Part 1

a)

```
local SumListS SumList Out1 Out2 in
    fun {SumList L} // Declarative recursive
       case L
       of nil then 0
       [] '|'(1:H 2:T) then (H + {SumList T})
       end
   end

    fun {SumListS L}                          // Stateful iterative C Help in
        newCell 0 C fun {Help L}
        case L
        of nil then @C
        [] '|'(1:H 2:T) then
        C:=(@C+H)
        {Help T}
        end
    end
    {Help L} end

    Out1 = {SumList [1 2 3 4]}
    Out2 = {SumListS [1 2 3 4]}
    skip Browse Out1
    skip Browse Out2
end
```

```
local FoldLS FoldL Out1 Out2 Z in
   fun {FoldL F Z L}                                // Declarative recursive
     case L of nil then Z
     [] '|'(1:H 2:T) then  {FoldL F {F Z H} T}
     end
   end

  fun {FoldLS F Z L} FoldLH C in                   // Stateful iterative
     newCell 0 C
     fun {FoldLH F Z L}
        case L of nil then @C
        [] '|'(1:H 2:T) then
        end
     end
   {FoldLH F Z L} end
   C := {F Z H} {FoldLH F @C T}

   Out1 = {FoldL fun {$ X Y} (X+Y) end 3 [1 2 3 4 5]}
   Out2 = {FoldLS fun {$ X Y} (X+Y) end 3 [1 2 3 4 5]}
   skip Browse Out1
   skip Browse Out2
end
```

Output:

```
*Hoz> runFull "stateful" "Part1a.txt" "Part1a_out.txt"

Out1 : 15

Out2 : 15

Out1 : 18

Out2 : 18
```

b)

```
local SumListS SumList Out1 Out2 in /*
    fun {SumList L}                                    // Declarative recursive case L
      case L of nil then 0
      [] '|'(1:H 2:T) then (H + {SumList T}) end
    end
*/


    fun {SumListS L}                      // Stateful iterative C Help in
    newCell 0 C fun {Help L}
      case L of nil then @C
      [] '|'(1:H 2:T) then C:=(@C+H)
      {Help T} end
    end
    {Help L} end


    Out1 = {SumList [1 2 3 4]}
    Out2 = {SumListS [1 2 3 4]}
    skip Browse Out1
    skip Browse Out2
    skip Full
```

```
 end
local FoldLS FoldL Out1 Out2 Z in /*
   fun {FoldL F Z L}                                         // Declarative recursive
     case L of nil then Z
     [] '|'(1:H 2:T) then {FoldL F {F Z H} T}
     end
   end
*/

   fun {FoldLS F Z L} FoldLH C in                   // Stateful iterative
      newCell 0 C
      fun {FoldLH F Z L}
         case L of nil then @C
         [] '|'(1:H 2:T) then
         end
      end
   {FoldLH F Z L} end
   C := {F Z H} {FoldLH F @C T}

  Out1 = {FoldL fun {$ X Y} (X+Y) end 3 [1 2 3 4]}
  Out2 = {FoldLS fun {$ X Y} (X+Y) end 3 [1 2 3 4]}
  skip Browse Out1
  skip Browse Out2
  skip Full
end
```

