Turing Machine Implementation

CPTS322 Software Engineering I

Christian Penick

April 29, 2024

// Module: Turing\_Machine

// File: Turing\_Machine.h

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Turing\_Machine.h provides the private members and public declarations

// for the class Turing\_Machine

#ifndef TURING\_MACHINE\_H

#define TURING\_MACHINE\_H

#include <string>

#include <vector>

#include "Tape.h"

#include "Input\_Alphabet.h"

#include "Tape\_Alphabet.h"

#include "Transition\_Function.h"

#include "States.h"

#include "Final\_States.h"

using std::string;

using std::vector;

class Turing\_Machine

{

private:

vector<string> description;

string initial\_state;

string current\_state;

string original\_input\_string;

int number\_of\_transitions = 0;

bool valid;

bool used;

bool operating;

bool accepted;

bool rejected;

Tape tape;

Input\_Alphabet input\_alphabet;

Tape\_Alphabet tape\_alphabet;

Transition\_Function transition\_function;

States states;

Final\_States final\_states;

ifstream definition;

public:

explicit Turing\_Machine(const string& definition\_file\_name);

void view\_definition();

void view\_instantaneous\_descriptions(int maximum\_number\_of\_cells);

void initialize(const string& input\_string);

void perform\_transitions(int maximum\_number\_of\_transitions);

void terminate\_operation();

string input\_string();

int total\_number\_of\_transitions() const;

bool is\_valid\_definition() const;

bool is\_valid\_input\_string(const string& value) const;

bool is\_used() const;

bool is\_operating() const;

bool is\_accepted\_input\_string() const;

bool is\_rejected\_input\_string() const;

};

#endif

// Module: Turing\_Machine

// File: Turing\_Machine.cpp

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Turine\_Machine.cpp provides the implementation of methods for the class Turing\_Machine.

#include "Turing\_Machine.h"

#include "TM\_Utils.h"

// Name: Turing\_Machine

// Purpose: this constructor reads in the definition file

// and validates the definition.

Turing\_Machine::Turing\_Machine(const string& definition\_file\_name)

{

valid = true;

used = false;

operating = false;

accepted = false;

rejected = false;

definition.open(definition\_file\_name);

if(!definition.is\_open())

throw Crash("Could not open: " + definition\_file\_name);

string value;

string first\_keyword("STATES:");

bool found\_keyword = false;

string accumulator\_str;

string prev\_str;

char cur\_char;

while(definition.get(cur\_char))

{

if(isspace(cur\_char))

{

if(uppercase(accumulator\_str) == first\_keyword)

{

found\_keyword = true;

if(!prev\_str.empty() && prev\_str.back() != '\n')

description.emplace\_back("\n");

break;

}

accumulator\_str.append(1, cur\_char);

prev\_str = accumulator\_str;

description.push\_back(accumulator\_str);

accumulator\_str.clear();

}

else

accumulator\_str.append(1, cur\_char);

}

if(!found\_keyword)

{

cout << red\_start << "Did not find keyword \"" << first\_keyword << "\" following description.\nExiting.\n" << color\_reset;

exit(0);

}

states.load(definition, valid);

input\_alphabet.load(definition, valid);

tape\_alphabet.load(definition, valid);

transition\_function.load(definition, valid);

// read in initial state

string second\_keyword("BLANK\_CHARACTER:");

vector<string> tmp\_initial\_state\_vec;

bool found\_initial\_state\_keyword = false;

while(definition >> value)

{

if(uppercase(value) == second\_keyword)

{

found\_initial\_state\_keyword = true;

break;

}

tmp\_initial\_state\_vec.push\_back(value);

}

if(!found\_initial\_state\_keyword)

cout << red\_start << "Did not find keyword \"" << second\_keyword << "\".\nExiting.\n" << color\_reset;

if(tmp\_initial\_state\_vec.size() != 1)

{

cout << "Initial states: must have one initial state\n";

valid = false;

}

if(tmp\_initial\_state\_vec.size() == 1)

initial\_state = tmp\_initial\_state\_vec.at(0);

current\_state = initial\_state;

tape.load(definition, valid);

final\_states.load(definition, valid);

// validate sequence

input\_alphabet.validate(tape\_alphabet, valid);

final\_states.validate(states, valid);

tape.validate(input\_alphabet, tape\_alphabet, valid);

transition\_function.validate(tape\_alphabet, states, final\_states, valid);

// validate initial state in set of states

if(!states.is\_element(initial\_state))

{

valid = false;

cout << yellow\_start << "initial state: \"" << initial\_state << "\" not in set of states.\n" << color\_reset;

}

if(!valid)

exit(0);

}

// Name: view\_definition

// Purpose: displays white-space preserved description of definition,

// followed by a formatted, formal definition of TM

void Turing\_Machine::view\_definition()

{

// view TM description

for(const auto &i : description) // print description from 'this' first

{

cout << i;

}

cout << endl;

states.view();

input\_alphabet.view();

tape\_alphabet.view();

transition\_function.view();

cout << "Q\_0: " << this->initial\_state << endl; // displaying initial state

tape.view();

final\_states.view();

}

// Name: view\_instantaneous\_description

// Purpose: view maximum\_number\_of\_cells to the left and right of the tape head

void Turing\_Machine::view\_instantaneous\_descriptions(int maximum\_number\_of\_cells)

{

cout << total\_number\_of\_transitions() << " ";

cout << tape.left(maximum\_number\_of\_cells);

cout << "[" << current\_state << "]";

cout << tape.right(maximum\_number\_of\_cells);

cout << endl;

}

// Name: intialize

// Purpose: set beginning of tape to input string, intialize values

void Turing\_Machine::initialize(const string& input\_string)

{

number\_of\_transitions = 0;

tape.initialize(input\_string);

original\_input\_string = input\_string;

}

// Name: perform\_transitions

// Purpose: perform up to maximum\_number\_of\_transitions

void Turing\_Machine::perform\_transitions(int maximum\_number\_of\_transitions)

