Madhuri Pyreddy
20 November 2019

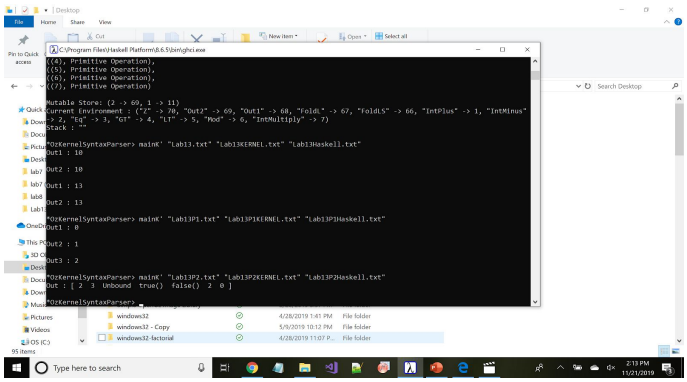
CSCI 117 Lab 13

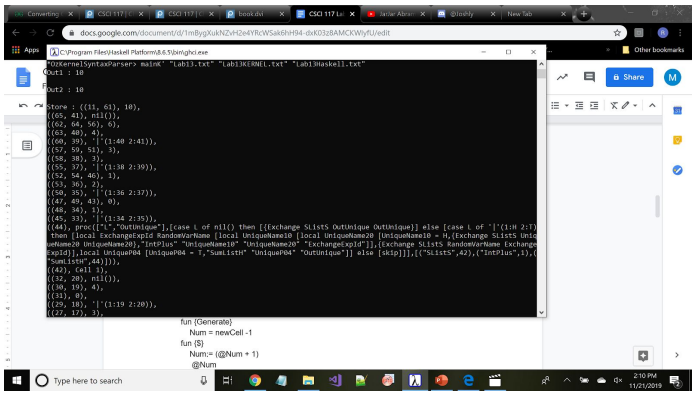
1.

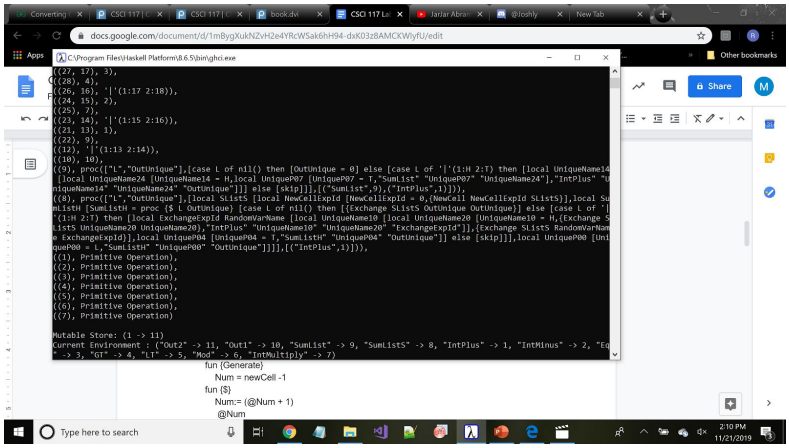
a.
```
local SumListS SumList Out1 Out2 in
        fun {SumList L}
          case L of nil then 0
          [] '|'(1:H 2:T) then (H + {SumList T})
          end
        end
        fun {SumListS L}
        SListS = newCell 0
        SumListH in
        fun {SumListH L}
        case L of nil then @SListS
        [] '|'(1:H 2:T) then SListS := (H + @SListS) {SumListH T}
        end
     end
        {SumListH 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}
          case L of nil then Z
          [] '|'(1:H 2:T) then
                {FoldL F {F Z H} T}
          end
        end
        fun {FoldLS F Z L}
        FLS  = newCell 0
        FoldLH in
        proc {FoldLH F Z L}
        case L of nil then FLS := @FLS
        [] '|'(1:H 2:T) then
        FLS := 0
        FLS := {F Z H}
```

