Identifying information will be printed here.

University of Waterloo
Midterm Examination
CS 145

Term: Fall Year: 2025

Date: October 27th, 2025
Time: 7:00 pm- 8:50 pm
Duration: 110 minutes
Sections: 001-002
Instructors: Vasiga

Student Signature: _____

UW Student ID Number: ANSWER KEY

Number of Exam Pages (including this cover sheet)	22 pages
Additional Material Allowed	UW-Approved Calculators

Question	Points
Q1	9
Q2	5
Q3	4
Q4	15
Q5	8

Question	Points
Q6	5
Q7	6
Q8	10
Q9	11

Total Points: 73

Instructions:

- All code is to use the Intermediate Student with lambda language level, unless otherwise stated.
- Unless otherwise stated, only the function body is required. Contracts, examples, and/or tests will be specified if required. Examples and tests must use `check-expect`. Unless otherwise allowed, tests and examples provided in the student solutions must be different from any examples supplied in the question. Examples also count as test cases.
- Unless otherwise specified, you may assume that all arguments provided to a function will obey the contract.
- 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 didn't do that part).
  - Any other built-in function or special form **discussed in the course**, unless specifically noted in the question.
  - Any built-in **mathematical** function.
- Unless otherwise specified, for questions where you are required to provide a value, you may use either `cons` or `list` notation. For stepper questions, switching between these notations does not count as a "step".
- Throughout the exam, you should follow good programming practices as outlined in the course such as appropriate use of constants and meaningful identifier names.
- You are **not** allowed to use any built-in Racket functions or special forms not discussed in lecture (e.g., `symbol->string`, `apply`, etc.).
- 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.
- 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.

Points on this page: 9

1. (9 points) **(a)** (3 points) Using mathematical induction, prove that $2^n > n$ for all $n > 1$, $n \in \mathbb{Z}$.

Base case: When $n = 1$, we have $2^n = 2^1 = 2 > 1 = n$. Thus, the base case holds.
Inductive hypothesis: Assume that when $n = k$ (for $k > 1$) that $2^k > k$ holds.
Inductive step: We must show that $2^{k+1} > k+1$. We have the following:

$$\begin{aligned} 2^{k+1} &= 2 \cdot 2^k \\ &> 2 \cdot k \\ &= k + k \\ &> k + 1 \end{aligned}$$

since $k > 1$. Thus, the inductive step holds, and $2^n > n$ for all $n > 1$ by mathematical induction. ■

(b) (3 points) Using the definition of big-O, prove that $3n \log_2 n \in O(n^2)$. You may use part (a) to help you.

We need a $c > 0$ and $n_0 > 0$ such that $3n \log_2 n \leq c \cdot n^2$ for all $n \geq n_0$.
Pick $c = 3$ and $n_0 = 1$.
Then, we have from part (a), for all $n > 1$:

$$\begin{aligned} 2^n &> n \\ \Rightarrow \quad n &> \log_2 n \\ \Rightarrow \quad n^2 &> n \log_2 n \\ \Rightarrow \quad 3n^2 &> 3n \log_2 n \end{aligned}$$

Therefore, since $3n \log_2 n \leq 3n^2$ for all $n > 1$, $3n \log_2 n \in O(n^2)$. ■

(c) (3 points) Using the definition of big-O, prove that $\frac{1}{100} n^3 \notin O(n^2)$.

By way of contradiction, assume that $\frac{1}{100}n^3 \in O(n^2)$. Then, there exists constants $c > 0$ and $n_0 > 0$ such that $\frac{1}{100}n^3 \leq cn^2$.
Pick $n_* = \max(101c, n_0 + 1)$. Notice that $n_* > 1$ since $n_0 > 0$. Additionally, we know $n_* \geq 101c$. Using this last inequality, we have:

$$\begin{aligned} & n_* \geq 101c \\ \Rightarrow \quad & n_*^3 \geq 101cn_*^2 \\ \Rightarrow \quad & \tfrac{1}{100}n_*^3 \geq \tfrac{101}{100}cn_*^2 \\ \Rightarrow \quad & \tfrac{1}{100}n_*^3 > cn_*^2 \end{aligned}$$

which contradicts that $\frac{1}{100}n^3 \leq cn^2$ for all $n > n_0$, since $n_* > n_0$. Therefore, $\frac{1}{100}n^2 \notin O(n^2)$. ■

Points on this page: 5

2. (5 points) Write the Racket function `minor` which consumes a list, `M`, of n lists of numbers, with each list being of length m, and two integers `i` and `j`. The function should produce a list of $n-1$ lists, each of length $m-1$ of numbers. The produced list should contain the same elements as `M`, except with the `i`th list removed, and the `j`th element from each list removed from the remaining lists. We assume that all lists are 0-indexed (i.e., the first element in a list is at index 0).

