COMP 250 Assignment 5

Christopher Hundt (110220945)

December 2, 2002

The input file "foo" contains the definitions as laid out in the assignment instructions. Each "bar" input file contains one of the expressions given as an example in the assignment instructions, and they are numbered (from 0) in the order that they occur in the instructions. All input files can be found attached to this assignment, as well as various relevant .java files. Also, every file related to this assignment can be found both on the attached diskette and at http://www.cs.mcgill.ca/~chundt/comp250.

Question 1

Syntax

```
java SymbolTable [defs]
```

Commands

```
java SymbolTable foo > Q1
```

Output

Q1

```
showdefs() output:

A is bound to 12
B is bound to 3
C is bound to 5
D is bound to 6
E is bound to 2
F is bound to 9
G is bound to 1
H is bound to 7
K is bound to 56
```

Question 2

Syntax

```
java Parser [exp]
```

Commands

```
java Parser bar1 > Q2-1
java Parser bar2 > Q2-2
java Parser bar3 > Q2-3
```

Output

Q2-1

```
E.postorder() output:
 A B C * +

E.preorder() output:
 + A * B C

E.inorder() output:
 A + B * C
```

Q2-2

```
E.postorder() output:
 A B + C + D + A B * *

E.preorder() output:
 * + + + A B C D * A B

E.inorder() output:
 A + B + C + D * A * B
```

Q2-3

```
E.postorder() output:
 A B C D E F G + H K * + + * + + +

E.preorder() output:
 + A + B + C * D + E + + F G * H K

E.inorder() output:
 A + B + C + D * E + F + G + H * K
```

Question 3

Syntax

```
java Evaluator [defs] [exp]
```

Commands

```
java Evaluator foo bar1 > Q3-1
java Evaluator foo bar2 > Q3-2
java Evaluator foo bar3 > Q3-3
```

Output

Q3-1

```
The value is: 27
```

Q3-2

```
The value is: 936
```

Q3-3

```
The value is: 2444
```

Question 4

Syntax

```
java Codegen [defs] [exp] [ouputFile]
```

Commands

```
java Codegen foo bar0 Q4-0
java Codegen foo bar1 Q4-1
java Codegen foo bar2 Q4-2
java Codegen foo bar3 Q4-3
```

Output

Q4-0

```
LOAD A
ADD B
STORE 1
LOAD G
ADD K
MUL 1
```

Q4-1

```
LOAD B
MUL C
ADD A
```

Q4-2

```
LOAD A
ADD B
ADD C
ADD D
MUL A
MUL B
```

Q4-3

```
LOAD H
MUL K
ADD F
ADD G
ADD E
MUL D
ADD C
ADD B
ADD A
```

Question 5

$$\Sigma = \{a, b, c\}$$

$$NT = \{S, A, B\}$$

Starting character is S.

Rules:

$$\begin{array}{rcl} S & \longrightarrow & aSc \\ S & \longrightarrow & AB \\ A & \longrightarrow & aAb \\ A & \longrightarrow & \epsilon \\ B & \longrightarrow & bBc \\ B & \longrightarrow & \epsilon \end{array}$$

Question 6

We use B to refer to a boolean expression and C to refer to a command. $\langle Stmt1 \rangle$ is a statement that can be converted to an "if/then" branch, while $\langle Stmt2 \rangle$ can be converted to an "if/then/else" branch. $\langle Stmt \rangle$ is the starting character, and can lead to either kind of branching statement.

$$\Sigma = \{\text{if}, \text{then}, \text{else}, B, C\}$$

$$NT = \{\langle Stmt \rangle, \langle Stmt1 \rangle, \langle Stmt2 \rangle\}$$

Rules:

$$\begin{array}{rcl} \langle Stmt \rangle & \longrightarrow & \langle Stmt1 \rangle \\ \langle Stmt \rangle & \longrightarrow & \langle Stmt2 \rangle \\ \langle Stmt1 \rangle & \longrightarrow & \text{if } B \text{ then } \langle Stmt \rangle \\ \langle Stmt2 \rangle & \longrightarrow & \text{if } B \text{ then } \langle Stmt2 \rangle \text{ else } \langle Stmt \rangle \\ \langle Stmt2 \rangle & \longrightarrow & C \end{array}$$

