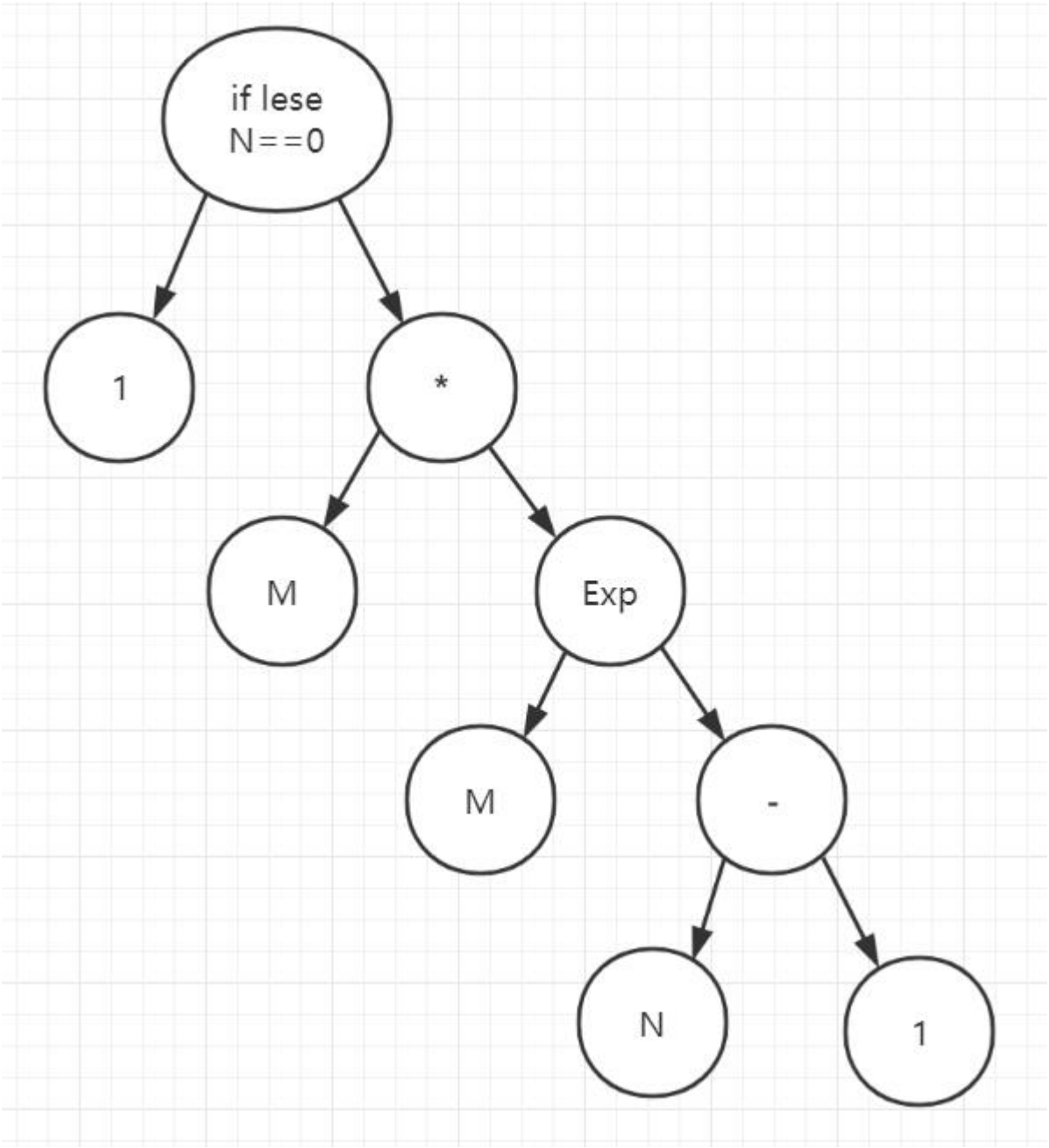
# Homework 2

James Zhang zhany111

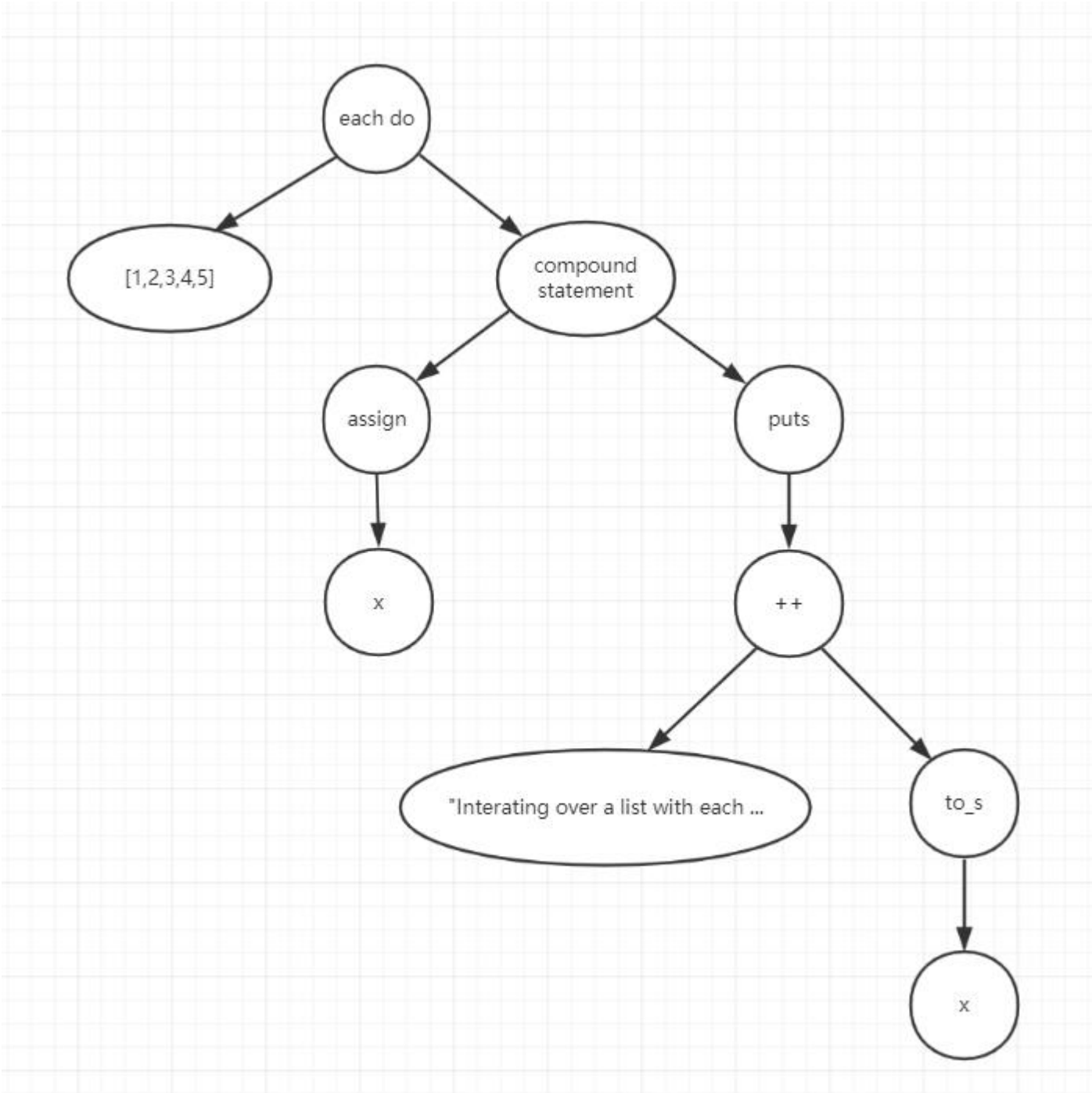
