Part 1

1.a.1   Display function

local Display Generate X N in

```
 fun {Generate N}          //an 'infinite' generator function
  fun {$}
   (N#{Generate (N+1)})     //first value and rest of stream
  end
 end
```

 Display = proc {$ X N}   //displays a list
  local DisplayHelp Out Result in
   DisplayHelp = proc {$ X N Out}   //produces a list
    local
     (H#F) = {X}   //break up the stream
    in
     if ((N-1)==0) then   //when we reach the end
      Out = [H]   //return the head of the stream
     else T in   //otherwise
      {DisplayHelp F (N-1) T}   //get the rest with recursion
      Out = (H|T)   //output the list
     end
    end
   end
   {DisplayHelp X N Result}   //call helper to get list
   skip Browse Result   //display the list
  end
 end

X = {Generate 4}   //generate a stream starting from 4
{Display X 5}   //outputs [4 5 6 7 8]

end

1.a.2

```
local Generate Times Y Display X Result in

  fun {Generate N}                //generate a stream
    fun {$}
      (N#{Generate (N+1)})
    end
  end

  fun {Times X N}                 //returns a list of (X*N)s
    fun {$}                       //where X is a stream
      (H#F) = {X}                 //break up stream
    in
      ((H*N)#{Times F N})         //head is the product, tail is recursive call
    end
  end

  Display = proc {$ X N}               //displays a list
    local DisplayHelp Out Result in
      DisplayHelp = proc {$ X N Out}      //produces a list
        local
          (H#F) = {X}                     //break up the stream
        in
          if ((N-1)==0) then              //when we reach the end
            Out = [H]                     //return the head of the stream
          else T in                       //otherwise
            {DisplayHelp F (N-1) T}       //get the rest with recursion
            Out = (H|T)                   //output the list
          end
        end
      end
      {DisplayHelp X N Result}            //call helper to get list
      skip Browse Result                  //display the list
    end
  end

X = {Generate 0}     //get X
Y = {Times X 3}      //get Y
{Display Y 5}        //display it, outputs [0 3 6 9 12]
End
```

1.a.3

```
local Merge X Y Display Generate3 Generate5 in

 fun {Generate3 N}              //generate a stream
  fun {$}
   ((N*3)#{Generate3 (N+1)})     //of (N*3)'s
  end
 end

 fun {Generate5 N}              //generate a stream
  fun {$}
   ((N*5)#{Generate5 (N+1)})     //of (N*5)'s
  end
 end

 fun {Merge X Y}                   //merges two generated streams
  fun {$}
   (Hx#Fx) = {X}               //break apart X
   (Hy#Fy) = {Y}                //break apart Y
  in
   if (Hx<Hy) then               //if head of X is less then
    (Hx#{Merge Y Fx})           //add it to list, rec. call with Y and X's tail
   else                         //otherwise
    if (Hx==Hy) then             //if the heads are equal then
     (Hy#{Merge Fx Fy})          //add one to list and rec. call with the tails
    else                        //otherwise
     (Hy#{Merge X Fy})           //add X's head to list, rec. call with X and Y's tail
    end                          //..recursive call with X and Y's tail
   end
  end
 end

 Display = proc {$ X N}                  //displays a list
  local DisplayHelp Out Result in
   DisplayHelp = proc {$ X N Out}        //produces a list
    local
     (H#F) = {X}                      //break up the stream
    in
     if ((N-1)==0) then                //when we reach the end
      Out = [H]                       //return the head of the stream
     else T in                        //otherwise
      {DisplayHelp F (N-1) T}          //get the rest with recursion
      Out = (H|T)                     //output the list
     end
    end
```

```
      end
       {DisplayHelp X N Result}        //call helper to get list
        skip Browse Result             //display the list
      end
   end

X = {Generate3 1}                   //stream contents: [3 6 9 12 15 18 21 ...]
Y = {Generate5 1}                   //stream contents: [5 10 15 20 ...]
{Display {Merge X Y} 8}             //output: [3 5 6 9 10 12 15 18]
End
```

