Lecture 03: Regular Expressions

Ryan Bernstein

Computer Science 311
Spring 2016

1 Introductory Remarks

- Homework 0 is due today
- Homework 1 is available or will be available tonight. It's due next Thursday?

1.1 Homework Solutions

1.2 Recapitulation

On Tuesday, we introduced the nondeterministic finite automaton. We considered NFAs as "shorthand" machines that can be used to describe regular languages. NFAs look very similar to DFAs, but they have a couple of corners that they can cut to make their creation easier. These are:

Undefined Transitions

NFAs need not have a fully-specified transition function. It's valid to have undefined entries in our state/input transition table.

Multiply-defined Transitions

On the other hand, it's also legal for NFAs to have multiply-defined transitions. In an NFA, our computational context — the state of our computation at any point during the processing of an input string — is a *set* of current states, rather than a DFA's single current state.

When we see a state/input pair for which multiple transitions are defined, we add all of these new states to the set that denotes the next context.

ϵ-transitions

It's also legal for an NFA to change state without taking a character from the input stream. The ϵ-closure of a state is a set composed of the state in question and any state that can be reached from it using only ϵ-transitions. When we enter this state, we immediately add every element of its ϵ-closure to our computational context.

Since we can convert any NFA to a DFA and vice versa, drawing an NFA is sufficient to prove the existence of an equivalent DFA, thus proving regularity of the language of the machine.

We also saw the benefits of using NFAs as shorthand. Once we'd proven their equivalence with DFAs, we were able to use nondeterminism to show that regular languages were closed under union, concatenation, and Kleene star. Nondeterminism allowed us to (very informally) prove all of these closures in the last ten minutes of class.

1.3 For Today

If there's one concept in this class that you're likely to use every day (or at least every week), it's probably the one we'll introduce today. Today we'll be talking about regular expressions.

This lecture is divided into two parts. First, we'll discuss regular expressions in computability theory, and then we'll discuss how regular expressions are used in industry.

2 Regular Expressions in Computability Theory

We introduced NFAs as shorthand for DFAs, but creating one is still a pretty similar amount of work. A state diagram of an NFA looks an awful lot like a state diagram of an NFA, and formally specifying the behavior of an NFA might actually be *more* work in some cases.

We'll use regular expressions as an even *shorter* shorthand to describe regular languages. Regular expressions are a way of succinctly representing a language as a single string of characters. We'll first provide a definition of regular expressions, and then show that regular expressions represent regular languages by proving their equivalence to DFAs.

Regular expressions are strings that define languages. This means that syntactically, we write them as a slightly strange combination of set notation and string notation.

2.1 A Recursive Definition of Regular Expressions

A regular expression is one of the following:

- The empty set $\emptyset$
- a for some single character a
- ϵ
- $(R_1) \cup (R_2)$, where R_1 and R_2 are other, smaller regular expressions
- $(R_1) \circ (R_2)$, where R_1 and R_2 are other, smaller regular expressions
- R^*, where R is another, smaller regular language

If a string s is an element of the language described by a regex R, we say that *s matches R*.

2.2 Examples of Regular Expressions

We'll now look at some examples of regular expressions. In an exam/homework context, any questions on regular expressions will use the more "practical" syntax introduced in the second part of the lecture, so we'll only look at a couple of small examples here.

2.2.1 Example 1: Strings ending in 00

Our expression should end in $R \circ 0 \circ 0$. We need R to match anything, including the empty string. While we do have the Kleene star, we have no expression for Σ.

Fortunately, we can create Σ as the union of all of its elements. Since we're using binary, Σ is presumably $\{0, 1\}$, so this is pretty simple.

$$(0 \cup 1)^* \circ 0 \circ 0$$

2.2.2 Example 2: Strings containing at least three ones

We haven't provided any method of "counting" yet. It's pretty simple, though — if we want at least three ones, we can simply put those three ones in our expression with Σ^* around and between them:

$$(0 \cup 1)^* \circ 1 \circ (0 \cup 1)^* \circ 1 \circ (0 \cup 1)^* \circ 1 \circ (0 \cup 1)^*$$

If we wanted *exactly* three ones, we would simply prevent ones from appearing in the star expressions, like so:

$$0^* \circ 1 \circ 0^* \circ 1 \circ 0^* \circ 1 \circ 0^*$$

Building a range (e.g. 3-5 ones) can be thought of as the union of three subexpressions that match strings with three, four, and five ones. This is pretty long, though, so I'm not going to write it out.

