Name and Section: Nick Palacio, CSCI-8000

Instructor: Dr. Gelfond

1. Kdlf
   1. Skd
   2. Lsdjf
2. Ljd
3. Lks
4. Lkajds
5. Ladj
6. Las
7. Laksdj
8. Lksa
9. Tail recursion is a particular application of recursion where the last instruction executed inside the recursive function is a recursive function call to itself. This can be contrasted with ‘standard’ recursion where the last instruction executed would be some other computation. Below I have defined two functions that sum the elements of a list. One is tail recursive (tail-recursive-sum) where we can see that the last function call that will execute is a recursive one. In the non tail recursive function we can see that the last function call is actually doing addition. The main idea to note here which makes the tail recursive implementation much faster is that is does not need to reverse back up the call stack in order to finish its calculation because it utilizes an accumulator to keep track of the current sum value as it moves along.

#lang racket

**(**require racket/trace**)**

**(**trace-define **(**recursive-sum **l)** **(cond**

**[(**null? **(cdr** **l))** **(car** **l)]**

**[**else **(+** **(car** **l)** **(**recursive-sum **(cdr** **l)))]))**

**(**trace-define **(**tail-recursive-sum-helper **l** acc**)** **(cond**

**[(**null? **l)** acc**]**

**[**else **(**tail-recursive-sum-helper **(cdr** **l)** **(+** **(car** **l)** acc**))]))**

**(**trace-define **(**tail-recursive-sum **l)** **(**tail-recursive-sum-helper **l** 0**))**

* 1. We can see the difference in these functions by looking at the invocations we get from using trace-define below. Specifically we can see that the non-tail recursive function has to make the recursive calls all the way down to the base case and then propagate those values back up the call stack. Where as the tail recursive function is actually finished once it hits it’s base case because it has been keeping the accumulator up to date with the current sum of the list so far.

