CSCE 355, Spring 2019 Programming Assignment Due Wednesday, 24 April 2019

For the programming portion of the course (15% of your grade) you are to write programs that perform two or more of a choice of six tasks involving DFAs. You may choose to implement two or more of the following functions:

Simulating a DFA. After reading the description of a DFA N from a text file, read any number of strings over N's alphabet from another text file, indicating (to standard output) whether N accepts or rejects each of the strings.

Minimizing a DFA. Read the description of a DFA from a text file, and write (to standard output) the description of a minimal equivalent DFA.

Text search. Read a string w from a text file, and write to standard output the description of a DFA that accepts a string x if and only if w is a substring of x.

Closure properties: complement and intersection. Read one or two descriptions of DFAs from text files (depending on the type of transformation asked for), and write to standard output a description of the transformed DFA—either the complementary DFA or the product construction.

Closure property: inverse homomorphic image. Read from a text file the description of a DFA A over some alphabet Γ and (from another text file) the description of a homomorphism $h : \Sigma^* \to \Gamma^*$ (where Σ is some alphabet), and write to standard output the description of a DFA recognizing $h^{-1}(L(A))$.

Determining properties of a DFA. Read from a text file the description of a DFA A and write to standard output (a) whether or not $L(A) = \emptyset$ and (b) whether or not $L(A)$ is finite.

Details

You are only required to implement your choice of *two* of the six alternatives above, but you may implement more than two if you want a modest amount of extra credit (1% of your final grade per additional task correctly implemented, which is 5 points of the 500-point raw point total). As you decide which two, notice that some combinations of tasks can share code. For example, all except the third require reading and internalizing a DFA description; the second, third, fourth, and fifth require writing a DFA description (see the section on DFA format, below).

Therefore we am requiring you to stick to a simple, uniform interface for your program: Your program must be able to be run via a simple command-line invocation on a computer in the departmental Linux lab, and all I/O will be ASCII text. Input is read either from text files (given as command line arguments) or standard input, and output is written to standard output. You may write your program in Java, C or C++, or Python 3.x. (Note: Python 2.7 is not acceptable.) To repeat: your program (after compiling, if necessary) should be a stand-alone executable that can be run directly from the Linux shell, requiring no special user interface to execute (e.g., Eclipse).[1] More about this below.

This assignment is not meant to be overly taxing or time-consuming. Most of the time spent will probably be to get the input read in correctly.

More specific details

Simulating a DFA. First read the description of the DFA. The description is in a file named by the first command line argument (see below for information about the format of this file). Next read a series of zero or more strings from another text file named by the second command line argument. These are inputs to the DFA. Each string will take up an entire line of text, ending with a newline character (which is not included in the string). For each string, write to standard output either "accept" or "reject" according to the behavior of the DFA on the string. End each output word with a newline.

[1]It's OK if we need to precede your main class name with "java" or "python" on the command line, although if that is necessary it would be considerate of you to provide a one-line script that makes that invocation and document the existence of that script in an appropriate README file.

Minimizing a DFA. The input DFA is read from a text file given as the first command line argument, and the output DFA is written to standard output. You should use the table-of-distinguishable-states method described in class to do the minimization. You do not need to output the distinguishability table, but you may optionally do so to standard error if you want. Both the input and output DFA descriptions adhere to the format described below. The ordering of the states of the output DFA is not completely determined, so correct answers may differ up to the ordering of states. That's OK; we'll take that into account using a own script that determines whether two DFAs are the same up to reordering states.

Text search. The name of the file containing the text string is given as the sole command line argument. You should assume that the input string and the output DFA are both over the alphabet consisting of the 26 lower case letters `abcdefghijklmnopqrstuvwxyz` only.

Closure properties: complement and intersection. There will be either one or two command line arguments. If one, then it is the name of a file containing the description of a DFA A, and you are to output the complementary DFA $\neg A$ (see Definition 7.4 of the course notes). If two arguments, then they both are names of text files containing DFAs A and B, and you are to output the product construction $A \wedge B$ (see Definition 7.4 of the course notes). A and B will have the same alphabet. In each case, write to standard output.

Closure property: inverse homomorphic image. The first command line argument is the name of a text file containing the description of the DFA, and the second argument gives a text file containing the description of the homomorphism. See below for the description format for the homomorphism. Output is to standard output.

Determining properties of a DFA. The sole command line argument gives the file describing the DFA. Output is to standard output and consists of just two words separated by whitespace: the first is either "`empty`" or "`nonempty`", and the second is either "`finite`" or "`infinite`". Note that the language of a DFA is nonempty if and only if some final state is reachable from the start state, and the language is infinite if and only if there is some cycle that is reachable from the start state and from which a final state is reachable.