{

if(!operating)

{

accepted = false;

rejected = false;

operating = true;

current\_state = initial\_state;

}

used = true;

char read\_char = tape.current\_character();

string destination\_state;

char write\_char;

Direction move\_direction;

bool found;

if(final\_states.is\_element(current\_state)) // check if in final state w/o any transitions

{

accepted = true;

return;

}

for(int i = 0; i < maximum\_number\_of\_transitions; i++)

{

transition\_function.find\_transition(current\_state, read\_char, destination\_state, write\_char, move\_direction, found);

if(found)

{

tape.update(write\_char, move\_direction);

current\_state = destination\_state;

read\_char = tape.current\_character();

++number\_of\_transitions;

if(final\_states.is\_element(current\_state))

{

accepted = true;

operating = false;

return;

}

}

else // crash

{

rejected = true;

operating = false;

throw(Crash("Could not find transition.\n"));

}

}

}

// Name: terminate\_operation

// Purpose: set all possible statuses of TM to false

void Turing\_Machine::terminate\_operation()

{

operating = false;

accepted = false;

rejected = false;

}

// Name: input\_string

// Purpose: returns class member variable: original\_input\_string

string Turing\_Machine::input\_string()

{

return original\_input\_string;

}

// Name: total\_number\_of\_transitions

// Purpose: returns class member variable: number\_of\_transitions

int Turing\_Machine::total\_number\_of\_transitions() const

{

return number\_of\_transitions;

}

// Name: is\_valid\_definition

// Purpose: returns class member variable: valid

bool Turing\_Machine::is\_valid\_definition() const

{

return valid;

}

// Name: is\_valid\_input\_string

// Purpos: check if string value is solely composed

// of characters from input alphabet

bool Turing\_Machine::is\_valid\_input\_string(const string& value) const

{

bool str\_valid = true;

for(const auto& i : value)

{

if(!input\_alphabet.is\_element(i)) // char in string not in input alphabet

{

cout << yellow\_start << "input string: char \"" << i << "\" not in input alphabet.\n" << color\_reset;

str\_valid = false;

}

}

return str\_valid;

}

// Name: is\_used

// Purpose: returns class member variable: is\_used

bool Turing\_Machine::is\_used() const {return used;}

// Name: is\_operating

// Purpose: returns class member variable: operating

bool Turing\_Machine::is\_operating() const {return operating;}

// Name: is\_accepted\_input\_string

// Purpose: returns class member variable: accepted

bool Turing\_Machine::is\_accepted\_input\_string() const {return accepted;}

// Name: is\_rejected\_input\_string

// Purpose: returns class member variable: rejected

bool Turing\_Machine::is\_rejected\_input\_string() const {return rejected;}

// Module: Transition

// File: Transition.h

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Transition.h provides the private members and public declarations

// for the class Transition

#ifndef TRANSITION\_H

#define TRANSITION\_H

#include <string>

#include "TM\_Utils.h"

using std::string;

class Transition

{

private:

const string source;

const char read;

const string destination;

const char write;

const Direction move;

public:

Transition(string source\_state, char read\_character,

string destination\_state, char write\_character,

Direction direction);

string source\_state() const;

char read\_character() const;

string destination\_state() const;

char write\_character() const;

Direction move\_direction() const;

};

#endif

// Module: Transition

// File: Transition.cpp

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Transition.cpp implements the methods of the class Transition

#include "Transition.h"

// Name: Transition

// Purpose: This constructor intializes all the member

// variables of the class Transition to create an

// object of said class.

Transition::Transition(string source\_state, char read\_character,

string destination\_state, char write\_character,

Direction direction) : source(source\_state), read(read\_character), destination(destination\_state),

write(write\_character), move(direction){}

// Name: source\_state

// Purpose: returns member variable: source

string Transition::source\_state() const {return source;}

// Name: read\_character

// Purpose: returns member variable: read\_character

char Transition::read\_character() const {return read;}

// Name: destination\_state

// Purpose: returns member variable: destination

string Transition::destination\_state() const {return destination;}

// Name: write\_character

// Purpose: returns member variable: write

char Transition::write\_character() const {return write;}

// Name: move\_direction

// Purpose: returns member variable: move

Direction Transition::move\_direction() const {return move;}

// Module: Transition\_Function

// File: Transition\_Function.h

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Transition\_Function.h provides the private members and public declarations

// for the class Transition\_Function.

#ifndef TRANSITION\_FUNCTION\_H

#define TRANSITION\_FUNCTION\_H

#include <fstream>

#include <iostream>

#include <vector>

#include <string>

#include "Tape\_Alphabet.h"

#include "States.h"

#include "Final\_States.h"

#include "TM\_Utils.h"

#include "Transition.h"

using std::ifstream;

using std::cout;

using std::endl;

using std::string;

using std::vector;

class Transition\_Function

{

private:

vector<Transition> transitions = {};

public:

void load(ifstream& definition, bool& valid);

void validate(const Tape\_Alphabet& tape\_alphabet, const States& states,

const Final\_States& final\_states, bool& valid) const;

void view() const;

void find\_transition(const string& source\_state, char read\_character,

string& destination\_state, char& write\_character,

Direction& move\_direction, bool& found) const;

};

#endif

// Module: Transition\_Function

// File: Transition\_Function.cpp

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Transition\_Function.cpp implements methods of the class Transition\_Function

#include "Transition\_Function.h"

#include "Tape\_Alphabet.h"

#include "States.h"

#include "Final\_States.h"

#include "TM\_Utils.h"

#include <string>

#include <vector>

#include <iostream>

using std::string;

using std::vector;

using std::cout;

// Name: load

// Purpose: reads in whitespace delimited tokens,

// attempting to create syntactically valid transitions;

// this process occurs until the next keyword is reached or EOF;

// also ensures that the set of transition functions only

// contains unique transitions.

void Transition\_Function::load(ifstream& definition, bool& valid)

