CSCE 355, Spring 2024
Programming Assignment

Due Thursday, April 11, 2024 at 11:30pm EDT
Late assignments accepted up to 24 hours late with a 20% penalty

March 27, 2024

For the programming portion of the course (15% of your total grade) you are to write programs that do two things:

1. Take the description of an ε-NFA and output an equivalent NFA without ε-transitions, and
2. Take the description of an NFA N (with no ε-transitions) and a list of strings as input and output the acceptance/rejection result of running N on each string in the list.

For (1), you should implement the method described in class or in the course notes starting on page 29 (Method 2). For (2), you should implement the set-of-states approach described in class.

Details

The grading of your work will be automated via scripts running on Linux (These scripts are found from the project homepage: `https://cse.sc.edu/~fenner/csce355/prog-proj1/index.html`). Therefore we are 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 one of the GNU/Linux boxes in the department's linux lab (e.g., `l-1d43-12.cse.sc.edu`), and all I/O will be ASCII text. Input is read either from a text file (given as a command line argument) or standard input (or both), and output is written to standard output. You may write your program in any programming language you want, provided it is implemented on the Linux machines in the CSCE lab. We recommend Java or C or C++ or Perl or Python or Ruby or ML or Haskell or Prolog or Scheme or .... 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.

Here are more specific details for each of the two items above:

Removing ε-moves. First read the description of an ε-NFA from a file named by the first command line argument (see below for information about the format of this description). Output the equivalent NFA *in the same format* to standard output. You should not change state labels (the output state set is the same as the input state set).

[1] It's OK if we need to invoke the JVM by preceding your main class name with "java" on the command line.

Simulating an NFA. First read the description of the NFA (same format as above, but with no ε-transitions) from a file named by the command line argument. Next read a series of zero or more strings from standard input. These are inputs to the NFA. 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 NFA on the string. End each output word with a newline.

In both cases, you may assume that the inputs adhere to their respective formats, i.e., you won't need to error-check the input. Note that all the input ε-NFAs are from text files given as command line arguments, and input strings for the simulation are read 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 program 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 and within the alotted time (11 seconds), we won't really look too closely at the source code. If it does not run correctly or times out, however, the source code style might make some difference.

ε-NFA and NFA description format

We will use a common ASCII text format for describing automata. If you read an ε-NFA or NFA 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 an NFA as output, you must use the same format. *The same format will be used for both ε-NFAs and NFAs without ε-transitions; the only difference is that for the latter, the ε-transitions will all be the empty set.* This allows the flexibility of using the output of one program as the input to the other. The format is meant to be both simple to parse and easily readable by humans.

Below is a sample description of an 8-state ε-NFA N that accepts a binary string iff it is either ab repeated zero or more times or aba repeated zero or more times. (That is, it recognizes the language $L((ab)^* + (aba)^*)$.) The tabular form is

	ε	a	b
$\rightarrow 0$	$\{1, 4\}$	$\emptyset$	$\emptyset$
$*1$	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\emptyset$	$\{3\}$
3	$\{1\}$	$\emptyset$	$\emptyset$
$*4$	$\emptyset$	$\{5\}$	$\emptyset$
5	$\emptyset$	$\emptyset$	$\{6\}$
6	$\emptyset$	$\{7\}$	$\emptyset$
7	$\{4\}$	$\emptyset$	$\emptyset$

The corresponding ASCII file format looks like this:

[2] We call the three I/O streams open by default on GNU/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 input and output are 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.

```
Number of states: 8
Alphabet size: 2
Accepting states: 1 4
{1,4}   {}      {}
{}      {2}     {}
{}      {}      {3}
{1}     {}      {}
{}      {5}     {}
{}      {}      {6}
{}      {7}     {}
{4}     {}      {}
```

Generally,

- The first line starts with "Number of states: " followed by a single positive decimal integer giving the number of states of the ε-NFA N. If N 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. You may assume that N will never have more than 64 states.

- The second line starts with "Alphabet size: " followed by a single positive decimal integer giving the size of N's alphabet. Assume that N's alphabet is an initial segment of the lowercase letters $a, b, c, \ldots, z$. So for example, if the alphabet size is 2, then N's alphabet is $\{a, b\}$; if the size is 5, then the alphabet is $\{a, b, c, d, e\}$, etc. The alphabet size will always be between 1 and 26, inclusive.

- The third line starts with "Accepting states: " followed by a list of nonnegative integers indicating the states that are accepting. Consecutive numbers are separated by *whitespace* (i.e., a string of one or more spaces and/or tabs) and should appear in increasing order.

