```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;
using System.Collections.Generic;
using System.Threading.Tasks;
using System.IO;

namespace Assignment3Theory
{
    class Program
    {

        static void Main(string[] args)
        {
            ///this is the log that we will be passing around to create the output file
            StreamWriter writer = new StreamWriter("log.log");

            //do we have ore input what is the input
            bool keepGoing = true;
            int result;

            //what is our data
            List<State> workingList = new List<State>();

            //until we want to quit
            while (keepGoing)
            {
                //make sure data is valid choice
                bool invalid = false;

                //get user data and handle the file they choose
                Console.WriteLine("Which example would you like to convert to minimized DFA?\n");
                Console.WriteLine("(1) Example 3.4.1\n(2) Example 4.4.1\n(3) Example 4.4.2\n(4)
                Example 4.8\n(5) Example 4.10\n(6) Exit");
                if(int.TryParse(Console.ReadLine(), out result))
                {
                    switch(result)
                    {
                        case 1:
                            workingList = HandleFile("3.4.1_input.txt", writer);
                            writer.WriteLine("Using 3.4.1 Example");
                            break;
                        case 2:
                            workingList = HandleFile("4.4.1_input.txt", writer);
                            writer.WriteLine("Using 4.4.1 Example");
                            break;
                        case 3:
                            workingList = HandleFile("4.4.2_input.txt", writer);
                            writer.WriteLine("Using 4.4.2 Example");
                            break;
                        case 4:
```

```
                        workingList = HandleFile("4.8_input.txt", writer);
                        writer.WriteLine("Using 4.8 Example");
                        break;
                    case 5:
                        workingList = HandleFile("4.10_input.txt", writer);
                        writer.WriteLine("Using 4.10 Example");
                        break;
                    case 6:
                        keepGoing = false;
                        invalid = true;
                        break;
                    default:
                        invalid = true;
                        break;
                }

                //if they have made valid choice then get a string of input
                if (!invalid)
                {
                    Console.WriteLine("What is the string you would like (a's and b's)");

                    //does the input get accepted or declined?
                    string word = Console.ReadLine();
                    writer.WriteLine(word);
                    if (isValid(workingList, word))
                    {
                        Console.WriteLine("Accepted!");
                        writer.WriteLine("Accepted!");
                    }
                    else
                    {
                        Console.WriteLine("Declined!");
                        writer.WriteLine("Declined!");
                    }
                }
            }

        }

        //close writer when done
        writer.Close();
    }

    /// <summary>
    /// this method is used to parse the file
    /// </summary>
    /// <param name="fileName"></param>
    /// <returns></returns>
    public static List<State> HandleFile(string fileName, StreamWriter logWriter)
    {
        //input string
        string inputString = string.Empty;
```

```
            //stream reader and final state list
            StreamReader streamReader = new StreamReader(fileName);
            List<State> tempStatesList = new List<State>();

            //what line of input are we on
            int count = 0;
            int numberOfStates = 0;

            //lists for holding all incoming data
            List<string> stateNames = new List<string>();
            List<string> stateInfo = new List<string>();
            List<string> acceptingStates = new List<string>();

            //used for parsing
            bool doneWithStates = false;

            //while we still have input
            while (!streamReader.EndOfStream)
            {
                //deal with it
                inputString = streamReader.ReadLine();
                if (doneWithStates)
                {
                    acceptingStates = inputString.Split(',').ToList<string>();
                }
                else if (count > 1)
                {
                    stateInfo.Add(inputString);
                    count++;
                    if (count == (numberOfStates + 2))
                    {
                        doneWithStates = true;
                    }
                }
                //just input handling
                else if (count == 1)
                {
                    stateNames = inputString.Split(',').ToList<string>();
                    count++;
                }
                else
                {
                    numberOfStates = int.Parse(inputString);
                    count++;
                }

