……...PART 1……..

```
local  N  X Y F Z Times Merge Display Xr  Yr G H V1 F1 V2 F3  GG Generate X1 X2 Y1 Y2 Xs
Ys  Two Three Five Take HamingTake DisplayH in

  fun {Generate N}
    fun {$} (N#{Generate (N+1)}) end
  end

//a. Times function

  fun {Times N H}
   //skip Browse N
    fun {$}
      (X#G) = {H} in
      ((N*X)#{Times N G })
     end
  end

//a. Merge function

  fun {Merge Xs Ys}
      fun {$}
        (X1#X2) = {Xs}
        (Y1#Y2) = {Ys} in
          if(X1<Y1) then
             (X1#{Merge X2 Ys})
          elseif (X1>Y1) then
             (Y1#{Merge Xs Y2 })
          else (X1#{Merge X2 Y2})
        end
      end
  end

  GG = {Generate 1}
  Two = {Times 2 GG}  // 2, 4 ,6,...
  Three = {Times 3 GG}   // 3, 6, ...
  Five = {Times 5 GG}   // 5, 10, ...

//Generate Haming Squence
  Z = {Merge Two {Merge Three Five}}

  local
    (V5#F5) = {Z}
```

```
   (V6#F6) = {F5}
   (V7#F7) = {F6}
   (V8#F8) = {F7}
   (V9#F9) = {F8}
   (V10#F10) = {F9}
   (V11#F11) = {F10} in
   skip Browse V5
   skip Browse V6
   skip Browse V7
   skip Browse V8
   skip Browse V9
   skip Browse V10
   skip Browse V11
  end

//b generate hamming sequence and display

   proc {Display X N}
   fun {DisplayH Z Num}
   if (Num == 0) then nil
   else
   (V#F) = {Z} in
   (V|{DisplayH F (Num-1)})
   end
   end
   local L in
   L = {DisplayH X N}
   skip Browse L
   end
   end
  //Testing
  X = {Generate 3}
  Y = {Times 3 X}
  {Display Y 5}


//c. take function to print 10 hamming sequence

  fun {Take N F }
    fun {$}
      if (N==0) then nil
      else
        (X#G) = {F} in
        (X#{Take (N-1) G})
```

```
    end
  end
end


HamingTake = {Take 10 Z}
local
  (V12#F12) = {HamingTake}
  (V13#F13) = {F12}
  (V14#F14) = {F13}
  (V15#F15) = {F14}
  (V16#F16) = {F15}
  (V17#F17) = {F16}
  (V18#F18) = {F17} in
  skip Browse V12
  skip Browse V13
  skip Browse V14
  skip Browse V15
  skip Browse V16
  skip Browse V17
  skip Browse V18
end


end
```

Part 1 B:

B)

b1.

```
  b. modG :: Gen Int -> Int -> Gen Int
  modG x n = let G(v,f)= x in
  G ((v mod n), (modG f n))
  *Main> G (v,f) = modG (gen 0) 2
  *Main> G (v1,f1) = f
  *Main> G (v2, f2) = f1
  *Main> v
  0
  *Main> v1
  1
```

```
  b2. interleave :: [Gen Int] -> Gen Int
    interleave (x:xs) = let G(v1,f1) = x in
     G(v1, (interleave (xs ++ [f1])))
```

```
*Main> G (v1,f) = interleave [gen 3, gen 7, gen 13]
*Main> G (v1,f1) = f
*Main> G (v2,f2) = f1
*Main> G (v3,f3) = f2
*Main> v
0
*Main> v1
7
*Main> v2
13
```