Output:

```
*Hoz> runFull "stateful" "Part1b.txt" "Part1b_out.txt"

Out1 : Unbound

Out2 : 10

Store : ((11, 44), 10), ((48, 24), nil()),

((45, 46, 39), 6),

((47, 23, 16), 4),

((43, 22), '|'(1:23 2:24)), ((40, 41, 34), 3),

((42, 21, 15), 3),

((38, 20), '|'(1:21 2:22)),

((35, 36, 29), 1),

((37, 19, 14), 2),

((33, 18), '|'(1:19 2:20)),

((30, 31, 27), 0),

((32, 17, 13), 1),

((28, 12), '|'(1:17 2:18)),

((25), Cell 1),

((26), proc(["L","EXU2"],[case L of nil() then [{Exchange A EXU2 EXU2}] else [case L of
'|'(1:H 2:T) then [local ["SCU3","GarbU3"] [local ["EXU4","EXU5"] [{Exchange A EXU4
EXU4},EXU5 = H,"IntPlus" "EXU4" "EXU5" "SCU3"],{Exchange A GarbU3
SCU3}],local ["EXU3"] [EXU3 = T,"Help" "EXU3" "EXU2"]] else [skip]]],[("A",25),
("IntPlus",1),("Help",26)])),

((8), proc(["L","EXU1"],[local ["A","Help"] [local ["NCU2"] [NCU2 = 0,{NewCell NCU2
A}],Help = proc {$ L EXU2} [case L of nil() then [{Exchange A EXU2 EXU2}] else [case
L of '|'(1:H 2:T) then [local ["SCU3","GarbU3"] [local ["EXU4","EXU5"] [{Exchange A
EXU4 EXU4},EXU5 = H,"IntPlus" "EXU4" "EXU5" "SCU3"],{Exchange A GarbU3
SCU3}],local ["EXU3"] [EXU3 = T,"Help" "EXU3" "EXU2"]] else [skip]]],local ["EXU2"]
[EXU2 = L,"Help" "EXU2" "EXU1"]]],[("IntPlus",1)])),
((9), Unbound),
((10), Unbound),
((1), Primitive Operation),
((2), Primitive Operation),
((3), Primitive Operation), ((4), Primitive Operation), ((5), Primitive Operation), ((6),
Primitive Operation), ((7), Primitive Operation)
Mutable Store: (1 -> 11)
Current Environment : ("SumListS" -> 8, "SumList" -> 9, "Out1" -> 10, "Out2" -> 11,
"IntPlus" -> 1, "IntMinus" -> 2, "Eq" -> 3, "GT" -> 4, "LT" -> 5, "Mod" -> 6, "IntMultiply"
-> 7)
Stack : "local [\"FoldLS\",\"FoldL\",\"Out1\",\"Out2\",\"Z\"] [FoldLS = proc {$ F Z L
EXU1} [local [\"FoldLH\",\"A\"] [local [\"NCU2\"] [NCU2 = 0,{NewCell NCU2
A}],FoldLH = proc {$ F Z L EXU2} [case L of nil() then [{Exchange A EXU2 EXU2}] else
[case L of '|'(1:H 2:T) then [local [\"SCU3\",\"GarbU3\"] [local [\"EXU4\",\"EXU5\"]
[EXU4 = Z,EXU5 = H,\"F\" \"EXU4\" \"EXU5\" \"SCU3\"],{Exchange A GarbU3
SCU3}],local [\"EXU3\",\"EXU4\",\"EXU5\"] [EXU3 = F,{Exchange A EXU4
EXU4},EXU5 = T,\"FoldLH\" \"EXU3\" \"EXU4\" \"EXU5\" \"EXU2\"]] else [skip]]],local
[\"EXU2\",\"EXU3\",\"EXU4\"] [EXU2 = F,EXU3 = Z,EXU4 = L,\"FoldLH\" \"EXU2\"
\"EXU3\" \"EXU4\" \"EXU1\"]]],local [\"EXU1\",\"EXU2\",\"EXU3\"] [EXU1 = proc {$ X
Y EXU4} [local [\"EXU5\",\"EXU6\"] [EXU5 = X,EXU6 = Y,\"IntPlus\" \"EXU5\"
\"EXU6\" \"EXU4\"]],EXU2 = 3,local [\"EXU4\",\"EXU5\",\"EXU6\",\"EXU7\"] [EXU4 =
1,EXU5 = 2,EXU6 = 3,EXU7 = 4,local [\"EXU8\",\"EXU9\"] [EXU8 = EXU4,local
[\"EXU10\",\"EXU11\"] [EXU10 = EXU5,local [\"EXU12\",\"EXU13\"] [EXU12 =
EXU6,local [\"EXU14\",\"EXU15\"] [EXU14 = EXU7,EXU15 = nil(),EXU13 =
'|'(1:EXU14 2:EXU15)],EXU11 = '|'(1:EXU12 2:EXU13)],EXU9 = '|'(1:EXU10
2:EXU11)],EXU3 = '|'(1:EXU8 2:EXU9)]],\"FoldLS\" \"EXU1\" \"EXU2\" \"EXU3\"
\"Out2\"],skip/BOut1,skip/BOut2,skip/f]"
Out1 : Unbound Out2 : 13
Store : ((52, 109, 102), 13),
((110, 68), nil()),
