MTE

- TRUE = $\lambda x.\lambda y.x$ = first
- FALSE = $\lambda x.\lambda y.y$ = second
- IF = $\lambda b.\lambda c_1.\lambda c_2.\ b\ c_1\ c_2$
- sq = $\lambda x.x\ x$ (self application)
- Recursion ⇒ $f = Yf = F(Yf)$
  where f → recursive function
  F → Non-recursive Equivalent
  Y → $\lambda t.(\lambda x.t(xx))(\lambda x.t(xx))$
  Yf is called fixed point of F
  and Y is fixed point generator
- successor ≡ $\lambda m.\lambda f.\lambda y.\ f(m\ f\ y)$
- Plus ≡ $\lambda m.\lambda n.\lambda f.\lambda y.\ m\ f\ (n\ f\ y)$
- iszero ≡ $\lambda n.n\ (\lambda x.FALSE)\ TRUE$
- multiply ≡ $\lambda m.\lambda n.\lambda f.\lambda y.\ n\ (m\ f)\ y$
- exponent ≡ $\lambda m.\lambda n.\ n\ m$ {m^n}
- subtraction ≡ $\lambda m.\lambda n.\ n$ pred m {m-n}
  {: subtract 1 from m, 'n' times.}
- Less than equal to (<=):-
  {if m<=n → true
  else false}
  LE ≡ $\lambda m.\lambda n.$ iszero (sub m n)
  pred 0 = 0
- Equality (eq):-
  eq ≡ $\lambda m.\lambda n.$ AND ((LE m n)) (LE n m)
  {m<=n and n<=m ⇔ n==m}
- OR ≡ $\lambda x.\lambda y.\ x\ x\ y$
- AND ≡ $\lambda x.\lambda y.\ x\ y\ x$
- NOT ≡ $\lambda x.\ x$ FALSE TRUE
  (OR)
  NOT ≡ $\lambda x.\ x$ FALSE TRUE
- XOR ≡ $\lambda x.\lambda y.\ x$ (NOT y) y
- twice = $\lambda f.\lambda g.\ f(f\ g)$

Pair :-

- Pair = $\lambda x.\lambda y.\lambda z.\ z\ x\ y$
  {(Pair 3 2 will form {3,2})}
- We have to design functions to access first & second element.
- fst ≡ $\lambda p.\ p$ TRUE
- snd ≡ $\lambda p.\ p$ FALSE
  {Both will take pair as input which is $\lambda z.zxy$ format}

**

List :-

- Just like a pair with head and tail
- [2,3,5] = 2 : [3,5]
  h t
- List = $\lambda x.\lambda y.\lambda z.zxy$
  ≡ $\lambda h.\lambda t.\lambda s.\ s\ h\ t$
- fst ≡ $\lambda L.\ L$ TRUE
- snd ≡ $\lambda L.\ L$ FALSE
- [2,3] ≡ $\lambda s.$ s 2 ($\lambda s.$ s 3 Nil)
  {Nil will choco the empty list}
  ⇓
  We want isempty function to check for it
- isempty
  ≡ $\lambda L.\ L(\lambda h.\lambda t.$ FALSE)
  where NIL ≡ $\lambda x.$ TRUE

**

· Implement map (of haskell) in Lambda Calculus :-

- Given a function 'f' and list 'L', apply function to all the elements of L.

⇒ map = λf. λL. IF (isempty L) L (List (f (frst L)) (map f (snd L)))

- recursive definition
- use Y combinator to get the answer.

· Implement filter (of haskell) in Lambda Calculus :-

⇒ Filter = λf. λL. IF (isempty L) L (IF (f (frst L)) (List (frst L) (Filter (snd L))) (Filter (snd L)))

- if an element satisfies the function or returns true, then include in the output list, otherwise not (drop it)

· Haskell code to find unique element in a list :-

(a)
```
take X :: a → [a] → [a]
take X z [] = []
take X z (x:xs)
        | (x == z) = take X z xs
        | (x /= z) = x : (take X z xs)
```
["Not equal to" in Haskell]

(b)
```
unique :: [a] → [a]
unique [] = []
unique (x:xs) = x : unique (take X x xs)
```
first remove all the (x) from remain list and then calculate unique elements.

- Haskell code to find neighbors of a cell in 2D infinite grid :-

{Number of Neighbors can be 3, 5, or 8.}

```
neighbors :: (Ord a1, Ord a2, Num a1, Num a2) -> a1
                -> a2 -> [(a1, a2)]
neighbors x y = [(x+dx, y+dy) | dx <- [-1,0,1],
                                dy <- [-1,0,1], (dx,dy) /= (0,0)]
```

#[Return list of Coordinates]

☆ Stucked Lambda term :-

- Are there lambda terms whose evaluation may stuck at any time ?
  - Lambda terms with free variables can stuck.
- Given a lambda term, is it possible to create an automatic analyzer that decides, yes or no, whether or not a lambda term will ever get stuck ?
  - No ;
  - LC is turing-complete.
  - suppose TM is a lambda term that simulates Turing machine ∅ $(\lambda x.\, y\ x)$ TM
  - If TM not halts, then it doesn't stuck otherwise it gets stucked, when TM halts on running into free variable y.
  - We can't decide if TM halt.
    {undecidable problem}