|  |  |
| --- | --- |
| Compiler design lab programs | Ashhar Hasan  13 BCS 0015 |

Contents

[Implement a DFA: 2](#_Toc459658519)

[Implement a NFA: 7](#_Toc459658520)

[Implement a NFA to DFA converter: 12](#_Toc459658521)

[Implement a Mealy machine: 15](#_Toc459658522)

[Implement a Moore machine: 19](#_Toc459658523)

[Implement a regular expression: 23](#_Toc459658524)

# Implement a DFA:

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

#define MAX\_STATES 100

#define a\_ASCII 97

#define EOL '\n'

using namespace std;

struct split

{

enum empties\_t

{

empties\_ok,

no\_empties

};

};

template <typename Container>

Container& splitString(Container& result,

const typename Container::value\_type& str,

const typename Container::value\_type& delimiters,

split::empties\_t empties = split::empties\_ok)

{

result.clear();

size\_t current, next = -1;

do

{

if (empties == split::no\_empties)

{

next = str.find\_first\_not\_of(delimiters, next + 1);

if (next == Container::value\_type::npos)

break;

next -= 1;

}

current = next + 1;

next = str.find\_first\_of(delimiters, current);

result.push\_back(str.substr(current, next - current));

} while (next != Container::value\_type::npos);

return result;

}

class DFA

{

public:

DFA() : initState{0},

numStates{0},

numInputs{0}

{

}

bool readInput(string automataFile = "01-dfa.txt",

string delimiters = " ,");

bool process(string inputFile = "01-dfa.input",

bool batchMode = false);

private:

size\_t initState;

size\_t finalStates[MAX\_STATES];

int automata[MAX\_STATES][MAX\_STATES];

size\_t numStates;

size\_t numInputs;

bool matchFA(string str);

};

bool DFA::readInput(string automataFile, string delimiters)

{

// Open the file for reading and define the delimiters.

fstream fp(automataFile, ios::in);

string line;

// Read the initial state.

getline(fp, line, EOL);

initState = stoi(line);

// Read the final states.

getline(fp, line, EOL);

vector<string> fields;

splitString(fields, line, delimiters);

size\_t i = 0;

for (auto field : fields)

{

finalStates[i] = stoi(field);

i++;

}

// Read the automata array.

size\_t j;

i = 0;

while (!fp.eof())

{

// Read a line and parse it.

getline(fp, line, EOL);

splitString(fields, line, delimiters, split::no\_empties);

j = 0;

for (auto field : fields)

{

automata[i][j] = stoi(field);

j++;

}

// Read j only once since it will be set to 0 on the last iteration of the while loop.

if (i == 0)

numInputs = j;

i++;

}

numStates = i - 1;

return true;

}

bool DFA::matchFA(string line)

{

// Start matching the line against the automata.

int curState = initState;

for (size\_t i = 0; i < line.length(); i++)

{

curState = automata[curState][line[i] - a\_ASCII];

if (curState == -1)

return false;

}

for (int finalState : finalStates)

if (curState == finalState)

return true;

return false;

}

bool DFA::process(string inputFile, bool batchMode)

{

if (batchMode)

{

fstream fp(inputFile, ios::in);

string line;

while (!fp.eof())

{

// Read a line from input file and run it against the automata.

getline(fp, line, EOL);

// If we read an empty line, we skip it.

if (line.empty())

continue;

if (matchFA(line))

cout << "Matched: " << line << endl;

else

cout << "Not Matched: " << line << endl;

}

}

else

{

string line;

getline(cin, line);

if (matchFA(line))

cout << "Matched: " << line << endl;

else

cout << "Not Matched: " << line << endl;

}

return true;

}

int main(int argc, char\*\* argv)

{

DFA fa;

switch (argc)

{

case 1:

// No arguments means the automata will be read from "01-dfa.txt" and input from stdin.

fa.readInput();

while (1)

fa.process();

case 2:

// One argument means the automata will be read from "01-dfa.txt" and input from the file specified.

fa.readInput();

fa.process(argv[1], true);

break;

case 3:

// Two arguments means the automata and input will be read from the arguments.

fa.readInput(argv[2]);

fa.process(argv[1], true);

break;

}

}

/////////////////////////////////////////////////////////////////////////////// 01-dfa.txt //

