Eric Smrkovsky
CSci 117
Lab 13
11/17/2019

1a)

```
local SumList Sum in

  fun {SumList L}            //SumList function
    S = newCell 0            //new cell (contents are the sum) starts at 0
    SumListH                 //declare helper
  in
    fun {SumListH L}         //begin function declaration
      case L of nil then @S  //after traversing list return cell contents
      [] '|'(1:H 2:T) then   //break up list to a head and tail
        S := (@S+H)          //cell contents = old contents + head
        {SumListH T}         //recursive call
      end                    //end case statement
    end                      //end helper function
    {SumListH L}             //call helper function
  end

  Sum = {SumList [1 2 3 4]}
  skip Browse Sum
end
  fun {FoldL F Z L}        //FoldL function
    C = newCell 0          //create a new cell starting at zero
    Help in                //declare helper function
    fun {Help F Z L}       //begin function declaration
      case L of nil then   //if list is empty
        C := {F Z @C}      //add Z to cell contents
        @C                 //return cell contents
      [] '|'(1:H 2:T) then //if L is a list break it up into H and T
        C := {F H @C}      //replace cell contents with X
        {Help F Z T}       //recursive call passing tail and original args
      end                  //end case statement
    end                    //end helper function
    {Help F Z L}           //call helper function
  end                      //end FoldL function

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

Eric Smrkovsky
CSci 117
Lab 13
11/17/2019

1b)

The first thing that I notice is that the declarative version of SumList has an empty mutable store and the mutable store in the stateful version has the has the store locations (1 -> 9). The declarative only needs to create store locations for the members of the list once during creation and once during the function execution using a total of 30 store locations. It is a different story with the execution of the stateful version.

When the stateful SumList program executes the mutable store starts off empty. Next when the function is called it uses 20 store locations for the mutable store, locations 10-18 used for the list and number 19 assigned to the cell. The program then runs the helper function 4 times recursively and returns. With each recursion the mutable store increases by 5 and new store locations are needed for the newly split lists and the operations that occur before the cell is updated. After the function has completed its execution the mutable store becomes (1->9) for the statements available in the environment and the store is left untouched.

With the FoldL functions it's the same thing, more store locations are needed for when the cell is updated. These 2 programs use about double the amount of store locations because of the use of the function that is passed into the FoldL function. Also, you see that the only time that the mutable store is used is when the cell is updated.

2a)

```
local Generate in
   fun {Generate}              //Generate function
      G = newCell 0            //create new cell
      GenF in                  //declare helper function
      fun {GenF}               //start function definition
         X=@G in               //assign X current value of cell
         G := (@G +1)          //update cell contents
         X                     //output old cell contents
      end                      //end helper function
      GenF                     //output helper function
   end

//Testing
   local GenF X Y Z in
   GenF = {Generate}
   X = {GenF}
   Y = {GenF}
   Z = {GenF}
   skip Browse X
   skip Browse Y
   skip Browse Z

   end
end
```

2b)

There is no way to access the previous values because each time the function Generate is called it updates the contents of the cell, causing the old contents to be lost forever.

Eric Smrkovsky
CSci 117
Lab 13
11/17/2019

3a)

```
local NewQueue S Pu Po IsE Av B1 B2 V1 V2 V3 A1 A2 Out in

  fun {NewQueue Size}
    Q = newCell nil
    S = newCell Size
    Push Pop IsEmpty SlotsAvailable Out O2 in

    proc {Push X}
    if (@S>0) then Q:=(X|@Q) S:=(@S-1)
    else Out = {Pop} Q:=(X|@Q) O2=@Q end
    end

    fun {SlotsAvailable} @S end
    fun {IsEmpty} (@Q==nil) end
    fun {Pop} B = @Q Rev in //START POP

        fun {Rev Xs}
        Rs = newCell nil
        ReverseH in
        proc {ReverseH L}
          case L of nil then skip Basic
          [] '|'(1:H 2:T) then Rs := (H|@Rs) {ReverseH T}
          end
        end
        {ReverseH Xs}
        @Rs
        end

        case B of '|'(1:X 2:S1) then HT L in
          HT = {Rev B}
        case HT of '|'(1:H 2:T) then L=T Q:={Rev T} H end
      end
    end   //END POP

    ops(push:Push pop:Pop isEmpty:IsEmpty avail:SlotsAvailable)
  end
```

Eric Smrkovsky
CSci 117
Lab 13
11/17/2019

```
  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}
  //A2 = {Av}
  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
```

3b) Because all the operations that access the data within the Queue are hidden within the implementation of the function NewQueue and inaccessible from the rest of the program.

3c) The function Stackops does operations on its data within its internal functions by calling itself recursively. In my queue program the operations on the data are done by utilizing cells. Each time a cell is updated it frees the previously used memory up. Also, when the memory is being accessed within the NewQueue function it is within a cell, but the memory is passed into the functions when Stack Ops is called recursively. Using cells is cheaper because the some of the memory ends up being freed up.