{

string current\_component("transition function");

string next\_keyword("INITIAL\_STATE:");

bool found\_keyword = false;

string source\_state;

char read\_char;

string dest\_state;

char write\_char;

Direction move\_direction;

enum read\_state

{

reading\_from\_state,

reading\_read\_char,

reading\_dest\_state,

reading\_write\_char,

reading\_move\_dir

};

read\_state rs = reading\_from\_state;

string value;

while(definition >> value)

{

if(uppercase(value) == next\_keyword)

{

found\_keyword = true;

if (rs != reading\_from\_state)

cout << red\_start <<"Incomplete transition function\n" << color\_reset;

break;

}

switch(rs)

{

case reading\_from\_state:

source\_state = value;

rs = reading\_read\_char;

value.clear();

break;

case reading\_read\_char:

if(value.length() != 1)

cout << yellow\_start << "transition function: read char must be single char \"" << value << "\".\n" << color\_reset;

read\_char = value[0];

rs = reading\_dest\_state;

value.clear();

break;

case reading\_dest\_state:

dest\_state = value;

rs = reading\_write\_char;

value.clear();

break;

case reading\_write\_char:

if(value.length() != 1)

{

cout << yellow\_start << "transition function: write char must be single char \"";

cout << value << "\".\n" << color\_reset;

}

write\_char = value[0];

rs = reading\_move\_dir;

value.clear();

break;

case reading\_move\_dir:

if(value.length() != 1)

{

cout << yellow\_start << "transition function: move char must be single char \"";

cout << value << "\".\n" << color\_reset;

}

if(toupper(value[0]) != 'L' && toupper(value[0]) != 'R')

{

valid = false;

cout << yellow\_start << "transition function: Invalid move direction \"";

cout << value[0] << "\".\n" << color\_reset;

}

move\_direction = (char)toupper(value[0]);

// push back fully formed transition

transitions.emplace\_back(source\_state, read\_char, dest\_state, write\_char, move\_direction);

rs = reading\_from\_state;

value.clear();

break;

}

}

if(!found\_keyword)

{

cout << red\_start << "Did not find keyword \"" << next\_keyword;

cout << "\" following " << current\_component << ".\nExiting.\n" << color\_reset;

exit(0);

}

// check for duplicates

int outer\_index = 0;

for(const auto& o : transitions)

{

int inner\_index = 0;

for(const auto& i : transitions)

{

if((o.source\_state() == i.source\_state()) &&

(o.read\_character() == i.read\_character()) &&

(o.write\_character() == i.write\_character()) &&

(o.destination\_state() == i.destination\_state()) &&

(o.move\_direction() == i.move\_direction()) &&

(outer\_index != inner\_index))

{

cout << yellow\_start << "transition function: duplicate found.\n" << color\_reset;

valid = false;

}

++inner\_index;

}

++outer\_index;

}

}

// Name: validate

// Purpose: for every transition, validate checks to make sure

// the source state is not in the set of final states,

// the source state is in the set of states,

// the destination state is in the set of states,

// the read character is in the tape alphabet,

// the write character is in the tape alphabet

void Transition\_Function::validate(const Tape\_Alphabet& tape\_alphabet, const States& states, const Final\_States& final\_states, bool& valid) const

{

for(const auto& i : transitions)

{

if(final\_states.is\_element(i.source\_state())) // if source state in set of final states (not ok)

{

cout << yellow\_start << "tf: source state " << i.source\_state() << " is in final states.\n" << color\_reset;

valid = false;

}

if(!states.is\_element(i.source\_state())) // if source state not in set of states (not ok)

{

cout << yellow\_start << "tf: source state " << i.source\_state() << " not in set of states.\n" << color\_reset;

valid = false;

}

if(!states.is\_element(i.destination\_state())) // if dest. state not in set of states (not ok)

{

cout << yellow\_start << "tf: desination state " << i.destination\_state() << " not in set of states.\n" << color\_reset;

valid = false;

}

if(!tape\_alphabet.is\_element(i.read\_character())) // if read char not in tape alphabet (not ok)

{

cout << yellow\_start << "tf: read character " << i.read\_character() << " not in tape alphabet.\n" << color\_reset;

valid = false;

}

if(!tape\_alphabet.is\_element(i.write\_character())) // if write char not in tape alphabet (not ok)

{

cout << yellow\_start << "tf: write character " << i.write\_character() << "not in tape alphabet.\n" << color\_reset;

valid = false;

}

}

}

// Name: view

// Purpose: displays the transitions functions

// in a formatted manner to the console

void Transition\_Function::view() const

{

for(const auto& i : transitions)

{

cout << delta << "(" << i.source\_state() << ", " << i.read\_character() << ") = ";

cout << "(" << i.destination\_state() << ", " << i.write\_character() << ", ";

cout << i.move\_direction() << ")\n";

}

}

// Name: find\_transition

// Purpose: given a source state and read charcter,

// find\_transition attempts to find the corresponding transition function,

// writing to the appropriate inout parameters and setting found to true,

// otherwise setting found to false

void Transition\_Function::find\_transition(const string& source\_state, char read\_character,

string& destination\_state, char& write\_character,

Direction& move\_direction, bool& found)const

{

for(const auto& i : transitions)

{

if((i.source\_state() == source\_state) && (i.read\_character() == read\_character))

{

destination\_state = i.destination\_state();

write\_character = i.write\_character();

move\_direction = i.move\_direction();

found = true;

return;

}

}

found = false;

}

// Module: TM\_utils

// File: TM\_utils.h

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// TM\_utils.h defines type Direction, and defines various output modifiers (Greek characters, colors);

// implements helper functions to check uniqueness of elements

// and convert string to uppercase

#ifndef UNIQUE\_SET\_H

#define UNIQUE\_SET\_H

#include <iostream>

#include <algorithm>

#include <vector>

#include <string>

#include "TM\_Utils.h"

using std::cout;

using std::count;

using std::vector;

using std::string;