/////////////////////////////////////////////////////////////////////////////

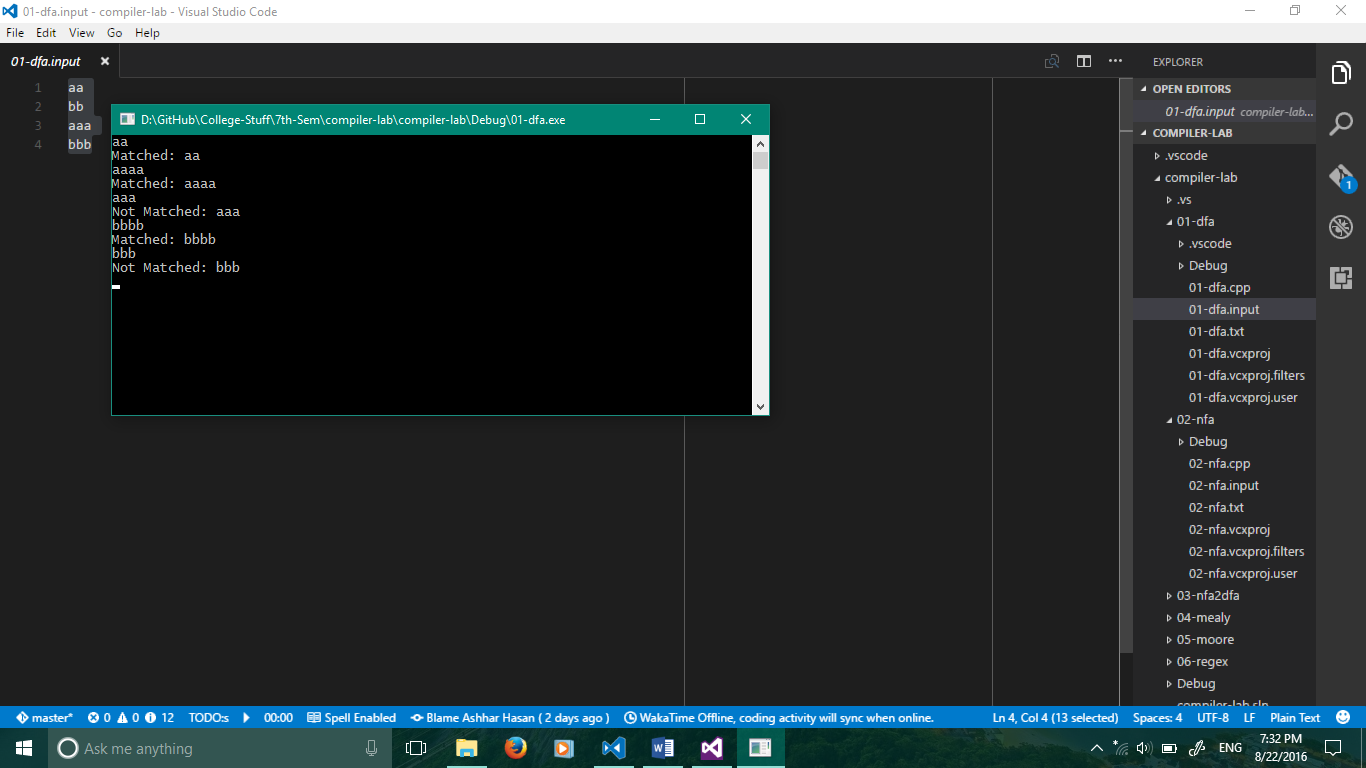
0

2

1 1

2 2

1 1



# Implement a NFA:

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

#include <map>

#include <set>

#include <iterator>

#include <algorithm>

#define a\_ASCII 97

#define EOL '\n'

using namespace std;

struct split

{

enum empties\_t

{

empties\_ok,

no\_empties

};

};

template <typename Container>

Container& splitString(Container& result,

const typename Container::value\_type& str,

const typename Container::value\_type& delimiters,

split::empties\_t empties = split::empties\_ok)

{

result.clear();

size\_t current, next = -1;

do

{

if (empties == split::no\_empties)

{

next = str.find\_first\_not\_of(delimiters, next + 1);

if (next == Container::value\_type::npos)

break;

next -= 1;

}

current = next + 1;

next = str.find\_first\_of(delimiters, current);

result.push\_back(str.substr(current, next - current));

} while (next != Container::value\_type::npos);

return result;

}

class NFA

{

public:

NFA() : initState{0},

numStates{0},

numInputs{0}

{

}

bool readInput(string automataFile = "02-nfa.txt",

string delimiterOuter = " ",

string delimiterInner = ",");

bool process(string inputFile = "02-nfa.input",

bool batchMode = false);

private:

size\_t initState;

vector<size\_t> finalStates;

multimap<pair<int, int>, int> automata;

size\_t numStates;

size\_t numInputs;

bool matchFA(string str);

};

bool NFA::readInput(string automataFile, string delimiterOuter, string delimiterInner)

{

// Open the file for reading and define the delimiters.

fstream fp(automataFile, ios::in);

string line;

// Read the initial state.

getline(fp, line, EOL);

initState = stoi(line);

// Read the final states.

getline(fp, line, EOL);

vector<string> fields;

splitString(fields, line, delimiterOuter);

for (auto field : fields)

finalStates.push\_back(stoi(field));

size\_t i = 0, j = 0;

// Read the transitions.

while (!fp.eof())

{

// Read a line and parse it.

getline(fp, line, EOL);

splitString(fields, line, delimiterOuter, split::no\_empties);

j = 0;

for (auto field : fields)

{

if (field.find(",") != string::npos)

{

// A comma was found, so we treat this differently and split on commas.

vector<string> innerFields;

splitString(innerFields, field, delimiterInner, split::no\_empties);

for (auto innerField : innerFields)

automata.insert(pair<pair<int, int>, int>(pair<int, int>(i, j), stoi(innerField)));

}

else

{

automata.insert(pair<pair<int, int>, int>(pair<int, int>(i, j), stoi(field)));

j++;

}

}

if (line.find(",") == string::npos && !line.empty())

// No comma found and line not empty.

numInputs = j;

i++;

}

numStates = i - 1;

return true;

}

bool NFA::matchFA(string line)

{

set<int> curStates = {}, nextStates = {};

// Add the initial state to the reachable current states.

curStates.insert(initState);

for (size\_t i = 0; i < line.length(); i++)

