CS270: LAB #3

Racket proofs

You may work in groups of one or two people (three is acceptable in the event of an unscheduled absence). Unless stated otherwise, the lab is due to be submitted into Gradescope at the end of the day.
In order to receive credit, follow these instructions:

[a] Every team member should be discussing simultaneously the same problem – do NOT try to divvy up the labor and assign different problems to different students since the material is cumulative.

[b] Directly edit this lab PDF using Sedja/PDFescape with your answers (extra pages can be added in the rare event you need more than the allotted space)

[c] Each lab, rotate which member has the responsibility of being the Scribe. This is the person that is typing the answers and uploading the final PDF – note that only a single copy of the filled in PDF is turned into Gradescope. Only one lab needs to be submitted for the entire team, and all members receive the same score. Make sure to use a font that your PDF editor is compatible with (otherwise you might find your answers appear as weird shapes/sizes or simply disappear entirely!)

[d] The Gradescope submission must have each answer properly tagged with the appropriate question. Moreover, every member of the team must be listed as a submitter. Although it is the Scribe which executes these actions, it is still the responsibility of the entire team to make certain this is done properly (thus it is highly recommended that the Scribe share their screen so the entire team can witness it). Answers which are improperly tagged cannot be seen by the grader and thus cannot be scored.

[e] **FOR REMOTE ONLY**: Each lab, rotate which member has the responsibility of being the Recorder. This is the person who hits the Zoom Record button (once the technical permission is granted by the TA/RCF/Professor) and ensures that everyone has their camera/microphone on. They are also the member that is responsible to make sure the DrexelStream video is marked as viewable and entered into the https://tinyurl.com/VidLinkForm webform before 11:59pm (they should also email the rest of their team as confirmation.) Note that the video file doesn't get created/processed until after the Recorder has quit Zoom.

[f] Each lab, rotate which member has the responsibility of being the Manager. This is the person that ensures that everyone is participating equally and honestly, keeps the group on task, ensures that all team members understand a solution before going on to the next question, and presses the "hand up" button in Zoom to summon a TA or the professor (but they only do so after surveying the group to make sure everyone has the same question).

Team Name (CS pioneer):	Tim Berners-Lee
Scribe name:	Brendan Hoag
Recorder name:	Cole Bardin
Manager name:	Jeremy Mathews
Other team member (if any):	Jackson Masterson

Question 1 : 14 points

How to Write a Proof

A **proof** is sequence of statements that justify a theorem is true. We will see a few ways to write proofs in this class, but they are all based on some common principles.

Every Proof has a **start**. These are the things that are known to be true at the start of the problem.

Every Proof has an **end**. This is the thing we want to justify. This should always be the last line of the proof.

Every Proof has a **middle**. The **middle** bridges the gap between the **start** and **end**.

Every statement in a **proof** has two parts. The first is an expression and the second is a justification. The justification is a short written explanation of why the expression must be true. Proofs generally also contain line numbers, these allow for easy cross-referencing.

Prove: The Racket expression (if (> 6 9) (< 4 5) (> 9 1)) will evaluate to **#t**.

1.	(if (> 6 9) (< 4 5) (> 9 1))	Premise
2.	(if #f (< 4 5) (> 9 1))	Evaluate >
3.	(> 9 1)	Evaluate If
4.	#t	Evaluate >

The justification "Evaluate >" explains why line 2 is a logical follow-up to line 1. Justifications should be short and easy to understand.

(a) (2 points) What does it mean to have **premise** as a justification?

Premise means that it is defining the initial statement

(b) (2 points) Which line(s) of the above proof are the **beginning**?

Line 1 is the beginning

(c) (2 points) Which line(s) of the above proof are the **middle**?

Lines 2 and 3 are the middle

(d) (2 points) Which line(s) of the above proof are the **end**?

Line 4 is the end

(e) (6 points) **Prove**: The Racket expression (**first** (**rest** (**first** '((2 3) 4 5)))) will return 3.

1. (first (rest (first '((2 3) 4 5))))	Premise
2. (first (rest ('(2 3))))	Evaluate First
3. (first ('(3)))	Evaluate Rest
4. 3	Evaluate First

Question 2 : 9 points

Examine the following Racket function. Note that in this lab, "=" is used in place of "equal?" just for brevity

```
;Computes x*y using addition
;Assuming x >=0 and y >= 0
(define (prod x y) (if (= y 0) 0 (+ x (prod x (- y 1)))) )
```

There are two simple justifications related to functional programming. A proof written by justifying code execution is called **Equational Reasoning**.

- **Evaluation**: Commands that are built into the program language are evaluated.
- **Application**: If we have a function, like prod above we can replace the function name with its definition.

Prove: The return value of (prod 3 2) is 6.