// define type Direction (char)

typedef char Direction;

// define Greek character codes

#define sigma "\u03A3"

#define gamma "\u0393"

#define delta "\u03B4"

// define command line color codes

#define red\_start "\033[1;31m"

#define yellow\_start "\033[1;33m"

#define color\_reset "\033[1;0m"

// check if a vector of elements has any duplicates and set valid to false if so

template<class T>

void check\_unique\_set(vector<T>& vec, const string& current\_component, bool&valid)

{

int index = 0;

for(const auto& i : vec)

{

if(std::count(vec.begin(), vec.end(), i) > 1)

{

cout << yellow\_start << current\_component << ": duplicate found \"" << i << "\".\n" << color\_reset;

valid = false;

vec.erase(vec.begin()+index); // remove element so duplicate doesn't get picked up twice

}

index++;

}

}

// convert string to upper case

string uppercase(const string& value);

// checks to make sure the string passed in only contains integers

int parse\_string\_to\_int(const string& to\_parse);

#endif

#include "TM\_Utils.h"

// Name: uppercase

// Purpose: convert string to upper case

string uppercase(const string& value)

{

string to\_return;

for(const auto& i : value)

{

to\_return.push\_back((char)toupper(i));

}

return to\_return;

}

// Name: parse\_string\_to\_int

// Purpose: checks to make sure the string

// passed in only contains positive integers;

// returns 0 if invalid string,

// else returns integer from string

int parse\_string\_to\_int(const string& to\_parse)

{

bool only\_digits = true;

if(to\_parse.empty())

return 0;

for(int i = 0; i < to\_parse.length(); i++)

{

if(isdigit(to\_parse[i]))

{

continue;

}

else

{

if((i == 0) && (to\_parse[i] == 45)) // if the first char is dash (negative), that's ok

{continue;}

only\_digits = false;

break;

}

}

if(!only\_digits)

{

cout << "only accept integers\n";

return 0;

}

int actual\_num = stoi(to\_parse);

if(actual\_num < 1)

{

cout << "must be >= 1\n";

return 0;

}

else

return actual\_num;

}

// Module: Tape

// File: Tape.h

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Tape.h provides the private members and public declarations

// for the class Tape

#ifndef TAPE\_H

#define TAPE\_H

#include <string>

#include <iostream>

#include <fstream>

#include "Tape\_Alphabet.h"

#include "Input\_Alphabet.h"

#include "TM\_Utils.h"

#include "Crash.h"

using std::string;

using std::ifstream;

using std::cout;

class Tape

{

private:

string cells;

int current\_cell;

char blank\_character;

public:

Tape();

void load(ifstream& definition, bool& valid);

void validate(const Input\_Alphabet& input\_alphabet, const Tape\_Alphabet& tape\_alphabet, bool& valid) const;

void view()const;

void initialize(const string& input\_string);

void update(char write\_character, Direction move\_direction);

string left(int maximum\_number\_of\_cells) const;

string right(int maximum\_number\_of\_cells) const;

char current\_character() const;

};

#endif

// Module: Tape

// File: Tape.cpp

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Tape.cpp implements the methods of the class Tape

#include "Tape.h"

#include "TM\_Utils.h"

// Name: Tape

// Purpose: This constructor intializes tape, current cell, and blank character

Tape::Tape(): cells(" "), current\_cell(0), blank\_character(' '){}

// Name: load

// Purpose: loads blank TM definition file and ensures that next string read in is next keyword

void Tape::load(ifstream& definition, bool& valid)

{

string value;

string current\_component("BLANK\_CHARACTER:");

string next\_keyword("FINAL\_STATES:");

if((definition >> value) && (value.length() == 1)

&& value[0] != '\\' && value[0] != '[' && value[0] != ']'

&& value[0] != '>' && value [0] != '<' && value[0] >= '!' && value[0] <= '~')

blank\_character = value[0];

else

{

cout << "Illegal blank\_character.\n";

valid = false;

}

if((!(definition >> value)) || (uppercase(value) != next\_keyword))

{

cout << red\_start << "Did not find keyword \"" << next\_keyword << "\" following " << current\_component << ".\nExiting.\n" << color\_reset;

valid = false;

exit(0);

}

}

// Name: validate

// Purpose: checks to make sure blank character

// is not in input alphabet and is in tape alphabet

void Tape::validate(const Input\_Alphabet& input\_alphabet, const Tape\_Alphabet& tape\_alphabet, bool& valid) const

{

if(input\_alphabet.is\_element(blank\_character)) // if blank char in input alphabet (not ok)

{

cout << yellow\_start << "Blank character " << blank\_character << " is element of input alphabet\n" << color\_reset;

valid = false;

}

if(!tape\_alphabet.is\_element(blank\_character)) // if blank char not in tape alphabet (not ok)

{

cout << yellow\_start << "Blank character " << blank\_character << " not in tape alphabet\n" << color\_reset;

valid = false;

}

}

// Name: view

// Purpose: displays blank character to console

void Tape::view()const

{

cout << "B: " << blank\_character << "\n";

}

// Name: initialize

// Purpose: initializes tape to the input string that is passed in

void Tape::initialize(const string& input\_string)

{

cells = input\_string + blank\_character;

current\_cell = 0;

}

// Name: update

// Purpose: update writes the passed in character to

// the current position of the tape head and

// attempts to move the tape head left or right,

// throwing an exception if the tape head attempts

// to move to the left of the starting cell

void Tape::update(char write\_character, Direction move\_direction)

{

move\_direction = (char)toupper(move\_direction);

if((move\_direction == 'L') && (current\_cell == 0)) // cannot move left of start cell (will crash)

{

throw Crash("Left move from first cell");

}

if(move\_direction == 'R' && (current\_cell == cells.length() - 1)) // will need to extend TM tape

{

cells += blank\_character;

}

cells[current\_cell] = write\_character;

if(move\_direction == 'L')

