CSCE 355, Spring 2025
Programming Assignment

Due Thursday April 24, 2025 at 11:30 pm EDT

For the programming portion of the course (15% of your total grade) you are to write routines to analyze regular expressions (regexes). Each of your routines will take a sequence of regexes, one per line, and do one of two things, depending on the type of routine:

1. answer an yes/no question about each regex, or
2. transform each regex into a different regex (illustrating a closure property of the class of regular languages).

Later in this handout are recursive rules to use to perform some of the tasks. You must produce an output consistent with these rules. These routines build on each other to some extent: you will often find it best for one routine to call another routine.

1 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-proj2/sp25/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., `l1d43-12`), and all I/O will be ASCII text. Input is read from standard input only, and normal output is written to standard output only; error messages are written to standard error (these three streams may have different names depending on which programming language you use). 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.

Your program should take one or two command-line arguments specifying which task it is to perform on each input regex r. The possible arguments, with their associated tasks, are given below. For each task answering a yes/no question, the output should be either "yes" or "no," followed by a newline, for each input regex. Formal rules for performing a few of these tasks will be given later in this handout. (Following the book's convention, we let $L(r)$ be the language denoted by r, for any regex r.) The strings in $L(r)$ are said to *match* r, or be *matched* by r. Any symbol

[1] It's OK if we need to invoke the JVM by preceding your main class name with "java" on the command line, or the python interpreter by preceding your program with "python" or "python3."

given below in `typewriter` font, such as "`a`," "`b`," etc. should be taken literally as an alphabet symbol, whereas italicized symbols, like "a," represent variables that range over symbols.

`--no-op` : Output r (unaltered). (This is not completely trivial, because you read in r in postfix form and write out r in prefix form.)

`--simplify` : Output a regex equivalent to r after applying some simplifying transformations (see **Simplifying a Regex**, below).

`--empty` : Determine whether or not $L(r) = \emptyset$, i.e., whether r matches no strings.

`--has-epsilon` : Determine whether $\varepsilon \in L(r)$, i.e., whether r matches the empty string.

`--has-nonepsilon` : Determine whether $L(r)$ contains some nonempty string.

`--uses` a : Determine whether $L(r)$ contains a string that has an "a" somewhere in it. (The symbol a is given on the command line as an additional argument.)

`--not-using` a : For the given symbol $a \in \Sigma$, output a regex r' such that $L(r)$ contains all those strings in $L(r)$ that do not include the symbol a anywhere in the string.

`--infinite` Determine whether $L(r)$ contains infinitely many strings.

`--starts-with` a : For the given symbol $a \in \Sigma$, determine whether $L(r)$ contains a string starting with a. (The symbol a is given on the command line as an additional argument.)

`--reverse` : Output a regex r' denoting the string reversal $L(r)^R$ of $L(r)$, that is, $L(r') = L(r)^R$.

`--ends-with` a : For the given symbol $a \in \Sigma$, determine whether $L(r)$ contains a string ending with a. (The symbol a is given on the command line as an additional argument.)

`--prefixes` : Output a regex r' denoting the language of all prefixes of strings in $L(r)$, that is, $L(r') = \{w \in \Sigma^* \mid (\exists x \in \Sigma^*)wx \in L(r)\}$.

`--bs-for-a` : Output a regex r' such that $L(r')$ consists of those strings obtained from strings $w \in L(r)$ by inserting a string of zero or more consecutive "'`b`"s in place of every occurrence of "`a`" in w, independently for every occurrence of "`a`." This task recalls one of the problems on the second midterm.

`--insert` a : For the given symbol $a \in \Sigma$, output a regex r' for the language of all strings obtained from strings $w \in L(r)$ by inserting the symbol a one time anywhere in w. That is, $L(r') = \{xay \mid x, y \in \Sigma^* \ \& \ xy \in L(r)\}$. (The symbol a is given on the command line as an additional argument.)

`--strip` a : For the given symbol $a \in \Sigma$, output a regex r' for the language obtained from $L(r)$ by stripping off the initial symbol a from all strings in $L(r)$ that start with a (while excluding any strings in $L(r)$ not starting with a). That is, $L(r') = \{x \in \Sigma^* \mid ax \in L(r)\}$. (The symbol a is given on the command line as an additional argument.)