For example, `(minor (list (list 1 2 3) (list 4 5 6) (list 7 8 9)) 2 1)` should produce `(list (1 3) (4 6))`.

```
;; a helper function to remove the ith element from the given list M
(define (remove-i M i)
  (cond
    [(zero? i) (rest M)]
    [else (cons (first M) (remove-i (rest M) (sub1 i)))]))

(define (minor M i j)
  (map (lambda (L) (remove-i L j)) (remove-i M i)))
```

Points on this page: 4

3. (4 points) Write the Racket function `bin-to-int` that consumes a list of boolean values, and produces the positive integer represented by that binary (base-2) number, when read from left to right. For example, `(bin-to-int (list true false true true))` should produce `11`, since 1011 in base-2 means $2^3 + 2^1 + 2^0 = 11$ in base-10. For full marks, use tail-recursion.

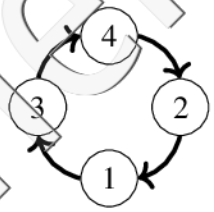
```
(define (bin-to-int B)
  (foldl (lambda (fir acc) (+ (if fir 1 0) (* 2 acc))) 0 B))

;; Alternative version with explicit tail recursion
(define (bin-to-int B)
  (bin-to-int-help B 0))
(define (bin-to-int-help B acc)
  (cond
    [(empty? B) acc]
    [(first B) (bin-to-int-help (rest B) (add1 (* 2 acc)))]
    [else (bin-to-int-help (rest B) (* 2 acc))]))
```

Points on this page: 2

4. (15 points) You are to implement an ADT Circular List (CList). A CList stores distinct numeric values and supports the operations:

- `(singleton k)` – produces a CList with the element `k` contained it in.
- `(insert-before v p C)` – places the element `v` into the Clist `C` before `p`. (You can assume that `v` is not in `C`, and that `p` is in `C`, when this function is called)
- `(contains? k C)` – produces `true` if `k` is in `C`, otherwise produces `false`.
- `(nextkey k C)` – produce the key which occurs after `k` in CList `C`. (You can assume that `k` is in `C`).
- `(prevkey k C)` – produce the key which occurs previously to `k` in CList `C`. (You can assume that `k` is in `C`).
- `(delete k C)` – produces a CList which is the same as `C`, except with element `k` removed
- `(list-cl k C)` – produce a list of all keys in `C`, starting at key `k`, going clockwise around `C`.

For example, if we perform
`(define C1 (insert-before 4 2 (insert-before 2 1 (insert-before 1 3 (singleton 3)))))`,
we get the following picture represent of `C1`:



Then, `(nextkey 4 C1)` produces 2, `(prevkey 4 C1)` produces 3, and `(list-cl 2 C1)` produces `(list 2 1 3 4)`.

Using a simple one-dimensional list to store the values, implement these operations below.

(a) (1 point) Write the Racket function for `singleton`.

```
(define (singleton k) (list k))
```

(b) (1 point) Write the Racket function for `contains?`.

```
(define (contains? k C) (member? k C))

;; alternative with filter
(define (contains? k C) (empty? (filter (lambda (x) (= x k)) C)))

;; alternative with explicit recursion
(define (contains? k C)
  (cond
    [(empty? C) false]
    [(= (first C) k) true]
    [else (contains? k (rest C))]))
```

Points on this page: 13

(c) (2 points) Write the Racket function for `delete`.

```
(define (delete k C)
  (cond
    [(= k (first C)) (rest C)]
    [else (cons (first C) (delete k (rest C)))]))
```

(d) (2 points) Write the Racket function for `insert-before`.

```
(define (insert-before v p C)
  (cond
    [(= (first C) p) (cons v C)]
    [else (cons (first C) (insert-before v p (rest C)))]))
```

(e) (3 points) Write the Racket function for `nextkey`.

```
(define (nextkey k C)
  (nextkey-help k C (first C)))
(define (nextkey-help k C orig-first)
  (cond
    [(= k (first C))
     (cond
       [(empty? (rest C)) orig-first]
       [else (first (rest C))])]
    [else (nextkey-help k (rest C) orig-first)]))
```

