Ren Hao Wong
CSCI 117 Lab 12

Part 1:

1.

```
proc {Eval E Out}
  local IsNumE in
{IsNumber E IsNumE}
if IsNumE then E
  else
    try
      case E
      of plus(X Y) then {Eval X} + {Eval Y}
      [] times(X Y) then {Eval X} * {Eval Y}
      [] divide(X Y) then
        try
          if {Eval Y} == 0 then raise divByZero(E) end
          else {Eval X} / {Eval Y} end
           Ill1 = illFormedExpr(E)
        catch Ill1 then
          raise illFormedExpr(E)
        end
        end
      else raise illFormedExpr(E) end
      end
     Ill2 = illFormedExpr(E)
    catch Ill2 then
      raise illFormedExpr(E)
    catch divByZero(D) then
      raise divByZero(D)
    end
    end
  end
end
end

try
  try
    *** S ***
  catch illFormedExpr(E) then
    {Browse E}
  end
```

```
DivZero = divByZero(divide(X _))
catch DivZero then
  {Browse X}
end
```

2.

The program contains two try blocks in its stack. The first try block handles exceptions related to division by zero or malformed expressions, while the second try block wraps the Browse statements. If an exception is raised, it is caught by the inner try block first and then passed to the outer try block if it is not handled. The program terminates if the outer try block fails to catch the exception, in which case all the Browse statements will be executed before the program quits.

In the first Browse statement, "Browse Eval plus(plus(5 5) 10)," the console will display the value 20 as expected. However, the second Browse statement, "Browse Eval divide(6 0)," will throw a divByZero exception when the denominator is zero. This exception will trigger the inner catch block, which will browse the value. As the program has stopped due to the exception raised by the second Browse statement, the third Browse statement, "Browse Eval times(6 11)," will not be executed.

3.

Yes, it is possible for the Eval function to raise additional exceptions. The operations involved within the Eval function such as Browse could contain additional try catches that raise other exceptions. This is done and designed to reduce the responsibility of the end-user for checking for different cases of errors that should raise exceptions.

Part 2:

1.

I1: Reverse(L) = Append(H, C) in H|T = V

2.

I1: C + SumList(V) = SumList(L) is true at every iteration.
C is the cumulative sum thus far and V consists of an array of numbers from L that has yet to be added to C. It is also a true statement before the iteration where C is initialized with 0 when V == L.

3.

Selection Sort:
I1: A[n-i..n-1] is sorted
A[0..n-i-1] <= A[n-i..n-1]

I2: A[bigindex] >= A[0..j-1]

- I1 is true when i = 0
    - A[n..n-1] is an empty range therefore it is sorted
    - A[0..n-1] <= A[n..n-1] is true because the second range is empty
- I2 is true when j = 1
    - A[bigindex] >= A[0..0] is true because the second range is empty
- If I2 is true at location 2, then it is true the next time we reach location 2 (i.e., after a possible reassignment of bigindex and j++).
    - Given a current scenario and the scenario in the following iteration:

      A[bigindex1] >= A[0..j-1] implies A[bigindex2] >= A[0..j]

      where A[bigindex1] <= A[bigindex2]

      The implication at the next iteration increases the RHS range by one element, and since A[bigindex1] has been acknowledged to be larger than or equal to every other element in the range except for the last element, the statement must be true by accounting a possible reassignment of bigindex to ensure the validity, in which it can only be the last element.
- If I1 was true at location 1, and j == n-i, then after i++, I2 implies I1.
    - A[bigindex] >= A[0..j-1] implies A[n-i..n-1] is sorted

      -implies A[0..n-i-1] <= A[n-i..n-1]
- When i = n-1, I1 implies that A[0..n-1] is sorted.
    - A[n-(n-1)..n-1] = A[1..n-1] indicates that all the elements after the first are sorted.
    - A[0..n-(n-1)-1] <= A[n-(n-1)..n-1]
      A[0..0] <= A[1..n-1] indicates that the first element is less than or equal to every other element after it, implying that A[0..n-1] is sorted.