1.1 Regex syntax

A regex over an alphabet Σ is built from *atoms*, each of which is either the empty set $\emptyset$ or a symbol from Σ, using three possible operations: union, concatenation, and the Kleene closure operator. Union and concatenation are binary operators, taking two operands, and the Kleene closure operator is a unary operator, taking one operand.

Formally, a regex r is either: (1) and atom, or (2) of the form $s + t$ or st or s^*, where s and t are regexes. This is the basis for the recursive rules defined later in this handout—the base cases are (1) and the recursive cases are (2). This describes the infix form of a regex, and parentheses are allowed to control grouping, overriding the normal precedence and associativity rules.

For this assignment, all regexes will be over the common alphabet Σ consisting of the ten decimal digits `0`, ..., `9` and the 26 lower-case letters `a`, ..., `z` from the English alphabet. (Thirty-six symbols in all.) The forward slash (`/`) will denote the empty set $\emptyset$. We will use the plus sign (`+`) for the union operator, the period (`.`) for the concatenation operator, and the star (`*`) for the Kleene closure operator. (The period for concatenation is only needed in prefix and postfix forms (see below); infix regexes use juxtaposition for concatenation).

1.2 Input/output formats

Your program should accept regexes in *postfix* form, that is, operators always appear *after* their operands, and your program should output regexes in *prefix* form, that is, where each operator appears *before* its operands. Here are some sample regexes given in the traditional infix form and their postfix and prefix equivalents:

infix	postfix	prefix
$a + b$	`ab+`	`+ab`
ab	`ab.`	`.ab`
ab^*	`ab*.`	`.a*b`
$(ab)^*$	`ab.*`	`*.ab`
$a + b + c$	`ab+c+`	`++abc`
abc	`ab.c.`	`..abc`
$ab + c$	`ab.c+`	`+.abc`
$a(b + c)$	`abc+.`	`.a+bc`
$\emptyset$	`/`	`/`
$\emptyset^*$	`/*`	`*/`

Note that we use the period (`.`) to denote concatenation explicitly in the postfix and prefix forms, as well as plus ($+$) for union and star ($*$) for Kleene closure as usual. Concatenation in the infix form is given by juxtaposition as usual. The last row is the regex matching ε and nothing else.

Postfix and prefix forms make regexes especially easy for a program to process. For one thing, parentheses are not needed. These forms are hard for a human being to read, however, given how much we are used to reading infix. For this reason, I have provided some utilities for converting between these forms: the program `in2post` reads infix regexes line by line and converts them to postfix; the program `pre2in` reads prefix regexes line by line and converts them to infix (with just enough parentheses to control grouping). Download the .zip file from the project homepage and unzip it. Then for convenience, move these executables to a directory in your path variable for finding executables (your personal bin directory, for example). All test files will have infix forms of

the regexes. When you test your program, you can run something like this for the yes/no answering tasks:

```
$ in2post | your_program --empty
```

then type in infix regexes at the keyboard (one per line, ending with Ctrl-D on an empty line). For the regex transformation tasks, run something like this:

```
$ in2post | your_program --reverse | pre2in
```

and see the results in infix of the regexes you type at the keyboard. To redirect from and to files, you can do this:

```
$ (in2post | your_program | pre2in) < your_input_file > your_output_file
```

or this:

```
$ cat your_input_file | in2post | your_program --some-arg | pre2in > your_output_file
```

and similarly for the yes/no answering tasks. The `<` and `>` symbols above are typed literally.

The two utilities `in2post` and `pre2in` allow spaces, tabs, and blank lines on the input, but strip them from their output. They also treat anything starting with a pound sign (`#`) as the start of a 1-line comment and ignore everything afterwards on the line. (The .zip file also includes the source code for these utilities, which should be highly portable, so you can compile and use them on your own system. The Makefile assumes `gcc`, but any reasonable C compiler should work. If you don't have a `make` utility available, just run the commands in the Makefile by yourself.)

You may assume that when we test your program for grading, the inputs will adhere to their respective formats, i.e., you won't need to error-check the input. You can send anything you want to standard error; the grading program will ignore it.[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. The script allows a generous 11 seconds for each run of your code. You'll be fine as long as your runs keep to this time limit; otherwise, you won't get credit.