{

nextStates.clear();

auto input = line[i] - a\_ASCII;

for (auto curState : curStates)

{

if (curState == -1)

continue;

// Find all curState, input pairs in automata and add all reachable states as nextStates.

auto iterpair = automata.equal\_range(pair<int, int>(curState, input));

for (auto it = iterpair.first; it != iterpair.second; ++it)

nextStates.insert(it->second);

}

curStates = nextStates;

}

set<int> resSet = {};

set<int> finals(finalStates.begin(), finalStates.end());

set\_intersection(curStates.begin(), curStates.end(), finals.begin(), finals.end(), inserter(resSet, resSet.end()));

if (resSet.size() > 0)

return true;

return false;

}

bool NFA::process(string inputFile, bool batchMode)

{

if (batchMode)

{

fstream fp(inputFile, ios::in);

string line;

while (!fp.eof())

{

// Read a line from input file and run it against the automata.

getline(fp, line, EOL);

// If we read an empty line, we skip it.

if (line.empty())

continue;

if (matchFA(line))

cout << "Matched: " << line << endl;

else

cout << "Not Matched: " << line << endl;

}

}

else

{

string line;

getline(cin, line);

if (matchFA(line))

cout << "Matched: " << line << endl;

else

cout << "Not Matched: " << line << endl;

}

return true;

}

int main(int argc, char\*\* argv)

{

NFA fa;

switch (argc)

{

case 1:

// No arguments means the automata will be read from "02-nfa.txt" and input from stdin.

fa.readInput();

while (1)

fa.process();

case 2:

// One argument means the automata will be read from "02-nfa.txt" and input from the file specified.

fa.readInput();

fa.process(argv[1], true);

break;

case 3:

// Two arguments means the automata and input will be read from the arguments.

fa.readInput(argv[2]);

fa.process(argv[1], true);

break;

}

}

/////////////////////////////////////////////////////////////////////////////// 02-nfa.txt //

/////////////////////////////////////////////////////////////////////////////

0

1 3

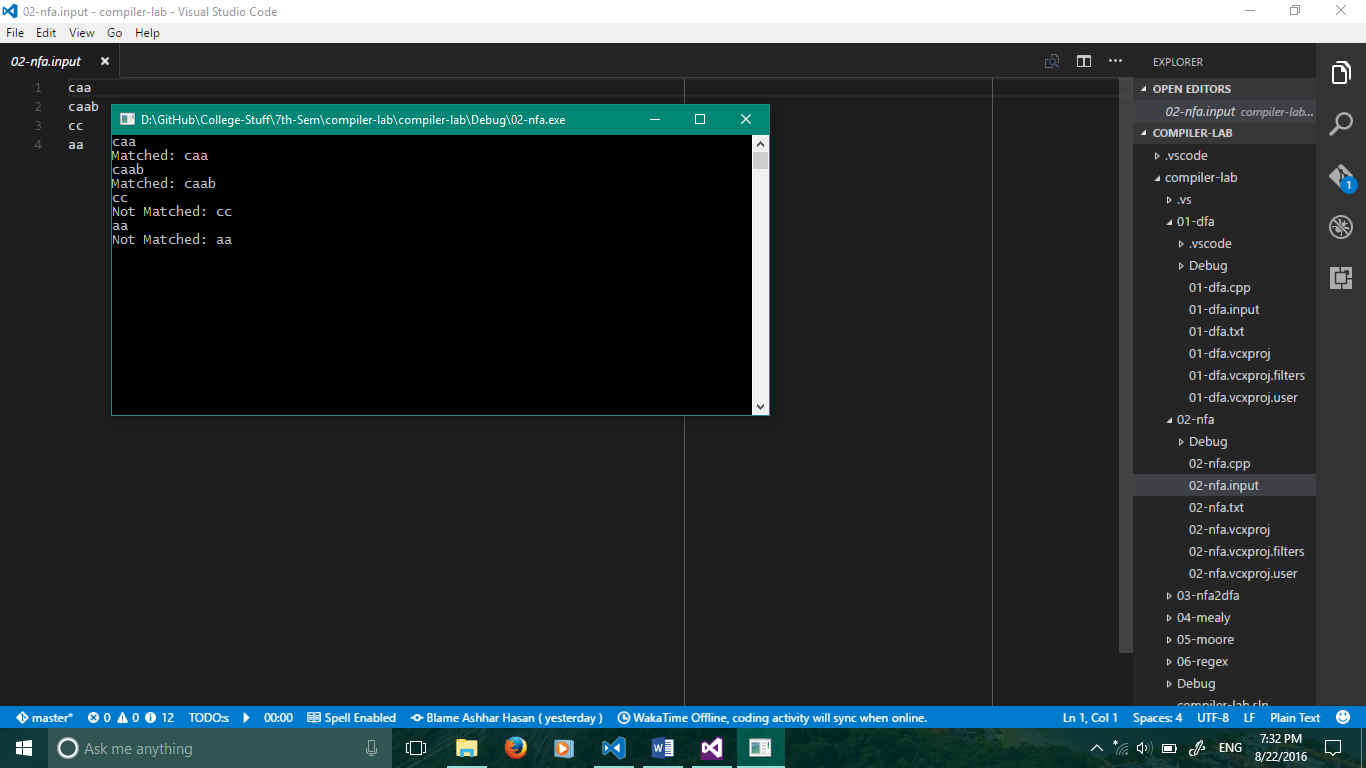
-1 -1 1,3

2 1 -1

1 2 -1

3 4 -1

4 3 -1



# Implement a NFA to DFA converter:

#include<iostream>

#include<vector>

#include<fstream>

#include<map>

#include<queue>

using namespace std;

int states, variables;

map<int, int> mp;

queue<int> q;

map<int, int>::iterator im;

vector<vector<int>> v;

void nfa\_dfa()