--current\_cell;

else

++current\_cell;

}

// Name: left

// Purpose: displays up to maximum\_number\_of\_cells

// to the left of the tape head, adding

// truncation symbol '<' if truncation occured

string Tape::left(int maximum\_number\_of\_cells) const

{

int first\_cell = std::max(0, current\_cell-maximum\_number\_of\_cells);

string value = cells.substr(first\_cell, current\_cell - first\_cell);

if(value.length() < maximum\_number\_of\_cells)

{

value.insert(0, "<");

}

return value;

}

// Name: right

// Purpose: displays up to maximum\_number\_of\_cells

// to the right of the tape head, including the tape head,

// adding truncation symbol '>' if truncation occured

string Tape::right(int maximum\_number\_of\_cells) const

{

int end\_cell = (int)cells.length() - 1;

while((end\_cell >= current\_cell) && (cells[end\_cell] == blank\_character)) // get to non-blank character

--end\_cell;

int last\_cell = std::min(end\_cell, current\_cell+maximum\_number\_of\_cells-1);

string value = cells.substr(current\_cell, last\_cell-current\_cell+1);

if(value.length() < end\_cell-current\_cell+1)

{

value.append(">");

}

return value;

}

// Name: current\_character

// Purpose: return the character at position of tape head

char Tape::current\_character() const

{

return cells[current\_cell];

}

// Module: Tape\_Alphabet

// File: Tape\_Alphabet.h

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Tape\_Alphabet.h provides the private members and public declarations

// for the class Tape\_Alphabet

#ifndef TAPE\_ALPHABET\_H

#define TAPE\_ALPHABET\_H

#include <vector>

#include <fstream>

#include <iostream>

using std::vector;

using std::ifstream;

using std::cout;

class Tape\_Alphabet

{

private:

vector<char> alphabet = {};

public:

void load(ifstream& definition, bool& valid);

void view();

bool is\_element(char value) const;

};

#endif

// Module: Tape\_Alphabet

// File: Tape\_Alphabet.cpp

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Tape\_Alphabet.cpp implements the methods of the class Tape\_Alphabet

#include "Tape\_Alphabet.h"

#include "TM\_Utils.h"

#include <string>

using std::string;

using std::endl;

// Name: load

// Purpose: reads in whitespace delimited characters

// from the TM definition file until the next

// keyword is read or EOF. It also ensures that

// every element in the tape alphabet is unique.

void Tape\_Alphabet::load(ifstream& definition, bool& valid)

{

string value;

string next\_keyword("TRANSITION\_FUNCTION:");

string current\_component("tape alphabet");

bool found\_keyword = false;

vector<string> tmp\_tape\_alphabet;

while (definition >> value)

{

if (uppercase(value) == next\_keyword)

{

found\_keyword = true;

break;

}

tmp\_tape\_alphabet.push\_back(value);

}

if (!found\_keyword)

{

cout << red\_start << "Did not find keyword \"" << next\_keyword << "\" following " << current\_component << ".\nExiting.\n" << color\_reset;

exit(0);

}

for (const auto &letter: tmp\_tape\_alphabet)

{

if (letter.length() != 1)

{

valid = false;

cout << yellow\_start << "tape alphabet: must be single char \"" << letter << "\".\n" << color\_reset;

}

else

{

if(value[0] != '\\' && value[0] != '[' && value[0] != ']'

&& value[0] != '>' && value [0] != '<' && value[0] >= '!' && value[0] <= '~')

alphabet.push\_back(letter[0]);

else

{

cout << yellow\_start << "tape alphabet: invalid char \"" << letter[0] << "\".\n" << color\_reset;

}

}

}

if(alphabet.empty())

{

cout << red\_start << "Tape alphabet is empty.\nExiting.\n" << color\_reset;

exit(0);

}

check\_unique\_set(alphabet, current\_component, valid);

}

// Name: view

// Purpose: displays the elements of the tape alphabet on the console

void Tape\_Alphabet::view()

{

int num\_elements = (int)alphabet.size();

int count = 0;

cout << gamma << ": {";

for(const auto &character : alphabet)

{

cout << character;

if(count < num\_elements-1)

cout << ", ";

count++;

}

cout << "}\n";

}

// Name: is\_element

// Purpose: checks whether the character passed

// in is an element of the tape alphabet

bool Tape\_Alphabet::is\_element(char value) const

{

bool flag = false;

for(const auto& i : alphabet)

{

if(i == value) // if given string is in set of states

{

flag = true;

break;

}

}

return flag;

}

// Module: States

// File: States.h

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// States.h provides the private members and public declarations

// for the class States

#ifndef STATES\_H

#define STATES\_H

#include <string>

#include <vector>

#include <iostream>

#include <fstream>

using std::string;

using std::vector;

using std::cout;

using std::ifstream;

using std::endl;

class States

{

private:

vector<string> names = {};

public:

void load(ifstream& definition, bool& valid);

void view();

bool is\_element(const string& value) const;

};

#endif

// Module: States

// File: States.cpp

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// States.cpp implements the methods of the class States

#include "States.h"

#include "TM\_Utils.h"

#include <algorithm>

// Name: load

// Purpose: reads whitespace delimited strings

// from the TM definition file until

// the next keyword is read in or EOF

// and ensures that every element in

// set of states is unique

void States::load(ifstream& definition, bool& valid)

{

string value;

string current\_component("states");

string next\_keyword("INPUT\_ALPHABET:");

bool found\_keyword = false;

while(definition >> value)

{

if (uppercase(value) == next\_keyword)

{

found\_keyword = true;

break;

}

names.push\_back(value);

}

if(!found\_keyword)

{

cout << red\_start << "Did not find keyword \"" << next\_keyword << "\" following "<< current\_component << ".\nExiting.\n" << color\_reset;

exit(0);

}

else if(names.empty())

{

cout << red\_start << "Set of states is empty.\nExiting.\n" << color\_reset;

exit(0);

