1.

Write the bytecode generation for the following BL Program. Use symbolic names for jump conditions and primitive instructions.

PROGRAM Mystery IS

INSTRUCTION myInstruction IS

turnright

IF next-is-enemy THEN

turnleft

END IF

END myInstruction

BEGIN

IF next-is-wall THEN

move

infect

WHILE next-is-not-empty

turnleft

END WHILE

ELSE

IF true THEN

myInstruction

END IF

skip

END IF

skip

END Mystery

Note: not all memory locations may be used

| 0 |  |
| --- | --- |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 |  |
| 8 |  |
| 9 |  |
| 10 |  |
| 11 |  |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 |  |
| 16 |  |
| 17 |  |
| 18 |  |
| 19 |  |
| 20 |  |
| 21 |  |
| 22 |  |
| 23 |  |
| 24 |  |

2.

Implement the following method that returns the location specified by the next unconditional jump in the given program, given what the bug sees, the current memory location and the compiled program. If there is no unconditional jump, it returns -1. Assume you are given methods conditionalJumpCondition and isPrimitiveInstructionByteCode, specified by the contracts below.

/\*\*

\* Returns the value of the condition in the given conditional jump

\* {@code condJump} given what the bug sees {@code wbs}. Note that if

\* {@code condJump} is the byte code for the conditional jump

\* JUMP\_IF\_NOT\_condition, the value returned is the value of the

\* "condition" part of the jump instruction.

\*

\* **@param** wbs

\* the {@code CellState} indicating what the bug sees

\* **@param** condJump

\* the byte code of a conditional jump

\* **@return** the value of the conditional jump condition

\* **@requires** [condJump is the byte code of a conditional jump]

\* **@ensures** <pre>

\* conditionalJumpCondition =

\* [the value of the condition of condJump given what the bug sees wbs]

\* </pre>

\*/

private static boolean conditionalJumpCondition(CellState wbs,

int condJump) {

/\*\*

\* Returns whether the given integer is the byte code of a BugsWorld

\* virtual

\* machine primitive instruction (MOVE, TURNLEFT, TURNRIGHT, INFECT,

\* SKIP, HALT).

\*

\* **@param** byteCode

\* the integer to be checked

\* **@return** true if {@code byteCode} is the byte code of a primitive

\* instruction or false otherwise

\* **@ensures** <pre>

\* isPrimitiveInstructionByteCode =

\* [true iff byteCode is the byte code of a primitive instruction]

\* </pre>

\*/

private static boolean isPrimitiveInstructionByteCode(int byteCode);

/\*\*

\* Returns the location specified by the next unconditional jump to

\* execute in

\* compiled program {@code cp} given what the bug sees {@code wbs} and

\* starting from location {@code pc}.

\*

\* **@param** cp

\* the compiled program

\* **@param** wbs

\* the {@code CellState} indicating what the bug sees

\* **@param** pc

\* the program counter

\* **@return** the location specified by the next unconditional jump to

\* execute

\* **@requires** <pre>

\* [cp is a valid compiled BL program] and

\* 0 <= pc < cp.length and

\* [pc is the location of an instruction byte code in cp, that is, pc

\* cannot be the location of an address]

\* </pre>

\* **@ensures** <pre>

\* [return the address specified by the next unconditional jump that

\* should be executed in program cp given what the bug sees wbs and

\* starting execution at address pc in program cp]

\* </pre>

\*/

public static int nextUnconditionalJumpAddress(int[] cp, CellState wbs,

int pc) {

//TODO - fill in body

}

3.

Write a recursive descent parser for the following grammar that parses a mathematical string of single digit integers into a Queue of ints. Each method is an instance method which will be called on a Queue<Integer>.

ex. <> and <1,2,3> are valid in this grammar. The result would be a Queue of integers with that mathematical representation.

mathString -> < [ elements ] >

elements -> entry {, entry }

entry -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

/\*\*

\* Parses the queue of tokens into this queue

\*

\* @replaces this

\*

\* @updates tokens

\*

\* @requires

\* tokens is a valid expression as defined by the grammar,

\* end\_of\_input is a suffix of tokens

\* @ensures

\* this = the queue modeled by the expression represented in tokens

\*/

public void parseMathString(Queue<String> tokens) {

}

/\*\*

\* Parses the queue of tokens into this queue

\*

\* @updates this

\*

\* @updates tokens

\*

\* etc.

