Final Specification
(The Regex - Peter Kraft, Lilly Shen, Kimberley Yu, Kat Zhou)

Timeline:
April 19-April 25th

- Implement to_nfa in Nfa
- Implement parse in Parser
    - We will have to implement the following helper functions:
        - val char : char -> char parser
        - val concat : 'a parser -> 'b parser -> ('a * 'b) parser
        - val choice : 'a parser -> 'a parser -> 'a parser
        - val map: ('a -> 'b) -> 'a parser -> 'b parser

    And then use them to build a recursive-descent parser

April 26-May 2nd

- Implement eval in Emulator
- Pull everything together in Regex and manage inputs/outputs
- Any extra functionality that we have time for - ex. wildcard, escape sequences/tokenizer
- Demo video

Signatures/Interfaces:

1. Parser.ml
    a. The Parse module exposes the pt abstract data type (representing a parse tree) and the parse function, which takes in a string as input (i.e. a regular expression) and returns a parse tree.
    b. The parse tree (type pt) is implemented as an algebraic datatype that represents our 6 types of regular expressions: Empty, Single of char (a single character), Paren of pt (parentheses for grouping), Cat of pt * pt (concatenation), Or of pt * pt (Boolean or), and Star of pt (closure/Kleene star).
2. Nfa.ml
    a. The Auto module defines the type nfa, which consists of either an Empty, a Single of schar * nfa ref, an Or of nfa ref * nfa ref, or a Star of nfa ref * nfa ref. These represent the types of nfa nodes that could exist in an nfa graph. It also includes a (not exposed) function lptr which takes an nfa and returns an nfa ref list of all nfa refs that point to empty; the "dangling arrows" that an incomplete nfa has. This function will help us "tie together" complex nfa's in order to ensure a Star always points back to itself or the left branch of a Cat points at the right branch. Lastly, to_nfa builds up the nfa graph and is exposed. It will probably recursively move up the tree, creating simple nfa's at first and then combining them using lptr.
3. Emulator.ml
    a. The Emulator exposes the eval function. The eval function takes in a string and the nfa for a regular expression. It will walk the the nfa graph to check whether the string matches the regular expression, returning true if it matches and false if

Commented [1]: We actually didn't need to do this

Commented [2]: In addition to all of this, Parser and Nfa expose additional functions that print DOT code for the parse tree and nfa

Commented [3]: Also ended up needing a tokenizer with escape sequences plus extensive error-checking code

Commented [4]: actually schar, which includes wildcards and character classes

Commented [5]: Empty and Paren weren't necessary, but we did need a new type for options ('?') and we had to modify the Single type to accommodate wildcards and character classes

Commented [6]: Once again, options added, along with wildcards and character classes

it doesn't. Emulator.ml might contain other exposed functions later as time permits.

4. Regex.ml
    a. Regex.ml handles I/O

Progress Report: (include snippets of code)

- We have written the interfaces for Parser, Nfa, and Emulator. We have begun to implement parts of Parser and Nfa. Please refer to those files in the attached zip file to see our progress on writing code.

Version Control: We are using GitHub: https://github.com/kraftp/CS51-Final-Project

Commented [7]: This was harder than expected

Commented [8]: In command line input, you can add flags to print DOT code for the parse tree and nfa.