}

for(const auto& state : names) // iterate over collected states

{

for(const auto& letter : state)// make sure all characters of state name are ok

{

if((letter>=65 && letter <= 90) || (letter >= 97 && letter <= 122) || (letter >= 48 && letter <= 57) || (letter == 45) || (letter == 95))

continue; // char is valid

else

{

cout << yellow\_start "states: invalid char '" << letter << "' in state \"" << state << "\"\n" << color\_reset;

valid = false;

}

}

}

check\_unique\_set(names, current\_component, valid);

}

// Name: view

// Purpose: displays the states of the TM on the console

void States::view()

{

int num\_states = (int)names.size();

int count = 0;

cout << "Q: {";

for(const auto &state : names)

{

cout << state;

if(count < num\_states - 1)

cout << ", ";

++count;

}

cout << "}\n";

}

// Name: is\_element

// Purpose: checks whether the state passed in is in the TM's set of states

bool States::is\_element(const string& value) const

{

bool flag = false;

for(const auto& i : names)

{

if(i == value) // if given string is in set of states

{

flag = true;

break;

}

}

return flag;

}

// Module: Input\_Alphabet

// File: Input\_Alphabet.h

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Input\_Alphabet.h provides the private members and public declarations

// for the class Input\_Alphabet

#ifndef INPUT\_ALPHABET\_H

#define INPUT\_ALPHABET\_H

#include <vector>

#include <fstream>

#include <iostream>

#include <string>

#include "Tape\_Alphabet.h"

using std::vector;

using std::ifstream;

using std::cout;

using std::string;

using std::endl;

class Input\_Alphabet

{

private:

vector<char> alphabet = {};

public:

void load(ifstream& definition, bool& valid);

void validate(const Tape\_Alphabet& tape\_alphabet, bool& valid);

void view();

bool is\_element(char value) const;

};

#endif

// Module: Input\_Alphabet

// File: Input\_Alphabet.cpp

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Input\_Alphabet.cpp implements the methods of the class Input\_Alphabet

#include "Input\_Alphabet.h"

#include "TM\_Utils.h"

// Name: load

// Purpose: reads whitespaced delimited characters until the next keyword is read or EOF

// and ensures that every element of the input alphabet is unique

void Input\_Alphabet::load(ifstream& definition, bool& valid)

{

string value;

string next\_keyword("TAPE\_ALPHABET:");

string current\_component("input alphabet");

bool found\_keyword = false;

vector<string> tmp\_input\_alphabet;

while (definition >> value)

{

if (uppercase(value) == next\_keyword)

{

found\_keyword = true;

break;

}

tmp\_input\_alphabet.push\_back(value);

}

if (!found\_keyword)

{

cout << red\_start << "Did not find keyword \"" << next\_keyword << "\" following " << current\_component << ".\nExiting.\n" << color\_reset;

exit(0);

}

for (const auto &letter: tmp\_input\_alphabet)

{

if (letter.length() != 1)

{

valid = false;

cout << yellow\_start << "input alphabet: must be single char \"" << letter << "\".\n" << color\_reset;

}

else

{

if(value[0] != '\\' && value[0] != '[' && value[0] != ']'

&& value[0] != '>' && value [0] != '<' && value[0] >= '!' && value[0] <= '~')

alphabet.push\_back(letter[0]);

else

{

cout << yellow\_start << "input alphabet: invalid char \"" << letter[0] << "\".\n" << color\_reset;

}

}

}

check\_unique\_set(alphabet, current\_component, valid);

}

// Name: view

// Purpose: displays the characters of the input alphabet to the console

void Input\_Alphabet::view()

{

int num\_states = (int)alphabet.size();

int count = 0;

cout << sigma << ": {";

for(const auto &character : alphabet)

{

cout << character;

if(count < num\_states - 1)

cout << ", ";

++count;

}

cout << "}\n";

}

// Name: validate

// Purpose: ensures that every element of the input alphabet is a member of the tape alphabet

void Input\_Alphabet::validate(const Tape\_Alphabet& tape\_alphabet, bool& valid)

{

for(const auto& i : alphabet)

{

if(!tape\_alphabet.is\_element(i))

{

valid = false;

cout << yellow\_start << "input alphabet: \"" << i << "\" is not member of tape alphabet.\n" << color\_reset;

}

}

}

// Name: is\_elements

// Purpose: checks if the character passed to it is in the input alphabet

bool Input\_Alphabet::is\_element(char value) const

{

for(const auto& i : alphabet)

{

if(i == value) // if given char is in input alphabet

return true;

}

return false; // char never appeared in input alphabet

}

// Module: Final\_States

// File: Final\_States.h

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Final\_States.h provides the private members and public declarations

// for the class Final\_States

#ifndef FINAL\_STATES\_H

#define FINAL\_STATES\_H

#include <string>

#include <vector>

#include <fstream>

#include <iostream>

#include "States.h"

using std::string;

using std::vector;

using std::ifstream;

using std::cout;

class Final\_States {

private:

vector<string> names = {};

public:

void load(ifstream& definition, bool &valid);

void validate(const States& states, bool &valid);

void view();

bool is\_element(const string &value) const;

};

#endif

// Module: Final\_States

// File: Final\_States.cpp

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Final\_states.cpp implements methods for the class Final\_States

#include "Final\_States.h"

#include "TM\_Utils.h"

using std::endl;

// Name: load

// Purpose: reads in whitespace delimited strings until EOF is reached

// and ensures that every element of final states is unique

void Final\_States::load(ifstream& definition, bool& valid)

{

string value;

string current\_component("FINAL\_STATES");

while(definition >> value)

{

names.push\_back(value);

}

check\_unique\_set(names, current\_component, valid);

}

// Name: validate

// Purpose: ensures that all final states are in the TM's set of states

void Final\_States::validate(const States& states, bool& valid)

{

for(const auto& i : names)

{

if(!states.is\_element(i))