Input Files

foo

```
C 5
A 12
D 6
B 3
E 2
K 56
G 1
F 9
H 7
```

bar0

```
(A+B)*(G+K)$
```

bar1

```
A+B*C$
```

bar2

```
(A+B+C+D)*(A*B)$
```

bar3

```
(A+(B+(C+(D*(E+(F+G+H*K))))))$
```

.java Files

The files are presented in alphabetical order.

BinTree.java

```
class BinTree {
  // The basic class used for many of the data structures in this program.
  // The nodes contain an object.  This can be cast to more specific types
  // as the situation demands.  There is nothing in this that forces this
  // to be a binary search tree or an expression tree.  So please write
  // this generically.

  // Instance Variables

  private Object node;
  private BinTree left, right;

  // Constructors

  public BinTree(Object root){
    // Makes a tree with one node and empty subtrees
    this.left = null;
    this.right = null;
    this.node = root;
  }

  public BinTree(Object root, BinTree l, BinTree r){
    //Makes a tree with a new root and given subtrees
    this.left = l;
    this.right = r;
    this.node = root;
  }

  // Instance Methods

  public Object root(){
      return this.node;
  }

  public BinTree left(){
      return this.left;
  }

  public BinTree right(){
      return this.right;
  }

  public void SetRoot(Object x){
      this.node = x;
  }

  public void SetLeft(BinTree t){
      this.left = t;
```

```
  }

  public void SetRight(BinTree t){
      this.right = t;
  }

  public boolean IsLeaf(){
      return (this.left==null && this.right==null);
  }

  public int length() {
      if (this.left==null && this.right==null) return 1;
      else return 1+java.lang.Math.max(this.left.length(), this.right.length());
  }

  //The next three methods are required to visit the nodes in the indicated
  //orders and print the object at the nodes.  It is somewhat silly to make
  //you do all three since if you do one you can do all three.  However I
  //think that even silly repetitive code will help some of this material
  //become automatic.
  public void preorder(){
  //Returns in root, left, right order--copied of out notes
      System.out.print(" " + this.node);
      if (this.left != null) this.left.preorder();
      if (this.right != null) this.right.preorder();
  }

  public void inorder(){
  //Returns in left, root, right order--copied of out notes
      if (this.left != null) this.left.inorder();
      System.out.print(" " + this.node);
      if (this.right != null) this.right.inorder();
  }

  public void postorder(){
  //Returns in left, right, root order--copied of out notes
      if (this.left != null) this.left.postorder();
      if (this.right != null) this.right.postorder();
      System.out.print(" " + this.node);
  }

}//end BinTree
```

