Takeaway Notes: Regular Sets/Languages/Expressions

Contents

1 Introduction

This writeup transitions our exploration of this rudimentary level of computation from *automata* (machines) to *languages*. The two have been linked from the beginning, but now we dive more deeply into the language aspect.

2 Key Concepts

As you've seen before, this family of languages has a name: the *regular* langauges or sets. It's time to put concrete meaning on what exactly "regular" is. Interestingly, one of the most foundational ideas in this regard is used more for identifying what is *not* regular. As a bonus, it has one of the more memorable names in theoretical computer science.

2.1 The Pumping Lemma

The *pumping lemma*, as the name implies, is a proven theorem whose primary function is to prove other theorems. The lemma itself states that "if a language is regular, then it has certain qualities." This takes the classic logical form $p \rightarrow q$. However, it is primarily used in its *contrapositive* form, $\neg q \rightarrow \neg p$: "if a language does *not* have certain qualities, then it is *not* regular." Note also that, also per traditional symbolic logic rules, the pumping lemma cannot be used to show that a language *is* regular: $q \rightarrow p$ does not necessary hold.

We need to go formal at this point: Given that L is a regular language, $\exists$ a constant n such that $\forall$ strings w in L with $|w| \geq n$, w can be broken up into three strings $w = xyz$ where these conditions hold:

1. $y \neq \epsilon$ (or, $|y| > 0$)
2. $|xy| \leq n$
3. $\forall\ k \geq 0, xy^k z \in L$

The third item is the critical condition, and where "pumping" gets into the name of the lemma: a regular language *always* has some string that has a substring within it which, if repeated ("pumped") any number of times, still results in a string that is in L.

Some insight that correlates the pumping lemma back to finite state machines: precisely because we have a *finite* number of states in the machine, say n states, in order to recognize a string of more than n symbols, then we must have visited at least one state more than once. This means there is a loop somewhere in the machine, and if there is a loop, then we can identify a substring that will loop infinitely. This is the intuition that drives the proof of the lemma.

The pumping lemma (or more specifically, its contrapositive) allows us to definitively say that the following languages are not regular, and thus lets us claim with complete confidence that no finite state automaton can be defined, deterministic nor nondeterministic, that will recognize their strings. Assume a two-symbol Σ in all cases:

- The language consisting of strings which are concatenation pairs of the same substring (i.e., of the form ww): try 0^n10^n1, where n is the n of the pumping lemma
- Any language consisting of *palindromes* (i.e., words that are the same when spelled backward): try 0^n110^n, where n is the n of the pumping lemma
- The language consisting of the same number of two different symbols concatenated: try 0^n1^n, where n is the n of the pumping lemma
- The language whose strings consist of the same symbol repeated a prime number of times: try 1^p where p is the smallest prime number that is larger than the n in the pumping lemma

If you stare at these examples long enough, you can build up an intuitive notion of the kinds of strings for which no finite state automaton can be defined. Further, the pumping lemma isn't the *only* way to prove that a language isn't regular; don't forget that you have a toolkit of closure rules which state how certain derivations of regular languages are also regular. Closure can then be combined with *proof by contradiction* to show how a language L can't be regular: first, assume that L is regular then derive another language L' via established closed operations; by closure, L' would need to be regular. However, if it can be shown that L' isn't regular (through other means...such as the pumping lemma!), then L could not possibly be regular as well.

2.2 Regular Expressions

Okay, so what *is* a regular language anyway? You asked for it (well, maybe not). Regular languages are sets of strings that can be written as *regular expressions* (REs), and these are written as follows. Given an alphabet Σ:

1. $s \in \Sigma$ is a regular expression
2. ϵ and $\emptyset$ are regular expressions
3. The union of regular expressions E and F, written as $E + F$, is a regular expression

4. The concatenation of regular expressions E and F, written as EF, is a regular expression

5. The *Kleene closure* of E, written as E^*, is a regular expression

…nothing more, nothing less. These are all self-explanatory except for E^*; E^* is the set of all strings that are concatenations of any number of strings represented by E. It's like a "mix and match anything as many times as you like" operation. Trivially, any regular expression E can be enclosed in parentheses, and that too is a regular expression: (E).