                logWriter.WriteLine(inputString);
            }

            tempStatesList = HandleString(numberOfStates, stateNames, stateInfo, acceptingStates
            , logWriter);

            return tempStatesList;
```

```
    }

    /// <summary>
    /// this method isused to handle all input and output it as the DFA
    /// </summary>
    /// <param name="file">All input from the file</param>
    public static List<State> HandleString(int count, List<string> names, List<string>
    states, List<string> acceptingStates, StreamWriter writer)
    {
        //list to hold our states
        List<State> tempList = new List<State>();

        //for every state
        for (int i = 0; i < count; i++)
        {
            //some paramaters are initialized
            bool isAccepting = false;
            string IncomingA = string.Empty;
            string IncomingB = string.Empty;

            //is the current state accepting
            for(int j = 0; j < acceptingStates.Count; j++)
            {
                if (String.Compare(names[i],acceptingStates[j]) == 0)
                {
                    isAccepting = true;
                    j = 100;
                }
            }

            //set up left and right next state names
            for (int j = 0; j < states.Count; j++)
            {
                var temp = states[j].Split(',');
                if (String.Compare(names[i], temp[0]) == 0)
                {
                    IncomingA = temp[1];
                    IncomingB = temp[2];
                }
            }

            //we have parsed the data correctly add the state object to the list
            tempList.Add(new State(names[i], IncomingA, IncomingB, isAccepting));
        }


        //set up pointers to next states and then start minimization
        return Minimize(SetUpPointer(tempList), writer);
    }

    //based on the names of the next states create pointers for the next state
    public static List<State> SetUpPointer(List<State> tempList)
    {
```

```
            //double loop so we can get them all
            for (int j = 0; j < tempList.Count; j++)
            {
                for (int i = 0; i < tempList.Count; i++)
                {
                    //compare it to the state name and if they match change the pointer to
                    point to it
                    if (string.Compare(tempList[j].StateA, tempList[i].stateName) == 0)
                    {
                        tempList[j].NextA = tempList[i];
                    }

                    //compare it to the state name and if they match change the pointer to
                    point to it
                    if (string.Compare(tempList[j].StateB, tempList[i].stateName) == 0)
                    {
                        tempList[j].NextB = tempList[i];
                    }

                }
            }

            //return the modified list
            return tempList;
        }

        /// <summary>
        /// minimize the DFA. This part sets up the equivilence table and calls the next part
        /// </summary>
        /// <param name="StatesList"></param>
        /// <returns></returns>
        public static List<State> Minimize(List<State> StatesList, StreamWriter writer)
        {
            //initial get into while loop true
            bool oneChanged = true;

            //create an empty list for the table
            //  the list is a 2d array created C style so one long array
            List<string> table = new List<string>();

            //every possible combination is represented here and put into the state table
            for (int j = 0; j < StatesList.Count; j++)
            {
                for (int i = 0; i < j; i++)
                {
                    //-1 means that it is not distinguishable
                    table.Add(StatesList[j].stateName + ',' + StatesList[i].stateName + ",-1");
                }
            }

            //word is used to find things that are distinguishable
            string word = string.Empty;
```

```
        //used to determine when a state was found to be distinguishable
        int turnCounter = 0;

        //if we dont have one change that means we never will
        while (oneChanged)
        {
            //counter used to navigate 2d array
            int counter = 0;

            //increase turn counter for current loop
            turnCounter++;

            //stop the looping unless we hit a match
            oneChanged = false;

            //start going through the lists
            for (int j = 1; j < StatesList.Count; j++)
            {
                for (int i = 0; i < j; i++)
                {
                    //determine if we found a good match
                    var stateOne = (WhereWouldIBe(StatesList[i], word));
                    var stateTwo = (WhereWouldIBe(StatesList[j], word));
                    string idCode = table[counter].Substring(table[counter].LastIndexOf(',')
                     + 1);
                    var alreadyMarked = (string.Compare(idCode, "-1") == 0);

                    //we found one fix it and keep looping
                    if ((stateOne.Final != stateTwo.Final) && alreadyMarked)
                    {
                        table[counter] = table[counter].Substring(0, table[counter].