Codegen.java

```
import java.io.*;
import java.util.StringTokenizer;

//****************** CODE GENERATION ************************************
class Codegen {
  // This is like parser in that there is a code generation object and it
  // has a method called codegen which produces the code.  Every time you
  // want to generate some code you make a new Codegen object which has the
  // ExpTree produced by the parser stored inside it.  I have written file
```

```
// IO code for you.  This is contained in a method called generateToFile
// which will prompt the user for the name of a file where the output
// should go.  This method calls the method "codegen" which you will
// write.

private int tempstore;
private ExpTree tree;
private String outFilename;
private PrintWriter out;

public Codegen(ExpTree t){
  tree = t;
  tempstore = 1;}

public void generateToFile(String outFilename) {
  FileInputStream inputData;
     try {
     out = new PrintWriter (new FileWriter(outFilename));
     codegen(tree, "", false);
     out.close();
     } catch(IOException exception){
       System.out.println("IO problem. Terminating.");
       System.exit(0);}
};

   private static String commandString(Character operChar) {
   //Given a symbol (in Character form), returns the command that should be
   //used when generating code.
       String operStr = operChar.toString();
       if (operStr.equals("+")) return "ADD";
       else if (operStr.equals("-")) return "SUBTRACT";
       else if (operStr.equals("_")) return "SUBFROM";
       else if (operStr.equals("*")) return "MUL";
       else if (operStr.equals("/")) return "DIVBY";
       else if (operStr.equals("\\")) return "DIVINTO";
       else if (operStr.equals("^")) return "POW";
       else {
           System.out.println("Error: unrecognized command: " + operChar);
           return "";
       }
   }

   private static Character switchOper(BinTree t) {
   //Returns the root symbol for division and subtraction to reflect change
   //in order.
       Character operChar = (Character) t.root();
       Character temp = operChar;
       String operStr = operChar.toString();
       if (operStr.equals("-")) temp = new Character("_".charAt(0));
       else if (operStr.equals("_")) temp = new Character("-".charAt(0));
       else if (operStr.equals("/")) temp = new Character("\\".charAt(0));
       else if (operStr.equals("\\")) temp = new Character("/".charAt(0));
       return temp;
```

```
}


private static void switchUp(BinTree t) {
//Switches the left and right sides of a tree, and changes the root command
//if necessary. (For instance, A/B --> B\A)
    t.SetRoot(switchOper(t));
    BinTree temp=t.left();
    t.SetLeft(t.right());
    t.SetRight(temp);
}

private static boolean associative(String oper1, String oper2) {
//Given two operators (in String form), returns true if you can use the
//associative property with the two operators (in that order). For example,
//associative("+", "-") returns true because (A+B)-C=A+(B-C), but
//associative("-", "+") returns false because (A-B)+C<>A-(B+C).
    if (oper1.equals("+"))
        return (oper2.equals("+") || oper2.equals("-") || oper2.equals("_"));
    else if (oper1.equals("*"))
        return (oper2.equals("*") || oper2.equals("/") || oper2.equals("\\"));
    else return false;
}

private boolean codegen(ExpTree t, String lastOper, boolean rightTree){
//t is the Tree to generate code for.
//lastOper is a String representing the operator from the calling tree,
//only used if rightTree is true.
//rightTree is whether t is a right tree of the calling tree.

    //OPTIMIZATION:
    //(1) If the right side is longer than the left side, switch them.
    if (t.left().length() < t.right().length()) switchUp(t);
    //(2) Otherwise, if the left side "associates," but the right side does
    //not, switch them, because association is only used when descending
    //down the right side of the tree. Don't bother switching if the right
    //side is just a variable, though, because this will not improve, and
    //may negatively affect, the quality resulting code.
    else if (!(t.right() instanceof Var)
             && associative(t.root().toString(),t.left().root().toString())
             && !associative(t.root().toString(),t.right().root().toString())) {
        switchUp(t);
    }

    boolean storeStuff = false; //used to indicate whether a "STORE"
                                //command was used, so the calling tree
                                //knows where to find its previous results

    if (t instanceof Oper) {
        String firstAction = "LOAD "; //Action to be taken if left is a Var
        if (rightTree) {
            if (associative(lastOper, t.root().toString()))
            //If we can use associative property, then we might not need
```

```
            //to STORE anything
            {
                if (!(t.left() instanceof Var)) {
                //If left is a Var, then we have to STORE, so we can load it
                    out.println("STORE " + tempstore++);
                    storeStuff = true;
                }
                //Otherwise, we can execute the command from the calling tree
                else firstAction =
                  commandString(new Character(lastOper.charAt(0))) + " ";
            }
            else {
            //If we can't use associative property, then we need to store
            //whatever the calling tree did.
                out.println("STORE " + tempstore++);
                storeStuff = true;
            }
        }
        if (t.left() instanceof Var) {
        //If left is a Var, then do something with it (LOAD or perform
        //some operation if we are using associativity.
            String rightStr = t.left().root().toString();
            out.println(firstAction + rightStr);
        }
        //Otherwise, call codegen() on left
        else codegen((ExpTree) t.left(), t.root().toString(), false);

        if (t.right() instanceof Var) {
        //If right is a Var, then we just perform our operation with it
            String rightStr = t.right().root().toString();
            out.println(commandString((Character) t.root()) + " " + rightStr);
        }
        else {
        //Otherwise, we need to go down the right tree:
            boolean storedStuff =
              codegen((ExpTree) t.right(), t.root().toString(), true);
            if (storedStuff) {
            //If the right tree stored our previous result (from the left)
            //then we need to run our command on the stored value. But we
            //need to switch the operator, because we waited until after
            //we did stuff on the right to perform the operation.
                out.println(
                  commandString(switchOper((BinTree) t)) + " " + --tempstore);
            }
            //Otherwise, they took care of our command.
        }
    }
    else {
        out.println("Error--non-operator node!");
    }
    return storeStuff; //Let the calling tree know whether we stored stuff
}//end method codegen
```

```
    public static void main(String[] args){
    SymbolTable S = SymbolTable.getdefs(args[0]);
    char[] Data = Tokenizer.scan(args[1]);
    Parser P = new Parser(Data);
    ExpTree E = P.parse();
    Codegen G = new Codegen(E);
    G.generateToFile(args[2]);
    }//end method main
}// end Codegen
```