1.3 Simplifying a Regex

To perform the `--simplify` task, you will apply a small handful of transformations on each regex r, outputting an equivalent, possibly simplified regex r'. By "simplified" I mean that r' has a shorter prefix representation than r (or equivalently, r' has a shorter postfix representation than r, or equivalently, the syntax tree of r' has fewer nodes than that of r). Simplification should occur bottom-up: you first recursively simplify the subregex(es) of r, then apply as many of the following transformations as possible at the top level (s stands for an arbitrary regex):

1. $s^{**} \to s^*$

[2]We call the three I/O streams open by default on GNU/Linux programs *standard input* (keyboard input by default), *standard output* (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.

2. $\emptyset + s \to s$ and $s + \emptyset \to s$

3. $\emptyset s \to \emptyset$ and $s\emptyset \to \emptyset$

4. $\emptyset^* s \to s$ and $s\emptyset^* \to s$

5. $(s + \emptyset^*)^* \to s^*$ and $(\emptyset^* + s)^* \to s^*$

If none of these simplifications occur, then the output should just be r.

Other transformations are possible, which may or may not simplify a regex but which might lead to subsequent simplification opportunities. These are all strictly optional. Here, s, t, u are arbitrary regexes. A double arrow indicates that the transformations can go either way.

1. $s + t \leftrightarrow t + s$

2. $(s + t) + u \leftrightarrow s + (t + u)$

3. $(st)u \leftrightarrow s(tu)$

4. $st + su \to s(t + u)$ and $su + tu \to (s + t)u$ (You will need to check that the left-hand side is factorable.)

NOTE: The other tasks that output regexes should not do any simplifications. Simplification should only be performed for the `--simplify` task alone.

2 Notes and Hints

You may store each regex internally any way you like, but I *strongly* urge you to store it as a syntax tree, where the leaves are atomic regexes and each internal node is an operator to be applied to its subtree(s). The advantage of a syntax tree is that you can apply the rules using tree recursion.

To build a syntax tree from a postfix regex, you maintain a stack of (pointers to) tree nodes while reading the regex character by character. When you encounter an atomic regex, you build a new leaf node for it and push it on the stack. When you encounter $*$ (the Kleene closure operator), you pop a tree node off the stack and make it the child of a new node for $*$, and push the new node back on the stack. When you encounter a binary operator (either + or .), you pop two nodes off the stack, make them the right and left children (respectively) of a new parent node for the operator, which you then push back on the stack. When you are done (assuming no syntax errors in the input regex), you will be left with exactly one item on the stack—this is the root of the final tree.

For all tasks except simplification, there should be no need to modify the tree once it is built; the only differences are in what is output. I suspect this is not true for the `--simplify` task, however. My own implementation for this task modifies the tree.

Within reason, you should not assume any bound on the length of an input regex. That means you should implement the tree as a linked structure, or, if you want to use one or more arrays instead, you should be able to resize them if needed.

3 Testing and Grading

As we mentioned, your project will be graded automatically. To grade your project, we will use the script `project-self-test.pl` (written in Perl) and test files in a test suite directory to test and grade your project. *All these files will be available to you 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-self-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-self-test.pl` script run on a CSE Linux Lab machine.

The scoring is calculated as follows:

- The score is out of 100 points. There are 110 points possible, but any score over 100 points will be truncated to 100 points.
- Implementing `--no-op` correctly on all test inputs earns 35 points.
- Implementing `--simplify` correctly on all test inputs earns 10 points. (I recommend saving this for last, as it may be the most time-consuming.)
- Each additional task implemented completely correctly on all test inputs counts for 5 additional points. (There are 13 additional tasks.)
- Partial credit will *not* be given for any individual task that does not work on *all* test inputs. Working correctly on some inputs but not others will not earn any credit for that task.

4 Submission

Submission will be via Blackboard. 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. a "build-run" text file giving Linux (bash) shell commands to compile and/or run your program. Don't include the command line arguments (e.g., `--has-epsilon`); we will supply those separately when we run your program. This file should be named `build-run.txt` and placed in the same directory as your other source files. See below for the contents of this file.