\*/

public void parseElements(Queue<String> tokens){

}

/\*\*

\* Parses the queue of tokens into this queue

\*

\* @updates this

\*

\* @updates tokens

\*

\* etc.

\*/

public void parseEntry(Queue<String> tokens){

}

4.

The following comparator sorts strings alphabetically regardless of case. Here is a sample of a list sorted according to this comparator:

alpha

bar

Beta

cat

Foo

foo

gamma

Zeta

zeta

private static class StringLT implements Comparator<String> {

@Override public int compare(String o1, String o2) {

return o1.compareToIgnoreCase(o2);

}

}

Change the comparator so that it sorts the same way, but is consistent with equals. If two strings are spelled the same but differ in case, it does not matter which comes first, as long as they appear consecutively in the sorted list.

private static class StringLT implements Comparator<String> {

@Override public int compare(String o1, String o2) {

//TODO - fill in body

}

}

5.

Suppose fast direct access has just been discovered for strings with dynamic size! You are implementing Sequence5 based on this recent discovery. You have implemented all of the kernel methods, but some of the secondary method implementations are too slow, and don’t take advantage of fast direct access. How can you make sure Sequence5 has different implementations for its secondary methods?

6.

Implement the equals method for Stack<T>

public boolean equals(Object obj){

//TODO - fill in body

}

7.

Write a method that, given a java.util Map<String, Integer> increments the value if it matches the given key

/\*\*

\* given {@code Map}, increments the value if it matches the given

\* {@code key}

\*

\* **@param** map

\* the given map

\* **@param** key

\* the given key

\* **@ensures** if map contains key, map.value(key) is incremented

\*/

public static void incrementValue(Map<String, Integer> map, String key) {

//TODO - fill in body

}

8.

Write a main method that takes a file as input, the name of an output location, and a number of lines to be copied from the input file and printed to the output file. First, declare that it simply throws an IOException, then catch and handle all the IOExceptions. Take the input as command-line arguments

public static void main(String[] args) throws IOException {

// TODO - fill in body

}

public static void main(String[] args) {

// TODO - fill in body

}

9.

Suppose you have a program that keeps track of all books in a library. A book is composed of a title and an author. Each time a book is created, it is added to the library. There can be more than one copy of a book, but if all the copies are checked out, the book is not available. Implement the following interface according to this idea and the method contracts, using static class members.

/\*\*

\* A book modeled by a title and author

\*

\* **@author** Izzy Smith

\*

\*/

public interface Book {

//im trying with this javadoc cut me some slack

/\*\*

\* reports the title

\*

\* **@return** the title of this book

\*

\* **@ensures** title = this.title

\*/

String title();

/\*\*

\* reports the author

\*

\* **@return** the author of this book

\*

\* **@ensures** author = this.author

\*/

String author();

/\*\*

\*

\* reports whether this is available

\*

\* **@return** whether this book is available in the global library

\*

\* **@ensures** available = (number of total copies of the book) != 0

\*/

boolean available();

/\*\*

\* checks this book out of the library

\*

\* **@requires** available == true;

\*

\* **@ensures** (number of total copies of the book) = (number of total

\* copies of the book - 1)

\*

\*/

void checkOut();

/\*\*

\* **@ensures** (number of total copies of the book) = (number of total

\* copies of the book + 1)

\*/

void checkIn();

}

import java.util.Map;

import java.util.TreeMap;

/\*\*

\* Implementation of the Book interface

\*

\* **@author** Izzy Smith

\*

\*/

public class Book1 implements Book {

//TODO - add private members

public Book1(String title, String author) {

//TODO - fill in body

}

@Override

public String title() {

//TODO - fill in body

}

@Override

public String author() {

//TODO - fill in body

}

@Override

public boolean available() {

//TODO - fill in body

}

@Override

public void checkOut() {

//TODO - fill in body

}

@Override

public void checkIn() {

//TODO - fill in body

}

@Override

public String toString() {

//TODO - fill in body

}

}

10.

