Henry Osei

This exercise concerns TM M2, whose description and state diagram appear in Example 3.7. In each of the parts, give the sequence of configurations that M2 enters when started on the indicated input string.

c. 000.

d. 000000

c: 000

d:000000

c.

q1000

⊔q200

⊔xq30

⊔x0q4⊔

⊔x0⊔qreject

d.

q1000000 ⊔x0x0q40 ⊔xq50x0x⊔ ⊔xxq3x0x⊔

⊔q200000 ⊔x0x0xq3⊔ ⊔q5x0x0x⊔ ⊔xxxq30x⊔

⊔xq30000 ⊔x0x0xq5x⊔ q5⊔x0x0x⊔ ⊔xxx0q4x⊔

⊔xq4000 ⊔x0xq50x⊔ ⊔q2x0x0x⊔ ⊔xxx0xq4⊔

⊔x0xq300 ⊔x0xq5x0x⊔ ⊔xq20x0x⊔ ⊔xxx0x⊔qreject

Give implementation-level descriptions of Turing machines that decide the following languages over the alphabet {0,1}.

b. {w| w contains twice as many 0s as 1s}

1. Scan tape and find the first unmarked 1. If no unmarked 1’s are found skip to stage 5 . Otherwise move the head back to the start of the tape
2. Scan the tape until an unmarked 0 is found, mark the 0, if no 0’s are found (reject)
3. Scan the tape again until an unmarked zero is found, mark the 0, if no 0’s are found reject
4. Move head back to the start of tape and go to step 1
5. Move the head back to the start of the tape. Scan tape to see if any unmarked 0’s are found. Otherwise accept (reject) if none are found

