LAB7
Part1

```
local Generate Zip Take Times Merge Hamming H X Y Z V S V1 F1 in


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

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

   Merge = fun {$ Xs Ys}
      fun {$}
         (X#G) = {Xs}
         (Y#Z) = {Ys} in
         if (X < Y) then
            (X#{Merge G Ys})
         else
            if (X > Y) then
               (Y#{Merge Xs Z})
            else
               (X#{Merge G Z})
            end
         end
      end
   end

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

   Hamming = fun {$ N}
      fun {$}
         (X#G) = {N} in
         (1#{Merge {Times 2 {Hamming G}} {Merge {Times 3 {Hamming G}} {Times 5
{Hamming G}}}})
      end
   end
```

```
   X = {Generate 1}
   Y = {Hamming X }
   V = {Take 10 Y}
   skip Browse V
end

//  runFull "declarative" "part1.txt" "part1.out"

part1 B
data Gen a = G (() -> (a, Gen a))

generate :: Int -> Gen Int
generate n = G (\_ -> (n, generate (n+1)))

gen_take :: Int -> Gen a -> [a]
gen_take 0 _ = []
gen_take n (G f) = let (x,g) = f () in x : gen_take (n-1) g    -- What's the type of f here? --  f will
be (Int, Gen Int)

times :: Int -> Gen Int -> Gen Int
times n (G f) = let (x,g) = f () in G(\_ -> ((n*x),times n g))

merge :: Gen Int -> Gen Int -> Gen Int
merge (G f) (G p) = let (x,g) = f () in let (y,k) = p() in
   if x < y then G (\_ -> (x,merge g (G p)))
   else if y < x then G (\_ -> (y,merge k (G f)))
   else  G (\_ -> (x, merge g k))

hamming :: Gen Int -> Gen Int
hamming (G f) = let (x,g) = f () in G (\_ -> (1,merge (times 2 (hamming g)) (merge (times 3
(hamming g)) (times 5 (hamming g)))))

l :: Gen Int
l = generate 1
y = hamming l

part2
local GateMaker AndG OrG NotG A B S IntToNeed Out MulPlex 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

  AndG ={GateMaker fun {$ X Y} if (x==0) then 0 else (X*Y)end end}   // Use GateMaker
  OrG = {GateMaker fun {$ X Y} if(X==0 Y==0) then 0 else X+Y-X*Y end}   // Use GateMaker

  fun {IntToNeed L}
    L = {IntToNeed x1, x2,... xn}
  end

  fun {MulPlex A B S}

  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

d1 :B   d2 same
```