**Turing Machine Simulation in Java**

**Overview**

This Java project implements a simulation of a Turing Machine, a theoretical computing device invented by Alan Turing. The project consists of multiple classes that work together to represent and simulate the behavior of a Turing Machine.

**Classes**

**1. TestTape**

This class serves as a test program for the **Tape** class. It demonstrates the usage of the **Tape** class methods, such as setting content, moving left and right, and displaying the tape contents.

**2. TestTapeGUI**

A graphical user interface (GUI) test program for the **Tape** class. This class creates a window with a graphical representation of the tape, including buttons for moving the current cell left or right, a text-input box for setting the content of the current cell, and a display of the tape contents.

**3. TestTuringMachine**

This class tests the **TuringMachine** class by creating three machines with different sets of rules and running them. It demonstrates the expected behavior of the Turing Machine simulation, including successful execution and handling of illegal states.

**4. TuringMachine**

The core class representing the Turing Machine. It includes methods to add rules to the machine's program and run the machine on a given tape. The machine executes rules based on the current state and content of the tape cells, moving left or right and updating the tape accordingly.

**5. Tape**

This class represents the tape of the Turing Machine using a doubly-linked list of cells. It provides methods for setting content, moving left or right, and obtaining the contents of the tape. The tape automatically expands to the left or right if the current cell is at the edge.

**6. Rule**

A simple class representing a rule for the Turing Machine. Each rule consists of the current state, current content, new state, new content, and a flag indicating whether to move left or right.

**7. Expressions**

A class for experimenting with expression trees. It includes several nested abstract classes and subclasses representing nodes in an expression tree. The class also provides methods for creating test data, copying expression trees, and evaluating expressions.

**8. Cell**

A basic class representing one cell on a Turing Machine tape. It includes the content of the cell and pointers to the next and previous cells in the tape.

**Programs**

// package turing;

// A test program for the Tape class that calls most of the methods in that class.

//

// The output from this program should be:  Tape Conents:  Hello World

//                                          Final position at the W

public class TestTape {

    public static void main(String[] args) {

        Tape tape = new Tape();

        for (int i = 0; i < "World".length(); i++) {

            tape.setContent("World".charAt(i));

            tape.moveRight();

        }

        for (int i = 0; i < "Hello World".length(); i++)

            tape.moveLeft();

        for (int i =0; i < "Hello".length(); i++) {

            tape.setContent("Hello".charAt(i));

            tape.moveRight();

        }

        System.out.println("Tape Conents:  " + tape.getTapeContents());

        tape.moveRight();

        System.out.println("Final position at the " + tape.getContent());

    }

}

// package turing;

// A test program for the Tape class that creates a window containing

// a graphical representation of the tape with an arrow pointing to

// the current cell.  There are buttons for moving the current cell to

// the left and to the right.  A text-input box shows the content of

// the current cell.  This box can be edited, and there is a "Set" button

// that copies the contents of the cell (actually just the first character)

// to the current cell.

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

public class TestTapeGUI extends JPanel {

    public static void main(String[] args) {

        JFrame window = new JFrame("Test Tape");

        window.setContentPane( new TestTapeGUI("Test") );

        window.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

        window.pack();

        window.setLocation(100,100);

        window.setVisible(true);

    }

    private Tape tape;

    private TapePanel tapePanel;

    private JButton moveLeftButton, moveRightButton, setContentButton;

    private JTextField contentInput;

    private class TapePanel extends JPanel {

        public void paintComponent(Graphics g) {

            super.paintComponent(g);

            int mid = getWidth() / 2;

            g.drawLine(0,30,getWidth(),30);

            g.drawLine(0,60,getWidth(),60);

            g.drawLine(mid,70,mid,110);

            g.drawLine(mid,70,mid+15,85);

            g.drawLine(mid,70,mid-15,85);

            int ct = (mid / 30) + 1;

            for (int i = 0; i <= ct; i++) {

                g.drawLine(mid + 30\*i - 15, 30, mid + 30\*i - 15, 59);

                g.drawString("" + tape.getContent(), mid + 30\*i - 8, 53);

                tape.moveRight();