In all cases, you may assume that the files adhere to their respective formats, i.e., you won't need to error-check an input file. Note that all input is from text files given as command line arguments; your programs should not expect any input from standard input. Your programs' "official" output goes to standard output (not a text file), but you may also print anything you want to standard error; the grading process will ignore anything sent to standard error.[2]

Your code should be economically written, well-structured, and well-commented, following the common stylistic guidelines of the programming language you use. The code should also be reasonably efficient, but this is a secondary requirement. If your code runs correctly, we won't really look too closely at the source code. If it does not run correctly, however, the source code style is more important.

DFA description format

We will use a common ASCII text format for describing DFAs. If you read a DFA description as input, you can expect it to be in this format, i.e., you don't need to error-check the description. If you write a DFA as output, you must use the same format. (This allows the flexibility of using the output of one program as the input to another.) The format is meant to be both simple to parse and easily readable by humans.

[2]The three I/O streams open by default on Linux programs *standard input* (buffered keyboard input by default), *standard output* (buffered screen output by default), and *standard error* (unbuffered output sent to the screen by default, even if standard output is redirected by the system). Some programming environments may use different names for these streams, e.g., C programs using `stdio.h` for high-level I/O call these stdin, stdout, and stderr, respectively; C++ programs typically use cin, cout, and cerr for the same purpose.

Here is a sample description of a 5-state DFA that accepts a binary string if and only if the numerical value of the string differs by 1 from a multiple of 5:

```
Number of states: 5
Accepting states: 1 4
Alphabet: 01
0 1
2 3
4 0
1 2
3 4
```

Generally,

- The first line starts with "Number of states: " followed by a single positive decimal integer giving the number of states of the DFA. If the DFA has n states, then we assume that the state set is $Q = \{0, 1, 2, \ldots, n-1\}$, with 0 *always* being the start state.

- The second line starts with "Accepting states: " followed by a list of nonnegative integers indicating the states that are accepting. The numbers are separated by white space (strings of one or more space characters) and should appear in increasing order.

- The third line starts with "Alphabet: " followed by a single string of characters that takes up the rest of the line. The characters in the string form the alphabet of the DFA. They should come in order of increasing ASCII code, with no duplicates. We will restrict our alphabets to printable ASCII characters only, i.e., no control characters like carriage returns, line feeds, newlines, etc. The space character is considered printable—and thus allowed—but the tab and DEL characters are not. This means that the printable ASCII characters run from 32 through 126 (decimal), inclusive. The alphabet string starts with the first character after the space after the first colon, and runs through the end of the line (not including the final newline).

- The rest of the description consists of the transition table. The rows of the table (each terminated with a newline character) correspond to the states of the DFA in numerical order. Each row consists of a sequence of nonnegative integers separated by white space, one number for each alphabet symbol. These numbers correspond to the destination states given the initial state and the alphabet symbol. The order of entries within each row must correspond to the order of characters in the alphabet string given earlier.

Thus the sample DFA description above corresponds to the more traditional tabular form

	0	1
$\rightarrow 0$	0	1
$*1$	2	3
2	4	0
3	1	2
$*4$	3	4

Homomorphism description format

This only applies to the fifth task. A homomorphism $h : \Sigma^* \rightarrow \Gamma^*$ will be represented in a text file by giving (a) the input alphabet Σ, (b) the output alphabet Γ, and (c) the values $h(a)$ for all $a \in \Sigma$ in increasing ASCII order. For example, if $\Sigma = \{0, 1, 2\}$, $\Gamma = \{a, b, c, d\}$, and $h(0) = abd$, $h(1) = \varepsilon$, and $h(2) = acda$, then the file will look like this:

```
Input alphabet: 012
Output alphabet: abcd
abd

acda
```

(Notice the blank line denoting the empty string for $h(1)$.) The format of each of the two lines specifying the alphabets is similar to the format of a DFA alphabet, above, with the same restrictions on the alphabet. Following the alphabet specifications is a series of strings (one per line) giving the value of the homomorphism applied to each of the input alphabet symbols, given in the same order as the symbols in the input alphabet.

Notes and hints

One way to check if $L(A) = \emptyset$ for some DFA A is to first minimize it then check if the result is the 1-state DFA with no accepting states. (More generally, an efficient way to check whether two DFAs are equivalent is to minimize both of them, then check whether the results are isomorphic (which is fairly quick, since there can be at most one isomophism between any two sane DFAs).) Minimizing also helps for checking whether $L(A)$ is finite. There can be at most one dead state (from which no final state is reachable) in a minimal DFA, so it it easy to isolate. A minimal DFA recognizes an infinite language if and only if it has a cycle not involving the dead state.

Generally, you cannot assume any *a priori* bounds on the number of states of a DFA or the length of $h(a)$ where h is a homomorphism and a is a symbol in the input alphabet of h. In other words, a DFA may less than 10 states or more than $10,000$ states. Similarly, the length of $h(a)$ may be 0 or over $10,000$. So you will not be able to declare a buffer to hold $h(a)$ values whose length is known at compile time.