{

valid = false;

cout << yellow\_start << "final states: \"" << i << "\" not in set of states.\n" << color\_reset << endl;

}

}

}

// Name: view

// Purpose: displays the final states to the console

void Final\_States::view()

{

int num\_final\_states = (int)names.size();

int count = 0;

cout << "F: {";

for(const auto &state : names)

{

cout << state;

if(count < num\_final\_states-1)

cout << ", ";

++count;

}

cout << "}\n";

}

// Name: is\_elements

// Purpose: checks if the state passed to it is in the set of states

bool Final\_States::is\_element(const string& value) const

{

bool flag = false;

for(const auto& final\_state : names)

{

if(final\_state == value) // if given string is in set of states

{

flag = true;

break;

}

}

return flag;

}

// Module: Crash

// File: Crash.h

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Crash.h provides the public declarations

// for the class Crash

#ifndef CRASH\_H

#define CRASH\_H

#include <stdexcept>

#include <string>

using std::string;

using std::runtime\_error;

class Crash : public runtime\_error

{

public:

Crash(const string& reason);

};

#endif

// Module: Crash

// File: Crash.cpp

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Crash.cpp implements the constructor for class Crash

#include "Crash.h"

#include <stdexcept>

// constructor takes string and initializes runtime\_error with it

Crash::Crash(const string& reason) : runtime\_error(reason.c\_str()){}

// Module: Main

// File: Main.cpp

// Application: Turing Machine

// Computer: MSI GS65 9SD Stealth

// Operating System: Ubuntu 22.04.4 LTS

// Course: CPT\_S 322 Software Design I

// Author: Christian L. Penick

// Date: 04/24/2024

// Description

// Main.cpp validates command lines arguments, intializes Turing Machine,

// validates user commands, and sends messages to the TM to perform commands

#include <vector>

#include <string>

#include <iostream>

#include <algorithm>

#include "Turing\_Machine.h"

#include "TM\_Utils.h"

using std::string;

using std::ofstream;

using std::ifstream;

using std::cout;

using std::cin;

using std::vector;

using std::endl;

using std::invalid\_argument;

using std::exception;

// Name: handle\_user\_command

// Purpose: takes a user command and if it's a valid command,

// performs the appropriate command, either through

// messages to the Turing Machine object, or using

// other data structures in main.

int handle\_user\_command(char cmd\_char, Turing\_Machine& TM,

vector<string>& input\_strings, bool& str\_list\_modified,

const string& input\_file\_name, int& num\_transition\_per\_run,

int& num\_cells\_in\_id)

{

cmd\_char = (char)toupper(cmd\_char);

string raw\_CL\_input;

if (cmd\_char == 'D') // delete

{

cout << "Enter number of line to delete: ";

getline(cin, raw\_CL\_input);

int user\_num = parse\_string\_to\_int(raw\_CL\_input);

if(user\_num == 0)

return 0;

if(user\_num - 1 < input\_strings.size())

{

str\_list\_modified = true;

input\_strings.erase((input\_strings.begin() + (user\_num-1)));

}

else

cout << "Invalid line to delete.\n";

return 0;

}

else if (cmd\_char == 'X') // exit

{

if(str\_list\_modified) // write to file

{

ofstream writeInputStringFile(input\_file\_name, std::ios::trunc);

if(writeInputStringFile.is\_open())

{

int num\_strings = (int)input\_strings.size();

int count = 0;

for(const auto& i : input\_strings)

{

if(i.empty())

writeInputStringFile << '\\';

else

writeInputStringFile << i;

if(count < num\_strings -1)

writeInputStringFile << endl;

++count;

}

cout << "Input string file written successfully.\n";

}

else

{

cout << "Could not write input string file successfully\n";

}

}

return 1; // return one will signal in main to exit program

}

else if (cmd\_char == 'H') // help

{

cout << "Delete (D): Delete input string from list\n";

cout << "Exit (X): Exit application\n";

cout << "Help (H): Help user\n";

cout << "Insert (I): Insert input strings into list\n";

cout << "List (L): List input strings\n";

cout << "Quit (Q): Quit operation of Turing machine on input string\n";

cout << "Run (R): Run Turing Machine on input string\n";

cout << "Set (E): Set maximum number of transitions to perform\n";

cout << "Show (W): Show status of application\n";

cout << "Truncate (T): Truncate instantaneous descriptions\n";

cout << "View (V): View Turing Machine\n";

return 0;

}

else if(cmd\_char == 'I') // insert

{

cout << "New input string: ";

getline(cin, raw\_CL\_input);

if(!raw\_CL\_input.empty())

{

if(raw\_CL\_input.length() == 1 && raw\_CL\_input[0] == '\\')

raw\_CL\_input = ""; // empty string

if(TM.is\_valid\_input\_string(raw\_CL\_input) && (std::count(input\_strings.begin(), input\_strings.end(), raw\_CL\_input) == 0))

{

str\_list\_modified = true;

input\_strings.push\_back(raw\_CL\_input);

}

}

return 0;

}

else if(cmd\_char == 'L') // list input strings

{

cout << endl;

cout << "Input strings:\n";

if(input\_strings.empty())

{

cout << "Empty.\n";

return 0;

}

int num = 1;

for(const auto& i : input\_strings)

{

cout << num << ") ";

if(i.empty()) // empty string

cout << "\\";

else

cout << i;

cout << endl;

++num;

}

return 0;

}

else if (cmd\_char == 'Q') // quit

{

if(TM.is\_operating())

{

TM.terminate\_operation();

cout << "Input string: " << TM.input\_string() << endl;

cout << "Total number of transitions performed: " << TM.total\_number\_of\_transitions() << endl;

cout << "Input string is neither accepted nor rejected.\n";

}

else

cout << "Turing Machine not running on input string.\n";

return 0;

}

else if (cmd\_char == 'R') // run

{

if(!TM.is\_valid\_definition())

{

cout << "Cannot operate invalid Turing Machine.\n";

return 0;