                        LastIndexOf(',')) + ',' + turnCounter;
                        oneChanged = true;
                    }

                    //increment 2d list pointer
                    counter++;
                }
            }

            //increment the string to search harder
            word = IncrementString(word);
        }
        //print out the table for grading purposes
        PrintTable(StatesList, table, writer);

        //using this table fix the dfa to be minimized
        return AdjustDFA(StatesList, EquivilentStateFinder(table));
    }

    public static List<State> AdjustDFA(List<State> states, List<string> table)
    {
        //this first loop removes the duplicates created by minimization
```

```
            for (int i = 0; i < table.Count; i++)
            {
                var split = table[i].Split(',');
                for (int j = 0; j < states.Count; j++)
                {
                    if (states[j].stateName == split[0])
                    {
                        states.Remove(states[j]);
                        j = int.MaxValue - 1;
                    }
                }
            }

            //this reassigns a => a,h etc so that the DFA still works in the assignment 3 part
            for (int i = 0; i < table.Count; i++)
            {
                var split = table[i].Split(',');
                for (int j = 0; j < states.Count; j++)
                {
                    if (states[j].stateName == split[1])
                    {
                        states[j].stateName = split[1] + "," + split[0];
                    }
                    if (states[j].StateA == split[1] || states[j].StateA == split[0])
                    {
                        states[j].StateA = split[1] + "," + split[0];
                    }
                    if (states[j].StateB == split[1] || states[j].StateB == split[0])
                    {
                        states[j].StateB = split[1] + "," + split[0];
                    }
                }
            }

            //use pointers because they are great
            return SetUpPointer(states);
        }

        public static List<string> EquivilentStateFinder(List<string> table)
        {
            //remove any state that isn't distinguishable
            for (int i = 0; i < table.Count; i++)
            {
                var split = table[i].Split(',');
                if (split[2] != "-1")
                {
                    table.Remove(table[i]);
                    i--;
                }
            }

            return table;
        }
```

```
//borrowed from
http://codereview.stackexchange.com/questions/57452/binary-addition-with-strings
public static string BinAdd(string a, string b)
{
    if (a.Length < b.Length)
    {
        string c = a;
        a = b;
        b = c;
    }

    var sum = new char[a.Length + 1];
    bool carry = false;

    for (int i = a.Length - 1, j = b.Length - 1, k = sum.Length - 1; i >= 0; i--, j--, k
    --)
    {
        char x = a[i];
        char y = j >= 0 ? b[j] : '0';

        if (carry)
        {
            sum[k] = x == y ? '1' : '0';
            carry = x == '1' || y == '1';
        }
        else
        {
            sum[k] = x == y ? '0' : '1';
            carry = x == '1' && y == '1';
        }
    }

    if (carry)
    {
        sum[0] = '1';
        return new string(sum);
    }

    return new string(sum, 1, sum.Length - 1);
}

/// <summary>
/// this method increases the string just like incrementing a binary string but with
a's and b's
/// </summary>
/// <param name="word"></param>
/// <returns></returns>
public static string IncrementString(string word)
{
    if (string.Compare(word, string.Empty) == 0)
    {
        return "a";
```

```
        }
        else
        {
            word = word.Replace('a', '0');
            word = word.Replace('b', '1');

            word = BinAdd(word, "1");

            word = word.Replace('0', 'a');
            word = word.Replace('1', 'b');

            return word;
        }
    }

    /// <summary>
    /// this function is used for finding given a word what state would the given state be in
    /// </summary>
    /// <param name="parmState"></param>
    /// <param name="word"></param>
    /// <returns>the state that it would end in</returns>
    public static State WhereWouldIBe(State parmState, string word)
    {
        while (word.Length != 0)
        {
            if (string.Compare(word[0].ToString(), "a") == 0)
            {
                parmState = parmState.NextA;
            }
            else
            {
                parmState = parmState.NextB;
            }
            word = word.Substring(1);
        }

        return parmState;
    }

    public static void PrintTable(List<State> States, List<string> table, StreamWriter
    writer)
    {
        int counter = 0;
        writer.WriteLine("This is with table descriptive information\n");
        for (int i = 1; i < States.Count; i++)
        {
            writer.Write(States[i].stateName + " ");
            for (int j = 0; j < i; j++)
            {
                var split = table[counter].Split(',');
                if (split[2] == "-1")
                {
                    writer.Write('0');
```

```
                    }
                    else
                    {
                        writer.Write(split[2]);
                    }
                    counter++;
                }
                writer.WriteLine();
            }
            writer.Write("  ");
            for (int k = 0; k < States.Count - 1; k++)
            {
                writer.Write(States[k].stateName);
            }

            writer.WriteLine();
            counter = 0;
            writer.WriteLine("This is without table descriptive information\n");
            for (int i = 1; i < States.Count; i++)
            {
                for (int j = 0; j < i; j++)
                {
                    var split = table[counter].Split(',');
                    if (split[2] == "-1")
                    {
                        writer.Write('0');
                    }
                    else
                    {
                        writer.Write(split[2]);
                    }
                    counter++;
                }
                writer.WriteLine();
            }
            writer.WriteLine();
        }

        /// <summary>
        /// this determines if the state is accepting same as where would i be but returns a
        bool its legacy
        /// </summary>
        /// <param name="list"></param>
        /// <param name="word"></param>
        /// <returns></returns>
        public static bool isValid(List<State> list, string word)
        {
            State CurrentState = list[0];

            return WhereWouldIBe(CurrentState, word).Final;
        }
    }
}
```