**import java.util.\* ;**

**public class NFAstate**

**{**

**/\***

**\* WARNING:**

**\***

**\* If we were to use this on the Unicode character set, we'd get**

**\* an array index out-of-bounds exception.**

**\***

**\* A ``proper'' implementation of this would not use arrays but**

**\* rather a dynamic data structure like Vector.**

**\*/**

**public static final int MAX\_CHAR = 255 ;**

**public boolean isFinal = false ;**

**private ArrayList<NFAState> onChar[] = new ArrayList[MAX\_CHAR] ;**

**private ArrayList<NFAState> onEmpty = new ArrayList() ;**

**/\***

**\* Add a transition edge from this state to next which consumes**

**\* the character c.**

**\*/**

**public void addCharEdge(char c, NFAState next) {**

**onChar[(int)c].add(next) ;**

**}**

**/\***

**\* Add a transition edge from this state to next that does not**

**\* consume a character.**

**\*/**

**public void addEmptyEdge(NFAState next) {**

**onEmpty.add(next) ;**

**}**

**public NFAState () {**

**for (int i = 0; i < onChar.length; i++)**

**onChar[i] = new ArrayList() ;**

**}**

**public boolean matches(String s) {**

**return matches(s,new ArrayList()) ;**

**}**

**private boolean matches(String s, ArrayList visited) {**

**/\***

**\* When matching, we work character by character.**

**\***

**\* If we're out of characters in the string, we'll check to**

**\* see if this state if final, or if we can get to a final**

**\* state from here through empty edges.**

**\***

**\* If we're not out of characters, we'll try to consume a**

**\* character and then match what's left of the string.**

**\***

**\* If that fails, we'll ask if empty-edge neighbors can match**

**\* the entire string.**

**\***

**\* If that fails, the match fails.**

**\***

**\*/**

**if (visited.contains(this))**

**/\* We've found a path back to ourself through empty edges;**

**\* stop or we'll go into an infinite loop. \*/**

**return false ;**

**/\* In case we make an empty transition, we need to add this**

**\* state to the visited list. \*/**

**visited.add(this) ;**

**if (s.length() == 0) {**

**/\* The string is empty, so we match this string only if**

**\* this state is a final state, or we can reach a final**

**\* state without consuming any input. \*/**

**if (isFinal)**

**return true ;**

**/\* Since this state is not final, we'll ask if any**

**\* neighboring states that we can reach on empty edges can**

**\* match the empty string. \*/**

**for (NFAState next : onEmpty) {**

**if (next.matches("",visited))**

**return true ;**

**}**

**return false ;**

**} else {**

**/\* In this case, the string is not empty, so we'll pull**

**\* the first character off and check to see if our**

**\* neighbors for that character can match the remainder of**

**\* the string. \*/**

**int c = (int)s.charAt(0) ;**

**for (NFAState next : onChar[c]) {**

**if (next.matches(s.substring(1)))**

**return true ;**

**}**

**/\***

**\***

**\***

**\*/**

**for (NFAState next : onEmpty) {**

**if (next.matches(s,visited))**

**return true ;**

**}**

**return false ;**

**}**

**}**

**}**

**/\***

**\* Here, an NFA is represented by an entry state and an exit state.**

**\***

**\* Any NFA can be represented by an NFA with a single exit state by**

**\* creating a special exit state, and then adding empty transitions**

**\* from all final states to the special one.**

**\***

**\*/**

**public class NFA**

**{**

**public NFAState entry ;**

**public NFAState exit ;**

**public NFA(NFAState entry, NFAState exit) {**

**this.entry = entry ;**

**this.exit = exit;**

**}**

**public boolean matches(String str) {**

**return entry.matches(str);**

**}**

**/\***

**\* c() : Creates an NFA which just matches the character `c'.**

**\*/**

**public static final NFA c(char c) {**

**NFAState entry = new NFAState() ;**

**NFAState exit = new NFAState() ;**

**exit.isFinal = true ;**

**entry.addCharEdge(c,exit) ;**

**return new NFA(entry,exit) ;**

**}**

**/\***

**\* e() : Creates an NFA which matches the empty string.**

**\*/**

**public static final NFA e() {**

**NFAState entry = new NFAState() ;**

**NFAState exit = new NFAState() ;**

**entry.addEmptyEdge(exit) ;**

**exit.isFinal = true ;**

**return new NFA(entry,exit) ;**

**}**

**/\***

**\* rep() : Creates an NFA which matches zero or more repetitions**

**\* of the given NFA.**

**\*/**

**public static final NFA rep(NFA nfa) {**

**nfa.exit.addEmptyEdge(nfa.entry) ;**

**nfa.entry.addEmptyEdge(nfa.exit) ;**

**return nfa ;**

**}**

**/\***

**\* s() : Creates an NFA that matches a sequence of the two**

**\* provided NFAs.**

**\*/**

**public static final NFA s(NFA first, NFA second) {**

**first.exit.isFinal = false ;**

**second.exit.isFinal = true ;**

**first.exit.addEmptyEdge(second.entry) ;**

**return new NFA(first.entry,second.exit) ;**

**}**

**/\***

**\* or() : Creates an NFA that matches either provided NFA.**

**\*/**

**public static final NFA or(NFA choice1, NFA choice2) {**

**choice1.exit.isFinal = false ;**

**choice2.exit.isFinal = false ;**

**NFAState entry = new NFAState() ;**

**NFAState exit = new NFAState() ;**

**exit.isFinal = true ;**

**entry.addEmptyEdge(choice1.entry) ;**

**entry.addEmptyEdge(choice2.entry) ;**

**choice1.exit.addEmptyEdge(exit) ;**

**choice2.exit.addEmptyEdge(exit) ;**

**return new NFA(entry,exit) ;**

**}**

**/\* Syntactic sugar. \*/**

**public static final NFA re(Object o) {**

**if (o instanceof NFA)**

**return (NFA)o ;**

**else if (o instanceof Character)**

**return c((Character)o) ;**

**else if (o instanceof String)**

**return fromString((String)o) ;**

**else {**

**throw new RuntimeException("bad regexp") ;**

**}**

**}**

**public static final NFA or(Object... rexps) {**

**NFA exp = rexps[0] ;**

**for (int i = 1; i < rexps.length; i++) {**

**exp = or(exp,re(rexps[i])) ;**

**}**

**return exp ;**

**}**

**public static final NFA s(Object... rexps) {**

**NFA exp = e() ;**

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

**exp = s(exp,re(rexps[i])) ;**

**}**

**return exp ;**

**}**

**public static final NFA fromString(String str) {**

**if (str.length() == 0)**

**return e() ;**

**else**

**return s(re(str.charAt(0)),fromString(str.substring(1))) ;**

**}**

**public static void main(String[] args) {**

**NFA pat = s(rep(or("baba","baa")),"") ;**

**String[] strings =**

**{ "baba" , "baa" ,**

**"foobaar", "forbaba", "babaofar" , "barfoo" ,**

**"foofoobarfooX" ,**

**"foofoobaarfoo" ,**

**} ;**

**for (String s : strings) {**

**System.out.println(s + "\t:\t" +pat.matches(s)) ;**

**}**

**}**

**}**

**Library works by converting each regular expression into an NFA.**

**Library works by building syntax for regular expression in java**

**NFA pattern = (s(“baba”),s(“baba”); or s(“baa”, “baa”)**