Ren Hao Wong
CSCI 117 Lab 11

Part 1

a)

```
local SumListS SumList Out1 Out2 in

    // Declarative Recursive
    fun {SumList L}
        case L
        of nil then 0
        [] (X|Xs) then (X + {SumList Xs})
        end
    end

    // Stateful Iterative
    fun {SumListS L} C Go in
        newCell 0 C

        proc {Go L}
            case L of (X|Xs) then
                C := (@C + X)
                {Go Xs}
            end
        end

        {Go L}
        @C
    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 in

    // Declarative Recursive
    fun {FoldL Function Value L}
        case L
```

```
            of nil then Value
            [] (X|Xs) then {FoldL Function {Function Value X} Xs}
        end
    end

    // Stateful Iterative
    fun {FoldLS Function Value L} C Go in
        newCell Value C

        proc {Go L}
            case L of (X|Xs) then
                C := {Function @C X}
                {Go Xs}
            end
        end

        {Go L}
        @C
    end

    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
end
```

b)

Using `skip Full`, the stateful function variants show the existence of a mutable store in the program. Both stateful functions of `SumList` and `FoldLS` have a mutable store labeled '1' and point to address 11, which is the address that is bounded to the output value.

Part 2

```
local Generate in
    fun {Generate} C in
        newCell 0 C
        fun{$}
            C := (@C + 1)
            @C
        end
    end

    local GenF Out1 Out2 Out3 in
        GenF = {Generate}
```

```
        Out1 = {GenF} // returns 1
        Out2 = {GenF} // returns 2
        Out3 = {GenF} // returns 3
        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
```

Part 3

a)

```
local NewQueue Out in
    fun {NewQueue Capacity}
        local Content Size Push Pop IsEmpty SlotsAvailable in
            proc {Push Value} C = @Content NewBack in
                if (@Size < Capacity) then
                    case C of (List # Back) then
                        Size := (@Size + 1)
                        Back = (Value|NewBack)
                        Content := (List # NewBack)
                    end
                end
            end

            fun {Pop} C = @Content Ub in
                if (@Size > 0) then
```

```
                    case C of (List # Back) then
                        case List of (X|Xs) then
                            Size := (@Size - 1)
                            Content := (Xs # Back)
                            X
                        end
                    end
                end
            end

            fun {IsEmpty} (@Size == 0) end

            fun {SlotsAvailable} (Capacity - @Size) end

            local Ub in
                newCell (Ub # Ub) Content
            end
            newCell 0 Size
            ops(push:Push pop:Pop isEmpty:IsEmpty
avail:SlotsAvailable)
        end
    end

    local S Pu Po IsE Av A1 A2 B1 B2 V1 V2 V3 Out in
        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   // Out : [ 2  3  Unbound  true()  false()
2  0 ]
    end
end
```

b)

This is a secure ADT because the variables and cells of the queue are inaccessible outside of the function, and the client can only utilize the provided functions to modify the queue correctly. This ensures that the client cannot harm the variable and cell structure that keeps the queue working, such as maintaining the state of the content as a difference list.

c)

The secure stateful ADT creates an object instance of the ADT and requires new names for each ADT operation that apply to this specific object instance. Because the operations apply directly to a specific object, it is much more memory efficient than the secure declarative variant. The secure declarative ADT on the other hand provides generalized operations that take in an ADT object as an argument to perform the operations on. This uses more memory because rather than operating on a specific object, the operations return a modified copy of the ADT, creating many instances of the ADT object.