September 23, 2019

## 1 Q1

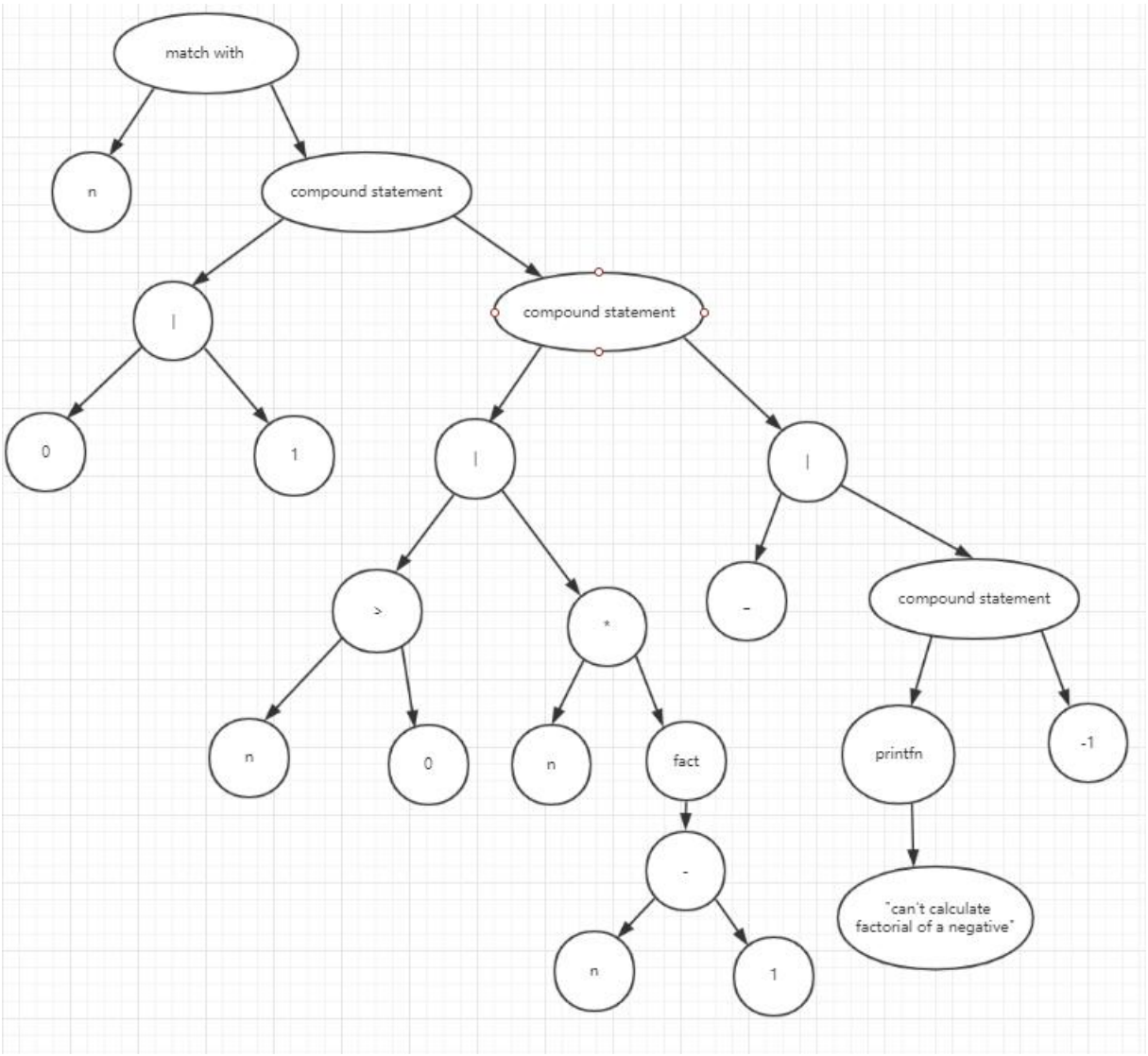
1)




2)


each do
[1,2,3,4,5]
compound statement
assign
puts
x
++
"Interating over a list with each ...
to_s
x

3)