(f) (3 points) Write the Racket function for `prevkey`.

```
(define (prevkey k C)
  (cond
    [(= k (first C)) (list-ref C (sub1 (length C)))]
    [else (prevkey-help k C)]))
(define (prevkey-help k C)
  (cond
    [(= (first (rest C)) k) (first C)]
    [else (prevkey-help k (rest C))]))
```

(g) (3 points) Write the Racket function for `list-cl`.

```
(define (list-cl start C)
  (append (after start C) (before start C)))
(define (before start C)
  (cond
    [(= (first C) start) empty]
    [else (cons (first C) (before start (rest C)))]))
(define (after start C)
  (cond
    [(= (first C) start) C]
    [else (after start (rest C))]))
```

Points on this page: 5

5. (8 points) Troy wants to store student records in a binary search tree, with the last name field used for comparisons in the BST (i.e., the key is the last name, and it is compared using `string<?`). As such, he defines the following structures:

```
(define-struct student (lname fname id grades left right))
```

to store the last name, first name, UW student id, and a list of grades for each student, along with the necessary BST fields.

(a) (2 points) Define a `Student` constant `ya` for student with lastname Agisav, first name Yort, with id number 20009999, and has grades 87, 60, 45, and 23 (in that order). This node also has two children: one student `sm` with last name Mitchell, and the other student `ea` with last name Aaron. You can assume that `sm` and `ea` have already been defined.

```
(define ya (make-student "Agisav" "Yort" 20009999 (list 87 60 45 23) ea sm))|
```

(b) (3 points) Write the function `mean`, which consumes a `Student`, `s`, and produces a `Posn` structure where the `x` field contains the UW id for the student, and the `y` field contains the the mean (i.e., average) grade for `s`. You can assume that each grade for the student is a percentage grade (i.e., the grades are out of the same total of 100). For full marks, do not write any helper functions and use abstract list functions to help you.

```
(define (mean s)
  (make-posn (student-id s)
             (/ (foldr + 0 (student-grades s)) (length s))))
```

Points on this page: 3

(c) (3 points) Write the function `classlist` that consumes a BST of `Students`, `T`, and produces a list of the last names of students in ascending alphabetical order (i.e., using `string<?` as the comparison function). For full marks, your solution should be $O(n)$ if there are n `Students` in `T`.

```
(define (classlist T)
  (classlist-acc T empty))

(define (classlist-acc T acc)
  (cond
    [(empty? T) acc]
    [else (classlist-acc (student-left T) (cons (student-lname T) (
       classlist-acc (student-right T) acc)))]))
```

Points on this page: 5

6. (5 points) A *Lucas sequence* is defined as:

$$\begin{aligned} L(n) &= P \cdot L(n-1) + Q \cdot L(n-2), \quad n \geq 2 \\ L(0) &= A \\ L(1) &= B \end{aligned}$$

for integers P, Q, A and B.

Write the **tail recursive function** `L`, which consumes integers `P`, `Q`, `A`, `B`, and `n` (in that order) and produces $L(n)$. You can assume that `n` is non-negative.

```
(define (L P Q A B n)
  (Lh P Q n B A))

(define (Lh P Q n Ln-1 Ln-2)
  (cond
    [(zero? n) Ln-2]
    [else (Lh P Q (sub1 n) (+ (* P Ln-1) (* Q Ln-2)) Ln-1)]))

(define (L-alt P Q A B n)
  (cond
    [(zero? n) A]
    [else (L-alt P Q B (+ (* P B) (* Q A)) (sub1 n))]))
```

Points on this page: 6

7. (6 points) **(a)** (3 points) Recall the definition for a BST node:

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

We will assume that all keys are `Num`s.

Consider the following incorrect code for `bst?`, which produces `true` if the given argument is a binary search tree, and `false` otherwise.

```
(define (bst? T)
  (cond
    [(empty? T) true]
    [(not (node? T)) false]
    [else (and (bst? (node-left T))
               (bst? (node-right T))
               (or (empty? (node-left T))
                   (< (node-key (node-left T)) (node-key T)))
               (or (empty? (node-right T))
                   (> (node-key (node-right T)) (node-key T))))]))
```

Draw an example argument (i.e., a picture, not a nested sequence of `make-node`s) which causes this `bst?` function to produce the incorrect value. State what the correct produced value should be, and briefly explain why this function does not produce the correct value. Your example argument should contain at most 7 `node`s.