            }

            for (int i = 0; i <= ct; i++)

                tape.moveLeft();

            for (int i = 1; i <= ct; i++) {

                g.drawLine(mid - 30\*i - 15, 30, mid - 30\*i - 15, 59);

                tape.moveLeft();

                g.drawString("" + tape.getContent(), mid - 30\*i - 8, 53);

            }

            for (int i = 1; i <= ct; i++)

                tape.moveRight();

        }

    }

    private class ButtonListener implements ActionListener {

        public void actionPerformed(ActionEvent evt) {

            if (evt.getSource() == moveLeftButton)

                tape.moveLeft();

            else if (evt.getSource() == moveRightButton)

                tape.moveRight();

            else {

                String content = contentInput.getText();

                if (content.length() == 0)

                    tape.setContent(' ');

                else

                    tape.setContent(content.charAt(0));

            }

            contentInput.setText("" + tape.getContent());

            contentInput.selectAll();

            contentInput.requestFocus();

            tapePanel.repaint();

        }

    }

    public TestTapeGUI(String initialContent) {

        tape = new Tape();

        if (initialContent != null && initialContent.length() > 0) {

            for (int i = 0; i < initialContent.length(); i++) {

                tape.setContent(initialContent.charAt(i));

                tape.moveRight();

            }

            tape.moveLeft();  // move back over last character written

        }

        ButtonListener listener = new ButtonListener();

        tapePanel = new TapePanel();

        tapePanel.setPreferredSize(new Dimension(500,130));

        tapePanel.setFont(new Font("Serif", Font.PLAIN, 24));

        tapePanel.setBackground(new Color(180,180,255));

        tapePanel.setBorder(BorderFactory.createLineBorder(Color.BLUE, 2));

        moveLeftButton = new JButton("Left");

        moveLeftButton.addActionListener(listener);

        moveRightButton = new JButton("Right");

        moveRightButton.addActionListener(listener);

        setContentButton = new JButton("Set");

        setContentButton.addActionListener(listener);

        contentInput = new JTextField(1);

        contentInput.setText("" + tape.getContent());

        contentInput.setFont(new Font("Serif", Font.PLAIN, 18));

        contentInput.addActionListener(listener);

        JPanel bottom = new JPanel();

        bottom.add(moveLeftButton);

        bottom.add(moveRightButton);

        bottom.add(Box.createHorizontalStrut(15));

        bottom.add(new JLabel("Content:"));

        bottom.add(contentInput);

        bottom.add(setContentButton);

        setLayout(new BorderLayout());

        add(tapePanel,BorderLayout.CENTER);

        add(bottom,BorderLayout.SOUTH);

    }

}

// package turing;

// A test program for the TuringMachine class.  It creates three machines

// and runs them.  The output from the program indictes the expected behavior.

public class TestTuringMachine {

    public static void main(String[] args) {

        TuringMachine writeMachine = new TuringMachine();

        writeMachine.addRules( new Rule[] {  // writes Hello on the tape then halts

                new Rule(0,' ',1,'H',false),

                new Rule(1,' ',2,'e',false),

                new Rule(2,' ',3,'l',false),

                new Rule(3,' ',4,'l',false),

                new Rule(4,' ',-1,'o',false)

        });

        System.out.println("Running machine #1.  Output should be:  Hello");

        String writeMachineOutput = writeMachine.run(new Tape());

        System.out.println( "Actual output is:                       " + writeMachineOutput );

        TuringMachine badMachine = new TuringMachine();

        badMachine.addRules( new Rule[] {  // writes ERROR on the tape then fails

                new Rule(0,' ',1,'R',true),

                new Rule(1,' ',2,'O',true),

                new Rule(2,' ',3,'R',true),

                new Rule(3,' ',4,'R',true),

                new Rule(4,' ',5,'E',true) // no rule for state 5!

