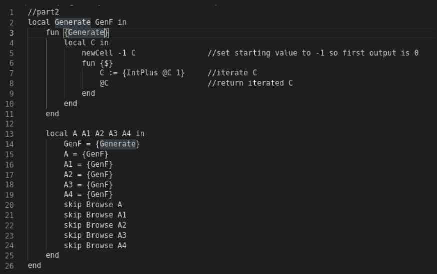
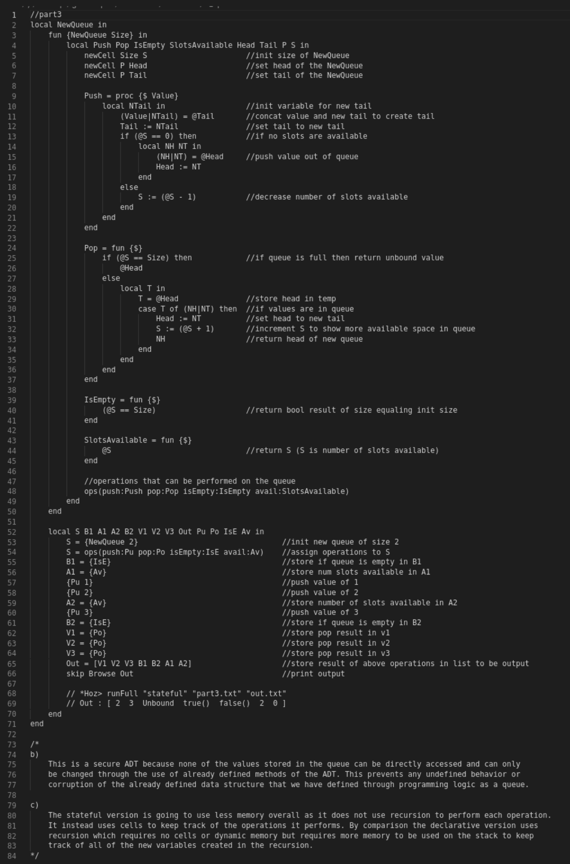
```
//part1
local SumList SumListS Out1 Out2 in
    //---------------SumList---------------

    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
        local SumListIt C in
            newCell 0 C
            SumListIt = proc {$ Ls}
                case Ls of '|'(1:X 2:Xs) then
                    C := {IntPlus @C X}
                    {SumListIt Xs}
                end
            end
            {SumListIt L}
            @C
        end
    end

    ////testing SumList
    //local A A1 B B1 in
    //    A = (1|(2|(3|nil)))         //test arrays
    //    A1 = (4|(2|(-7|(10|nil))))
    //
    //    B = {SumListS A}            //test array A
    //    B1 = {SumListS A1}           //test array A1

    //    skip Browse B
    //    skip Browse B1
    //end

    //test results
    // *Hoz> runFull "stateful" "part1.txt" "out.txt"
    // B : 6
    // B1 : 9

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

local FoldL FoldLS Out1 Out2 in
    //---------------FoldL---------------

    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 O Z L}
        local FoldLIt C in
            newCell Z C
            FoldLIt = proc {$ Op Z Ls}
                case Ls of '|'(1:X 2:Xs) then
                    C := {Op @C X}
                    {FoldLIt Op Z Xs}
                end
            end
            {FoldLIt O Z L}
            @C
        end
    end

    ////testing FoldL
    // local A A1 B B1 in
    //     A = (1|(2|(3|nil)))         //test arrays
    //     A1 = (4|(2|(-7|(10|nil))))

    //     B = {FoldLS IntPlus 0 A}           //test array A
    //     B1 = {FoldLS IntMultiply 1 A1}     //test array A1

    //     skip Browse B
    //     skip Browse B1
    // end

    //test results
    // *Hoz> runFull "stateful" "part1.txt" "out.txt"
    // B : 6
    // B1 : -560

    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

// Looking at the declarative recursive version of each function compared to the stateful iterative it is evident
// that they are working differently. The declarative versions use less store variables as they do not have
// a function and a procedure to deal with, rather having only one function. The stateful versions however have
// an extra procedure which causes there to be more store variables used.

