shift-list` again.
- If `(first loi)` is zero, produce `loi`.

Inspect the following examples of `shift-list` carefully.

```
(shift-list '(-1 0 20 8)) => '(0 20 8 -1)
(shift-list '(-2 0 1 5)) => '(0 1 5 -2)
(shift-list '(-2 0 -1 0 8 9)) => '(0 8 9 -2 0 -1)
(shift-list '(0 5 4 -2 1)) => '(0 5 4 -2 1)
(shift-list '(1 3 0 7 0 1 1)) => '(0 1 1 1 3 0 7)
```

You may use `shift-left` and `shift-right`, even if you did not complete them.

```
(define (shift-list loi)
```

(d) (1 point) Does `shift-list` terminate for all consumed values? (Circle one)

Yes No

If yes, explain why. If no, give an example of a list (in quoted list or `list` notation) that would cause it to run forever.

Points on this page: 4

10. (4 points) Graphs

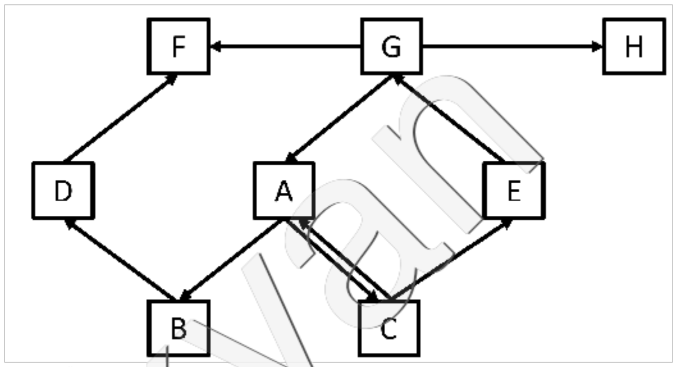
Given the following definitions:

```
;; A Node is a Sym
;;
;; A Graph is a (listof (list Node (listof Node)))

(define a-graph '((A (B C)) (B (D)) (C (A E))   (D (F))
                  (E (G))   (F ())  (G (A F H)) (H ())))
```

Note that the nodes and lists of neighbours in `a-graph` have been sorted alphabetically.

`a-graph` can be represented pictorally as:



For the following questions, use `find-route` as defined on your reference sheet, which is the version that handles cycles but is not efficient on diamond graphs. "Visiting" a node means that it is assigned to `orig` in a call to `find-route/acc`. If an expression does not terminate, write "Does not terminate".

(a) (1 point) Write the nodes visited in order when `(find-route 'B 'H a-graph)` is evaluated. (For example, `(find-route 'A 'F a-graph)` visits `A B D F`.)

(b) (1 point) What is produced by `(find-route 'B 'H a-graph)`?

(c) (1 point) Write the nodes visited in order when `(find-route 'A 'H a-graph)` is evaluated.

(d) (1 point) What is produced by `(find-route 'A 'H a-graph)`?

This page is intentionally left blank for your use. Do NOT remove it from your booklet. If you require more space to answer a question, you may use this page, but you **must clearly indicate** in the provided answer space that you have done so. If this page is used for continuing an answer, the marks for work done here will be included in that question's page mark.

This page is intentionally left blank for your use. Do NOT remove it from your booklet. If you require more space to answer a question, you may use this page, but you **must clearly indicate** in the provided answer space that you have done so. If this page is used for continuing an answer, the marks for work done here will be included in that question's page mark.