        });

        System.out.println("\nRunning machine #2.  Should throw an IllegalStateExcpetion.");

        try {

            badMachine.run( new Tape() );

            System.out.println("No Error was thrown.");

        }

        catch (IllegalStateException e) {

            System.out.println("Caught Illegal Argument Exception, with error message:");

            System.out.println(e.getMessage());

        }

        String input = "aababbbababbabbaba";  // a string of a's and b's for input to the copy machine

        Tape tape = new Tape();

        for (int i = 0; i < input.length(); i++) {

            tape.setContent(input.charAt(i));

            tape.moveRight();

        }

        tape.moveLeft();  // now, input is written on the tape, with the machine on the rightmost character

        TuringMachine copyMachine = new TuringMachine();  // copies a string of a's and b's;

                                                          // machine must start on leftmost char in the string

        copyMachine.addRules(new Rule[] {

                new Rule(0,'a',1,'x',true),  // rules for copying an a

                new Rule(1,'a',1,'a',true),

                new Rule(1,'b',1,'b',true),

                new Rule(1,' ',2,' ',true),

                new Rule(2,'a',2,'a',true),

                new Rule(2,'b',2,'b',true),

                new Rule(2,' ',3,'a',false),

                new Rule(3,'a',3,'a',false),

                new Rule(3,'b',3,'b',false),

                new Rule(3,' ',3,' ',false),

                new Rule(3,'x',0,'x',true),

                new Rule(3,'y',0,'y',true),

                new Rule(0,'b',4,'y',true),  // rules for copying a b

                new Rule(4,'a',4,'a',true),

                new Rule(4,'b',4,'b',true),

                new Rule(4,' ',5,' ',true),

                new Rule(5,'a',5,'a',true),

                new Rule(5,'b',5,'b',true),

                new Rule(5,' ',7,'b',false),

                new Rule(7,'a',7,'a',false),

                new Rule(7,'b',7,'b',false),

                new Rule(7,' ',7,' ',false),

                new Rule(7,'x',0,'x',true),

                new Rule(7,'y',0,'y',true),

                new Rule(0,' ',8,' ',false),  // rules that change x and y back to a and b, then halt

                new Rule(8,'x',8,'a',false),

                new Rule(8,'y',8,'b',false),

                new Rule(8,' ',-1,' ',true)

        });

        System.out.println("\nRunning machine #3.  Output should be: " + input + " " + input);

        String copyMachineOutput = copyMachine.run(tape);

        System.out.println("Actual output is:                      " + copyMachineOutput);

    }

}

// package turing;

import java.util.ArrayList;

/\*\*

 \* This class represents Turing machines.  A Turing machine is a simple

 \* computing device that moves moves back and forth along a Tape

 \* (see {@link Tape}), reading and writing characters.  The machine

 \* has a state, which is represented as an integer.  It also has

 \* a program, which consists of a list of rules (@see {@link Rule}).

 \*/

public class TuringMachine {

    private ArrayList<Rule> rules = new ArrayList<Rule>();  // This machine's program.

    /\*\*

     \* Adds one rule to the machine's program.

     \* @param rule A non-null rule to be added to the program.  A null

     \* value will not cause an immediate error, but will produce NullPointExceptions

     \* when run() method is called.

     \*/

    public void addRule(Rule rule) {

        rules.add(rule);

    }

    /\*\*

     \* A convenience method for adding an entire array of rules to this machine's program.

     \* This method simply calls {@link #addRule(Rule)} for each rule.

     \* @param rules  The array of rules that are to be added to the program.

     \* Each rule should be non-null.

     \*/

    public void addRules(Rule[] rules) {

        for (Rule rule : rules)

            addRule(rule);

    }

    /\*\*

     \* Run this Turing machine on a given Tape.  The machine starts in state

     \* zero and continues as long as the state is greater than or equal to

     \* zero.  At each step of the computation, it finds the applicable rule

     \* in its program (the one whose currentState variable matches the machine's

     \* current state and whose currentContent value matches the content of

     \* the current Cell on the tape), and it executes the action part of the

     \* rule (by setting the state of the machine to the rule's newState

     \* variable, and setting the content of the current Cell to the rule's

     \* newContent variable, and moving either left or right, depending on the

     \* value of the rule's moveLeft variable.  (See {@link Rule}.)