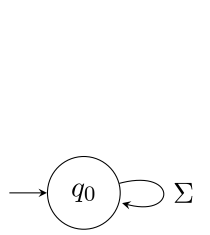
2.3 Proof of Equivalence with DFAs

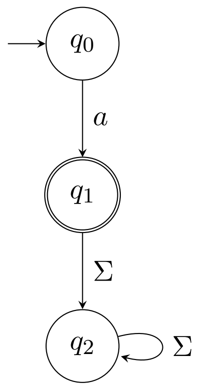
We know that regular expressions define some languages. To prove that they're used to describe *regular* languages, though, we'll have to do exactly what we did with NFAs: prove their equivalence by showing how to transform them to a DFA and vice versa.

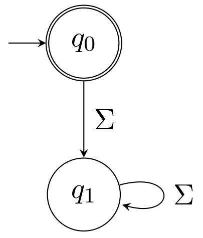
2.3.1 Converting a Regular Expression to a Finite Automaton

To show how to build a finite automaton out of a regular expression, we can simply iterate over the items in our recursive definition and show how to build a finite automaton for every possibility.

Building DFAs that decide the first three possibilities — $\emptyset$, a, and ϵ — is pretty trivial:







As for the final three cases — union, concatenation, and Kleene star of other regular expressions — we actually already proved that regular languages are closed under these operations at the end of the last lecture.

We have therefore shown that we can build a finite automaton equivalent to any regular expression.

2.3.2 Converting a Finite Automaton to a Regular Expression

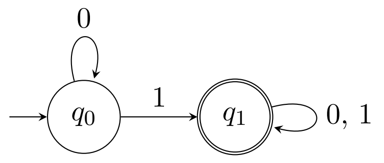
Generalized NFAs Converting a DFA to a regular expression requires us to introduce a new type of machine, called a generalized nondeterministic finite automaton or GNFA.

A generalized NFA has exactly one start state and one accept state. It has the following properties:

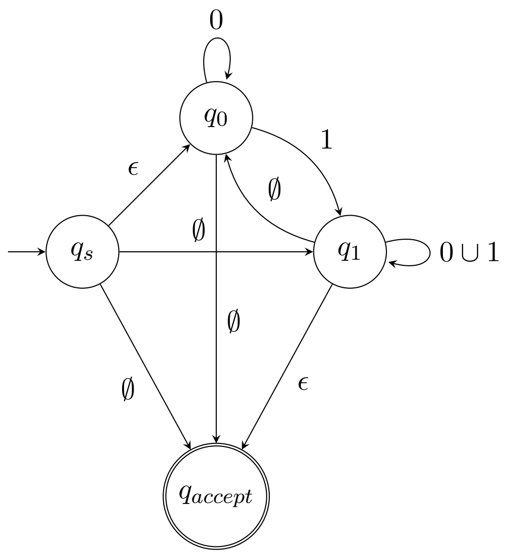
- The start state has a transition defined to every other state in the machine
- The accept state has a transition defined *from* every other state in the machine
- Every state in $Q-\{q_{start}, q_{accept}\}$ has a transition defined to every state in $Q-\{q_{start}, q_{accept}\}$ including itself

Instead of single characters, the transitions in a GNFA are labeled with regular expressions. Remember that we can construct a DFA for every possible regular expression. This means that we can think of each transition in a GNFA as having a machine that decides the language of that regular expression hidden inside it.

As an example, consider the following DFA, which decides $\{s \in \{0,1\}^* \mid s \text{ contains a one}\}$:



We can construct a GNFA for this language that looks like this:



What was the process that we used to construct this? We can formalize it as follows:

- The new start state has an ϵ-transition to the old start state. It has $\emptyset$-transitions to everything else.
- Any state that was accepting in the DFA has an ϵ-transition to the accept state. Any state that was *not* accepting in the DFA has an $\emptyset$-transition to the accept state
- Any path between two nodes that did not exist in the DFA is an $\emptyset$-transition
- Any transition that was defined for one state with multiple input characters in the DFA is now defined with the union of those characters