When you have operators, you have precedence rules: here, $*$ takes top precedence, followed by concatenation, followed by union. As with other kinds of expressions, parentheses can be used to denote an order that is different from what these precedence rules would have otherwise dictated.

Some examples, all using $\Sigma = \{0, 1\}$:

- $01^* + 10^*$
- $(01)^*$
- $(01)^* + (10)^* + 0(10)^* + 1(01)^*$
- $1(01)^*0$
- $(\epsilon + 1)(01)^*(\epsilon + 0)$
- $01^* + 1$
- $(10 + 0)^*(\epsilon + 1)(01 + 1)^*(\epsilon + 0)$
- $(1 + \epsilon)(00^*1)^*0^*$

We can do this all day…and this is just for a two-symbol alphabet!

2.2.1 A Couple of Shortcuts

Some regular expressions appear so frequently that they have shortcut notation, so that we don't have to write them out fully every time:

- The "Kleene plus" (in quotes because it doesn't really have a formal name): $L^+ = LL^* = L^*L$ (analogous to L^+ as strictly applied to languages as any set of strings)

- The “optional” expression: $L? = \epsilon + L$

The definitions pretty much say it all; not much more to do here except to be aware of these notational shortcuts.

2.2.2 Algebraic Properties of Regular Expressions

When you have sets of objects and operators on those objects, you have an *algebra*. And when you have an algebra, you expect to see algebraic *properties* that describe how the various operators in the algebra relate to each other.

Well, regular expressions constitute a set of objects (with a regular expression serving as notation for a regular language), and union, concatenation, and Kleene closure are operators on those objects. So, we have an algebra and algebraic properties. They are all provable by (a) interpreting regular expressions as the regular languages that they describe, (b) interpreting the operators in terms of the new sets that they create (a language being a set of strings), then finally (c) looking at the strings that belong in those languages to see how they relate to the original operands.

The letter variables below all interchangeably represent regular expressions or the regular languages that they describe:

- Union is commutative: $L + M = M + L$
- Union is associative: $(L + M) + N = L + (M + N)$
- Concatenation is associative: $(LM)N = L(MN)$
- Concatenation is *not* commutative: $LM \neq ML$ —see if you can prove this
- The identity element of union is $\emptyset$: $\emptyset + L = L + \emptyset = L$
- The identity element of concatenation is ϵ: $\epsilon L = L\epsilon = L$
- Concatenation has an *annihilator* element, $\emptyset$: $\emptyset L = L\emptyset = \emptyset$ (analogous to how numeric multiplication has 0 as its annihilator)
- Concatenation is distributive over union, but because it is not commutative, *left distributivity* vs. *right distributivity* are distinct:
  - Left distributivity: $L(M + N) = LM + LN$
  - Right distributivity: $(M + N)L = ML + NL$
  - However $(M + N)L \neq L(M + N)$
- Union is idempotent: $L + L = L$

2.2.3 A Closer Look at Closure

Kleene closure also has a few well-known identities or laws:

- $(L^*)^* = L^*$
- $\emptyset^* = \epsilon$
- $\epsilon^* = \epsilon$
- $L^* = L^+ + \epsilon$

Again, these are all provable by interpreting the regular expressions as regular languages and therefore as sets of strings, then showing that membership in one set is true if and only if membership in the other is also true.