1.a.4
local E Generate Ham Times Merge Display Result in

```
fun {Generate N}                //generate a stream
  fun {$}
   (N#{Generate (N+1)})
  end
end


  fun {Merge X Y}           //merges two generated streams
   fun {$}
    (Hx#Fx) = {X}          //break apart X
    (Hy#Fy) = {Y}          //break apart Y
   in
     if (Hx<Hy) then                  //if head of X is less then
      (Hx#{Merge Y Fx})               //add it to list, rec. call with Y and X's tail
     else                             //otherwise
      if (Hx==Hy) then                //if the heads are equal then
       (Hy#{Merge Fx Fy})             //add one to list and rec. call with the tails
      else                            //otherwise
       (Hy#{Merge X Fy})              //add X's head to list, rec. call with X and Y's tail
      end                             //..recursive call with X and Y's tail
     end
   end
  end


  Display = proc {$ X N}                   //displays a list
   local DisplayHelp Out Result in
    DisplayHelp = proc {$ X N Out}         //produces a list
     local
      (H#F) = {X}                          //break up the stream
     in
      if ((N-1)==0) then                   //when we reach the end
       Out = [H]                           //return the head of the stream
      else T in                            //otherwise
       {DisplayHelp F (N-1) T}             //get the rest with recursion
       Out = (H|T)                         //output the list
      end
     end
    end
    {DisplayHelp X N Result}        //call helper to get list
    skip Browse Result              //display the list
   end
  end
```

```
  fun {Times X N}          //returns a list of (X*N)s
   fun {$}                  //where X is a stream
    (H#F) = {X}             //break up stream
   in
    ((H*N)#{Times F N})    //head is the product, tail is recursive call
   end
  end

  fun {Ham X}
   fun {$}
    (H#F) = {X}             //break up stream
   in                       //below is list with merged stream for a tail using recursion
    (H#{Merge {Times {Ham X} 2} {Merge {Times {Ham X} 3} {Times {Ham X} 5}}})
   end
  end

E = {Generate 1}          //a stream starting with 1
Result = {Ham E}           //start the Hamming function with 1
{Display Result 10}        //display first 10 values (below)
end
```
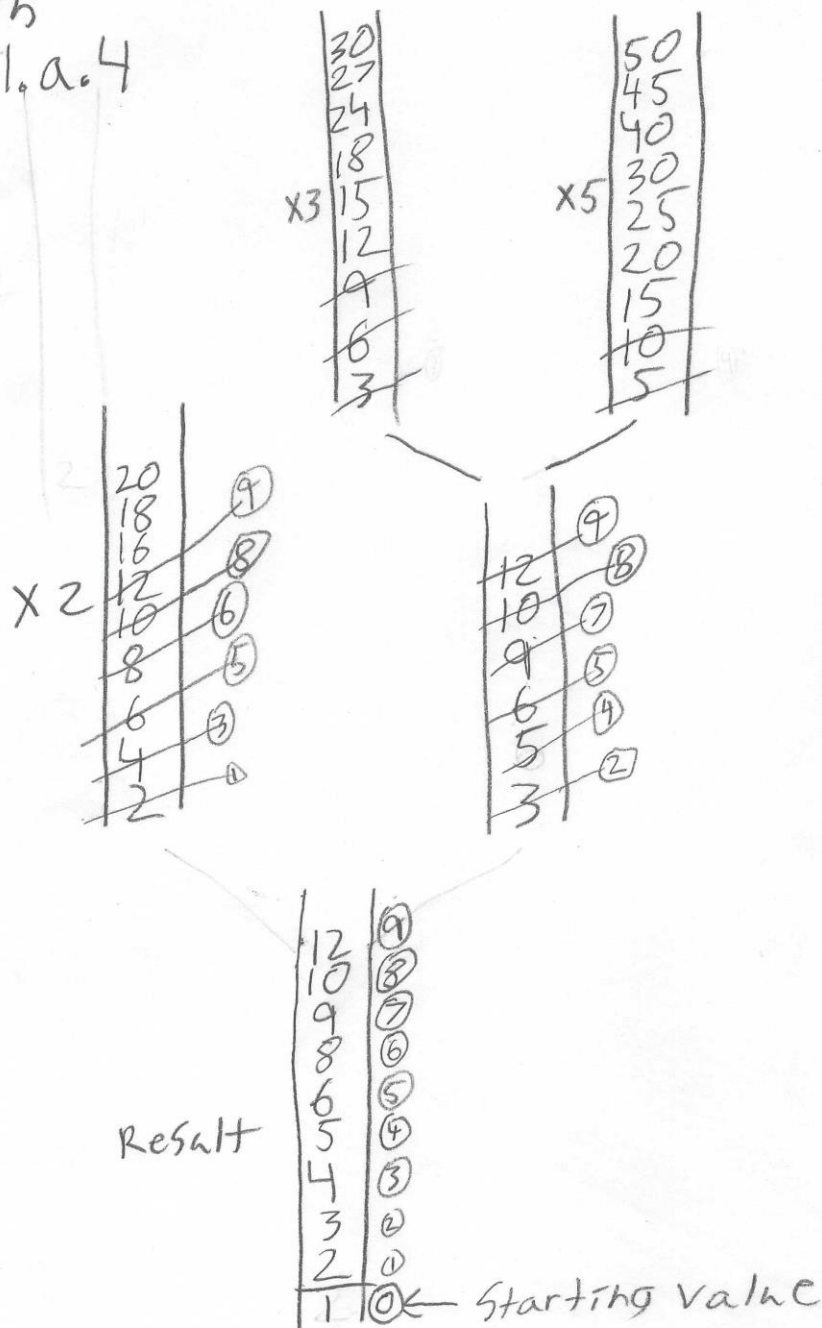
////Output for this program below////
//*OzKernelSyntaxParser> thread_mainK (Finite 1) "hamming.txt" "blah1.txt"
"blah.2txt"
//Result : [1 2 3 4 5 6 8 9 10 12]

