# Extending `thegrep`

PS07 - COMP590 - Spring 2019

## Overview

In the third part of `thegrep` you will extend its capabilities across three dimensions:

1. Given a regular expression, generate random strings that match it with the addtion of a `-g` or `--gen` flag to the program.
2. Implement operator overloading on `NFA` such that you can concatenate two `NFA` structs with the `+` operator.
3. Add the extended regular expression operator for one-or-more occurrences (`+`).

This part of the problem set gives experience with:

1. Relying upon documentation from public, open source projects to provide specific functionality without reinvention.
2. Growing the scope of a program across many cross-cutting layers of its implementation.

## Getting Started

This assignment will extend your previous version of `thegrep` and you should use it as your starting point.

## Part 1 - Generating Random Acceptable Input Strings

Add a command-line flag to generate random, acceptable input strings given a regular expression. Example usage:

```
$ cargo run -- -g 4 'omg( loll*| ha(ha)*)*'
omg ha lol
omg hahahaha ha
omg lolll loll
omg
```

```
$ cargo run -- --gen 3 '(tarr*|heee*ll*ss*)'
tarr
heelssss
heeeellllsss
```

```
$ cargo run -- -g 5 'pass: s.a.f.e'
pass:  sLauf2e
pass:  sQaIfve
pass:  sQaKfFe
pass:  sKa4fIe
pass:  s2aZfde
```

For documentation on how to add a command-line argument that takes a number parameter, please refer to the `structopt` documentation: https://docs.rs/structopt/0.2.15/structopt/

To generate random values you should make use of the `rand` crate:

- Crate: https://crates.io/crates/rand
- Crate Documentation: https://rust-random.github.io/rand/rand/index.html

An intended challenge of this part of the problem set is figuring out how to use a 3rd party library on your own. For generating a random boolean, consider the `random` function. For generating a random char, consider the `Alphanumeric` distribution. You should put some effort into properly using these components of the `rand` crate as their example uses may not match your exact need.

You should add at least a few unit tests to test this functionality. In your unit test you are permitted to rely upon the assumption your `accepts` method is properly implemented (and tested) separately.

## Part 2 - Operator Overloading

Overload the addition operator of `NFA` such that you can concatenate two `NFA` structs with `+` to result in a new `NFA` struct. Refer to Midterm 1's question 6.2 on Gradescope, "Suppose you want to override the addition operator for `NFA`..."

Please note there is already a private helper method defined on `NFA` named `add`. One of the challenges in adding new capabilities to existing programs is overcoming design decisions made without knowing of future extensions. The private `add` method will conflict with the Add trait's `add` method. You should begin by renaming the existing `add` method to something else descriptive and meaningful besides `add` before attempting to introduce the addition operator overload.

One simplification was made in the pseudo-code of the midterm to avoid bogging you down in unnecessary detail: `NFA`'s `from` constructor returns a `Result`. More accurate example usages look like this:

```
let ab = NFA::from("ab").unwrap();
let cd = NFA::from("cd").unwrap();
let abcd = ab + cd;
assert!(abcd.accepts("abcd"));
assert!(!abcd.accepts("abcde"));

let a_star = NFA::from("a*").unwrap();
let b_star = NFA::from("b*").unwrap();
let ab = a_star + b_star;
assert!(ab.accepts("a"));
assert!(ab.accepts("b"));
assert!(ab.accepts("ab"));
assert!(ab.accepts("aabbb"));
```

*Words of Wisdom:* Before you begin any code, generate DOT diagrams of the LHS and the RHS of the two examples above paying particularly close attention to each state's ID. Then, draw out what the resulting `NFA` should look like with careful consideration to what each state's ID in the resulting `NFA` will be. For simplicity's sake in this part of the assignment, your `NFA` is permitted to have a second, dummy `Start` state in the middle of the NFA at the point of concatenation.

The introduction of the addition operator is primarily for conceptual exposure and practice with the problems of operator overloading and relocation in memory. This operator will not be accessible via any command-line functionality. It would be useful if you were building a regular expression library, though. As such, unit tests must be written to prove its functionality. Be sure your unit tests cover the test cases above, as well as test cases where alternation is the topmost level operator of the right-hand side. These will be checked in hand grading not in autograding.

## Part 3 - Extending the Regular Expression Syntax with Kleene Plus

The final part of this problem set adds the Kleene Plus ("one-or-more times") operator to your regular expression engine's capabilities. After completing this part of the problem set, the following tokens and parse outputs are expected:

```
$ cargo run -- --tokens 'ab|().*+'
Char('a')
Char('b')
UnionBar
LParen
RParen
AnyChar
KleeneStar
KleenePlus
```

```
$ cargo run -- --parse '.+'
OneOrMore(AnyChar)
```

You will need to add your own variants to Token and AST. You should add test cases for the Kleene plus. Once added, your program should properly filter against the pattern (some matches omitted):

```
$ cargo run -- 'uu+' ~/dict
continuum
continuums
muumuu
muumuus
vacuum
vacuumed
vacuuming
vacuums
```

Finally, the `--gen` flag of Part 1 should *also* produce acceptable strings for patterns that utilize the Kleene Plus. Why should the `--gen` functionality *just work* without special effort to handle the Kleene Plus?

## Grading Rubric Breakdown

Autograding will only test Part 1 and Part 3's implementation.

1. 30pts - Part 1. `--gen` flag.
2. 30pts - Part 3. Kleene Plus implementation.

Hand-graded Points

1. 30 points - Part 2. Concatenation implementation.
2. 10 points - Unit tests for Part 1 and Part 3.