DefsTree.java

```
class DefsTree extends BinTree {
  //This is a tree of bindings.  This is organized as a binary SEARCH tree
  //with the alphabetic order of the keys used for ordering purposes.  You
  //have to signal that a value is being redefined if that happens.  All yu
  //have to do is print a message to the screen saying what variable is
  //being redefined.  If a search failed to find what you are looking for
  //you have to print a message to the screen.  There is no delete.

  public DefsTree(Binding b){super(b);}

  public void insert(Binding b){
      DefsTree rightTree, leftTree;
      Binding r = (Binding) this.root();//cast Object to Binding
      //use Unicode for enforcing order:
      int rootUnicode = Character.getNumericValue(r.GetKey());
      int insertedUnicode = Character.getNumericValue(b.GetKey());
      if (insertedUnicode > rootUnicode) {
      //If we're inserting a "higher" letter than the root, put it on the right:
          if (this.right() == null) this.SetRight(new DefsTree(b));
          else {
              rightTree = (DefsTree) this.right();
              rightTree.insert(b);
          }
      }
      else if (insertedUnicode < rootUnicode){
      //If we're inserting a "lower" letter than the root, put it on the left:
          if (this.left() == null) this.SetLeft(new DefsTree(b));
          else {
              leftTree = (DefsTree) this.left();
              leftTree.insert(b);
          }
      }
      else {
      //insertedUnicode = rootUnicode, replacing variables
          System.out.println("Redefining " + r.GetKey());
          r.SetVal(b.GetVal());
      }
  }// end method insert

  public int lookup(char name){
      DefsTree rightTree, leftTree;
      Binding r = (Binding) this.root();//cast Object to Binding
```

```
      //Use Unicode to search:
      int rootUnicode = Character.getNumericValue(r.GetKey());
      int searchUnicode = Character.getNumericValue(name);
      if (searchUnicode > rootUnicode) {
      //If the number we're looking for is "higher" than the root, go to the right:
          if (this.right() == null) System.out.println("Error--no such variable.");
          else {
              rightTree = (DefsTree) this.right();
              return rightTree.lookup(name);
          }
      }
      else if (searchUnicode < rootUnicode){
      //If the number we're looking for is "lower" than the root, go to the left:
          if (this.left() == null) System.out.println("Error--no such variable.");
          else {
              leftTree = (DefsTree) this.left();
              return leftTree.lookup(name);
          }
      }
      else {
      //searchUnicode = rootUnicode, we found it
          return r.GetVal();
      }
      return 0;
  }//end method lookup

}// end DefsTree
```