Graph
for 1.a.4

X3
30
27
24
18
15
12
9
6
3

X5
50
45
40
30
25
20
15
10
5

X2
20
18
16
12
10
8
6
4
2

⑨
⑧
⑥
⑤
③
①

12
10
9
6
5
3

⑨
⑧
⑦
⑤
④
②

Result
12
10
9
8
6
5
4
3
2
1

⑨
⑧
⑦
⑥
⑤
④
③
②
①
⓪ ← Starting value

Result: [1 2 3 4 5 6 8 9 10 12]

1.b
```
-- Generator
data Gen a = G (a, Gen a)

gen:: Int -> Gen Int
gen n = G(n, gen (n+1))

-- Generator Testing
-- G(v, f) = gen 5        -- v=5
-- G(v1,f1) = f          -- v1=6
-- G(v2,f2) = f1         -- v2=7

-- Part b.1          --modG--
-- The function modG, which takes a generator X and integer N as input,
-- and returns a generator where the values produced by X are modulated by N.

modG :: Gen Int -> Int -> Gen Int
modG (G(x, y)) n = G(mod x n, modG (gen (x+1)) n)

-- modG Testing
-- G (v,f) = modG (gen 7) 9  -- v=7
-- G (v1,f1) = f           -- v1=8
-- G (v2,f2) = f1          -- v2=0

-- Part b.2         --interleave--
--  The function interleave, described in the notes, except it takes in a
--  list of generators of arbitrary size as input, interleaves them in order.

interleave :: [Gen Int] -> Gen Int
interleave ((G(x, y)):gs) = (G(x, interleave (gs++[gen (x+1)])))

G (v,f) = interleave [gen 3, gen 7, gen 13]
G (v1,f1) = f    -- v=3
G (v2,f2) = f1   -- v=7
G (v3,f3) = f2   -- v=13
G (v4,f4) = f3   -- v=4
G (v5,f5) = f4   -- v=8
G (v6,f6) = f5   -- v=14
G (v7,f7) = f6   -- v=5
```

Part 2

2.a, 2.b and 2.c below in bold

```
local GateMaker AndG OrG NotG A B S IntToNeed Out MulPlex in
        //gatemaker is on pg 316
        GateMaker = fun {$ F}
                fun {$ Xs Ys} T
                        GateLoop = fun {$ 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
                        in
                        T = thread X = {GateLoop Xs Ys} in X end // thread wasn't added to
expressions
                        T
                end
        end

        NotG = fun {$ Xs} T
                Loop = fun {$ Xs}
                  case Xs of nil then nil
                  [] '|'(1:X 2:Xr) then ((1+(X*-1))|{Loop Xr})
                  end
                end
                in T = thread X = {Loop Xs} in X end T
        end
```

2.b

```
AndG = {GateMaker fun {$ Xs Ys} O in
        if (Xs==0) then O = 0        //short circuit
        else O = (Xs*Ys)             //use expression
        end
        O                            //output
end}

OrG = {GateMaker fun {$ Xs Ys} O in
        if (Xs==1) then O = 1        //short circuit
        else O = (Xs+Ys)             //rest are true
        end
        O                            //output
end}
```

2.a

```
IntToNeed = fun {$ Xs} Loop O in
  Loop = fun {$ Xs}
            case Xs of nil then              //empty list case
               nil
  [] '|'(1:X 2:Xr) then O in                 //list is populated
      byNeed fun {$} X end O                  //turn head into byneed
      (O|{Loop Xr})              //pass byneed var as head and repeat with tail
     end
    end
    O = {Loop Xs}
    O
  end
```

2.c

```
fun {MulPlex A B S} And1 And2 Not Or in
      And1 = {AndG S B}              //first And gate
      Not  = {NotG S}               //reverse bits of S
      And2 = {AndG Not A}           //second And gate
      Or = {OrG And1 And2}          //run both lists from And gates through Or gate
      Or                            //output
  end

 A = {IntToNeed [0 1 1 0 0 1]}
      B = {IntToNeed [1 1 1 0 1 0]}
      S = [1 0 1 0 1 1]
      Out = {MulPlex A B 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

end
```

2.d

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

S = [1 0 1 0 1 1]
Out = {MulPlex A B S}

2.d.1

The AndG function tests if the members of the list of S bits are 0 and if they are it "short-circuits" automatically returning 0 (false). So whichever value of the A or B bit lists that is in that specific position is not processed.
I will bold the members of the lists that are not processed.
S = [1 0 1 0 1 1]
B = [1 **1** 1 **0** 1 0]
Now for the A list the S list has had its bit's flipped when ran through the Not gate.
(not) S = [0 1 0 1 0 0]
        A = [**0** 1 **1** 0 **0** 1]
As you can see here it is always parallel with the zeros of the S list. The members of A that are not being accessed are in the positions of all the members of the B list that were accessed due to the Not gate.

2.d.2

The needed values have store locations which all point to ones or zeros.

169 -> 1
245 -> 1
308 -> 1
331 -> 1
345 -> 0
385 -> 0

I believe that the way this program is executed, the store locations are assigned in a sort of staggered way due to the threading processes of the program So the bits I have shown you are not assigned in the order that they appear in the list. The right number of ones and zeros are present though! Three of the ones and one zero is from B and the remaining one and zero is from A.