- The rest of the description consists of the guts of the transition table. The rows of the table (each terminated with a newline character) correspond to the states of N in numerical order. Each row consists of a sequence of sets of nonnegative integers, optionally separated by whitespace. The first set gives the possible ε-transitions and the rest give the transitions on each alphabet symbol in alphabetical order. Each set consists of a comma-separated list of zero or more nonnegative integers in increasing order, surrounded by curly braces. No whitespace is allowed anywhere between the braces. Whitespace is allowed (but not required) between sets and at the beginning and/or end of the line.

You may assume that all input ε-NFAs adhere to these format rules, and your output automaton must also adhere to these rules. Remember: the format for an NFA without ε-transitions is the same as with ε-NFA; there will still be entries (the first entry of each row) corresponding to ε-transitions, but all these entries will be the empty set. In the example above, here is the equivalent NFA without ε-transitions, given in the proper format:

```
Number of states: 8
Alphabet size: 2
Accepting states: 0 1 3 4 7
{}      {2,5}   {}
{}        {2}       {}
{}        {}        {3}
{}        {2}       {}
{}        {5}       {}
{}        {}        {6}
{}        {7}       {}
{}        {5}       {}
```

To ensure a unique output to your ε-transition removal program, you are required to preserve the order of states in the output, i.e., don't permute states. You are *not* required to line up the sets neatly in columns (as I did above), and there will be no penalty for ugly output, as long as the formatting rules are observed. (The output will be checked automatically by a script that strips all whitespace from each row of the transition table.) The format rules allow you the option of indenting and lining up the columns for your own sake, however, and you must also allow these options in the input files.

For your simulation program, you may assume that the first set in each row of the transition table is the empty set `{}`.

Notes and Hints

You can always assume that the input ε-NFA will have 64 or fewer states. It is no coincidence that 64 is the number of bits in a machine word on a modern Intel architecture. (One completely optional way to take advantage of this is to represent a set of states as a bit mask in your simulation program.) You cannot assume any *a priori* bound on the number or lengths of strings on which you need to simulate the NFA, however. A string may be ε (indicated by an empty line) or it may be millions of characters long.

One way to test your ε-transition removal program is to take its output as input to the same program. In this case, the resulting output should be identical to the input.

Your programs perform some tasks that are common between them, like reading an automaton from input. To economize code and simplify debugging, it is a good idea to share common code rather than duplicating it between your programs.

Testing and Grading

As we mentioned, your project will be graded automatically. We will use the Perl script `project-test.pl` and test files in a test suite directory to test and grade your project. *All these files will be 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 with owner privileges using one of the Linux lab machines. 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.

Submission

Submission will be via 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. an optional file `readme.txt` with anything you want to tell us (we will read this with our own eyes), and
3. for each of your two programs, a "build-run" text file giving Linux (bash) shell commands to compile and/or run the program performing each task. The two build-run files should be named `e-remove.txt` and `simulate.txt`, respectively, for the ε-transition removal program and the simulation program. See below for the contents of these files.

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, which is proprietary to Windows (I personally do not have Windows on any machine I use). If you deviate from the allowed formats, you risk getting zero credit for the entire assignment. Keep in mind that Linux file names are case-sensitive.

Examples of build-run files

Suppose you implement the ε-transition removal program in Java, and your main class is called `MyEpsilonRemover`. Then your `e-remove.txt` file would look like this:

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

For another example, suppose you implement the NFA simulation program in C as a single compilation unit called `my_simulator.c`. Then your `simulate.txt` file would look something like this:

```
Build:
  gcc my_simulator.c
  mv a.out my_simulator
Run:
  ./my_simulator
# 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_simulator` and maybe `my_e_remover` as well. Then the `simulate.txt` file can just look something like this:

```
Build:
  make -B my_simulator
Run:
  ./my_simulator
```

and the `e-remove.txt` file would look similar. (Use the `-B` option or `--always-make` option with `make`; it will build your entire program from source regardless of any intermediate files.)

As a final example, suppose you implement the simulator in Python, which is a scripting language that can be run directly without a compilation step. Then your `simulate.txt` file might look like this:

```
Build:
Run:
  python my_simulator.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 OSX and similar systems expect only an `\n` ending each line. We strongly recommend against doing your development on a Windows box, but if you absolutely must, be aware that the files `e-remove.txt` and `simulate.txt` may not work properly with the test script if they are 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 (see the next paragraph), but you may not copy code directly from any other source, even if you modify it afterwards. Likewise, you must take all reasonable precautions not to let your code be copied by anyone else, either in this class or in future classes. This includes uploading or developing your code on a web platform—such as SourceForge or GitHub—in a way that can be seen by others. 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.

Discussing the project with others in the class is allowed (even encouraged), but you must include in your `readme.txt` file the names of those with whom you discussed the project.

If you have any questions about what this policy means, please review the relevant section of the course syllabus or ask me.