((108, 99, 90, 81, 72, 54), proc(["X","Y","EXU4"],[local ["EXU5","EXU6"] [EXU5 =
X,EXU6 = Y,"IntPlus" "EXU5" "EXU6" "EXU4"]],[("IntPlus",1)])),
((103, 106, 104, 100, 93), 9),
((107, 105, 67, 60), 4),
((101, 66), '|'(1:67 2:68)),
((94, 97, 95, 91, 84), 6),
((98, 96, 65, 59), 3),
((92, 64), '|'(1:65 2:66)),
((85, 88, 86, 82, 75), 4),
((89, 87, 63, 58), 2),
((83, 62), '|'(1:63 2:64)),
((76, 71), 0),
((80, 78, 61, 57), 1),
((79, 77, 73, 55), 3),
((74, 56), '|'(1:61 2:62)),
((69), proc(["F","Z","L","EXU2"],[case L of nil() then [{Exchange A EXU2 EXU2}] else
[case L of '|'(1:H 2:T) then [local ["SCU3","GarbU3"] [local ["EXU4","EXU5"] [EXU4 =
Z,EXU5 = H,"F" "EXU4" "EXU5" "SCU3"],{Exchange A GarbU3 SCU3}],local
["EXU3","EXU4","EXU5"] [EXU3 = F,{Exchange A EXU4 EXU4},EXU5 = T,"FoldLH"
"EXU3" "EXU4" "EXU5" "EXU2"]] else [skip]]],[("A",70),("FoldLH",69)])),
((70), Cell 2),
((49), proc(["F","Z","L","EXU1"],[local ["FoldLH","A"] [local ["NCU2"] [NCU2 = 0,
{NewCell NCU2 A}],FoldLH = proc {$ F Z L EXU2} [case L of nil() then [{Exchange A
EXU2 EXU2}] else [case L of '|'(1:H 2:T) then [local ["SCU3","GarbU3"] [local
["EXU4","EXU5"] [EXU4 = Z,EXU5 = H,"F" "EXU4" "EXU5" "SCU3"],{Exchange A
GarbU3 SCU3}],local ["EXU3","EXU4","EXU5"] [EXU3 = F,{Exchange A EXU4
EXU4},EXU5 = T,"FoldLH" "EXU3" "EXU4" "EXU5" "EXU2"]] else [skip]]],local
["EXU2","EXU3","EXU4"] [EXU2 = F,EXU3 = Z,EXU4 = L,"FoldLH" "EXU2" "EXU3"
"EXU4" "EXU1"]]],[])),
((50), Unbound),
((51), Unbound),
((53), Unbound),
((11, 44), 10),
((48, 24), nil()),
((45, 46, 39), 6),
((47, 23, 16), 4),
((43, 22), '|'(1:23 2:24)),
((40, 41, 34), 3),
((42, 21, 15), 3),
((38, 20), '|'(1:21 2:22)),
((35, 36, 29), 1),
((37, 19, 14), 2),
((33, 18), '|'(1:19 2:20)),
((30, 31, 27), 0),
((32, 17, 13), 1),
((28, 12), '|'(1:17 2:18)),
((25), Cell 1),
((26), proc(["L","EXU2"],[case L of nil() then [{Exchange A EXU2 EXU2}] else [case L of
'|'(1:H 2:T) then [local ["SCU3","GarbU3"] [local ["EXU4","EXU5"] [{Exchange A EXU4
EXU4},EXU5 = H,"IntPlus" "EXU4" "EXU5" "SCU3"],{Exchange A GarbU3
SCU3}],local ["EXU3"] [EXU3 = T,"Help" "EXU3" "EXU2"]] else [skip]]],[("A",25),
("IntPlus",1),("Help",26)])),
((8), proc(["L","EXU1"],[local ["A","Help"] [local ["NCU2"] [NCU2 = 0,{NewCell NCU2
A}],Help = proc {$ L EXU2} [case L of nil() then [{Exchange A EXU2 EXU2}] else [case
L of '|'(1:H 2:T) then [local ["SCU3","GarbU3"] [local ["EXU4","EXU5"] [{Exchange A
EXU4 EXU4},EXU5 = H,"IntPlus" "EXU4" "EXU5" "SCU3"],{Exchange A GarbU3
SCU3}],local ["EXU3"] [EXU3 = T,"Help" "EXU3" "EXU2"]] else [skip]]],local ["EXU2"]
```

```
[EXU2 = L,"Help" "EXU2" "EXU1"]]],[("IntPlus",1)])),

((9), Unbound),

((10), Unbound),

((1), Primitive Operation), ((2), Primitive Operation), ((3), Primitive Operation), ((4),

Primitive Operation), ((5), Primitive Operation), ((6), Primitive Operation), ((7), Primitive

Operation)

Mutable Store: (2 -> 52, 1 -> 11)

Current Environment : ("FoldLS" -> 49, "FoldL" -> 50, "Out1" -> 51, "Out2" -> 52, "Z" ->

53, "IntPlus" -> 1, "IntMinus" -> 2, "Eq" -> 3, "GT" -> 4, "LT" -> 5, "Mod" -> 6,

"IntMultiply" -> 7) Stack : ""
```

