Project name: Regular Expressions

Team members:

Peter Kraft: pkraft@college.harvard.edu

Kimberley Yu: kimberleyyu@college.harvard.edu

Lilly Shen: lshen01@college.harvard.edu

Kat Zhou: kathleenzhou@college.harvard.edu

Brief Overview:

We're implementing regular expressions in OCaml for our project. To demonstrate an example of what a regular expression is and why it is useful, let's say we have a document with a list of strings, some of which are 'CS50' and others of which are 'CS51'. We want to be able to parse through the document and find those that are either 'CS50' or 'CS51'. A regular expression is a sequence of characters that would allow us to pattern match for CS50/CS51 in the document. In this case, 'CS5[01]' (using Python syntax for the sake of example, we will probably implement a different syntax but aren't sure yet) could be the regular expression. This indicates we are looking for a string of characters starting with 'CS5' and ending with either '0' or '1'. For our project, we want to write a program that takes a regular expression (such as 'CS5[01]') and turns it into a parse tree. We will then write a program that takes the parse tree to build a finite automaton which can take in a string and return a boolean indicating whether the string matches the regular expression (is a 'CS50' or 'CS51').

**Commented [1]:** In our final code, this would be 'CS5(0|1)'

**Commented [2]:** nondeterministic

**Commented [3]:** We also needed an emulator to run the nondeterministic finite automaton

Feature list:

The core of our project is a program that takes regular expressions as input and does matching with them. The program will have two parts. The first is a parser that takes a regular expression as input and returns a parse tree (an abstract data type that we'll define) as output.

**Commented [4]:** Three, including the NFA emulator

The second is a matcher that takes a string and a parse tree as input, builds a finite automaton, and outputs a boolean that indicates whether the string matches the parse tree.

For the core of our project, we will only work with basic regular expressions and the most basic functions involving them, the core regular language functionality. This means that our regular expressions will either be *empty*, be a *single character,* be a *parenthesis* of a regular expression (used to indicate ordering), be a *concatenation* of a regular expression and another regular expression, be an *or* of a regular expression and another regular expression, or be a *closure* (Kleene star) of a regular expression. If we have time, will will extend this to contain other operators that are not necessary to make regular expressions complete but that would be convenient, like a *complement* operator of a regular expression.

**Commented [5]:** Instead, we implemented wildcards, options, character classes, and escape sequences. We didn't think having a complement operator would be particularly useful

Similarly, the core of our project involves only writing a parser and a matcher. Those two programs include much of the theoretically interesting functionality of regular expressions by implementing the theoretical equivalence between a regular expression and a finite automaton (we probably aren't going to touch deterministic finite automata) and then using it to do pattern-matching on strings. Most of the truly interesting problems--the writing of a parser, the creation of the automaton--are done there. However, if we have some time, we might implement some more functions that make regular expressions easier to use, probably taking ideas from Python's *re* module. These include a *search* function that takes in a string and a regular expression and returns a list of substrings of the string that match the regular expression, a *split* function that takes a regular expression as a delimiter, a *sub* function that takes two strings and a regular expression as input and replaces all substrings of the first string that match the regular expression with the second string, and several others.

**Commented [6]:** We also decided to implement the DOT code generators, which we thought were really interesting and novel and emphasized the theoretical slant of our project by visualization of our abstract code.

**Commented [7]:** We prioritized extending our regular expression grammar over implementing more functions because we thought that having these functions over a limited grammar wouldn't be very useful.

Our minimum complete project is the parser of basic regular expressions and the string matcher. This contains most of the computationally and theoretically interesting components of regular expressions, and we think that this project will be sufficiently challenging.

**Commented [8]:** Got all that (with a tokenizer and error-checking code), plus wildcards, character classes, the option ('?') operator, and the DOT code generator

Technical Specification

We're modularizing our project into the parser and the matcher. Doing that is actually very intuitive and fairly straightforward because the parser outputs a parse tree and the matcher inputs a parse tree. So if we define an algebraic data type for a parse tree as a team beforehand along with clear invariants that leave no ambiguity as for what the parse tree will look like for a given regular expression, one or two of us can work on the parser and two or three of us can work on the matcher completely independently, each knowing exactly what the other will do and not caring about how they'll do it. The parser will probably be implemented as a module that exposes the "parse_tree" abstract data type and a "parse (str : string) : parse_tree" function. The matcher will probably be implemented as a module that exposes a "match (str: string) (pt : parse_tree) : bool" function (and maybe a few other functions if we have time, as described above). The matcher itself contains two parts, one part that builds a data structure to hold a non-deterministic finite automaton and one part that emulates the automaton; these parts can also be abstracted and worked on separately using an algebraic data structure that holds the output of the builder and is an input into the emulator.

**Commented [9]:** Parser, compiler, and emulator

**Commented [10]:** Needed another one for the NFA

**Commented [11]:** we largely coded together

The parser will probably be a recursive-descent parser using a regular expression grammar that we'll design to make our lives easier. That's obviously non-optimal from the user perspective, but we're simplifying it because we've never written a parser before. Right now, we're thinking that our parse tree will actually be a tree, where every node will be an operator whose branches will be another operator which is the next operator in the regular expressions, plus one (closure, parenthesis) or two (concatenation, or) other operator-nodes which are the regular expression inputs into the operator. From this, we will be able to determine some clear invariants so that there's no ambiguity as to what tree a regular expression will be parsed into.

**Commented [12]:** Actually, a standard grammar turned out to be optimal from both perspectives

**Commented [13]:** In reality, every node is an instance of the regular expression type holding one (closure, option) or two (concatenation, or) other instances of the regular expression type. Singles hold chars, character classes, or wild cards instead.

The matcher will probably implement a non-deterministic finite automaton (NFA), mostly because we feel like we understand them and how to use them better than we do DFA's and because they let us avoid the issue of greediness in our implementation. We're going to use abstract data types to create a special data structure to hold the NFA that we'll build by walking the parse tree, but we don't yet know what type of data structure is best for holding one. Then, after we've written the NFA, we're going to have to find a way to emulate it using the data structure in order to actually match the string. Overall, we're not entirely sure how we're going to write a program that builds an NFA yet, but we have a lot of resources available and will research into it.

**Commented [14]:** A special abstract data type that's basically a directed graph node with pointers to two other directed graph nodes.

Our language of choice is OCaml. That's because OCaml has algebraic data types, which are great from the parser perspective because they make it easy for us to make our parse tree whatever we want. It's also recursion-optimized, which is (we think) ideal for an NFA because recursion makes it relatively easy (or at least intuitive) to explore all possible state-sequences and find the ones that match. And easy recursion makes a recursive-descent parser easier to write. Immutability and (nearly, other than I/O) functional purity are fine because we can't anticipate any need for side effects (again, other than I/O), so working in a language that (mostly) has none just means less ways our code can break. OCaml just seems like the perfect language for writing languages.

**Commented [15]:** Graphs required pointers, and the emulator made clever use of side effects to check in constant time if a node had already been visited

**Commented [16]:** In hindsight, yes, OCaml was the right choice

Next steps:

We have decided to use OCaml for our project (as explained above), and will define an algebraic data type for parse trees. From here, we will set up the modules for the parser and the matcher. Our most immediate next steps are to conduct more research about regular expressions and finite state automata, and to determine what syntax we want to use for our regular expressions. (Our example in "Brief Overview" used Python syntax, but we will define our own simpler syntax for this project.) We will consult various resources such as the *Algorithms, 4th edition* textbook by Sedgwick and Wayne. We will also set up GitHub for version control and collaboration.