Formally, a GNFA is a 5-tuple $(Q, \Sigma, \delta, q_{start}, q_{accept})$. Most of this should be self-explanatory by now. δ is the only thing that's a bit different. Now, δ is a function that takes a pair of states and yields the regular expression defined for the transition between them. We know that the first state will never be the accept state and the second state will never be the start state, so we can say:

$$\delta : (Q - \{q_{accept}\}) \times (Q - q_{start}) \rightarrow R$$

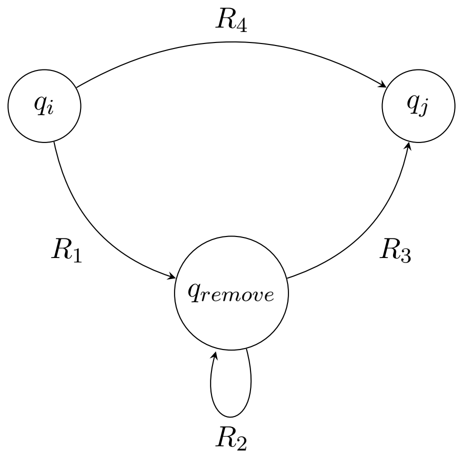
Reducing GNFAs Once we've converted our DFA to a GNFA, we can reduce that GNFA using the following algorithm. Reducing a GNFA involves removing states and combining transitions. When we've fully reduced a GNFA, it will contain *only* the start and accept states. The single transition between them will contain a regular expression that describes the entire language.

We reduce the GNFA G one state at a time and then convert it to a regular expression using the following recursive function, which we'll call $convert()$:

- Let $k = |Q|$
- If $k = 2$, then G is just a start state and accept state with a single transition between them labeled R
  - Return R
- If $k > 2$:
  - Select a state $q_{remove} \in (Q - \{q_{start}, q_{accept}\})$

- Construct a GNFA $G' = (Q', \Sigma, \delta', q_{start}, q_{accept})$ such that:
  - $Q' = Q - \{q_{remove}\}$
  - For all pairs $q_i \in (Q' - \{q_{accept}\})$ and $q_j \in (Q' - \{q_{start}\})$, $\delta'(q_i, q_j) = ((R_1) \circ (R_2)^* \circ (R_3)) \cup (R_4)$, where:
    - $R_1 = \delta(q_i, q_{remove})$
    - $R_2 = \delta(q_{remove}, q_{remove})$
    - $R_3 = \delta(q_{remove}, q_j)$
    - $R_4 = \delta(q_i, q_j)$
- Return $convert(G')$

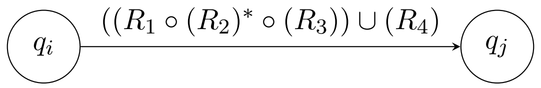
Why does this work? For any q_i, q_j, and q_{remove}, our GNFA contains a subgraph that looks something like this:



Looking at this diagram, we can see that there are two paths from q_i to q_j. One is the direct route, annotated with R_4. The second path is the indirect route that passes through q_{remove}. To take this route, we'll have to see a string of the form $w_1w_2w_3$ such that:

1. w_1 matches R_1 to get us to q_{remove}
2. w_2 matches $(R_2)^*$, which takes us around the loop zero or more times
3. w_3 matches R_3, which gets us from q_{remove} to q_j.

Thus the full string $w_1w_2w_3$ should match $(R_1) \circ (R_2)^* \circ (R_3)$. Since we could take either route, we take the union of this and R_4. We can then remove q_{remove}, leaving us with the following direct route:



Note that if R_1 or R_3 are $\emptyset$, this is equivalent to just R_4. If $R_4 = \emptyset$, this is equivalent to just $(R_1 \circ (R_2)^* \circ (R_3)$. If *both* are $\emptyset$, the result is still $\emptyset$.

Since we can use this method to create a regular expression from any DFA and we can create a finite automaton equivalent to any regular language, regular expressions and finite automata are equivalent in power.

Sometimes, it will be easier to describe a language with a regular expression. This is often true when we're building a language around some immediately obvious visual property of the strings inside it (e.g. $\{s \in \{a,b\}^* \mid s \text{ ends in } bab\}$). Sometimes, though, we're concerned with languages such as binary multiples of three, where our states have some associated meaning. In these cases, it's often easier to build a state machine that decides the language.

2.4 A New (actually useful) Definition of Regular Languages

Since we've shown that regular expressions and DFAs are equivalent in power, we can use the definition of a regular expression to create a new definition of regular languages. This is the first definition that we've seen that actually tells us something about what a regular language looks like, rather than stating the existence of some computational structure.

The Recursive Definition of a Regular Language A regular language is one of the following:

- The empty set $\emptyset$
- The singleton language $\{a\}$, where a is a string composed of a single character from some alphabet Σ
- The singleton language $\{\epsilon\}$, which contains only the empty string
- $A \cup B$, where A and B are other, smaller regular languages
- $A \circ B$, where A and B are other, smaller regular languages
- A^*, where A is another, smaller regular language.