}

if(!TM.is\_operating()) // if not operating

{

cout << endl;

cout << "Enter line of input string to run: ";

getline(cin, raw\_CL\_input);

if(raw\_CL\_input.empty())

return 0;

int user\_num = parse\_string\_to\_int(raw\_CL\_input);

if(user\_num == 0)

return 0;

if(user\_num - 1 >= input\_strings.size())

{

cout << "Not valid choice from input string list.\n";

return 0;

}

TM.initialize(input\_strings.at(user\_num-1));

TM.view\_instantaneous\_descriptions(num\_cells\_in\_id);

}

try

{

TM.perform\_transitions(num\_transition\_per\_run);

}

catch(Crash& ec)

{

cout << yellow\_start << ec.what() << color\_reset;

}

TM.view\_instantaneous\_descriptions(num\_cells\_in\_id);

if(TM.is\_accepted\_input\_string() || TM.is\_rejected\_input\_string())

{

cout << endl;

cout << "Total number of transitions performed: " << TM.total\_number\_of\_transitions() << endl;

cout << "Original input string: ";

if(TM.input\_string().empty())

cout << "\\" << endl;

else

cout << TM.input\_string() << endl;

}

if(TM.is\_accepted\_input\_string())

{

cout << "Accepted.\n";

}

else if (TM.is\_rejected\_input\_string())

{

cout << "Rejected.\n";

}

return 0;

}

else if (cmd\_char == 'E') // set

{

// set

cout << "Maximum number of transitions[" << num\_transition\_per\_run << "]: ";

getline(cin, raw\_CL\_input);

int user\_num = parse\_string\_to\_int(raw\_CL\_input);

if(user\_num > 0)

{

num\_transition\_per\_run = user\_num;

cout << "Setting changed.\n";

}

return 0;

}

else if(cmd\_char == 'W') // show

{

// accounting info

cout << "CPTS 322\nSpring 2024\nDr. Corrigan\nChristian Penick\nVersion: Final Submission\n";

// default settings

cout << "Max number of transition to perform at a time: " << num\_transition\_per\_run << endl;

cout << "Max number of cells to left and right of tape head to display in ID: " << num\_cells\_in\_id << endl;

// name of TM

cout << "Name of Turing Machine: " << input\_file\_name.substr(0, input\_file\_name.length()-4) << endl;

// status of TM

if(!TM.is\_used())

cout << "Not used.\n";

else if(TM.is\_operating())

cout << "Running.\nInput string: " << TM.input\_string() << endl << "Transitions performed: " << TM.total\_number\_of\_transitions() << endl;

else // TM ran on string

{

cout << "Last input string: " << TM.input\_string() << endl << "Transitions performed: " << TM.total\_number\_of\_transitions() << endl;

if(TM.is\_accepted\_input\_string())

cout << "Accepted.\n";

else if(TM.is\_rejected\_input\_string())

cout << "Rejected.\n";

else

cout << "Quit before determination of acceptance/rejection.\n";

}

if(str\_list\_modified)

cout << "List of input strings will be written to " << input\_file\_name << endl;

return 0;

}

else if(cmd\_char == 'T') // truncate

{

// truncate amount shown on ID

cout << "Number of cells[" << num\_cells\_in\_id << "]: ";

getline(cin, raw\_CL\_input);

int user\_num = parse\_string\_to\_int(raw\_CL\_input);

if(user\_num != 0)

{

num\_cells\_in\_id = user\_num;

cout << "Setting changed.\n";

}

return 0;

}

else if (cmd\_char == 'V') // view

{

TM.view\_definition();

return 0;

}

else

{

cout << "Invalid command.\n";

return 0; // return 0 if invalid cmd given

}

}

// Name: load\_input\_strings

// Purpose: read in newline delimited input strings from the input string file

vector<string> load\_input\_strings(const string& input\_string\_file\_name,

const Turing\_Machine& tm,

bool& str\_list\_modified)

{

vector<string> vector\_of\_input\_strings;

ifstream readFile;

readFile.open(input\_string\_file\_name);

if(!readFile.is\_open())

{

return vector\_of\_input\_strings;

}

string value;

while(getline(readFile, value))

{

if(value.length() == 1 && value[0] == '\\')

value = ""; // empty string

if(value.back() == '\n') // remove newline if exists in string

value.pop\_back();

if(count(vector\_of\_input\_strings.begin(), vector\_of\_input\_strings.end(), value) == 0) // check if dup

{

if(tm.is\_valid\_input\_string(value)) // if input string solely composed of input alphabet

vector\_of\_input\_strings.push\_back(value);

else

str\_list\_modified = true;

}

else

str\_list\_modified = true;

}

readFile.close();

return vector\_of\_input\_strings;

}

// Name: main

// Purpose: validates command lines arguments,

// intializes Turing Machine,

// performs command loop control

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

{

if(argc < 2)

{

cout << "Incorrect program invocation: ./progname TM\_def\_file\n\n";

return 0;

}

// define config settings

int num\_transition\_per\_run = 1;

int num\_cells\_in\_id = 32;

string definition\_file\_name = string(argv[1]) + ".def";

string input\_string\_file\_name = string(argv[1]) + ".str";

Turing\_Machine tm(definition\_file\_name);

bool str\_list\_modified = false;

vector<string> input\_strings = load\_input\_strings(input\_string\_file\_name, tm, str\_list\_modified); // load input strings

string user\_input;

int cmd\_flag = 0;

cout << "Loaded!\n";

while (cmd\_flag != 1)

{

cout << "\nCommand: ";

getline(cin, user\_input);

if(user\_input.empty())

continue;

if(user\_input.length() != 1) // more than char and newline, so skip user input

{

cout << "Invalid input.\n";

continue;

}

cmd\_flag = handle\_user\_command(user\_input[0], tm, input\_strings, str\_list\_modified, input\_string\_file\_name, num\_transition\_per\_run, num\_cells\_in\_id);

}

return 0;

}