{

int combine, pos;

while (!q.empty())

{

combine = q.front();

q.pop();

pos = 0;

if (mp.find(combine) == mp.end())

{

mp[combine] = ++states;

vector<int> vt(variables, 0);

while (combine)

{

if ((combine & 1))

{

for (int j = 0; j < variables; j++)

vt[j] |= v[pos][j];

}

combine = (combine >> 1);

pos++;

}

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

q.push(vt[i]);

v.push\_back(vt);

}

}

}

int main()

{

int i, j, temp;

ifstream inp("03-nfa2dfa.txt", ios::in);

inp >> states >> variables;

v.resize(states);

for (i = 0; i < states; i++)

{

v[i].resize(variables);

}

//we can use push\_back here too.

mp[0] = 0;

temp = 1;

for (i = 0; i < states; i++)

{

mp[temp] = i + 1;

temp = temp << 1;

}

for (i = 0; i < states; i++)

{

for (j = 0; j < variables; j++)

{

inp >> v[i][j];

if (v[i][j] != 0)

{

temp = v[i][j];

while (temp % 2 == 0) temp /= 2;

if (temp != 1)

{

q.push(v[i][j]);

}

}

}

}

inp.close();

nfa\_dfa();

/\*cout << "Binary representations of each number in the DFA will give you the state combination. ";

cout << "Ex: 4 is 100 thus state q2, 5 id 101 thus q2 and q0\n";\*/

cout << "\nThus all the states of DFA are:\n";

for (im = mp.begin(); im != mp.end(); im++)

{

cout << im->first << endl;

}

cout << "\nThe final DFA table:\n\n";

for (i = 0; i < states; i++)

{

for (j = 0; j < variables; j++)

{

cout << v[i][j] << " ";

}

cout << endl;

}

getchar();

return 0;

}

/////////////////////////////////////////////////////////////////////////////// 03-nfa2dfa.txt //

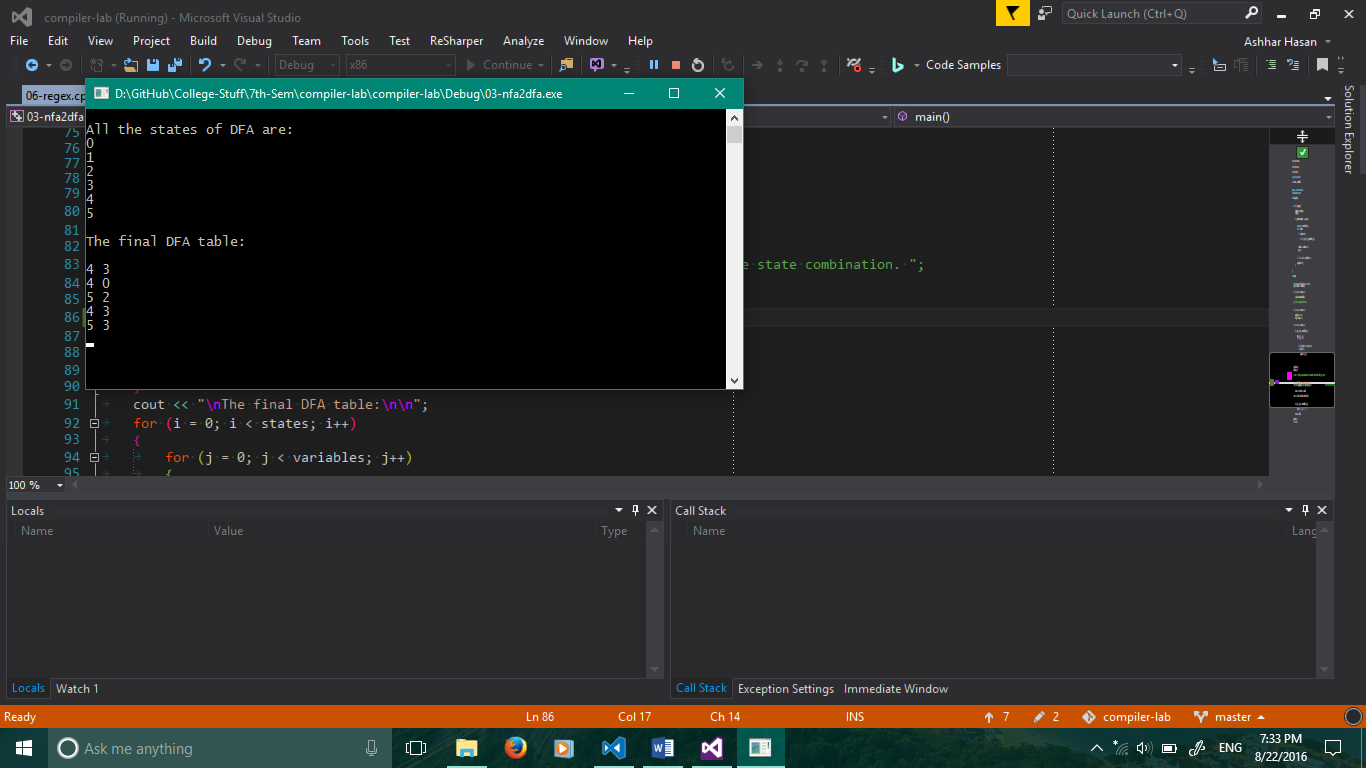
/////////////////////////////////////////////////////////////////////////////