1.	(prod 3 2)	Premise
2.	(if (= 2 0) 0 (+ 3 (prod 3 (− 2 1))))	Apply Def. of Prod
3.	(if #f 0 (+ 3 (prod 3 (− 2 1))))	Evaluate equals
4.	(+ 3 (prod 3 (− 2 1)))	Evaluate if
5.	(+ 3 (prod 3 1))	Evaluate Subtraction
6.	(+ 3 (if (= 1 0) 0 (+ 3 (prod 3 (− 1 1)))))	Apply Def. of Prod
7.	(+ 3 (if #f 0 (+ 3 (prod 3 (− 1 1)))))	Evaluate equals
8.	(+ 3 (+ 3 (prod 3 (− 1 1))))	Evaluate if
9.	(+ 3 (+ 3 (prod 3 0)))	Evaluate Subtraction
10.	(+ 3 (+ 3 (if (= 0 0) 0 (+ 3 (prod 3 (− 0 1))))))	Apply Def. of Prod
11.	(+ 3 (+ 3 (if #t 0 (+ 3 (prod 3 (− 0 1))))))	Evaluate Equals
12.	(+ 3 (+ 3 0))	Evaluate If
13.	(+ 3 3)	Evaluate Addition
14.	6	Evaluate Addition

(a) (3 points) Which lines in the proof apply function definitions?

Lines 2, 6, 10

(b) (6 points) **Prove**: The Racket expression (sqr 3) returns 9. The definition of sqr is given below.

```
(define (sqr x) (  x x))
```

1.	(sqr 3)	Premise
2.	(* 3 3)	Apply definition of sqr
3.	9	Evaluate multiplication

Question 3 : 10 points

When writing a proof justifying code works correctly, Equational Reasoning is a crucial skill.

The below is an example of using equational reasoning to show that (prod 4 2) returns 8.

Prove: The Racket expression (= 8 (prod 4 2)) will evaluate to #t.

1.	(= 8 (prod 4 2))	Premise
2.	(= 8 (if (= 2 0) 0 (+ 4 (prod 4 (- 2 1)))))	Apply Def. of Prod
3.	(= 8 (if #f 0 (+ 4 (prod 4 (- 2 1)))))	Evaluate Equals
4.	(= 8 (+ 4 (prod 4 (- 2 1))))	Evaluate If
5.	(= 8 (+ 4 (prod 4 1)))	Evaluate Subtraction
6.	(= 8 (+ 4 (if (= 1 0) 0 (+ 4 (prod 4 (- 1 1))))))	Apply Def. of Prod
7.	(= 8 (+ 4 (if #f 0 (+ 4 (prod 4 (- 1 1))))))	Evaluate Equals
8.	(= 8 (+ 4 (+ 4 (prod 4 (- 1 1)))))	Evaluate If
9.	(= 8 (+ 4 (+ 4 (prod 4 0))))	Evaluate Subtraction
10.	(= 8 (+ 4 (+ 4 (if (= 0 0) 0 (· · ·)))))	Apply Def. of Prod
11.	(= 8 (+ 4 (+ 4 (if #t 0 (· · ·)))))	??
12.	(= 8 (+ 4 (+ 4 0)))	??
13.	(= 8 (+ 4 4))	Evaluate Addition
14.	(= 8 8)	Evaluate Addition
15.	#t	Evaluate Equals

(a) (4 points) The Equational Reasoning on Line 10 has be shorthanded using an Ellipsis · · ·. What should be written to replace the · · · on Line 10 to fully write out the proof?
(Just so that it doesn't look weird, please also write down the enclosing parentheses in your answer)

(= 8 (+ 4 (+ 4 (if (= 0 0) 0 (+ 4 (prod 4 (- 0 1)))))))

(b) (2 points) No Justification has been provided for Line 11. What is the missing justification?

Evaluate Equals

(c) (2 points) No Justification has been provided for Line 12. What is the missing justification?

Evaluate If

(d) (2 points) Was it important to know what was shorthanded by the · · · on lines 10 or 11 to understand the proof? Explain why or why not.

Yes, because firstly by filling the ellipses in line 10, you can show that you know what the definition of prod is as well as how to plug in arguments. On line 11, its also important to know that the information where the ellipses are doesn't change from line 10 to 11 because you are just evaluating the equals function.

Question 4 : 29 points

Complete the following proof by filling in the blanks.

Prove: The Racket expression (prod 0 2) is equals to 0.