Write a recursive static method that reverses all the elements of a tree.

example:

before-

7

/ | \

4 6 5

/ | \ / \

3 2 1 3 9

after-

7

/ | \

5 6 4

/ \ / | \

9 3 1 2 3

/\*\* im trying pls ik these method contracts are whack

\* reverses all the elements of a tree

\*

\* **@param** <T>

\* the type of elements in the tree

\* **@param** t

\* the tree of be reversed

\* **@updates** t

\* **@requires** t is not null

\* **@ensures** reverses the order of the children of t and reverses each

\* child

\*/

public static <T> void reverse(Tree<T> t) {

//TODO - fill in body

}

11.

Implement Set on Stack in the following class:

import java.util.Iterator;

import components.set.Set;

import components.set.SetSecondary;

import components.stack.Stack;

import components.stack.Stack1L;

/\*\*

\* {@code Set} represented as a {@code Queue} of elements with implementations

\* of primary methods.

\*

\* **@param** <T>

\* type of {@code Set} elements

\* **@convention** |$this.elements| = |entries($this.elements)|

\* **@correspondence** this = entries($this.elements)

\*/

public class Set3<T> extends SetSecondary<T> {

/\*

\* Private members --------------------------------------------------------

\*/

/\*\*

\* Elements included in {@code this}.

\*/

private Stack<T> elements;

/\*\*

\* Finds {@code x} in {@code s} and, if such exists, moves it to the top

\* of {@code s}.

\*

\* **@param** <T>

\* type of {@code Stack} entries

\* **@param** s

\* the {@code Stack} to be searched

\* **@param** x

\* the entry to be searched for

\* **@updates** s

\* **@ensures** <pre>

\* perms(q, #q) and

\* if <x> is substring of q

\* then <x> is prefix of q

\* </pre>

\*/

private static <T> void moveToTop(Stack<T> s, T x) {

assert s != null : "Violation of: s is not null";

//assume correct implementation

}

/\*\*

\* Creator of initial representation.

\*/

private void createNewRep() {

//TODO - fill in body

}

/\*

\* Constructors -----------------------------------------------------------

\*/

/\*\*

\* No-argument constructor.

\*/

public Set3() {

//TODO - fill in body

}

/\*

\* Standard methods -------------------------------------------------------

\*/

//clear, newInstance and transferFrom not included for cleanliness

/\*

\* Kernel methods ---------------------------------------------------------

\*/

@Override

public final void add(T x) {

assert x != null : "Violation of: x is not null";

assert !this.contains(x) : "Violation of: x is not in this";

// TODO - fill in body

}

@Override

public final T remove(T x) {

assert x != null : "Violation of: x is not null";

assert this.contains(x) : "Violation of: x is in this";

// TODO - fill in body

}

@Override

public final T removeAny() {

assert this.size() > 0 : "Violation of: this /= empty\_set";

// TODO - fill in body

}

@Override

public final boolean contains(T x) {

assert x != null : "Violation of: x is not null";

// TODO - fill in body

}

@Override

public final int size() {

// TODO - fill in body

}

//Iterator removed for cleanliness

}

12.

Implement renameInstruction. Sorry no twists or pizazz here just good practice.

/\*\*

\* Refactors the given {@code Statement} by renaming every occurrence of

\* instruction {@code oldName} to {@code newName}. Every other statement is

\* left unmodified.

\*

\* **@param** s

\* the {@code Statement}

\* **@param** oldName

\* the name of the instruction to be renamed

\* **@param** newName

\* the new name of the renamed instruction

\* **@updates** s

\* **@requires** [newName is a valid IDENTIFIER]

\* **@ensures** <pre>

\* s = [#s refactored so that every occurrence of oldName is

\* replaced by newName]

\* </pre>

\*/

public static void renameInstruction(Statement s, String oldName,

String newName) {

switch (s.kind()) {

case BLOCK:

//TODO - fill in body

break;

case IF:

//TODO - fill in body

break;

case IF\_ELSE:

//TODO - fill in body

break;

case WHILE:

//TODO - fill in body

break;

case CALL:

//TODO - fill in body

break;

default:

//nothing to do here

break;

}

}

13.