     \* Note that it is possible for this method to run forever, if the

     \* state of the machine never becomes negative.

     \* @param tape  The tape on which this machine will run.  The position of

     \* the current cell on the tape and the content of all the cells on the

     \* tape when this method is called constitute the input to the machine's

     \* computation.

     \* @return the content of the tape at the end of the computation.  The return

     \* value is obtained by calling {@link Tape#getTapeContents()}.

     \* @throws IllegalStateException if at any point during the computation,

     \* no applicable rule can be found.  This is taken to indicate a bug in the

     \* machine's program.

     \*/

    public String run(Tape tape) throws IllegalStateException {

        int currentState = 0;

        while (currentState >= 0) {

            char currentContent = tape.getContent();

            Rule applicableRule = null;

            for (Rule rule : rules) {

                if (rule.currentContent == currentContent && rule.currentState == currentState) {

                    applicableRule = rule;

                    break;

                }

            }

            if (applicableRule == null)

                throw new IllegalStateException("Cannot find an applicable rule; tape contents = "

                        + tape.getTapeContents());

            currentState = applicableRule.newState;

            tape.setContent(applicableRule.newContent);

            if (applicableRule.moveLeft)

                tape.moveLeft();

            else

                tape.moveRight();

//          System.out.println(applicableRule.currentState + " "+ applicableRule.currentContent

//                  + " " +applicableRule.newState + " " +applicableRule.newContent

//                  + " " +applicableRule.moveLeft);  // for testing.

        }

        return tape.getTapeContents();

    }

}

// package turing;

/\*\*

 \* Represents a Turing machine tape using a doubly-linked list.

 \*/

public class Tape {

    private Cell currentCell;

    /\*\*

     \* Constructs a tape that initially consists of a single cell.

     \* The cell contains a blank space, and the current cell pointer points to it.

     \*/

    public Tape() {

        currentCell = new Cell();

    }

    /\*\*

     \* Returns the pointer that points to the current cell.

     \*

     \* @return The pointer to the current cell.

     \*/

    public Cell getCurrentCell() {

        return currentCell;

    }

    /\*\*

     \* Returns the character from the current cell.

     \*

     \* @return The character from the current cell.

     \*/

    public char getContent() {

        return currentCell.content;

    }

    /\*\*

     \* Changes the character in the current cell to the specified value.

     \*

     \* @param ch The new character for the current cell.

     \*/

    public void setContent(char ch) {

        currentCell.content = ch;

    }

    /\*\*

     \* Moves the current cell one position to the left along the tape.

     \* If the current cell is the leftmost cell, a new cell is created

     \* and added to the tape at the left of the current cell.

     \* The content of the new cell is a blank space.

     \*/

    public void moveLeft() {

        if (currentCell.prev == null) {

            Cell newCell = new Cell();

            newCell.next = currentCell;

            currentCell.prev = newCell;

        }

        currentCell = currentCell.prev;

    }

    /\*\*

     \* Moves the current cell one position to the right along the tape.

     \* If the current cell is the rightmost cell, a new cell is created

     \* and added to the tape at the right of the current cell.

     \* The content of the new cell is a blank space.

     \*/

    public void moveRight() {

        if (currentCell.next == null) {

            Cell newCell = new Cell();

            newCell.prev = currentCell;

            currentCell.next = newCell;

        }

        currentCell = currentCell.next;

    }

    /\*\*

     \* Returns a String consisting of the characters from all the cells on the tape,

     \* read from left to right. Leading or trailing blank characters are discarded.

     \*

     \* @return The tape contents as a String.

     \*/

    public String getTapeContents() {

        StringBuilder tapeContents = new StringBuilder();

        Cell pointer = currentCell;

        // Move to the leftmost non-blank cell

        while (pointer.prev != null && pointer.prev.content != ' ') {

            pointer = pointer.prev;

        }

        // Append non-blank characters to the StringBuilder

        while (pointer != null && pointer.content != ' ') {

            tapeContents.append(pointer.content);

            pointer = pointer.next;

        }

        return tapeContents.toString();

    }

}

// package turing;

/\*\*

 \* Represents one of the rules of a Turing Machine.

