Lecture 6: Regular Expressions

built on 2021/01/16 at 16:32:00

If you've used `grep`, `awk`, etc., you probably have seen weird-looking patterns that are nonetheless useful. Examples include `^[ \t]+` which somehow looks for white spaces at the beginning of a line. This is an example of (extended) regular expressions. They have seen applications not only in these bread-and-butter *NIX applications but also in your favorite programming languages (Python, Java, C#, Ruby, among others). They are an indispensable tool for string processing.

In this lecture, we'll look at some practical use of regular expressions (regex) and connect the dots between regex and regular languages.

1 Regular Expressions

When we write code, we form an expression by putting together (simple) bits and pieces, leading to a well-formed statement that expresses what we intend to write. In the same way, there is a tool for putting together languages, called *regular expressions* (regex). An example is

$$a(a+b)^*$$

What this encodes is a language. This is the language that starts with an a followed by any number of as and bs.

How do we decode this? We just have to break down this expression into its parts.

- First, there is the symbol a; this represents the language $\{a\}$—the language containing only the string "a".
- Consider also $(a+b)$; this says a or b, or more precisely the language $\{a, b\}$. The star operator, as we have seen before, says to take any number of strings from this language. Hence, $(a+b)^*$ expresses the language $\{a, b\}^*$.
- Then, when we write a next to $(a+b)^*$, we mean the language expressed by a concatenated with the language expressed by $(a+b)^*$. Hence, it represents the language $\{a\} \cdot \{a, b\}^*$.

Let's look at another regular expression. Say $\Sigma = \{a, b\}$. What does the regular expression $(a\Sigma^*) + (\Sigma^* b)$ represent? Once again, we can break this down into its parts: The expression takes the form A or B. The former part says it has to start with an a followed by anything (Σ^*). The latter part says it has to end with a b. (Before that, it could be anything.)

Precedence: We have PEMDAS for arithmetic. We have some precedence order for regex as well: the star operation (think of it as exponentiation) is done first. Then, concatenation (think of it as multiply). Then, plus.

1.1 Regular Expression More Formally

We define regular expressions more formally:

Definition 1.1 *An expression R is a regular expression if R is*

- *σ for some symbol $\sigma \in \Sigma$,*
- *ε (denoting the empty string),*
- *$\emptyset$ (denoting the empty language),*
- *$R_1 + R_2$, where R_1 and R_2 are both regular expressions,*
- *$R_1 R_2$, where R_1 and R_2 are both regular expressions, or*
- *R_1^*, where R_1 is regular expression.*

The definition of regular expressions are seemingly circular. It is not! We're always defining a regular expression in terms of smaller regular expressions. Hence, this is your familiar inductive/recursive definition.

Other Fun Things: We are a lazy bunch. We write R^+ to mean RR^* (that is, R one or more times). We also write R^k to mean repeat R exactly k times.

To be clear, in the same way that we have a machine M vs. the language that M recognizes $L(M)$, we'll write R for a regular expression and $L(R)$ for the language corresponding to R.

Examples: What language does each of the following represent? (Examples from Sipser) Let $\Sigma = \{0, 1\}$.

1. 0^*10^*. A: contains a single 1
2. $\Sigma^*1\Sigma^*$ A: contains at least one 1
3. $\Sigma^*001\Sigma^*$ A: contains the substring 001.
4. $1^*(01^+)^*$ A: every 0 is followed by at least one 1.
5. $(\Sigma\Sigma)^*$ A: string that has even length.
6. $01 + 10$ A: $\{01, 10\}$
7. $0\Sigma^*0 + 1\Sigma^*1 + 0 + 1$ A: starts and ends with the same symbol.
8. $(0 + \varepsilon)(1 + \varepsilon) = \{\varepsilon, 0, 1, 01\}$.
9. $1^*\varnothing$ A: empty
10. $\varnothing^* = \{\varepsilon\}$

A few things are clear from these examples:

- $R + \varnothing = R$ and $R\varepsilon = R$
- BUT: $R + \varepsilon$ may or may not be equal to R. Example: $R = 0$, then $R + \varepsilon = \{0, \varepsilon\}$.
- Also, $R\varnothing$ may or may not be equal to R. Example: $R = 1$, then $R\varnothing = \varnothing$.