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.

4.1 Examples of build-run files

Suppose you implement your program in Java, and your main class is called `MyTaskMaster`. Then your `build-run.txt` file would look like this:

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

For another example, suppose you implement your program in C as a single compilation unit called `my_task_master.c`. Then your `build-run.txt` file would look something like this:

```
Build:
  gcc my_task_master.c
  mv a.out my_task_master
Run:
  ./my_task_master
# 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_task_master`. Then the `build-run.txt` file can just look something like this:

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

(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 your program 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_task_master.py
# You still need to say "Build:" even though there are no build commands.
# For the linux machines you may have to use python3 instead of python,
# as python may default to version 2.x.x.
```

5 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.

6 Rules for Processing Regexes

The rules you will apply to process each regex r (either to answer a yes/no question about r or to output a transformed regex r') are recursive, based on the syntax of the regex. That means that atomic regexes form the base case(s), while processing a nonatomic regex may make recursive calls to its subexpressions, depending on the top-level operator used.

Here are the rules for some selected tasks. For each, we let r denote the input regex. For tasks answering yes/no questions, we let $Q_0(r), Q_1(r), Q_2(r), \ldots$ (defined below) be the statements about $L(r)$ to be answered (think of Q_i as a Boolean-valued predicate for $i = 0, 1, 2, \ldots$). For tasks outputting transformed regexes, we let r' denote the output regex. Each rule says what to do depending on the form of r. For the recursive cases, s and t denote arbitrary regexes, and c denotes any single symbol in Σ. "Or" is always interpreted inclusively.

6.1 Yes/no questions

`--empty` : $Q_0(r) :=$ "$L(r) = \emptyset$"

r	$Q_0(r)$?	comment
$\emptyset$	yes	of course
c	no	for any $c \in \Sigma$
$s + t$	iff $Q_0(s)$ and $Q_0(t)$	
st	iff $Q_0(s)$ or $Q_0(t)$	
s^*	no	s^* always contains ε

`--has-epsilon` : $Q_1(r) :=$ "$\varepsilon \in L(r)$"

r	$Q_1(r)$?	comment
$\emptyset$	no	
c	no	
$s + t$	iff $Q_1(s)$ or $Q_1(t)$	
st	iff $Q_1(s)$ and $Q_1(t)$	
s^*	yes	s^* always contains ε

`--has-nonepsilon` : $Q_2(r) :=$ "$L(r)$ contains a nonempty string" (you fill in the rules for this one)

`--uses` a : $Q_3(r) :=$ "$L(r)$ contains a string in which a occurs" (you fill in the rules for this one)

`--infinite` : $Q_4(r) :=$ "$L(r)$ contains infinitely many strings" (you fill in the rules for this one; hint: this uses Q_4 recursively as well as Q_0 and Q_2)

`--starts-with` a : $Q_5(r) :=$ "$L(r)$ contains a string starting with a" (you fill in the rules; this predicate depends implicitly on the value of a)

`--ends-with` a : $Q_6(r) :=$ "$L(r)$ contains a string ending with a" (this predicate depends implicitly on the value of a; you provide the rules for this one; hint: to save code, you might try applying Q_5 to the reverse regex)

6.2 Regex transformations

`--simplify` : Described above.

`--reverse` : You fill in the rules for this one.

`--prefixes` :

r	r'	comment
$\emptyset$	$\emptyset$	
c	$c + \emptyset^*$	recall $L(\emptyset^*) = \{\varepsilon\}$
$s + t$	$s' + t'$	
st	if $Q_0(t)$ then $\emptyset$ else $s' + st'$	
s^*	if $Q_0(s)$ then $\emptyset^*$ else s^*s'	

`--bs-for-a` : You provide the rules for this one.

`--insert` a : You provide the rules for this one.

`--strip` a :

r	r'	comment
$\emptyset$	$\emptyset$	
c	if $c = a$ then $\emptyset^*$ else $\emptyset$	recall $L(\emptyset^*) = \{\varepsilon\}$
$s + t$	$s' + t'$	
st	if $Q_1(s)$ then $s't + t'$ else $s't$	
s^*	$s's^*$	