 \* The rule applies when the machine's state is equal to currentState and

 \* the character in the current cell on the tape is equal to currentContent.

 \* The rule says that the machine will change to state newState, will

 \* write newContent into the current cell on the tape, and will then move

 \* eitehr to the left or to the right on the tape, depending on whether

 \* the value of moveLeft is true or false.

 \*/

public class Rule {

    public int currentState;

    public char currentContent;

    public int newState;

    public char newContent;

    public boolean moveLeft;

    /\*\*

     \* Create a rule with default values for the instance variables.

     \*/

    public Rule() {

    }

    /\*\*

     \* Create a rule with specified values for the instance variables.

     \*/

    public Rule(int currentState, char currentContent, int newState, char newContent, boolean moveLeft) {

        this.currentState = currentState;

        this.currentContent = currentContent;

        this.newState = newState;

        this.newContent = newContent;

        this.moveLeft = moveLeft;

    }

}

/\*\*

 \* A class for experimenting with expression trees.  This class includes

 \* a nested abstract class and several subclasses that represent nodes in

 \* an expression tree.  It also includes several methods that work with these

 \* classes.

 \*/

public class Expressions {

    /\*\*

     \* The main routine tests some of the things that are defined in this class.

     \*/

    public static void main(String[] args) {

        System.out.println("Testing expression creation and evaluation...\n");

        ExpNode e1 = new BinOpNode('+', new VariableNode(), new ConstNode(3));

        ExpNode e2 = new BinOpNode('\*', new ConstNode(2), new VariableNode());

        ExpNode e3 = new BinOpNode('-', e1, e2);

        ExpNode e4 = new BinOpNode('/', e1, new ConstNode(-3));

        System.out.println("For x = 3:");

        System.out.println("   " + e1 + " = " + e1.value(3));

        System.out.println("   " + e2 + " = " + e2.value(3));

        System.out.println("   " + e3 + " = " + e3.value(3));

        System.out.println("   " + e4 + " = " + e4.value(3));

        System.out.println("\nTesting copying...");

        System.out.println("   copy of " + e1 + " gives " + copy(e1));

        System.out.println("   copy of " + e2 + " gives " + copy(e2));

        System.out.println("   copy of " + e3 + " gives " + copy(e3));

        System.out.println("   copy of " + e4 + " gives " + copy(e4));

        ExpNode e3copy = copy(e3);  // make a copy of e3, where e3.left is e1

        ((BinOpNode)e1).left = new ConstNode(17);  // make a modification to e1

        System.out.println("   modified e3: " + e3 + "; copy should be unmodified: " + e3copy);

        System.out.println("\nChecking test data...");

        double[][] dt = makeTestData();

        for (int i = 0; i < dt.length; i++) {

            System.out.println("   x = " + dt[i][0] + "; y = " + dt[i][1]);

        }

    }

    /\*\*

     \* Given an ExpNode that is the root of an expression tree, this method

     \* makes a full copy of the tree.  The tree that is returned is constructed

     \* entirely of freshly made nodes.  (That is, there are no pointers back

     \* into the old copy.)

     \*/

    static ExpNode copy(ExpNode root) {

        if (root instanceof ConstNode)

            return new ConstNode(((ConstNode)root).number);

        else if (root instanceof VariableNode)

            return new VariableNode();

        else {

            BinOpNode node = (BinOpNode)root;

            // Note that left and right operand trees have to be COPIED,

            // not just referenced.

            return new BinOpNode(node.op, copy(node.left), copy(node.right));

        }

    }

    /\*\*

     \* Returns an n-by-2 array containing sample input/output pairs. If the

     \* return value is called data, then data[i][0] is the i-th input (or x)

     \* value and data[i][1] is the corresponding output (or y) value.

     \* (This method is currently unused, except to test it.)