……...PART 2……

```
local GateMaker AndG OrG NotG A B S IntToNeed Out MulPlex ByNeedImpl Ls X G Y Andd
Xxx Ys Xr Xx Yy Orr MulPlexAndG MulPlexOrG MulPlexRes InToNeedOut in

  fun {GateMaker F}
     fun {$ Xs Ys} GateLoop T in
        fun {GateLoop Xs Ys}
           case Xs of nil then nil
                 [] '|'(1:X 2:Xr) then
                    case Ys of nil then nil
                          [] '|'(1:Y 2:Yr) then
                             ({F X Y}|{GateLoop Xr Yr})
                    end
           end
        end
        T = thread {GateLoop Xs Ys} end   // thread isn't (yet) a returnable expression
           T
     end
  end

  fun {NotG Xs} NotLoop T in
     fun {NotLoop Xs}
        case Xs of nil then nil
              [] '|'(1:X 2:Xr) then ((1-X)|{NotLoop Xr})
        end
     end
     T = thread {NotLoop Xs} end          // thread isn't (yet) a returnable expression
     T
  end

//A. IntToNeed
```

```
   fun {IntToNeed L} Y in
      case L of nil then nil
        [] '|'(1:X 2:Xs) then
         byNeed fun {$} (X|{IntToNeed Xs}) end Y
       Y
       end
   end



//B. And gate using short circuiting
  fun {Andd X Ys}
     case X of nil then nil
       [] '|'(1:Xx 2:Xr) then
          if(Xx==0) then
            (Xx|{Andd Xr Ys})
          else
             case Ys of nil then nil
                [] '|'(1:Yy 2:Yr) then
                   (Yy|{Andd X Yr})
             end
          end
     end
  end

// B. Or gate using short circuiting
  fun {Orr Xs Ys}
     case Xs of nil then nil
       [] '|'(1:Xx 2:Xr) then
          if(Xx==1) then
            (1|{Orr Xr Ys})
          else
             case Ys of nil then nil
                [] '|'(1:Yy 2:Yr) then
                   (Yy|{Orr Xs Yr})
             end
          end
     end
  end

  Xxx =[0 1 1 0 0 1]
  Y =[0 1 1 0 0 1]
  AndG = {GateMaker {Andd Xxx Y}}   // Use GateMaker
  OrG =  {GateMaker {Orr Xxx Y}} // Use GateMaker
```

```
fun {MulPlex A B S}
  MulPlexAndG = {Andd {NotG S} A}
  MulPlexOrG = {Andd S B}
  MulPlexRes = {Orr MulPlexAndG MulPlexOrG}

  MulPlexRes
end

// C. calling MulPlex wihtout using IntToNeed
S = [1 0 1 0 1 1]
Out = {MulPlex Xxx Y S}

// run a loop so the MulPlex threads can finish before displaying Out
local Loop in
   proc {Loop X}
      if (X == 0) then skip Basic
      else {Loop (X-1)} end
   end
  {Loop 1000}
end

skip Browse Out



//D. calling MulPlex with IntToNeed
   A = {IntToNeed [0 1 1 0 0 1]}
   B = {IntToNeed [1 1 1 0 1 0]}
   S = [1 0 1 0 1 1]
   InToNeedOut = {MulPlex Xxx Y S}
   // run a loop so the MulPlex threads can finish before displaying Out
   local Loop in
      proc {Loop X}
         if (X == 0) then skip Basic
         else {Loop (X-1)} end
      end
     {Loop 10}
   end

   skip Browse InToNeedOut

   /*
```

(d.1) It depends on the value of S. If S is zero, it does not need the values of A and B. If S is 1, then it needs the value of the other variables. For instance, if A = 0, B=1, and S=1, then S will take the values of other variables like A= 0 and B=1. If A= 0, B=0, and S=0, then S will not take the values of other variables because it doesn't need it.

(d.2) Yes, they match up with what I got in (d.1).

*/

end

/*

output for MulPlex output

case 1 IntoNeed is not used in this case

here i have used input

Xxx =[0 1 1 0 0 1]

Y =[0 1 1 0 0 1]

output is Out : [0 1 1 0 0 1]

Case2 when InToNeed is used in MulPlex

same input is used here also like case 1

A = {IntToNeed [0 1 1 0 0 1]}

B = {IntToNeed [1 1 1 0 1 0]}

S = [1 0 1 0 1 1]

ouput

output is Out : [0 1 1 0 0 1]

*/