1. Consider a binary tree defined as:

**class** Tree {int val; Tree left; Tree right;}

Define an **iterator** bf\_elements(t) that yields the values of a Tree t in **breadth-first order**.

Answer:

First I found a piece of breadth-first order search code on internet(<http://codesam.blogspot.com/2011/01/breadth-first-search-in-binary-tree.html>).

*class Tree { val: int, left: Tree, right: Tree}*

*iterator int elements(Tree t) {*

*if (null(t))*

*{return;}*

*else {*

*LinkedList queue = new LinkedList();*

*Node tempNode;*

*queue.addLast(t);*

*while(!queue.isEmpty())*

*{*

*tempNode = queue.remove();*

*yield tempNode.val;*

*if (tempNode.left!=null) {queue.addLast(tempNode.left);}*

*if (tempNode.right!=null) {queue.addLast(tempNode.right);}*

*}*

*}*

*}*

2. Just as iterator constructs can be compiled using procedure parameters, one might wonder whether **coroutine** constructs could also be translated in a similar manner. Explain briefly why such a translation is not feasible, by highlighting what aspect of the use of coroutines would pose the greatest difficulty for translation.

Answer:

From my viewpoint, “iterator constructs can be compiled using procedure parameters” essentially because:

Iterator need to yield mutimes and can save the intterupted flow control when yield, when next time call iterator again, the flow control can resume without run from the beginning of iterator.Using Procddure paramter, move the loop funciton from the caller of iterator to insde, then all loops run in a continuing mode.

But for coroutine, it is diffcult to find the loop body in a concentrated place, and this loop body should be move inside to procedure parameter.So it diffcult to do such tranlsation.

3.

Assuming that a **stack** of n elements e1 e2 … en is represented by the following lambda-term, where e1 is at the top of the stack and en is at the bottom of the stack:

λf. λx.((f e1 ) ((f e2) …((f en ) x) …)).

Show *non-recursive* lambda-calculus definitions for the following operations on a stack. Assume that the empty stack is represented as: λf. λx.x

***let stack0 = Lf.Lx.x***

***let stack1 = Lf.Lx.((f e1) x)***

***let stack2 = Lf.Lx.((f e1) ((f e2) x))***

**1.(top stk)**: return the top element of the stack **stk**;

Top can be defined as λl.((l λx.λy.x) a) as List in note

***let top = Ll.((l Lx.Ly.x) a)***

run on simulator

(top stack1)=> e1

(top stack2)=> e2

2. **(nonempty stk)**: return a boolean indicating whether **stk** is not empty;

***let nonempty = Ln.((n Lx.Lz.true) false)***

run on simulator

(nonempty stack0)=> false

(nonempty stack1)=> true

3.**(size stk)**: return a Church numeral indicating the number of elements in **stk**;

***Let size = Ln.Lf.(n Lx.f)***

>>> (Ln.Lf.(n Lx.f) stack2)

-----------------------

==> Lf.(stack2 Lx.f)

= Lf.(Lf.Lx.((f e1) ((f e2) x)) Lx.f)

--------------------------------

==> Lf.Lx.((Lx.f e1) ((Lx.f e2) x))

---------

==> Lf.Lx.(f ((Lx.f e2) x))

---------

==> Lf.Lx.(f (f x)

(iv) **((push e) stk)**: given an element **e** and a stack **stk**, return a new stack by placing

element **e** on top of the stack **stk**.

***let push = Ln1.Ln2.Lf.Lx.((f n1) ((n2 f) x))***

>>> ((push a) stack1)

= ((Ln1.Ln2.Lf.Lx.((f n1) ((n2 f) x)) a) stack1)

-------------------------------------

==> (Ln2.Lf.Lx.((f a) ((n2 f) x)) stack1)

-------------------------------------

==> Lf.Lx.((f a) ((stack1 f) x))

= Lf.Lx.((f a) ((Lf.Lx.((f e1) x) f) x))

--------------------

==> Lf.Lx.((f a) (Lx.((f e1) x) x))

-----------------

==> Lf.Lx.((f a) ((f e1) x))

Test your answers using the Lambda Calculus simulator located at:

http://www.cse.buffalo.edu/LRG/CSE505/Lambda

Start with the ‘readme’ file in that directory.