Description: The SumList and FoldL functions are offered in both declarative and stateful versions. The declarative SumList function returns the sum of a list [1 2 3 4], and the stateful version, using an iterative method with a local memory cell, also delivers the correct output. FoldL functions are implemented the same, with declarative and stateful versionss.

However, in the declarative version, the mutable storage is empty. Furthermore, it lacks the phrases NewCell and ExChange. In the stateful variant, the mutable store is not empty. Keywords such as NewCell and ExChange are also present.

PART 2

```
local Generate Num GenF Out1 Out2 Out3
    in fun {Generate}
    Num = newCell -1
    fun {$}
       Num:= (@Num + 1)
      @Num
       end
    end

    GenF = {Generate}
    Out1 = {GenF}
    Out2 = {GenF}
    Out3 = {GenF}
     skip Browse Out1
     skip Browse Out2
     skip Browse Out3
end

local Client GenF Sum in
    GenF = {Generate}
    fun {Client} Value in
        Value = {GenF}
        if (Value > 100)
        then 0
        else (Value + {Client})
        end
     end
    Sum = {Client}

    skip Browse Sum

    end

end
```

Output:

```
*Hoz> runFull "stateful" "Part2.txt" "Part2out.txt"

Out1 : 0

Out2 : 1

Out3 : 2
```

PART 3

a)

```
local NewQueue S Pu Po IsE Av A1 A2 B1 B2 V1 V2 V3 Out Append Out1 in Append =
    fun {$ Ls Ms}
        case Ls of nil then (Ms|nil)
        [] '|'(1:X 2:Lr) then Y in
        end
    end
    fun {NewQueue L}
        C = newCell nil
        Y = {Append Lr Ms} (X|Y)
        S = newCell 0
        Push Pop IsEmpty SlotsAvailable in
        proc {Push X}
        if (@S==L) then
        B = @C in
        case B of '|'(1:Y 2:S1) then C:=S1 end
        C:={Append @C X }
        S:=(@S+1)
        else
        C:={Append @C X}
         S:=(@S+1) end
    end
   fun {Pop} B = @C in
        case B of '|'(1:X 2:S1) then C:=S1 X end
    end
    fun {IsEmpty} (@C==nil) end
    fun {SlotsAvailable} B in B = (L - @S)
    B end
    ops(push:Push pop:Pop isEmpty:IsEmpty avail:SlotsAvailable) end
    S = {NewQueue 2}
    S = ops(push:Pu pop:Po isEmpty:IsE avail:Av) B1 = {IsE}
    A1 = {Av}
    {Pu 1}
    {Pu 2}
    A2 = {Av}
    {Pu 3}
    B2 = {IsE}
    V1 = {Po}
    V2 = {Po}
    V3 = {Po}
    Out = [V1 V2 V3 B1 B2 A1 A2]
    skip Browse Out
end
```

Output:

```
*Hoz> runFull "stateful" "Part3.txt" "Part3out.txt"
Out:[2 3 Unbound true() false() 2 0]
```

Trial and error for different Queue sizes:

```
*Hoz> runFull "stateful" "Part3.txt" "Part3out.txt" Out:[1 2 3 true() false() 3 1]
*Hoz> runFull "stateful" "Part3.txt" "Part3out.txt" Out:[1 2 3 true() false() 5 3]
*Hoz> runFull "stateful" "Part3.txt" "Part3out.txt" Out : [ 2 3 Unbound true() false() 1 -1 ]
```

b)

The code implements a queue data structure with push and pop operations, as well as tools to determine whether the queue is empty and the number of available slots. The encapsulation and controlled access to the underlying data structure make this Abstract Data Type (ADT) secure. The queue's internal status is handled via local cells (C, S) and methods (Push, Pop, IsEmpty, SlotsAvailable) and are not accessible outside the function. This encapsulation prevents direct external access to the queue's internal representation, ensuring that queue operations are performed via well-defined interfaces (push and pop methods). This helps to prevent unauthorized or unintentional changes to the queue's status. The code also adds boundary condition checks, such as checking if the queue is empty before popping elements and confirming slot availability before pushing new elements. These contribute to the ADT's security by preventing frequent failures and ensuring the queue's state integrity.

c)

The stack is generated or instantiated every time the StackOps function is run in the declarative version, causing the store's size and memory usage to rise dynamically. Instead of operating on a specific object, more memory is used to construct a modified copy of the ADT, resulting in several instances of the ADT object.

However, in the stateful version, new operators just need to have their value updated rather than being produced each time. It creates an object instance for which each ADT operation requires unique names that are unique to this object. This **avoids** a considerable increase in storage (memory usage). As a result, this version takes less memory than the declarative version.