Evaluator.java

```
class Evaluator {
  // This just has one static method and uses existing objects.  It could
  // have been put into say the ExpTree class or somewhere else but
  // conceptually it really is seperate.  One can also imagine making an
  // evaluator object as I did for parser.

  public static int myPow(int base, int exp){
  //Used to find powers. Takes only integer bases and exponents.
        int t = 1;
        int m = exp;
        int n = base;
        if (m == 0) return 1;
        while (m != 1) {
            if (m % 2 == 0) {
                n = n*n;
                m = m/2;
            }
            else {
                m = m - 1;
                t = t * n;
            }
        }
        return t*n;
    }//method myPow
```

```
  public static int eval(SymbolTable S, ExpTree t){
      if (t == null) return 0;
      else if (S == null) {
        System.out.println("No definitions");
      return 0;}
      else {
          String r = t.root().toString();
          if (t instanceof Oper) {
          //If we have an operator, do whatever that operator says to do:
              if (r.equals("+")) return
                eval(S, (ExpTree) t.left()) + eval(S, (ExpTree) t.right());
              else if (r.equals("-")) return
                eval(S, (ExpTree) t.left()) - eval(S, (ExpTree) t.right());
              else if (r.equals("*")) return
                eval(S, (ExpTree) t.left()) * eval(S, (ExpTree) t.right());
              else if (r.equals("/")) return
                eval(S, (ExpTree) t.left()) / eval(S, (ExpTree) t.right());
              else if (r.equals("^")) return
                myPow(eval(S, (ExpTree) t.left()), eval(S, (ExpTree) t.right()));
          }
          else if (t instanceof Var) {
          //If we have a var, return the result
              char c = ((Character) t.root()).charValue();
              return S.lookup(c);
          }
          else {
              System.out.println("ERROR: invalid node");
          }
          return 0;
      }
  }//end method eval

  public static void main(String[] args){
    SymbolTable S = SymbolTable.getdefs(args[0]);
    char[] Data = Tokenizer.scan(args[1]);
    Parser P = new Parser(Data);
    ExpTree E = P.parse();
    System.out.println("The value is: " + eval(S,E));};
}//end Evaluator
```