# 2 Q2 Fsharp

The following code includes defining of tree type, with functions that transfer list to tree and tree to list. There are also test cases by requirment. I decide to use a sorted but unbalanced tree, because it is easy to create. There are different functions that use Leaf or Leaf option as input.

```
//Define the leaf type
type Leaf = {value: int; l:Leaf option; r: Leaf option}
```

```
///The method of inserting new leaf into a sorted tree(tree can be empty)
let rec insertLeaf (tree: Leaf option) (newValue: int) : Leaf =
    match tree with
    ///if tree is not empty, do the comparson to see if the newvalue
    ///should be insert left or right to do it, we use recursion.
    | Some t -> if newValue < t.value then {t with l = Some (insertLeaf t.l newValue)}
                else if newValue > t.value then {t with r = Some (insertLeaf t.r newValue)}
                else t
    //if tree is empty, insert new value into the leaf
    | None -> {value=newValue; l = None; r = None}

///A method that inserts newvalue into the tree that cannot be empty
let rec insertLeaf1 (tree: Leaf) (newValue: int) : Leaf =
    if newValue < tree.value then {tree with l = Some (insertLeaf tree.l newValue)}
    else if newValue > tree.value then {tree with r = Some (insertLeaf tree.r newValue)}
    else tree

///method that generates sorted tree from a list of int
let listToTree (l:List<int>) =
    //if the list is empty, return leaf with value of 0
    if l.IsEmpty then
        printf "empty tree is not a tree \n"
        let a = {value = 0; l = None; r = None}
        a
    else
        let mutable t = {value = l.Head; l = None; r = None}
        for i in l do
            t <- insertLeaf1 t i
        t

///method that changes tree(which can be empty) to a sorted list of int
let rec goThro (tree:Leaf option) : list<int> =
    match tree with
    ///if a leaf has 2 branches compute left first then right
    | Some t -> if not(t.l.IsNone || t.r.IsNone) then goThro(t.l) @ [t.value] @ goThro(t.r)
                else if not t.l.IsNone then goThro(t.l) @ [t.value] //when only left branch
                else if not t.r.IsNone then goThro(t.r) @ [t.value] //when only right branch
                else [t.value]  //when no branches
    | None -> []

///method that changes tree(that is not empty) to a sorted list of int
let rec goThro1 (tree:Leaf) : list<int> =
    if not(tree.l.IsNone || tree.r.IsNone) then goThro(tree.l) @ [tree.value] @ goThro(tree.r)
    else if not tree.l.IsNone then goThro(tree.l) @ [tree.value]
    else if not tree.r.IsNone then goThro(tree.r) @ [tree.value]
    else [tree.value]

///tree0 that is a empty tree
let tree0 = listToTree([])
///tree1 that test if the functions can generate tree from list, and can generate list from tree
let tree1 = listToTree([50; 34; 40; 70; 20; 6])
let l1 = goThro1(tree1)
///Three other test trees
let tree2 = listToTree([1;2;3;4;5;6;7])
let tree3 = listToTree([7;6;5;4;3;2;1])
let tree4 = listToTree([4;3;5;2;6;1;7])

printfn "This is an empty tree: \n%O" tree0
printfn "This is tree of [50 34 40 70 20 6]:\n %O" tree1
printfn "This is the list transfered from last tree:\n %O" l1
printfn "This is tree of [1 2 3 4 5 6 7]:\n %O" tree2
printfn "This is tree of [7 6 5 4 3 2 1]:\n %O" tree3
printfn "This is tree of [4 3 5 2 6 1 7]:\n %O" tree4
```

# 3 Q3

The belowing code is defining of class Leaf and Tree. I am sure that it does not run, because of imcompletement of Tree class. I don't have time to finish it because I have other work to do. The idea is creating a sorted tree. There are just some synitc errors.

```
#define Leaf class with value and two children
class Leaf
  attr_reader :value
  attr_accessor :left, :right

  def initialize(value=nil)
    @value = value
    left = nil;
    right = nil;
    return self
  end
end

#Define Tree that is connection of Leaf
class Tree
  #root_leaf is the first leaf of a tree
  attr_accessor :root_leaf

  def initialize(root_value=nil)
    @root_leaf = Leaf.new(root_value)
    return self
  end

  #method that insert new value into the tree
  def insert(leaf, value)
    if @value == nil
      @value = value
      return self
    elsif @value == value
      return leaf
    elsif value < @value
      insert(leaf.left, value)
    elsif value > @value
      insert(leaf.right, value)
    end
  end
end

#Function that transfer a hash to a Tree
```

```
def hashToTree(hsh, t)
  hsh.each_value {|v| t.insert(t, v)}
  return t
end

h = {1=>1, 2=>2, 3=>3, 6=>6, 5=>5}
t = Tree.new()
puts "#{hashToTree(h, t)}"
```