This is a minimal set of possibilities from which we can create every regular language. It's because of their use in this definition that union, concatenation, and Kleene star are known as the *regular operations.*

3 Regular Expressions in the Real World

In industry, regular expressions are an example of a domain-specific language. We pass them into programs such as grep when we're trying to search for strings that match some pattern. Often, this functionality is implemented by constructing a DFA or NFA from the regex in question and feeding the input strings through it. Many modern programming languages also now offer language support for regular expression matching.

3.1 Regular Expression Syntax

The syntax that we introduced in the first half of the lecture works fine when writing regular expressions on paper, but when was the last time you used the concatenation key on your computer? In industry, we have a more keyboard-friendly syntax that also includes some "syntactic sugar" for commonly used compositions. While many of these shortcuts produce behavior not specified by our minimal definition above, all of these constructs can be made by composing elements of that definition.

That syntactic sugar means two things:

- We're about to introduce a lot of ways to solve the same problem
- You *do not* have to remember all of these shortcuts. As long as you remember one way of accomplishing what you're trying to do — or, say, a minimal set of operations from which you could construct any of these methods — you should be able to construct an equally correct regular expression

3.1.1 Concatenation, Association, and General Structure

Regex are strings of printable (and typeable) characters. There is no concatenation operator — a regex that matches the string "abc" is simply written `abc`.

This can introduce some ambiguity when we start to add new operators. We can disambiguate regular expressions manually by adding parentheses. There is a defined order of operations, but I'm guessing that adding parens is easier than trying to memorize it.

3.1.2 Alternatives

What we previously called unions we now call *alternatives.* The union of two smaller regular expressions is now written with a vertical pipe between them. `ab | cd` matches either the string "ab" or the string "cd". It's easy to see where the ambiguity we just discussed comes in. If I want to match strings that are an "a", followed by either a "b" or a "c", followed by a "d", I can just add some parentheses and write `a(b|c)d`.

This quickly gets tedious if we have a large number of possibilities. If we wanted to recognize some digit, for example, we'd be writing something like `(0|1|2|3|4|5|6|7|8|9|)`. To address this, we have a shorthand notation: a series of characters between square brackets matches any one of those characters. We could thus instead write `[0123456789]`.

Even this is kind of a lot of work, though. We can also match any characters in some range using the succinct expression `[0-9]` or `[a-z]`. Ranges are based on ASCII ordinals.

What if we want to match *anything but* some set of characters? If a bracketed range begins with a caret, it matches any character *not* found in those brackets. We can therefore match any non-numeric character using the expression `[^0-9]`.

3.1.3 Repetitions

Often, we'll want to repeat some subexpression. This is equivalent to having a loop in our state machine.

Since it *is* available on the keyboard, the Kleene star has made it into our concrete syntax. Tagging a subexpression will match zero or more repetitions of that expression. There are some other, more restrictive cases of repetition that we can match as well, though:

- Tagging a subexpression with a question mark will match zero or one occurrences of strings that expression, effectively making it "optional"
- Tagging a subexpression with a plus is shorthand for "one or more repetitions"

There are also cases where we want to bound the number of repetitions somehow. We can specify bounds using curly braces.

- `a{3}` matches exactly three "a"s
- `a{3,5}` matches three, four, or five "a"s
- We can also omit the upper bound — `a{3,}` matches three or more "a"s

3.1.4 Shortcuts for Character Sets

We have the ability to pick from any set of characters using the strategies for alternatives described above. However, we often find ourselves rewriting the same sets. As a result, regular expressions include support for commonly-used sets, including:

- `\d`, which matches any digit (equivalent to `[0-9]`)
- `\s`, which matches any whitespace character
- `.`, which is a wildcard that matches *any* character.

3.1.5 Line/String Positioning

In industry, regular expressions are sometimes used for partial matches. Given a long string of text, this would be used to select *portions* of the string that matched some expression. This is often if we're attempting to, say, pull numbers out of some file or program output.

This makes it difficult to decide some of the languages that we've been discussing though — particularly, those of the form "s starts with" or "s ends with".

To accommodate these cases, we have a couple of operators that define where a match occurs.

- Expressions that begin with `^` match only things that start at the beginning of the line
- Expressions that end with `$` match only things that appear at the end of the string

3.2 Testing Regular Expressions

We've introduced a lot of rules here, and these can be difficult to remember. Fortunately, people have been using these things for a long time, so there are tools that can help us ensure that our regexes are (at least mostly) correct. I recommend `https://regex101.com` in particular to assist with debugging regular expressions on homework assignments or programming projects.