1.	(= 0 (prod 0 2))	Premise
2.	(= 0 (if (= 2 0) 0 (+ 0 (prod 0 (- 2 1)))))	Apply Def. of Prod
3.	(= 0 (if **A**))	Evaluate Equals
4.	(= 0 (+ 0 (prod 0 (- 2 1))))	Evaluate If
5.	(= 0 (+ 0 (prod 0 1)))	Evaluate Subtraction
6.	(= 0 (+ 0 (if (= 1 0) 0 (+ 0 (prod 0 (- 1 1))))))	**B**
7.	(= 0 (+ 0 (if #f 0 (+ 0 (prod 0 (- 1 1))))))	Evaluate Equals
8.	(= 0 (+ 0 (+ 0 (prod 0 (- 1 1)))))	Evaluate If
9.	**C**	Evaluate Subtraction
10.	(= 0 (+ 0 (+ 0 (if (= 0 0) 0 (+ 0 (prod 0 (- 0 1)))))))	Apply Def. of Prod
11.	(= 0 (+ 0 (+ 0 (if #t 0 (+ 0 (prod 0 (- 0 1)))))))	Evaluate Equals
12.	(= 0 (+ 0 (+ 0 0)))	Evaluate If
13.	**D**	**E**
14.	(= 0 0)	Evaluate Addition
15.	#t	Evaluate Equals

(a) (4 points) Fill in blank A

(= 0 (if #f 0 (+ 0 (prod 0 (- 2 1)))))

(b) (4 points) Fill in blank B

Apply Def. of prod

(c) (5 points) Fill in blank C

(= 0 (+ 0 (+ 0 (prod 0 0))))

(d) (5 points) Fill in blank D

(= 0 (+ 0 0))

(e) (5 points) Fill in Blank E

Evaluate addition

(f) (6 points) Propose a solution to improve performance when multiplying by zero. You do not need to rewrite the function, just explain how you would change it.

Change the check in the if from (= x 0) to (or (= x 0) (= y 0)). This will result in faster performance when either x or y are 0 instead of just if x is 0.

Question 5 : 15 points

Review the following function

```
(define/contract (f x) (-> positive? positive?)
 (if (equal? x 1)
     x
     (+ x (f (- x 1)))))
```

Prove: The Racket Expression (**equal**? 3 (f 2)) will evaluate to #t.

note that for now you can ignore the contract when applying the definition

1.	(equal? 3 (f 2))	Premise
2.	(equal? 3 (if (equal? 2 1) 2 (+ 2 (f (- 2 1))))))	Apply definition of f
3.	(equal? 3 (if #f 2 (+ 2 (f (- 2 1))))))	Evaluate equal?
4.	(equal? 3 (+ 2 (f (- 2 1))))	Evaluate #f
5.	(equal? 3 (+ 2 (f 1)))	Evaluate minus
6.	(equal? 3 (+ 2 (if (equal? 1 1) 1 (+ 1 (f (- 1 1))))))	Apply definition of f
7.	(equal? 3 (+ 2 (if #t 1 (+ 1 (f (- 1 1))))))	Evaluate equal?
8.	(equal? 3 (+ 2 1))	Evaluate #t
9.	(equal? 3 3)	Evaluate addition
10.	#t	Evaluate equal?

Question 6 : 8 points

Originally, we assumed only positive numbers were given as input to **prod**. We will now see what happens when we evaluate negative numbers.

```
(define/contract (prod x y)(-> integer? integer? integer?)
    (if (zero? y) 0 (+ x (prod x (- y 1)))))
```

Open a Racket Session and define the prod function.

(a) (2 points) Execute (prod 2 2). Does it work correctly? Why or why not?

yes it did work. it returned 4

(b) (2 points) Execute (prod −2 2). Does it work correctly? Why or why not?

yes it did work. it returned -4

(c) (2 points) Execute (prod 2 −2). Does it work correctly? Why or why not?

No, it turned into an infinite loop and ran out of memory.
This happened because the only stop condition was when y was equal to 0, not less than zero.
So the function decrements y infinitely since y was negative during the first call and can never be decremented to zero

(d) (2 points) Execute (prod −2 −2). Does it work correctly? Why or why not?

No, for the same reason above.

Question 7 : 15 points

The prod function below does not quite work correctly for negative numbers.

```
(define/contract (prod x y) (-> integer? integer? integer?) (if (zero? y) 0 (+ x (prod x (− y 1)))))
```

Modify the function to work correctly for negative numbers.
It is helpful to keep in mind that since Racket subtraction is variadic, (- x) would evaluate to the opposite of x.
−

Also, the cond operator may be used in this question, if desired.

```
(cond
        [(Bool Expr1) (Return if Expr1 is True)]
        [(Bool Expr2) (Return if Expr2 is True)]
        ...
        [else (Return if none of the above)]
)
```

Your code should pass the following tests

```
(prod 5 7) ; Returns 35
(prod −2 10) ; Returns −20
(prod 7 −3) ;Returns −21
(prod −3 −6) ; Returns 18
```

```
(define/contract (prod x y)
(-> integer? integer? integer?)
(cond
 [(zero? x) 0]
 [(zero? y) 0]
 [(> y 0) (+ x (prod x (- y 1)))]
 [else (+ (* -1 x) (prod x (+ y 1)))]
 )
)
```