The following method is an instance method for a doubly-linked list with two smart nodes (same instance variables as the ListWithRetreat project). What does mystery(7) do? Draw the result of this method given the example list below.

public void mystery(T x){

Node p = new Node();

p.data = x;

this.lastLeft.next = p;

p.prev = this.lastLeft;

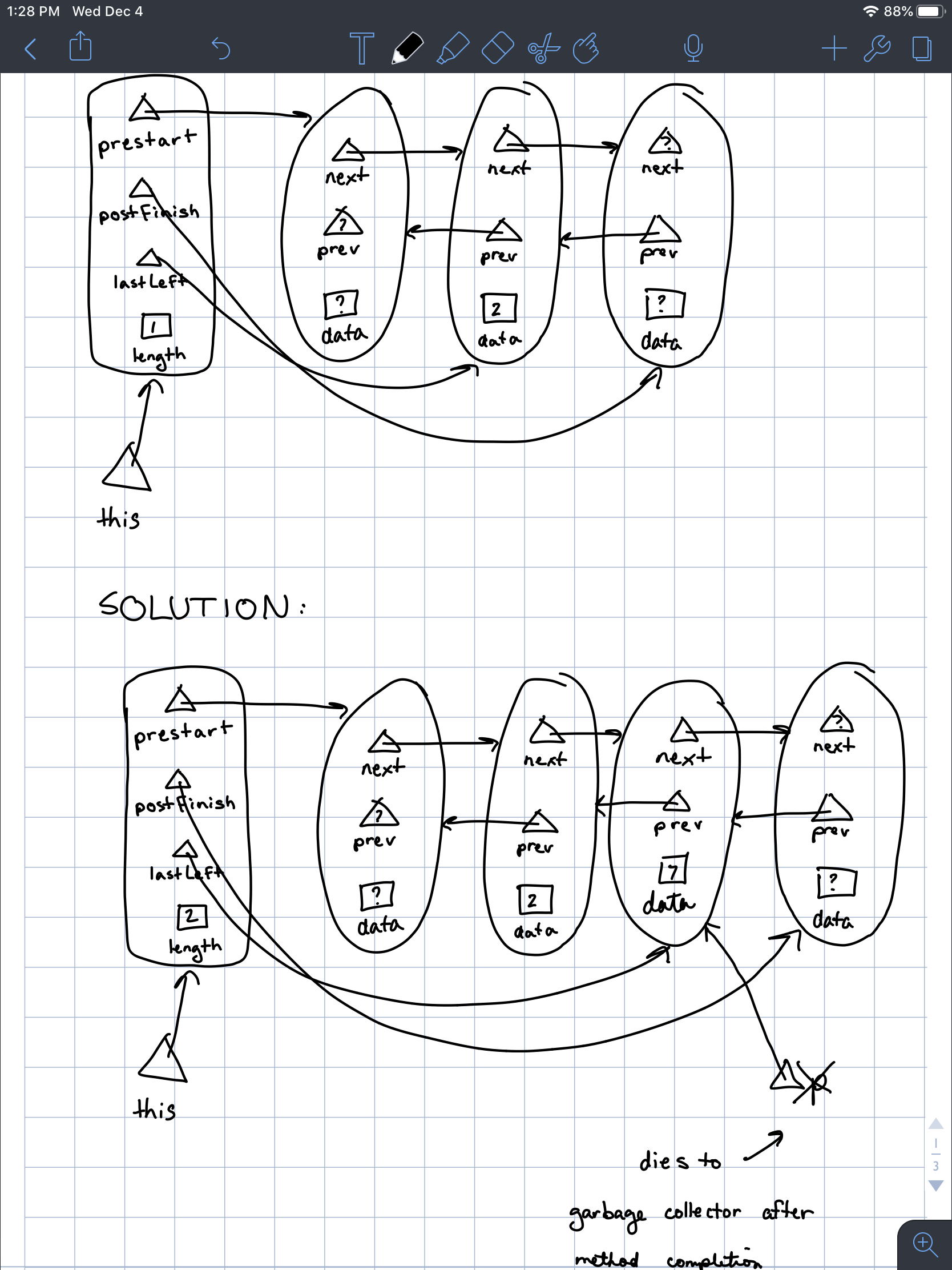
p.next = this.postFinish;

this.postFinish.prev = p;

this.lastLeft = p;

this.length++;