     \*/

    static double[][] makeTestData() {

        double[][] data = new double[21][2];

        double xmax = 5;

        double xmin = -5;

        double dx = (xmax - xmin) / (data.length - 1);

        for (int i = 0; i < data.length; i++) {

            double x = xmin + dx \* i;

            double y = 2.5\*x\*x\*x - x\*x/3 + 3\*x;

            data[i][0] = x;

            data[i][1] = y;

        }

        return data;

    }

    //------------------- Definitions of Expression node classes ---------

    /\*\*

     \* An abstract class that represents a general node in an expression

     \* tree.  Every such node must be able to compute its own value at

     \* a given input value, x.  Also, nodes should override the standard

     \* toString() method to return a fully parameterized string representation

     \* of the expression.

     \*/

    static abstract class ExpNode {

        abstract double value(double x);

        // toString method should also be defined in each subclass

    }

    /\*\*

     \* A node in an expression tree that represents a constant numerical value.

     \*/

    static class ConstNode extends ExpNode {

        double number;  // the number in this node.

        ConstNode(double number) {

            this.number = number;

        }

        double value(double x) {

            return number;

        }

        public String toString() {

            if (number < 0)

                return "(" + number + ")"; // add parentheses around negative number

            else

                return "" + number;  // just convert the number to a string

        }

    }

    /\*\*

     \* A node in an expression tree that represents the variable x.

     \*/

    static class VariableNode extends ExpNode {

        VariableNode() {

        }

        double value(double x) {

            return x;

        }

        public String toString() {

            return "x";

        }

    }

    /\*\*

     \* A node in an expression tree that represents one of the

     \* binary operators +, -, \*, or /.

     \*/

    static class BinOpNode extends ExpNode {

        char op;  // the operator, which must be '+', '-', '\*', or '/'

        ExpNode left, right;  // the expression trees for the left and right operands.

        BinOpNode(char op, ExpNode left, ExpNode right) {

            if (op != '+' && op != '-' && op != '\*' && op != '/')

                throw new IllegalArgumentException("'" + op + "' is not a legal operator.");

            this.op = op;

            this.left = left;

            this.right = right;

        }

        double value(double x) {

            double a = left.value(x);  // value of the left operand expression tree

            double b = right.value(x); // value of the right operand expression tree

            switch (op) {

            case '+': return a + b;

            case '-': return a - b;

            case '\*': return a \* b;

            default:  return a / b;

            }

        }

        public String toString() {

            return "(" + left.toString() + op + right.toString() + ")";

        }

    }

}

// package turing;

/\*\*

 \* Represents one cell on a Turing Machine tape.

 \*/

public class Cell {

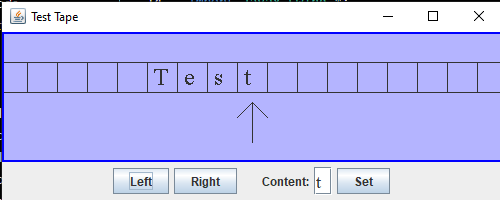
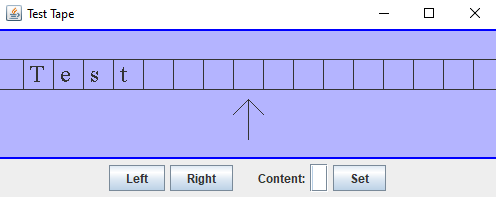
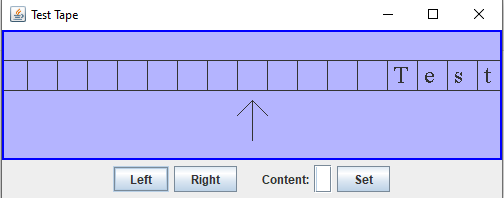
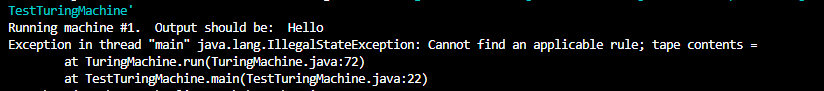
    public char content;  // The character in this cell.

    public Cell next;     // Pointer to the cell to the right of this one.

    public Cell prev;     // Pointer to the cell to the left of this one.

}

**Outputs**

  
  
  
  
  
  
  
  
  
  
  
**End**