3 2

4 3

4 0

5 2



# Implement a Mealy machine:

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

#define MAX\_STATES 100

#define numInputs 2

#define numOutputs 2

#define ZERO\_ASCII 48

#define EOL '\n'

using namespace std;

struct split

{

enum empties\_t

{

empties\_ok,

no\_empties

};

};

template <typename Container>

Container& splitString(Container& result,

const typename Container::value\_type& str,

const typename Container::value\_type& delimiters,

split::empties\_t empties = split::empties\_ok)

{

result.clear();

size\_t current, next = -1;

do

{

if (empties == split::no\_empties)

{

next = str.find\_first\_not\_of(delimiters, next + 1);

if (next == Container::value\_type::npos)

break;

next -= 1;

}

current = next + 1;

next = str.find\_first\_of(delimiters, current);

result.push\_back(str.substr(current, next - current));

} while (next != Container::value\_type::npos);

return result;

}

class Mealy

{

public:

Mealy() : initState{0},

curState{0},

numStates{0}

{

}

bool readInput(string automataFile = "04-mealy.txt",

string delimiters = " ,");

bool process(string inputFile = "04-mealy.input",

bool batchMode = false);

private:

size\_t initState;

int automata[MAX\_STATES][MAX\_STATES];

int curState;

size\_t numStates;

string runMealy(string inputs);

};

bool Mealy::readInput(string automataFile, string delimiters)

{

// Open the file for reading.

fstream fp(automataFile, ios::in);

string line;

// Read the initial state.

getline(fp, line, EOL);

initState = stoi(line);

// Read the automata array.

vector<string> fields;

size\_t i = 0, j;

while (!fp.eof())

{

getline(fp, line, EOL);

splitString(fields, line, delimiters, split::no\_empties);

j = 0;

for (auto field : fields)

{

automata[i][j] = stoi(field);

j++;

}

i++;

}

numStates = i - 1;

return true;

}

bool Mealy::process(string inputFile, bool batchMode)

{

if (batchMode)

{

fstream fp(inputFile, ios::in);

string line;

while (!fp.eof())

{

// Read a line from input file and run it against the automata.

getline(fp, line, EOL);

// If we read an empty line, we skip it.

if (line.empty())

continue;

// Process the automata.

cout << "Output: " << runMealy(line) << endl << endl;

}

}

else

{

string line;

getline(cin, line);

// If we read an empty line, we skip it.

if (line.empty())

return false;

// Process the automata.

cout << "Output: " << runMealy(line) << endl << endl;

}

return true;

}

// Take a string of integers as input and output corresponding outputs as string of integers.

string Mealy::runMealy(string inputs)

{

curState = initState;

string output = "";

for (auto input\_char : inputs)

{

auto input = input\_char - ZERO\_ASCII;

output.append(to\_string(automata[curState][2 \* input + 1]));

curState = automata[curState][2 \* input];

if (curState == -1)

{

cout << "Invalid Input! No transitions found." << endl;

return output;

}

}

return output;

}

int main(int argc, char\*\* argv)

{

Mealy mealy;

switch (argc)

{

case 1:

// No arguments means the automata will be read from file and input from stdin.

mealy.readInput();

while (1)

mealy.process();

case 2:

// One argument means the automata will be read from file and input from the file specified.

mealy.readInput();

mealy.process(argv[1], true);

break;

case 3:

// Two arguments means the automata and input will be read from the arguments.

mealy.readInput(argv[2]);

mealy.process(argv[1], true);

break;

}

}

/////////////////////////////////////////////////////////////////////////////// 04-mealy.txt //