Parser.java

```
class Parser {
  //This class packages the parser code and expression to be parsed as an
  //object.  The tokens are stored in an array which is private to the
  //parser object.  The parsing routines will be instance methods and they
  //will call each other with mutual recursion.  The main method is called
  //parse and it returns an ExpTree.  To see how tokenizer and parser are
  //used see the final main method of the class Codegen.  The basic idea is
  //that every time you want to parse an expression you will create a new
  //parser object which encapsulates the parsing routines and the string of
  //characters which have to be parsed.
```

```
private char[] tokens;
private int cursor;
private char sym;

private void getsym(){
  //This gets a new symbol for processing and it advances the cursor.
  //The next comment is for debugging purposes.  Uncomment it if needed.
  //    System.out.println("Entering getsym " + cursor + " " + sym);
  cursor++; sym = tokens[cursor]; }

private void error(){System.out.println("Parsing error occurred.");}

public Parser(char[] A){
  //Creates a new parser object with the characters to be parsed stored
  //in it and the sym and cursor initialized.
  tokens = A;
  cursor = 0;
  sym = tokens[cursor];}

public ExpTree parse(){
//Does addition. Every time it sees addition, it calls the next most tightly
//binding operation (subtraction).
    ExpTree result = null;
    ExpTree temp = null;
    ExpTree operRight = null;
    boolean newResult = false;
    temp = this.parsesub();
    result = temp; //Do this in case there's no addition performed, so this
                   //function still returns something.
    Character currentSym = new Character(this.sym);
    while (currentSym.toString().equals("+")) {
        getsym();
        if(!newResult) {
        //If this is the first result, we want to create a new tree.
            result = new Oper(currentSym, temp, this.parsesub());
            newResult = true;
        }
        else {
        //If we've already built a tree, we build "up" on it.
            result = new Oper(currentSym, result, this.parsesub());
        }
        currentSym = new Character(this.sym);
    }
    return result;
}// end method parse

public ExpTree parsesub(){
//Does subtraction. Every time it sees subtraction, it calls the next most tightly
//binding operation (multiplication).
    ExpTree result = null;
    ExpTree temp = null;
    ExpTree operRight = null;
    boolean newResult = false;
```

```
    temp = this.term();
    result = temp; //Do this in case there's no subtraction performed, so this
                   //function still returns something.
    Character currentSym = new Character(this.sym);
    while (currentSym.toString().equals("-")) {
        getsym();
        if(!newResult) {
        //If this is the first result, we want to create a new tree.
            result = new Oper(currentSym, temp, this.term());
            newResult = true;
        }
        else {
        //If we've already built a tree, we build "up" on it.
            result = new Oper(currentSym, result, this.term());
        }
        currentSym = new Character(this.sym);
    }
    return result;
}// end method parsesub

public ExpTree term(){
//Does multiplication. Every time it sees multiplication, it calls the next most
//tightly binding operation (division).
    ExpTree result = null;
    ExpTree temp = null;
    ExpTree operRight = null;
    boolean newResult = false;
    temp = this.termdiv();
    result = temp; //Do this in case there's no multiplication performed, so this
                   //function still returns something.
    Character currentSym = new Character(this.sym);
    while (currentSym.toString().equals("*")) {
        getsym();
        if(!newResult) {
        //If this is the first result, we want to create a new tree.
            result = new Oper(currentSym, temp, this.termdiv());
            newResult = true;
        }
        else {
        //If we've already built a tree, we build "up" on it.
            result = new Oper(currentSym, result, this.termdiv());
        }
        currentSym = new Character(this.sym);
    }
    return result;
}//end method term

public ExpTree termdiv(){
//Does division. Every time it sees division, it calls the next most tightly
//binding operation (powers).
    ExpTree result = null;
    ExpTree temp = null;
    ExpTree operRight = null;
```

```
    boolean newResult = false;
    temp = this.pow();
    result = temp; //Do this in case there's no division performed, so this
                   //function still returns something.
    Character currentSym = new Character(this.sym);
    while (currentSym.toString().equals("*")||currentSym.toString().equals("/")) {
        getsym();
        if(!newResult) {
        //If this is the first result, we want to create a new tree.
            result = new Oper(currentSym, temp, this.pow());
            newResult = true;
        }
        else {
        //If we've already built a tree, we build "up" on it.
            result = new Oper(currentSym, result, this.pow());
        }
        currentSym = new Character(this.sym);
    }
    return result;
}//end method termdiv

public ExpTree pow(){
//Does powers. Every time it sees powers, it calls primary(), because powers
//are the most tightly binding operation
    ExpTree result = null;
    ExpTree temp = null;
    ExpTree operRight = null;
    boolean newResult = false;
    temp = this.primary();
    result = temp; //Do this in case there's no powers performed, so this
                   //function still returns something.
    Character currentSym = new Character(this.sym);
    while (currentSym.toString().equals("^")) {
        getsym();
        if(!newResult) {
        //If this is the first result, we want to create a new tree.
            result = new Oper(currentSym, temp, this.primary());
            newResult = true;
        }
        else {
        //If we've already built a tree, we build "up" on it.
            result = new Oper(currentSym, result, this.primary());
        }
        currentSym = new Character(this.sym);
    }
    return result;
}//end method term

public ExpTree primary(){
    ExpTree result = null;
    Character currentSym = new Character(this.sym);
    //Use Unicode to check if it's a letter:
    int currentUnicode = Character.getNumericValue(this.sym);
```

```
      if (10 <= currentUnicode && currentUnicode <= 35) { //it's a variable
          result = new Var(new Character(this.sym));
          getsym();
          return result;
      }
      else if (currentSym.toString().equals("(")) { //left parenthesis
      //If we are in parentheses, we "start over"
          getsym();
          result = this.parse();
          currentSym = new Character(this.sym);
          //Check for right parenthesis:
          if (!currentSym.toString().equals(")")) {
              error();
              return null;
          }
          else {
              getsym();
              return result;
          }
      }
      else return null;
  }// end method primary

  public static void main(String[] args){
    char[] Data = Tokenizer.scan(args[0]);
    Parser P = new Parser(Data);
    ExpTree E = P.parse();
    System.out.println("E.postorder() output: ");
    E.postorder();System.out.println();
    System.out.println("\nE.preorder() output: ");
    E.preorder();System.out.println();
    System.out.println("\nE.inorder() output: ");
    E.inorder();
  }

}//end Parser
```