Example: How do recognize a numerical constant? For instance, 14, 3.14159, −2.5.

$$(S)(D^+ + D^+.D^+)$$

where $S = \{+, -, \varepsilon\}$ and $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

2 Equivalence With Finite Automata

As we'll see, regular expressions and finite automata are equivalent in their expressiveness. This means precisely that regular expressions can express any regular language, and vice versa. Despite their superficial differences, any regular expression can be converted into a corresponding finite automaton, and any DFA/NFA can be converted into a corresponding regular expression. More precisely, we'll attempt to prove (rather sketch a proof of) the following theorem:

Theorem 2.1 *A language L is regular if and only if some regular expression describes it.*

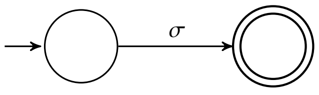
Because the theorem has two implication directions, we'll state and prove two separate lemmas.

Lemma 2.2 *For every regular expression R, there is a DFA that recognizes the language $L(R)$.*

Proof idea: It is sufficient for us to build an NFA that recognizes this language. The theorem we proved previously will take care of converting it into a DFA.

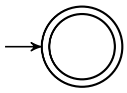
Proof: (Sketch) The proof will be inductive, following the *six* ways a regular expression can be constructed[1].

- $R = \sigma$: Build an NFA that accepts one symbol $\sigma \in \Sigma$.

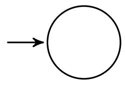


- $R = \varepsilon$: Build an NFA that accepts the empty string.

[1]Some people call this structural induction.



- $R = \varnothing$: Build an NFA that accepts nothing.

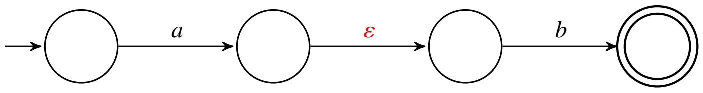


- You know how to do the rest from previous lectures.

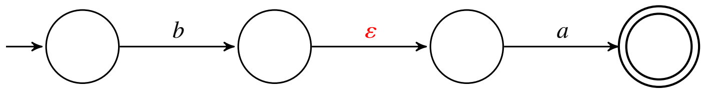
∎

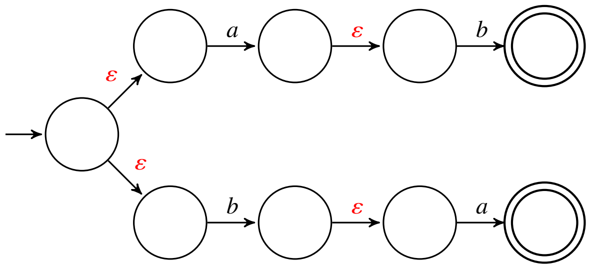
Let's use this knowledge to convert $(ab + ba)^*$ to an NFA (and we'll defer the conversion to DFA to you in your copious spare time).

- First, ab:

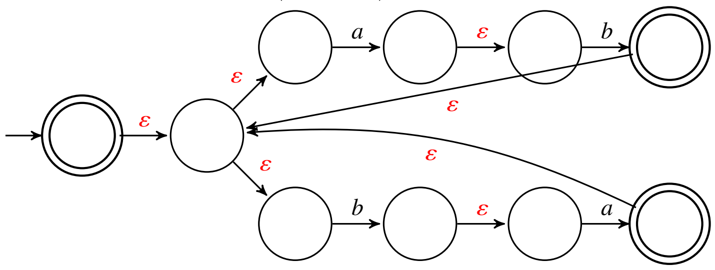


- Then, ba:





- Then $(ab + ba)$:
- Then, the star operation $(ab + ba)^*$:



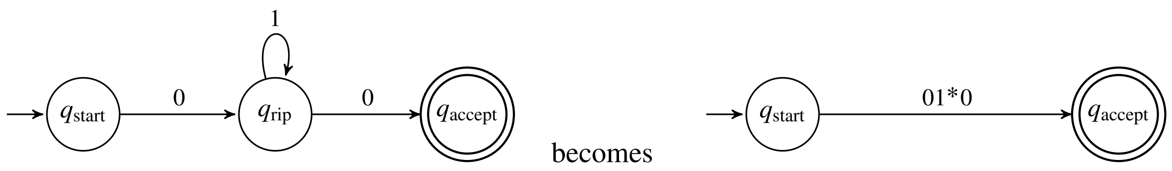
2.1 The Other Direction

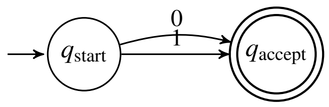
Lemma 2.3 *For every DFA M, there is a regular expression R such that $L(R) = L(M)$.*

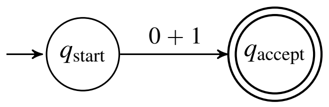
Proof idea: 1. Add two special states: q_{start} and q_{accept} to unify starting and ending states. 2. We'll repeatedly remove states one by one, replacing symbols on the arrows with regular expressions as we go along. That is,

```
while (machine has > 2 states)
  1. pick q_rip different from q_start and q_accept
  2. rip it out and "shortcut" the arrows incident on it with
     regex-labeled arrows
```

To get an idea for what we have to do, consider the following NFA, which has been augmented with q_{start} and q_{accept}.

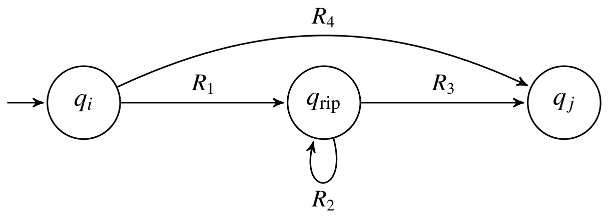


Another example: 

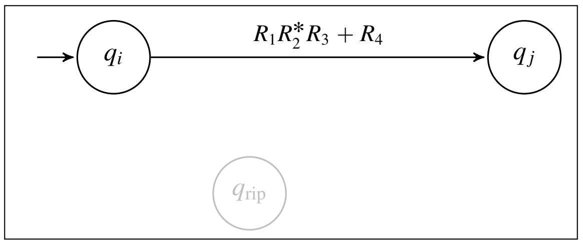
becomes 

So after removing q_{rip}, how should we alter the regular-expression labels to repair the machine? The new labels should compensate for the removal of q_{rip} by adding back in the computation that was removed.

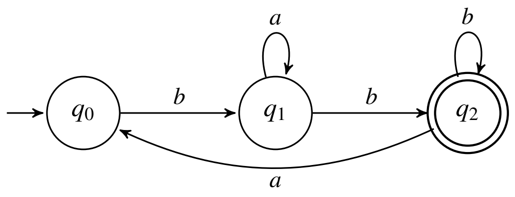
More generally: say we're removing q_{rip}. There may be states q_i and q_j where there's an edge from $q_i \to q_{\text{rip}} \to q_j$. (If there are multiple pairs, this applies to all of them.) Pictorially, the situation looks as follows:



After removing q_{rip}, the regular expressions R_1, R_2, R_3, R_4 should be "fused" somehow to reflect the absence of the ripped node. Guided by our examples above, we can do the following:

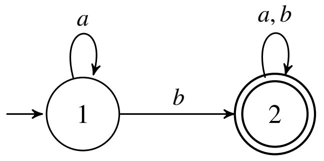


Work In Action: Equipped with this knowledge, let us derive a regex for the following machine (on the alphabet $\Sigma = \{a, b\}$):

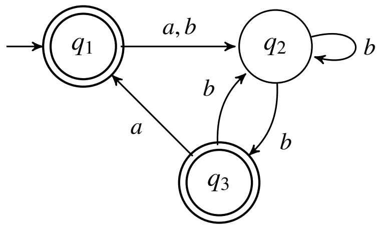


3 Practice

1. Write a regex that accepts all strings that start and end with the same symbol (on the alphabet $\Sigma = \{a, b\}$).
2. Convert $(0+1)^*000(0+1)^*$ to an NFA.
3. Convert the following DFA to a regex:



4. Convert NFA to regex:


q_1
a, b
q_2
b
b
a
b
q_3