/////////////////////////////////////////////////////////////////////////////

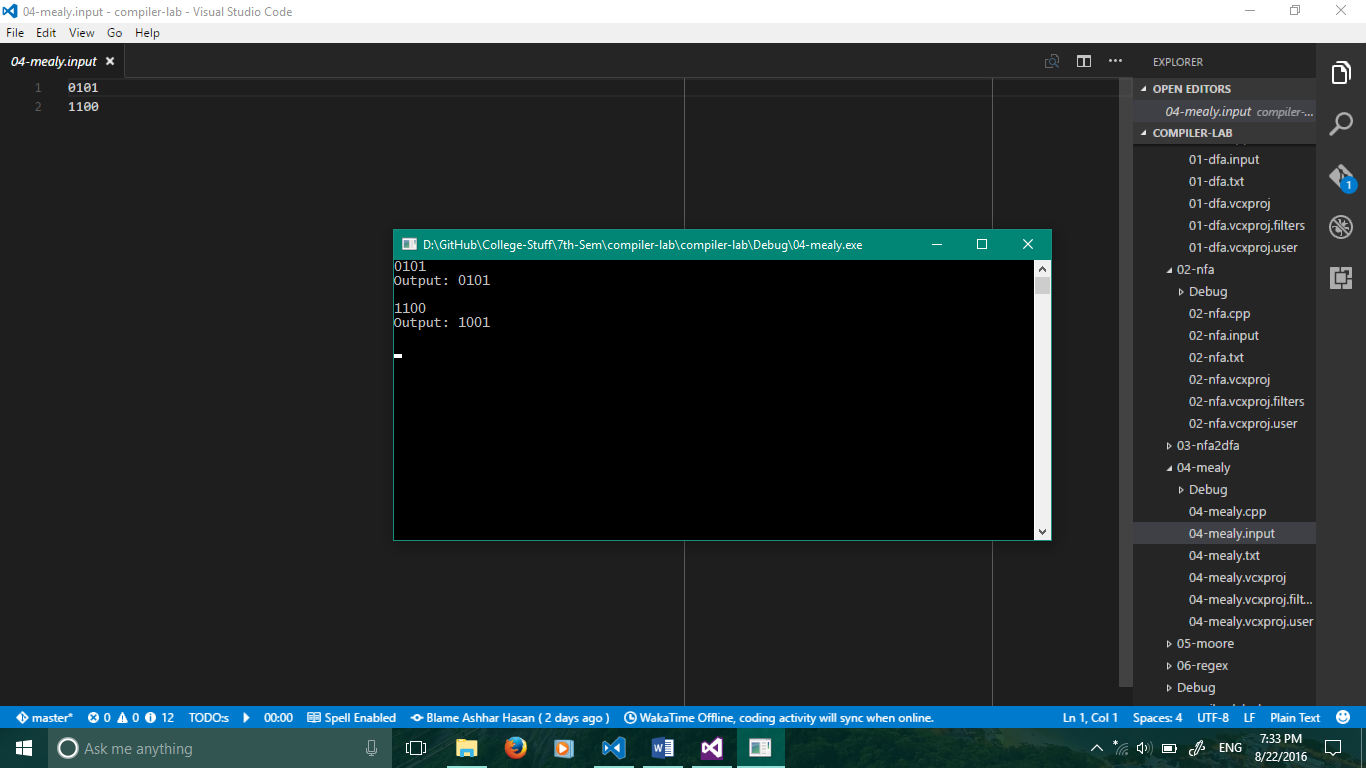
0

3 0 1 1

0 1 3 0

2 1 2 0

1 0 0 1



# Implement a Moore machine:

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

#define MAX\_STATES 100

#define numInputs 2

#define numOutputs 2

#define ZERO\_ASCII 48

#define EOL '\n'

#define OUTPUT\_COL\_INDEX 2

using namespace std;

struct split

{

enum empties\_t

{

empties\_ok,

no\_empties

};

};

template <typename Container>

Container& splitString(Container& result,

const typename Container::value\_type& str,

const typename Container::value\_type& delimiters,

split::empties\_t empties = split::empties\_ok)

{

result.clear();

size\_t current, next = -1;

do

{

if (empties == split::no\_empties)

{

next = str.find\_first\_not\_of(delimiters, next + 1);

if (next == Container::value\_type::npos)

break;

next -= 1;

}

current = next + 1;

next = str.find\_first\_of(delimiters, current);

result.push\_back(str.substr(current, next - current));

} while (next != Container::value\_type::npos);

return result;

}

class Moore

{

public:

Moore() : initState{0},

curState{0},

numStates{0}

{

}

bool readInput(string automataFile = "05-moore.txt",

string delimiters = " ,");

bool process(string inputFile = "05-moore.input",

bool batchMode = false);

private:

size\_t initState;

int automata[MAX\_STATES][MAX\_STATES];

int curState;

size\_t numStates;

string runMoore(string inputs);

};

bool Moore::readInput(string automataFile, string delimiters)

{

// Open the file for reading.

fstream fp(automataFile, ios::in);

string line;

// Read the initial state.

getline(fp, line, EOL);

initState = stoi(line);

// Read the automata array.

vector<string> fields;

size\_t i = 0, j;

while (!fp.eof())

{

getline(fp, line, EOL);

splitString(fields, line, delimiters, split::no\_empties);

j = 0;

for (auto field : fields)

{

automata[i][j] = stoi(field);

j++;

}

i++;

}

numStates = i - 1;

return true;

}

bool Moore::process(string inputFile, bool batchMode)

{

if (batchMode)

{

fstream fp(inputFile, ios::in);

string line;

while (!fp.eof())

{

// Read a line from input file and run it against the automata.

getline(fp, line, EOL);

// If we read an empty line, we skip it.

if (line.empty())

continue;

// Process the automata.

cout << "Output: " << runMoore(line) << endl << endl;

}

}

else

{

string line;

getline(cin, line);

// If we read an empty line, we skip it.

if (line.empty())

return false;

// Process the automata.

cout << "Output: " << runMoore(line) << endl << endl;

}

return true;

}

// Take a string of integers as input and output corresponding outputs as string of integers.

string Moore::runMoore(string inputs)

{

curState = initState;

string output = "";

for (auto input\_char : inputs)

{

auto input = input\_char - ZERO\_ASCII;

output.append(to\_string(automata[curState][OUTPUT\_COL\_INDEX]));

curState = automata[curState][input];

if (curState == -1)

{

cout << "Invalid Input! No transitions found." << endl;

return output;

}

}

return output;

}

int main(int argc, char\*\* argv)

{

Moore moore;

switch (argc)

{

case 1:

// No arguments means the automata will be read from file and input from stdin.

