Part 1: Exception Handling

Q1.

⟨s⟩ ::= skip

⟨s⟩1 ⟨s⟩2

local ⟨x⟩ in ⟨s⟩ end

⟨x⟩1 =⟨x⟩2

⟨x⟩=⟨v⟩

if ⟨x⟩ then ⟨s⟩1 else ⟨s⟩2 end

case ⟨x⟩ of ⟨pattern⟩ then ⟨s⟩1 else ⟨s⟩2 end -> 2 cases of exception

{⟨x⟩ ⟨y⟩1 ... ⟨y⟩n}

try ⟨s⟩1 catch ⟨x⟩ then ⟨s⟩2 end

case <X>

<X1>

<X2>

raise ⟨x⟩ end

Q2.

a.

1. Push Semantic Statements:
   - Push the semantic statement "(catch ⟨x⟩ then ⟨s⟩2 end, E)" onto the stack.
   - Push the pair "((⟨s⟩1, E))" onto the stack. In this case, E represents the ill-formed expression "minus(7 10)".
2. Exception Handling:
   - Pop elements off the stack until a catch statement is encountered.
   - Identify the catch handler that matches the pattern "illFormedExpr(E)". In this case, the catch handler is the "(raise ⟨x⟩ end, E)" statement.
3. Error Handling:
   - Execute the catch handler's code, which is "It will print error message E -> “minus(7 10)”".

- This results in printing the error message "minus(7 10)".

The execution terminates after handling the raised exception.

b.

1. Push Semantic Statements:
    - Push the semantic statement "(catch ⟨x⟩ then ⟨s⟩2 end, E)" onto the stack.
    - Push the pair "((⟨s⟩1, E))" onto the stack. In this case, E represents the expression "plus(plus(5 5) 10) {Eval divide(6 0)} {Eval times(6 11)}".
2. Exception Handling:
    - Attempt to evaluate the expression "plus(plus(5 5) 10) {Eval divide(6 0)} {Eval times(6 11)}".
    - Encounter a division by zero error when evaluating "{Eval divide(6 0)}".
3. Error Handling:
    - Pop elements off the stack until a catch statement is encountered or the stack is empty.
    - Identify the catch handler that matches the pattern "illFormedExpr(E)". In this case, the catch handler is the "(raise ⟨x⟩ end, E)" statement.
    - Since the catch handler doesn't handle division by zero errors, the stack is emptied without finding a suitable handler.
4. Uncaught Exception:
    - Terminate execution with the error message "Uncaught exception".
5. Remaining Statements:
    - Despite the uncaught exception, the remaining statements in the original expression, "{Eval times(6 11)}", are not executed.

Therefore, the overall execution results in an "Uncaught exception" error message, and the remaining statements are not executed.

Q3. Yes, it is possible for the Eval function to raise an exception that is not caught by either of the two catch statements. This may happen if the exception is not of the types that the catch statements are meant to handle. For example, if the Eval function tries to evaluate a syntactically wrong expression, it will report a syntax error that will not be caught by the catch statements.

```
(try
    Eval "x + y"
catch
     IllFormedArg(x) then "x is not a number"
     IllFormedArg(y) then "y is not a number"
end)
```

Part 2: Invariants

Q1.

Loop invariant: @C = Reverse(L, V)

Explanation:

This invariant states that the value of cell C at the start of each loop iteration matches to the reverse of the sublist of L starting from cell V.

Initialization: The invariant holds at the start of the loop because C is initialized to nil because the reverse of an empty list is an empty list.

Maintenance:

Assume that the invariant holds before the loop's ith iteration. This means that before the ith iteration, @C = Reverse(L, V).

The loop's ith iteration changes C to H|@C, where H is the head of the current cell V and @C is the value of cell C prior to the update. This means that the new value of C equals the current cell V's head plus the old value of C.

The loop's ith iteration also updates V to T, where T is the tail of the current cell V. That is, the new value of V is the tail of the current cell V.

Since the reverse of a list is the same as the reverse of the tail of the list appended to the head of the list, after the ith iteration, @C = Reverse(L, V).

Termination:

The loop terminates when V = nil, indicating that the current cell V is empty. This signifies that the list has been totally inverted, with C representing the inverse of the entire list L.

As a result, at the end of the loop, the loop invariant holds, and the iterative helper function for Reverse successfully reverses the list L.

Q2.

Loop invariant: @C = SumList(L, V)

Explanation:

This invariant states that the value of cell C at the start of each loop iteration is equal to the sum of the sublist of L beginning with cell V.

Initialization: The invariant holds at the start of the loop since C is set to 0 and the sum of an empty list is 0.
Maintenance:

Assume that the invariant holds before the loop's ith iteration. This means that before the ith iteration, @C = SumList(L, V).

The loop's ith iteration changes C to H + @C, where H is the head of the current cell V and @C is the value of cell C before to the update. This means that the new value of C is the sum of the current cell V's head and the previous value of C.

The loop's ith iteration also updates V to T, where T is the tail of the current cell V. That is, the new value of V is the tail of the current cell V.

Since the sum of a list is the sum of the tail and the head of the list, it follows @C = SumList(L, V) after the ith iteration.

Termination:

The loop ends when V = nil, indicating that the current cell V is empty. This indicates that the list has been fully summed, and the value of C is the total of the entire list L.

As a result, the loop invariant holds at the end of the loop, and the SumList iterative helper function correctly sums the list L.

Q3.

Insertion Sort invariants

I1 is true when i = 1

When i = 1, the insertion sort algorithm's outer loop has just begun. A[0] is the only element entered into the sorted area of the array, and because there is only one element, it is trivially sorted. As a result, I1 is true when i = 1.

I1 implies that I2 is true when j = i

When j = i, the insertion sort algorithm's inner loop has just begun. As established by I1, the component of the array A[0..i-1] is already sorted. Because no elements have been moved to the right of A[i], the array section A[i+1..i] is also sorted. Furthermore, because A[i] is the element being inserted, it must be bigger than or equal to all of the elements in A[0..i-1] and smaller than all of the elements in A[i+1..i]. As a result, I2 is true when j = i.

If I2 is true at location 2, then it is true the next time we reach location 2 (i.e., after the swap and j--)

If I2 is true at position 2, then A[0..j-1] is sorted, A[j+1..i] is sorted, and A[0..j-1] = A[j+1..i]. When we return to location 2, the only thing that may change is the value of A[j]. When we swap, we simply move an element from A[j+1..i] to A[0..j-1], which has no effect on the sorted order of either half of the array. As a result, I2 stays true even after the swap and j--.

If I1 was true at location 1, and the inner loop ends because j == 0, then after i++, I2 implies I1

If I1 was true at location 1, we know that A[0..i-1] is sorted. If the inner loop ends because j == 0, then A[j] = A[j-1] for every j > 0. This indicates that A[0..i-1] is still sorted when the inner loop finishes. As a result, following i++, I2 implies I1.

If I1 was true at location 1, and the inner loop ends because *A[j] >= A[j-1], then after i++, I2 implies I1*

If I1 was true at position 1, then A[0...i-1] is sorted. If the inner loop terminates because *A[j] >= *A[j-1], then A[j] is in the correct place in the array's sorted part. This means that A[0..i-1] is still sorted when the inner loop ends. As a result, I2 implies I1 after i++.

When i = n (i.e., when the outer loop ends), I1 implies that A[0..n-1] is sorted

When i = n, the insertion sort algorithm's outer loop is finished. We know that A[0..n-1] is sorted because I1 is true at this point. As a result, the full array A[] is sorted.