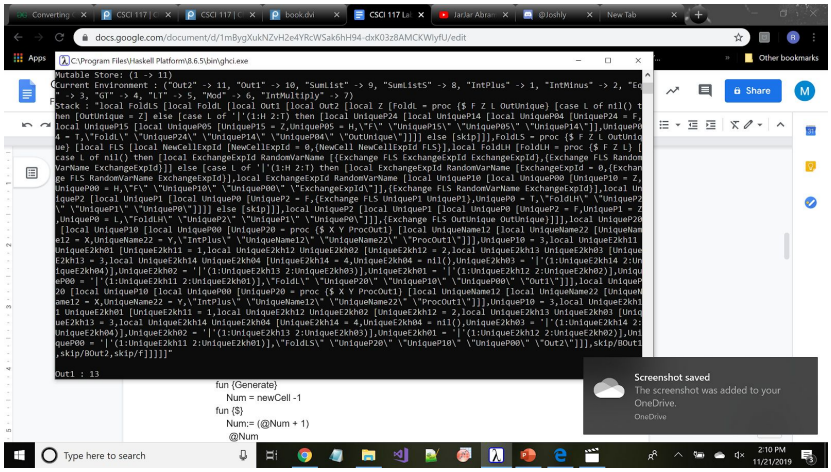
```
            {FoldLH F @FLS T}
            end
        end
            {FoldLH F Z L}
            @FLS
        end
            Out1 = {FoldL fun {$ X Y} (X+Y) end 3 [1 2 3 4]}
            Out2 = {FoldLS fun {$ X Y} (X+Y) end 3 [1 2 3 4]}
            skip Browse Out1
            skip Browse Out2
            skip Full
    end
```

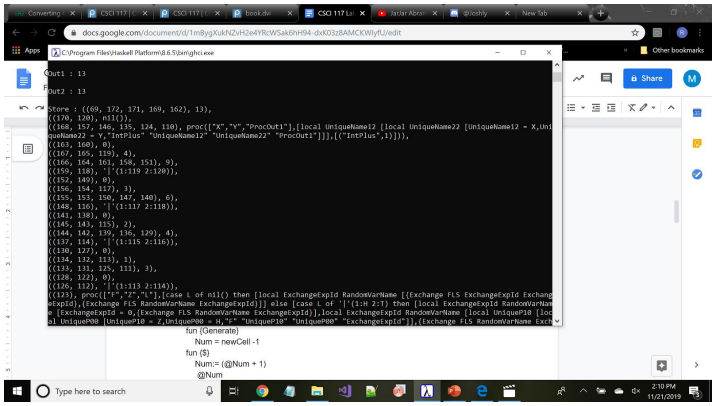
b. When you add Skip Full in SumList and FoldL, it will give the values of Output1 and Output2. But, it will also output kernel syntax, as well as multiple stores and environments for these two functions.







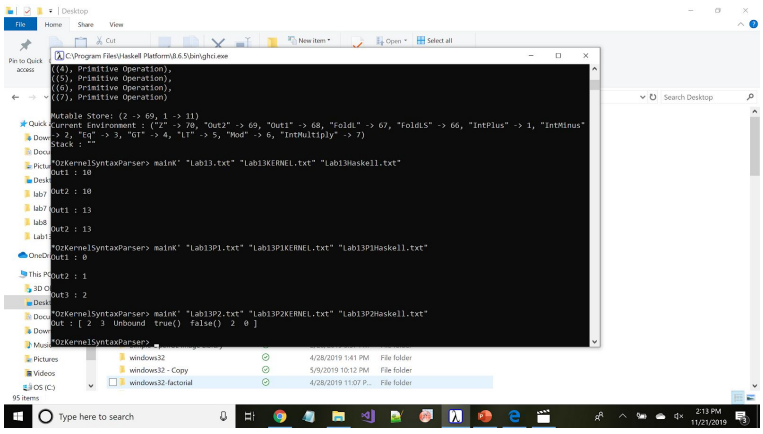




2.

a. 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

b. No, it is not possible to go backwards and recover previous streams because you cannot add possible values after updating with new cells.



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
Y = {Append Lr Ms}
(X|Y)
end
end

```
fun {NewQueue L}
  C = newCell nil
  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
```

b. What makes a secure ADT is information hiding. You cannot change the data in the NewQueue function. You have to have some “token” that will allow you to access the data like Push, Pop, isEmpty, and SlotsAvailable that were not given access to you. In

other words, you are only pushing and popping the elements in the queue without changing the value of the elements in the queue.

c. This compares to a secure declarative ADT on page 431 in relation to memory usage because in secure declarative ADT, it does not use wrapping, since wrapping is only needed for unbundled ADTs. Also, the function with ops takes a list S and returns record of procedure values ops(pop:Pop push:Push isEmpty: isEmpty) in which the list S is hidden by lexical scoping. In other words, it allocates more memory by creating more variables that set equal to the operations that don't return a value.