Testing and Grading

As mentioned, your project will be graded mostly automatically. We will (probably) use the Perl script `project-test.pl` and test files in a test suite directory to test and grade your project. *All these files are available to you soon from the project homepage,* so that you can see how your code will be tested and even run the test program yourself to see in advance how well you do. Just to be perfectly clear: we will grade your project by running the script `project-test.pl` on it. we will not run your code personally. The comments produced by that script will determine your grade. This means that you will not get credit for attempting to do something. You will only get credit for what actually works, as determined by the `project-test.pl` script.

Generally, your output does not have to match the solution output exactly. For example, if you minimize a DFA, your output (a minimal DFA) only needs to match our solution up to relabeling (actually, re-ordering) of states. If two DFAs are the same up to relabeling/re-ordering of states, we say that they are *isomorphic.* The Myhill-Nerode theorem says that two minimal DFAs are equivalent if and only if they are isomorphic; therefore, the minimal DFA is unique *up to isomorphism.* We have a program (that we will also make available to you) that tests whether two DFAs are isomorphic, and we will use this program to compare your output with ours. For another example, if you implement the fourth task, which requires you to build a

DFA for the intersection $L(A) \cap L(B)$ for two given DFAs A and B, we will only check that your output DFA is *equivalent* to the product construction, which suffices to recognize $L(A) \cap L(B)$. Since the product construction does this already, the number of states of your output DFA should be *no more* than the product of the sizes of the state sets of A and of B.

Submission

Submission will be via the CSE Departmental Dropbox (Moodle). Upload a single file, either a .zip file or a .tar.gz file, containing

1. all your source code files, which should all be in the same directory, i.e., no subdirectories (and no automatically generated files, please),
2. a README file indicating which two of the options above you are implementing for full credit, along with anything else you want to tell us (we will actually read this), and
3. for each task you implement, a "build/run" text file giving commands to compile and/or run the program performing the task (see below for the name and contents of these files).

The text files in item (3.) should be named `simulator.txt`, `minimizer.txt`, `searcher.txt`, `boolop.txt`, `invhom.txt`, and `properties.txt`, respectively.

IMPORTANT NOTE: You *must* use either the ZIP format (file extension .zip) or the GZIPPED TAR format (file extension .tar.gz) for your submission file. Your file will be de-archived either with `unzip` or with `gunzip; tar -xf`, depending on your file name's extension. Do not use any other archive format, particularly the RAR format. If you deviate from the allowed formats, you risk getting zero credit for the entire assignment.

Examples of build/run files

Suppose you implement a DFA simulator (first task, above) in Java, and your main class is called `MySimulator`. Then your submission zip file should include a text file named `simulator.txt` whose contents look like this:

```
# Lines like these are comments and will be ignored
Build:
  javac MySimulator.java
Run:
  java MySimulator
# Don't include command line arguments to the run command!
# The indenting is optional.
```

For another example, suppose you implement a DFA minimizer in C as a single compilation unit called `my_minimizer.c`. Then include a file named `minimizer.txt` with something like the following contents:

```
Build:
  gcc my_minimizer.c
  mv a.out my_minimizer
Run:
  ./my_minimizer
# Again, no command line arguments, please.  They will be supplied automatically.
```

Note that you can have any number of build commands, and they will be executed in order (in the directory containing your source files) before the run command. Always give the Build commands first before the Run command.

Suppose instead that you have several compilation units for your programs, including shared code, and a complicated build procedure, but you have a single Makefile controlling it all, capable of producing an executable called `my_minimizer` and maybe other executables as well. Then the `minimizer.txt` file can just look something like this:

```
Build:
  make my_minimizer
Run:
  ./my_minimizer
```

and similarly for the other program(s) you implement.

As a final example, suppose you implement the inverse homomorphic image closure property in Python, which can be run directly without a compilation step. Then your `invhom.txt` file might look like this:

```
Build:
Run:
  python my_inv_hom_maker.py
# You still need to say "Build:" even though there are no build commands.
```

Finally, be sure your CSE Dropbox account exists and is accessible. Do this early on to avoid last-minute glitches.

A Windows vs. Linux pitfall

Windows-based text files end each line with the two-character sequence `\r\n` (carriage return, newline), and GNU/linux/Mac OS X and similar systems expect only an `\n` ending each line. We recommend not doing your development on a Windows box, but if you absolutely must, be aware that the files `simulator.txt`, etc. may not work properly with the test script if they are just copied over without newline conversion. (This is the cause of many mysterious failures when running the test script.) Our linux boxes support the `scc` command. Running

```
scc my_text_file.txt
```

converts all `\r\n` sequences to `\n` in `my_text_file.txt`. (Note that this command alters the contents of the file and does not produce a backup copy.)

Do your own work

The code you write and submit must be yours alone. You may discuss the homework with others at the conceptual level only, but you may not copy code directly from any other source, even if you modify it afterwards. Likewise, you must take reasonable precautions not to let your code be copied by anyone else. Violating this policy constitutes a violation of the Carolina Honor Code, and will have serious consequences, including, but not limited to, failure of the course.

If you have any questions about what this policy means, please ask.