Javier Escareno
CSCI 117 Lab 11

Part 1A.)

```
local SumListS SumList Out1 Out2 SumList2 in
   fun{SumList L} //Declarative iterative
      case L of nil then 0
      []'|'(1:H 2:T) then (H+ {SumList T})
      end
   end

   fun{SumListS L} //Stateful iterative
   C = newCell 0
   {SumList2 C L}
   end

   fun{SumList2 C L}
      case L of nil then @C
      []'|(1:H 2:T) then C :=(H + @C) {SumList2 C T}
      end
   end

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

end

local foldL2 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{foldL2 F Z L}{
      FL2 = newCell 0
      foldL3 in
      proc {foldL3 F Z L}
      case L of nil then FL2 := @FL2
```

```
        [] '|'(1:H 2:T) then
        FLS := 0
        FLS := {F Z H}

        {foldL3 F @FLS T}
        end
    end

        {foldL3 F Z L}
        @FLS
    end

        Out1 = {foldL fun P {$ X Y} (X + Y) end 3 [1 2 3 4]}
        Out2 = {foldL2 fun {$X Y} (X+Y) end 3 [1 2 3 4]}
        skip Browse Out1
        skip Browse Out2
        skip Full
    }
```

Part1B.)

When using skipfull on the FoldL and SumList it outputs the values under Out1 and Out2 as well as in the Kernal Syntax. This also outputs the stores and the environment for the two functions as well.

Part2.)

```
local Generate Num GenF Out1 Out2 Out3 in

fun{Generate}
        Num = newCell - 1

fun{$}
        Num :=(@Num + 1)
        @Num
        end
end

GenF = {Generate}
Out1 = {GenF} // returns 0
Out2 = {GenF} // returns 1
Out3 = {GenF} // returns 2

skip Browse Out1
skip Browse Out2
skip Browse Out3
```

end

Part 3A.)

```
local NewQueue S Pu PPo IsE Av A1 A2 B1 B2 V1 V2 V3 Out Appened Out 1 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   // Out : [ 2  3  Unbound  true()  false()  2  0 ]

end
```

Part 3B.)

This would be a secure ADT because it is able to hide information. The data stored inside cant be changed unless you have a specific token that gives you access to the data which are : Pop, Push,isEmpty, and SlotsAvailable.

Part 3C.)

In comparison to the ADT given in the book the program given is related to the memory usage that is needed only for unbundled ADTs as wrapping is needed for those types.The functions StackOps takes the list S and returns the values of the procedure operations like push and pop. This can be considered a declarative type of OOP.