moore.readInput();

while (1)

moore.process();

case 2:

// One argument means the automata will be read from file and input from the file specified.

moore.readInput();

moore.process(argv[1], true);

break;

case 3:

// Two arguments means the automata and input will be read from the arguments.

moore.readInput(argv[2]);

moore.process(argv[1], true);

break;

}

}

/////////////////////////////////////////////////////////////////////////////// 05-moore.txt //

/////////////////////////////////////////////////////////////////////////////

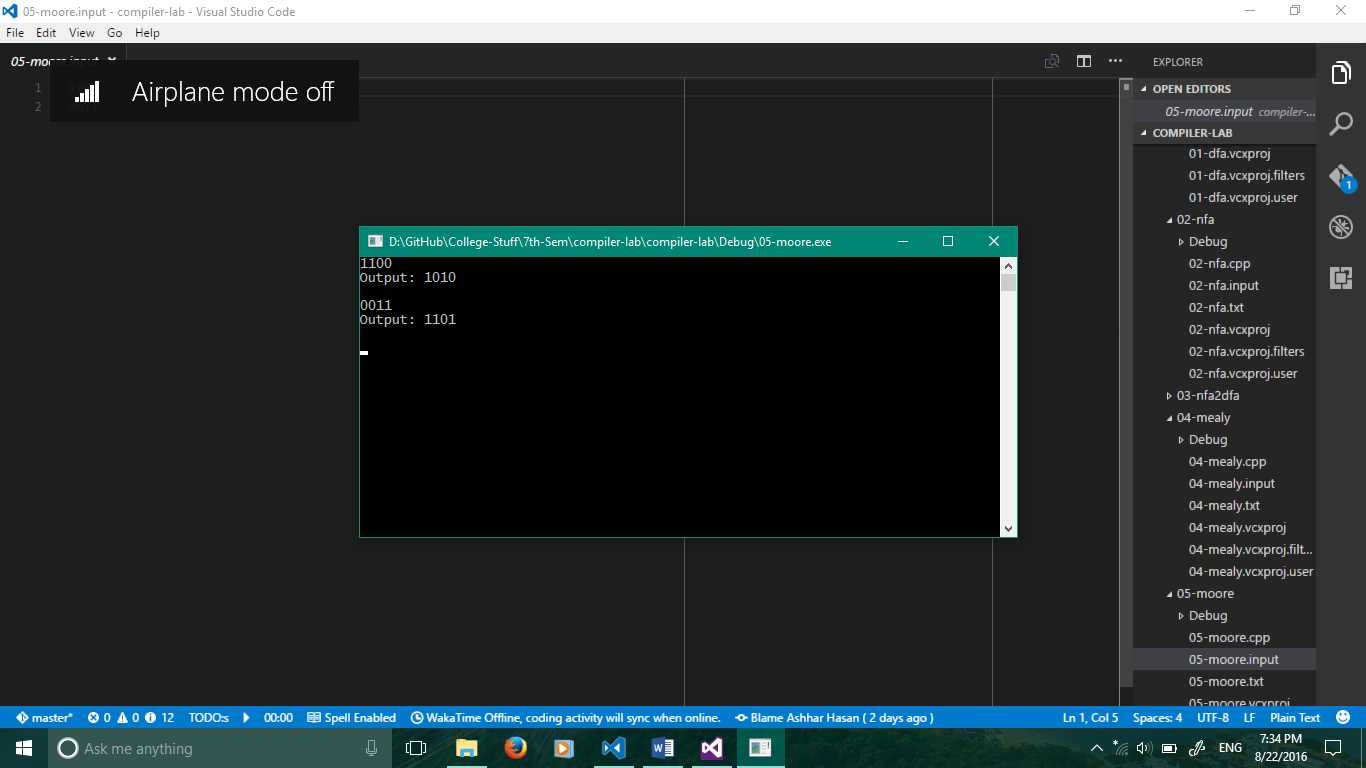
0

3 1 1

0 3 0

2 2 0

1 0 1



# Implement a regular expression:

#include <iostream>

#include <fstream>

#include <string>

#include <sstream>

#include <set>

using namespace std;

set<int> expression;

set<int>::iterator it;

int main()

{

ifstream file("06-regex.txt");

string line;

// Separating the units combined with a dot

getline(file, line);

istringstream line\_stream(line);

string dot\_sep;

while (getline(line\_stream, dot\_sep, '.'))

expression.insert(atoi(dot\_sep.c\_str()));

it = expression.begin();

while (it != expression.end())

{

cout << \*it << endl;

++it;