SymbolTable.java

```
import java.util.StringTokenizer;
import java.io.*;

class SymbolTable {
  // This class uses a private DefsTree as its basic data structure.  It
  // reads a filename and looks up the definitions from that file and puts
  // all the data in its private DefsTree using the BST insert which you
  // implemented.  I have done the method that does the IO for you but you
  // have to code the other methods.  Methods like insert perform some
  // basic tests (like is the DefsTree empty) and deal with boundary cases
  // but in the end they will invoke the DefsTree method.  This class
  // allows one to have a real symbol table object even if there are no
  // bindings (i.e. the private DefsTree is null).

  private DefsTree table;
```

```
// Constructors
public SymbolTable(){table = null;};
public SymbolTable(DefsTree t){table = t;};

// Class Methods
public static SymbolTable getdefs(String fname){
  //This function reads the definitions from the file named by the
  //argument and constructs a Symbol Table containing the variable-value
  //bindings.
  FileInputStream inputData;
  int val; char var;
  int blank = (int) ' ';
  SymbolTable result = new SymbolTable();

  try {
  inputData = new FileInputStream(fname);
  BufferedInputStream bis = new BufferedInputStream(inputData);
  DataInputStream dis = new DataInputStream(bis);
  String currentDef = null;
  StringTokenizer defST = null;
  //StringTokenizer rawdefs = new StringTokenizer (filein.readLine(), " ,");
  while ((currentDef = dis.readLine()) != null){
    defST = new StringTokenizer(currentDef, " ");
    //var = rawdefs.nextToken().charAt(0);
    //val = Integer.parseInt(rawdefs.nextToken());
    var = defST.nextToken().charAt(0);
    val = Integer.parseInt(defST.nextToken());
    result.insert(var,val);}
  } catch(IOException exception){
    System.out.println("Input problem. Terminating.");
    System.exit(0);}
  return result;}//end method getdefs

// Instance Methods

public void insert(char name, int val){
  // Insert is used to insert values into the symbol table.
  //It takes a name and a value.  The modification is made in-place and
  //nothing is returned.
    Binding bar = new Binding(name, val);
    //If we don't have a tree yet, make a new one. Otherwise, use the
    //DefsTree insert:
    if (table == null) table = new DefsTree(bar);
    else table.insert(bar);

}// end method insert

public int lookup(char name){
    //use the DefsTree lookup, unless there is no tree, in which case return 0
    if (table == null) return 0;
    else return table.lookup(name);
}// end method lookup
```

```
  void showdefs(){
    //The function showdefs is used to display the bindings stored in the
    //symbol table.

    if (table == null) System.out.println("The table is empty");
    else table.inorder();};

  public static void main(String[] args){
    SymbolTable S = getdefs(args[0]);
    System.out.println("showdefs() output:");
    S.showdefs();
  }
}// end SymbolTable
```