Notice that the tree rooted at 3 is a BST, but when the root node (5) is examined, it only looks at the value of the root of it's left child (3), which does meet the condition of being less than it's value (5). However, the node 8 should not be in a subtree to the left of 5.

(b) (3 points) Write the Racket function `tree-max` that consumes a BST `T`, and produces the largest element in `T`, and `false` if `T` is `empty`. The running time should be $O(h)$ where h is the height of `T`.

```
(define (tree-max T)
  (cond
    [(empty? T) false]
    [(empty? (node-right T)) (node-key T)]
    [else (tree-max (node-right T))]))
```

Points on this page: 6

8. (10 points) Determine the first and second substitution steps of each of the following expressions, along with the final value. If the evaluation would result in an error, describe the error. Assume that the following `define` appear on the left of all the expressions in this question, and has been fully evaluated. Note that you should use the *Intermediate Student* language level in evaluating these expressions, except for part (e), where you should use the *Intermediate Student with lambda* language level.

```
(define (f v w)
  (cond [(empty? v) w]
        [(zero? w) w]
        [else (+ (first v) w (f (rest v) (add1 w)))]))
```

(a) (2 points) `(posn? (posn-x (make-posn (make-posn 3 5) (make-posn 4 7))))`

[1st] ⇒ `(posn? (make-posn 3 5))`

[2nd] ⇒ `true`

[final] ⇒ `true`

(b) (2 points) `(first (rest (first (list (list 6 5) (list 4 3) (list 2 1)))))`

[1st] ⇒ `(first (rest (list 6 5)))`

[2nd] ⇒ `(first (list 5))`

[final] ⇒ `5`

(c) (2 points) `(f (list 1 2 3) (add1 1))`

[1st] ⇒ `(f (list 1 2 3) 2)`

[2nd] ⇒ `(cond [(empty? (list 1 2 3)) 2] [(zero? 2) 2] [else (+ (first (list 1 2 3)) 2 (f (rest (list 1 2 3)) (add1 2)))])`

[final] ⇒ `20`

Points on this page: 4

(d) (2 points) `(reverse (rest (reverse (rest (list 1 2 3 4)))))`

[1st] ⇒`(reverse (rest (reverse (list 2 3 4))))`

[2nd] ⇒`(reverse (rest (list 4 3 2)))`

[final] ⇒`(list 2 3)`

(e) (2 points) `((lambda (a b) (a b b)) (lambda (c d) (- (sqr c) d)) 6)`

[1st] ⇒`((lambda (c d) (- (sqr c) d)) 6 6)`

[2nd] ⇒`(- (sqr 6) 6)`

[final] ⇒`30`

Points on this page: 11

9. (11 points) For the following questions, you cannot use explicit recursion: that is, a function may not call itself, nor may you have any named helper functions. **You must use abstract list functions.**

(a) (3 points) Write the Racket function `my-map` which consumes a function `f` (with contract `f: X -> Y`), and a list, `L`, containing elements of type `X`, and produces the same thing as `(map f L)`. Your function must be one application of `foldr`, and cannot use the built-in `map` function.

```
(define (my-map f L)
        (foldr (lambda (frst rror) (cons (f frst) rror)) empty L))
```

(b) (3 points) Write the Racket function `member-alf?`, which consumes a value `v` and a list `L`, and does exactly what `(member? v L)` does (i.e., produce `true` if `v` is in `L`, and `false` otherwise). You cannot use the built-in `member` or `member?` functions. Additionally, give the **contract** for `member-alf?`.

```
;; member-alf?: Any (listof Any) -> Bool
(define (member-alf? v L)
  (not (empty? (filter (lambda (x) (equal? v x)) L))))
```

(c) (3 points) Write the Racket function `list-min` that consumes a non-empty list of numbers, `L`, and produces the minimum value contained in the list. You may not use `apply`.

```
(define (list-min L)
  (foldr (lambda (frst rror) (min frst rror)) (first L) L))
```

(d) (2 points) There is a built-in abstract list function called `andmap` that consumes a function, `f`, and a list, `L`, and will produce `true` if all the elements in `L` produce `true` when individually given as the argument to `f`, and `false` otherwise. Write the contract for `andmap`.

```
;; andmap: (X -> Bool) (listof X) ->  Bool

;; alternative
;; andmap: (Any -> Bool) (listof Any) ->  Bool

;; pedantic, but accepted
;; (Any -> Any) (listof Any) -> Bool
```

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.

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.