// There is also a difference in the mutable store. The declarative functions do not use the mutable store
// as they use no cells. The stateful functions do use the mutable store though as they create a cell which
// is mutable.
```



```
//part2
local Generate GenF in
    fun {Generate}
        local C in
            newCell -1 C                //set starting value to -1 so first output is 0
            fun {$}
                C := {IntPlus @C 1}     //iterate C
                @C                      //return iterated C
            end
        end
    end

    local A A1 A2 A3 A4 in
        GenF = {Generate}
        A = {GenF}
        A1 = {GenF}
        A2 = {GenF}
        A3 = {GenF}
        A4 = {GenF}
        skip Browse A
        skip Browse A1
        skip Browse A2
        skip Browse A3
        skip Browse A4
    end
end
```



```
//part3
local NewQueue in
    fun {NewQueue Size} in
        local Push Pop IsEmpty SlotsAvailable Head Tail P S in
            newCell Size S                      //init size of NewQueue
            newCell P Head                      //set head of the NewQueue
            newCell P Tail                      //set tail of the NewQueue

            Push = proc {$ Value}
                local NTail in                  //init variable for new tail
                    (Value|NTail) = @Tail       //concat value and new tail to create tail
                    Tail := NTail               //set tail to new tail
                    if (@S == 0) then           //if no slots are available
                        local NH NT in
                            (NH|NT) = @Head     //push value out of queue
                            Head := NT
                        end
                    else
                        S := (@S - 1)           //decrease number of slots available
                    end
                end
            end

            Pop = fun {$}
                if (@S == Size) then            //if queue is full then return unbound value
                    @Head
                else
                    local T in
                        T = @Head               //store head in temp
                        case T of (NH|NT) then  //if values are in queue
                            Head := NT          //set head to new tail
                            S := (@S + 1)       //increment S to show more available space in queue
                            NH                  //return head of new queue
                        end
                    end
                end
            end

            IsEmpty = fun {$}
                (@S == Size)                    //return bool result of size equaling init size
            end

            SlotsAvailable = fun {$}
                @S                              //return S (S is number of slots available)
            end

            //operations that can be performed on the queue
            ops(push:Push pop:Pop isEmpty:IsEmpty avail:SlotsAvailable)
        end
    end

    local S B1 A1 A2 B2 V1 V2 V3 Out Pu Po IsE Av in
        S = {NewQueue 2}                                //init new queue of size 2
        S = ops(push:Pu pop:Po isEmpty:IsE avail:Av)    //assign operations to S
        B1 = {IsE}                                      //store if queue is empty in B1
        A1 = {Av}                                       //store num slots available in A1
        {Pu 1}                                          //push value of 1
        {Pu 2}                                          //push value of 2
        A2 = {Av}                                       //store number of slots available in A2
        {Pu 3}                                          //push value of 3
        B2 = {IsE}                                      //store if queue is empty in B2
        V1 = {Po}                                       //store pop result in v1
        V2 = {Po}                                       //store pop result in v2
        V3 = {Po}                                       //store pop result in v3
        Out = [V1 V2 V3 B1 B2 A1 A2]                    //store result of above operations in list to be output
        skip Browse Out                                 //print output

        // *Hoz> runFull "stateful" "part3.txt" "out.txt"
        // Out : [ 2  3  Unbound  true()  false()  2  0 ]
    end
end

/*
b)
    This is a secure ADT because none of the values stored in the queue can be directly accessed and can only
    be changed through the use of already defined methods of the ADT. This prevents any undefined behavior or
    corruption of the already defined data structure that we have defined through programming logic as a queue.

c)
    The stateful version is going to use less memory overall as it does not use recursion to perform each operation.
    It instead uses cells to keep track of the operations it performs. By comparison the declarative version uses
    recursion which requires no cells or dynamic memory but requires more memory to be used on the stack to keep
    track of all of the new variables created in the recursion.
*/
```