}



14.

implement the set kernel methods using a hashtable representation

public class Set4<T> extends SetSecondary<T> {

/\*

\* Private members --------------------------------------------------------

\*/

/\*\*

\* Default size of hash table.

\*/

private static final int DEFAULT\_HASH\_TABLE\_SIZE = 101;

/\*\*

\* Buckets for hashing.

\*/

private Array<Set<T>> hashTable;

/\*\*

\* Total size of abstract {@code this}.

\*/

private int size;

/\*\*

\* Computes {@code a} mod {@code b} as % should have been defined to work.

\*

\* **@param** a

\* the number being reduced

\* **@param** b

\* the modulus

\* **@return** the result of a mod b, which satisfies 0 <= {@code mod} < b

\* **@requires** b > 0

\* **@ensures** <pre>

\* 0 <= mod and mod < b and

\* there exists k: integer (a = k \* b + mod)

\* </pre>

\*/

private static int mod(int a, int b) {

assert b > 0 : "Violation of: b > 0";

int result = a % b;

if (result < 0) {

result += b;

}

return result;

}

/\*\*

\* Creator of initial representation.

\*

\* **@param** hashTableSize

\* the size of the hash table

\* **@requires** hashTableSize > 0

\* **@ensures** <pre>

\* |$this.hashTable.entries| = hashTableSize and

\* for all i: integer

\* where (0 <= i and i < |$this.hashTable.entries|)

\* ($this.hashTable.entries[i, i+1) = <{}> and

\* i is in $this.hashTable.examinableIndices) and

\* $this.size = 0

\* </pre>

\*/

private void createNewRep(int hashTableSize) {

// TODO - fill in body

}

/\*

\* Constructors -----------------------------------------------------------

\*/

/\*\*

\* No-argument constructor.

\*/

public Set4() {

// TODO - fill in body

}

/\*\*

\* Constructor resulting in a hash table of size {@code hashTableSize}.

\*

\* **@param** hashTableSize

\* size of hash table

\* **@requires** hashTableSize > 0

\* **@ensures** this = {}

\*/

public Set4(int hashTableSize) {

// TODO - fill in body

}

/\*

\* Standard methods removed for cleanliness

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

\*/

/\*

\* Kernel methods ---------------------------------------------------------

\*/

@Override

public final void add(T x) {

assert x != null : "Violation of: key is not null";

assert !this.contains(x) : "Violation of: key is not in DOMAIN(this)";

// TODO - fill in body

}

@Override

public final T remove(T x) {

assert x != null : "Violation of: key is not null";

assert this.contains(x) : "Violation of: key is in DOMAIN(this)";

// TODO - fill in body

}

@Override

public final T removeAny() {

assert this.size() > 0 : "Violation of: this /= empty\_set";

// TODO - fill in body

}

@Override

public final boolean contains(T x) {

assert x != null : "Violation of: key is not null";

// TODO - fill in body

}

@Override

public final int size() {

// TODO - fill in body

}

//Iterator removed for cleanliness

}

15.

Write a recursive instance method that reverses all the elements of a binary tree. (assume reverses means the same as the similar question above).

/\*

\* reverses {@code this}

\* @**requires** this is not null

\* **@ensures** this = compose(root, right.reverse(), left.reverse())

\*/

public <T> void reverse(){  
 // TODO - fill in body

}

16.

