Javier Escareno
CSCI Lab 12

Question 1 A.)

```
{Browse {Eval minus(7 10)}}
{Browse {Eval plus(plus(5 5) 10)}}
{Browse {Eval times(6 11)}}
```

When *** S *** is replaced the procedure operation in the stack gets popped out and no statements gets executed. The reason is that there is no case function for minus in the eval function. According to the try-catch statement when an operation runs into an error the next operation will pop out of the stack. In this case because of the lack of a case function for minus their is an error on the procedure. The next 2 procedures then get popped off the stack.

Question 1 B.)

```
{Browse {Eval plus(plus(5 5) 10)}}

{Browse {Eval divide(6 0)}}

{Browse {Eval times(6 11)}}
```

When *** S *** is replaced the procedure is going to be passed into the function because theirs is a case function because there is a case function for plus so it's getting passed but the 6/0 isn't getting passed since you can't divide by 0 so it raises an error. The next procedure gets popped off the stack in accordance with the try catch statement.

Question 1 Part 3.)

It isn't possible for the program to raise an error that is not caught by either of the two catch statements because when you get an error on your first try catch statements it wouldn't get passed and the remaining statements would get popped out of the stacks.

Question 2 Part 1.)

Proof for reverse

Reverse : P= {rev (@v) ++ @c = = rev(L)

Loop:

@v =L , @C = Nil

rev (@v) ++ @c = rev(L)++ Nil = rev(L)

Iteration : Let @v = H|T @C' = H|@C @V=T

Assume:

rev(T) ++ @C

rev(T) ++ H|@C : By substitution

rev(T) ++[H] ++ @C : By associativity

rev(H|T) ++@C : By definition

Rev (@V)++ @C: By sub

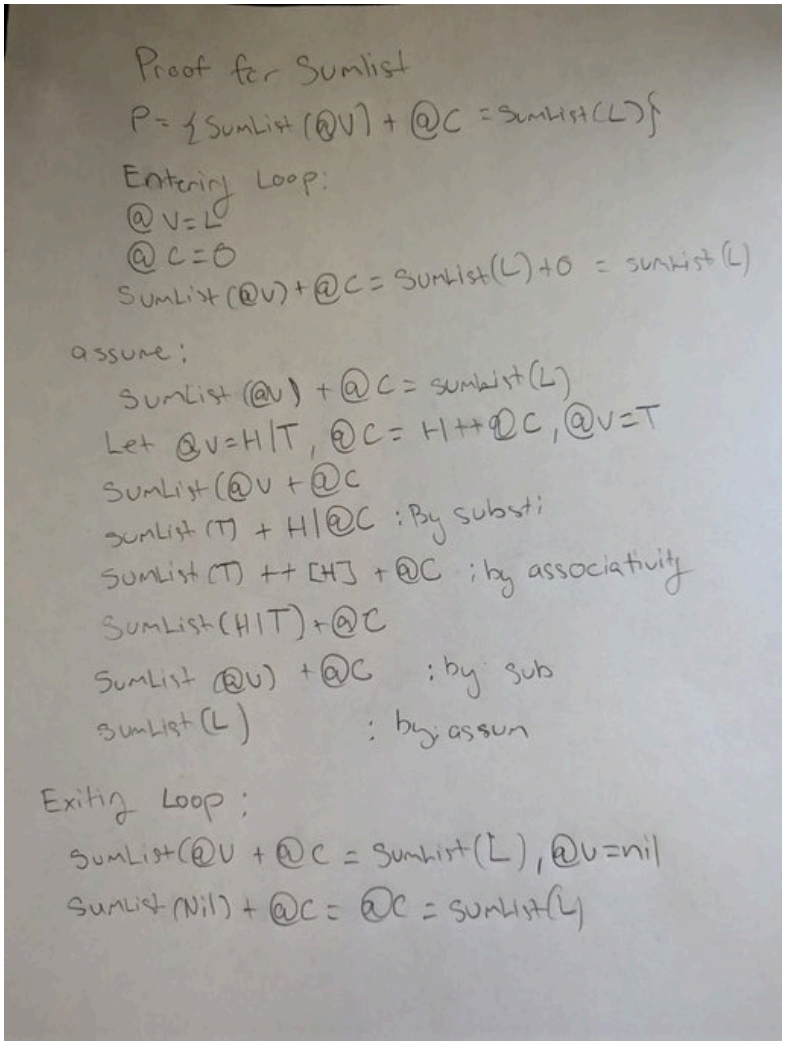
Exit Loop

rev(L)

rev(@v) ++ @c = rev(L), @V=Nil

rev(Nil) ++ @c = @c= rev(L)

Question 2 part 2.)



Question 2 Part 3.)

The insertion sort algorithm works by gradually building a sorted subarray at the beginning of the array. The algorithm maintains certain invariants to ensure correctness. The first invariant (I1) states that at the start of each iteration of the outer loop, the subarray A[0..i-1] is sorted. This is true when the outer loop begins at (i = 1) because a single-element subarray is inherently sorted. The second set of invariants (I2) holds true within the inner loop. When j = i (entering the inner loop for the first time during each iteration of the outer loop), A[0..i-1] is sorted as per I1. Additionally, A[j+1..i] is empty and sorted. As the algorithm proceeds and swaps and adjustments are made this ensures that I1 and I2 remain true ending in a fully sorted array by the end of the algorithm as guaranteed by I1. These invariants ensure that the insertion sort algorithm correctly builds and maintains a sorted array throughout its execution.