// part1

// a)

local SumListS SumList Out1 Out2 in

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 Helper C Out in

newCell 0 C

fun {Helper L C}

case L

of nil then @C

[] '|'(1:H 2:T) then

C := (@C+H)

{Helper T C}

end

end

Out = {Helper L C}

Out

end

end

Out1 = {SumList [1 2 3 4]}

Out2 = {SumListS [1 2 3 4]}

skip Browse Out1

skip Browse Out2

end

local FoldLS FoldL Out1 Out2 in

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 F Z L} // Stateful iterative

local Helper C Out in

newCell Z C

fun{Helper F C L}

case L

of nil then @C

[] '|'(1:H 2:T) then

C := {F @C H}

{Helper F C T}

end

end

Out = {Helper F C L}

Out

end

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)

// I see in the declarative version, function SumList and FoldL were called mutiple times before get the output, and these functions changed

// everytime they were called.

// For the stateful version, function SumListS and FoldLS were called only once before get the output, and these funtions stay the same

// everytime they were called.

//part2

fun {Generate}

local C Gen in

newCell 0 C

Gen = fun {$}

@C

end

C:=(@C+1)

Gen

end

end

//part3

// a)

fun {NewQueue S}

local Front Back Size Pu Po IsE Av in

newCell 0 Size

newCell nil Front

newCell nil Back

Pu = proc {$ N}

if (@Size == 0) then

Front := (N|@Front)

Back := (N|@Back)

Size := (@Size + 1)

else

if {LT @Size S} then

Back := (N|@Back)

Size := (@Size + 1)

else

(H|T) = @Front

(H1|T1) = @Back in

Front := (H1|T)

Back := (N|T1)

end

end

end

Po = fun {$}

if {GT @Size 1} then

(H|T) = @Front

(H1|T1) = @Back in

Front := (H1|T)

Back := T1

Size := (@Size - 1)

H

else

if (@Size == 1) then

(H|T) = @Front in

Size := (@Size -1)

Front := nil

Back := nil

H

end

end

end

IsE = fun {$}

(@Size == 0)

end

Av = fun {$}

(S-@Size)

end

ops(push:Pu pop:Po isEmpty:IsE avail:Av)

end

end

// test

S = {NewQueue 2}

ops (push:Pu pop:Po isEmpty:IsE avail:Av) = S

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 ]

// b)

// It is secure because all variables inside the data structure are declared locally, and not returned in the output.

// and the client only can see the operators of the data structure, not the codes. So, they only can operate the data structure

// but not change the code of the data structure.

// c)

// Compare both declarative ADT on page 431 and the secure ADT relating to memory usage, declarative ADT uses

// (0.02 secs, 14,745,480 bytes) while secure ADT is using (0.02 secs, 16,393,656 bytes). Therefore, secure ADT

// takes more memory usage then the declarative ADT does.