Draw the AST for the following BL Statement:

WHILE true DO

turnleft

IF next-is-empty THEN

skip

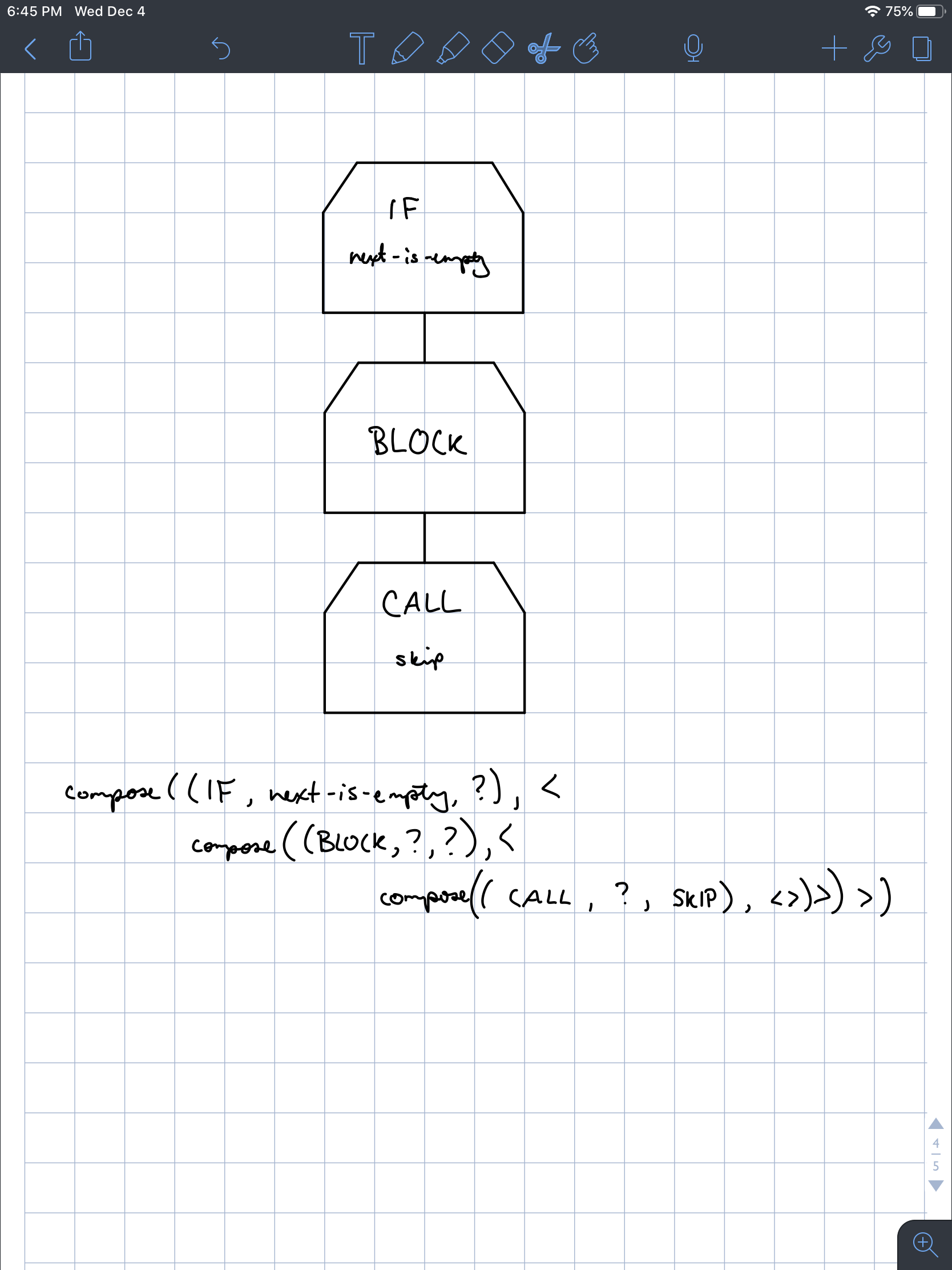
ELSE

move

END IF

END WHILE

17. write the following AST in mathematical notation



18.

Write the mathematical notation for the following program (draw the AST instead of using the compose function for any trees)

PROGRAM MyProgram IS

BEGIN

WHILE true DO

turnleft

IF next-is-empty THEN

skip

ELSE

move

END IF

END WHILE