Regular Expressions

Peter Kraft, Lilly Shen, Kimberley Yu, Kat Zhou

Overview:
We implemented regular expression matching in the command line by emulating a non-deterministic finite automaton.

Demo Video:
Our video can be found here.

Links to Specs:

- Draft Spec
- Final Spec

Instructions:

1. Unzip the file
2. Compile using `make`
3. Run in the command line using the following format:
   `./Regex.native “regular expression” “input”`
   Add a `-dn` or `-dp` flag for DOT code describing the NFA or parse tree respectively
4. Our grammar for regular expressions (BNF, wildcards are periods):
   - <re> ::= <re1> | <re1> “|” <re>
   - <re1> ::= <re2> | <re2> “*”
   - <re2> ::= <re3> | <re3> “?”
   - <re3> ::= <atom> | <atom> <re3>
   - <atom> ::= “(“<re>”)” | <schar>
   - <schar> ::= char | Wild | [char, char]

Planning and Process Report:
Our draft specification and final specification are in the main project folder. See the next section for specific details on implementation. Our actual project actually went mainly as planned. In both specifications, we outlined the structure of our project, and we basically stuck to that structure. We ended up running into unexpected issues with data structures that were not properly exposed, so we had to create special modules for core data structures. We also wrote new exposed functions for DOT code generation. However, other than that, we did not modify the original signatures.

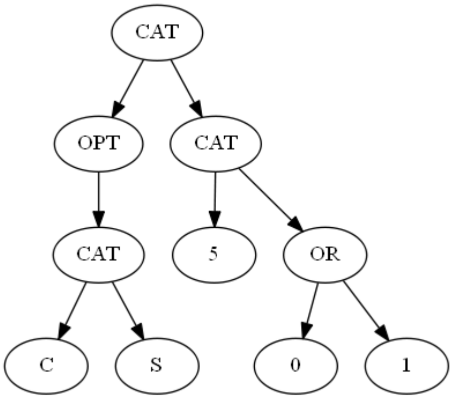
We think that the amount of thought we put into detailed planning helped the process go a lot smoother. Because we had a solid initial plan, we always had a clear idea of where we were going. Our early splitting of the project into distinct modules with pre-defined interfaces helped us keep track of all the moving parts of the project, which helped a lot by letting us organize and compartmentalize our work. We are also very glad we used OCaml, not only because it was familiar, but because its strengths as a language (recursion, abstract data types, modules etc) provided highly intuitive solutions to our problems.

In general, we set realistic goals for ourselves, and kept to the timeline that we mentioned in the final specification. We ended up being just a little bit ahead, which allowed us to also implement wildcards, escape sequences, character classes, and the '?' (option) operator, as well as special functions that printed DOT code that allows visualization of our NFA's and parse trees. If we had additional time, we might have added more features, such as additional operators, substring matching, or other functions found in common regular expression implementations (like Python's *re* module).

Design and Implementation:

- Regex
  - We learned how to create a command-line interface so that users could use Regex directly from the terminal. We added some flags that activated the DOT code generators.

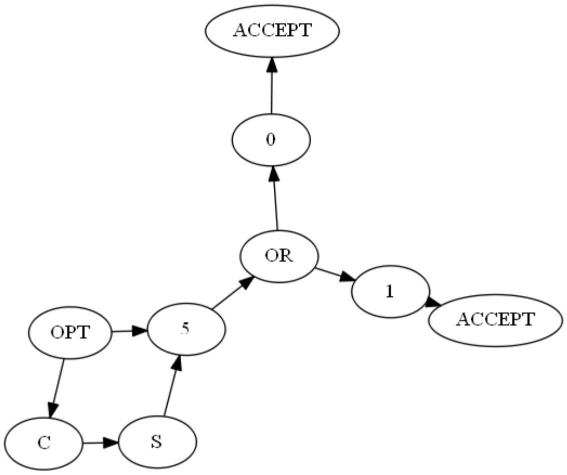
- Parser
  - Based on the grammar we defined, we wrote a recursive-descent parser, using a top-down approach to build a parse tree. Parse trees were implemented using the same algebraic data type described in the specs. An example parse tree corresponding to regular expression "(CS)?5(0|1)" is to the right.
  - We didn't need to have the helper functions that we mentioned in the final specification. We did, however, need many unexpected helper functions, mostly to make sure input regular expressions were valid. We ended up moving the tree type into its own module, which ensured it was properly exposed.



  - We added a tokenizer to help us better process inputted expressions. This allowed us to escape (using the tilde, ~) our regular expression operators so they could be inputted as characters. We also wrote a post-processor for the tokenizer that created a special type of character token for character classes that massively simplified parsing.
  - We also wrote a function that prints the DOT code that describes the parse tree of an input regex. The above tree was created with this function.

- Nfa

- This went as planned, although our initial plan was rather vague. We ended up implementing graphs as a collection of abstract data type structures linked by pointers. The result was a directed graph, where every node contains pointers to one or two other nodes. Thompson's algorithm was used to recursively build the NFA graph from the bottom level of the parse tree up by making small NFA's and recursively combining them. An example of the NFA for regular expression "(CS)?5(0|1)" is to the right.



- We also wrote another function that prints the DOT code that describes the NFA corresponding to an input regex. The above graph was created with this function.

- Emulator
  - Despite all warnings, we fell into the backtracking trap and stayed in it for a while. It led to pretty elegant-seeming code, but exponential asymptotic complexity is bad. Eventually we tested some pathological cases and discovered how our code performed very inefficiently in these cases. We then completely rewrote it using ideas from Russ Cox.
  - The resulting code works by taking a starting state, building a list of states that are one epsilon-transition away from the starting state (that is, a list of all states the emulated NFA could be in), making sure all those states are unique (this is the crucial step because the total number of possible existing states is superlinear, so if all states are unique the function should not have exponential growth), checking each state to see if it matches the first character of the input string, and then recursively doing the same thing with the other *n* characters of the input string and a list of all unique matching states *n* epsilon-transitions away from the starting state. When the last character in the input string is reached, the emulator indicates a match if any of the states it is in are accept states. We used an int ref tag in all singles in the NFA data structure to ensure the non-redundancy step could be done in constant time.

Reflection:

Overall, we're pretty happy with how the project turned out. We all got together and put in work to make good progress on our project, and we made sure to meet regularly enough that we would actually get work done and be in a good place for all checkpoints. One thing that might have made our project even more efficient (though it went pretty smoothly) would be if we actually split up tasks over the various project parts we had. The point of abstraction is that you can work on separate parts of a project and be able to put it together at the end. We tended to all work on one computer and contribute, helping fix problems that came up, and thinking of how we would implement our functions. Most of our code was written that way, but we usually worked independently on less fundamental extensions like escape sequences or character classes.

Advice for Future Students:

Make sure to start early and set goals for yourself along the way. Like anything in CS, things that you think are simple to implement end up taking far longer than you expect due to unforeseen bugs. Pick a topic that everyone finds interesting and engaging. It’s much easier to motivate yourself if you have a genuine curiosity in the project. Find a group of people that you can work well with and don’t be afraid to speak your mind. At the end of the day, sounding “dumb” is far less significant of an issue than not understanding what’s going on.

Coding specific tips: Even though you aren’t “required” to show testing, you will end up doing it anyway, so write helpful testing functions. They will save you a lot of hassle at the end of the day. Also, try to find a good workspace. We hooked up one of our laptops to a TV and coded together.