}

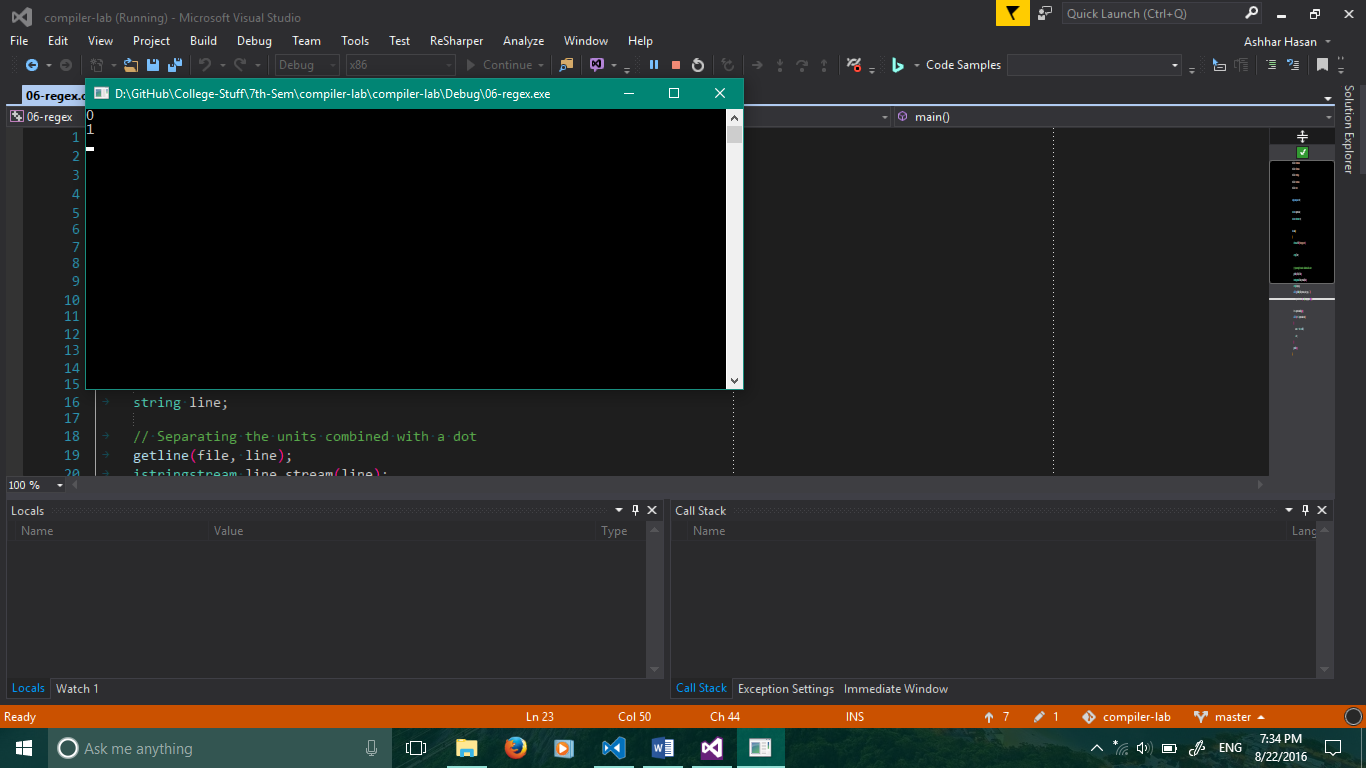
getchar();

}

/////////////////////////////////////////////////////////////////////////////// 06-regex.txt //

/////////////////////////////////////////////////////////////////////////////

1.0\*.(1+1)



# Find leaders and basic blocks from a given Three Address Code:

#include <fstream>

#include <vector>

#include <string>

#include <set>

#include <sstream>

#include <iostream>

using namespace std;

int main()

{

ifstream tacFile;

tacFile.open("tac.txt", ios::in);

// Stores line numbers to mark leaders.

set<int> leaders;

// First line is always a leader.

leaders.insert(1);

auto line\_number = 1;

cout << "The input TAC is: \n";

string line;

// Stores the entire tac. This is bad but can be done for small programs.

vector<string> tac;

while (getline(tacFile, line))

{

tac.push\_back(line);

// Print the program itself.

cout << line\_number << ": " << line << "\n";

// Tokenise and find goto's.

line\_number++;

istringstream line\_stream(line);

do

{

string token;

line\_stream >> token;

if (token == "goto")

{

line\_stream >> token;

// Target of goto label is a leader.

leaders.insert(atoi(token.c\_str()));

// Line after a jump is also a leader.

leaders.insert(line\_number);

}

} while (line\_stream);

}

cout << endl;

tacFile.close();

// Print out leaders.

cout << "Leaders are: \n";

for (auto leader : leaders)

{

cout << leader << ": " << tac.at(leader) << "\n";

}

cout << endl;

// Print out basic blocks.

cout << "Basic blocks are: \n";

auto block\_number = 1;

for (auto i = 0; i < tac.size(); i++)

{

auto found = leaders.find(i + 1);

if (found != leaders.end())

{

cout << block\_number << ":\n";

cout << "\t" << i + 1 << ": " << tac.at(i) << "\n";

block\_number++;

}

else

{

cout << "\t" << i + 1 << ": " << tac.at(i) << "\n";

}

}

cout << endl;

getchar();

return 0;

}

/////////////////////////////////////////////////////////////////////////////// tac.txt //

/////////////////////////////////////////////////////////////////////////////

i := m-1

j := n

t1 := 4\*n

v := a[t1]

i := i+1

t2 := 4\*i

t3 := a[t2]

if t3 < v goto 5

j := j-1

t4 := 4\*j

t5 := a[t4]

if t5 > v goto 9

if i >= j goto 23

t6 := 4\*i

x := a[t6]

t7 := 4\*i

t8 := 4\*j

t9 := a[t8]

a[t7] := t9

t10 := 4\*j

a[t10] := x

goto 5

t11 := 4\*i

x := a[t11]

t12 := 4\*i

t13 := 4\*n

t14 := a[t13]

a[t12] := t14

t15 := 4\*n

a[t15] := x