2.2.4 POSIX Regular Expressions

The notation above may seem familiar; in fact, they are the basis of so-called *POSIX regular expressions*, or *regexes* (or *regexen*, the way "oxen" is plural for "ox"). They aren't identical, however. We'll elide details here because there are lots of references out there for these. Suffice it to say that this expression standard is used in many many tools and programming languages to characterize patterns of strings. They are based on formal regular expressions but actually denote more than just regular languages. When you get to know regexen more deeply (likely in other courses or on the job), see if you can identify the operators or symbols that "break" the formal definition of regular expressions, vs. the ones that directly correspond to it.

2.3 Equivalency of REs, NFAs, and DFAs

Similarly to how DFAs can be constructed from NFAs, NFAs can be constructed from REs. Transitively, this shows that all three approaches describe regular languages. Generally speaking, unions are like choosing between alternative states, concatenations are like transitioning to new states along the concatenated symbols, and closures constitute a loop structure. We need the flexibility of NFAs (multiple transitions on the same symbol; option for no transition for a given symbol; ϵ moves) to make this conversion more convenient, and that's just fine because we can readily convert an NFA to a DFA anyway.

2.3.1 DFA to RE

The intuition behind this conversion is to think of regular expressions as "paths" through the equivalent finite state machine that lead to an accepting state. In order to cover every possibility, we determine the regular expressions involved for *every* "journey" from one state to the next. The regular expression that is equivalent to a DFA is then the union of regular expressions whose initial state is the start state of the DFA and whose destination state is an accepting one.

If we let n be the number of states in a DFA, then number the states from 1 to n with 1 as the start state, the building blocks of this construction are as follows. They are all notated as $R_{ij}^{(k)}$, where i and j are the starting and ending states of a "path" and k is the "latest" state that we pass through (as determined by the naming scheme). The final regular expression is the union of all expressions $R_{1j}^{(n)}$ where j is an accepting state.

- $R_{ij}^{(0)} = \emptyset$ if there is no direct transition from i to j
- $R_{ij}^{(0)} = a$ if there is only one symbol a that transitions from i to j
- $R_{ij}^{(0)} = a_1 + a_2 + a_3 + \cdots + a_m$ if the symbols a_1 through a_m all have direct transitions from i to j

From these building blocks, we can move to longer and longer paths (i.e., increasingly higher k) by using this construction:

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$$

Intuitively, this represents the idea that, for paths between states i and j, we have *either* the paths that don't pass through state k at all ($R_{ij}^{(k-1)}$), or, for the paths that do go through k, we can break those up into the path from i to k ($R_{ik}^{(k-1)}$) followed by zero or more loops through k ($(R_{kk}^{(k-1)})^*$), finally followed by the path from k to j ($R_{kj}^{(k-1)}$). Note how the sequencing of subpaths is equivalent to concatenation.

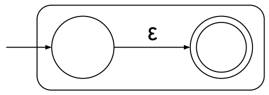
2.3.2 RE to NFA

To complete the equivalency, we need to show that we can go in the other direction, from regular expression to a finite state machine. We go to a nondeterministic machine in this case, but since we already know that we can convert a nondeterministic machine to a deterministic one, then it is sufficient to go to an NFA.

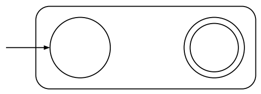
The approach to this conversion is to build "mini-machines" based on the known variations of regular expressions. The "mini-machines" are then composed into the full machine.

First, the basis conversions:

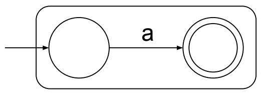
- ϵ: a two-state machine with an ϵ- or λ-move from the start state to an accepting state



- $\emptyset$: a two-state machine with *no* moves from the start state to the accepting state

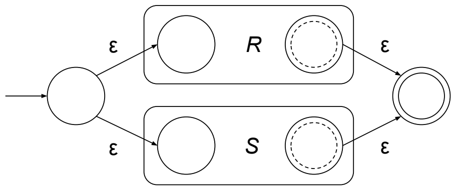


- A single symbol a from the alphabet: a two-state machine with a transition on a from the start state to the accepting state

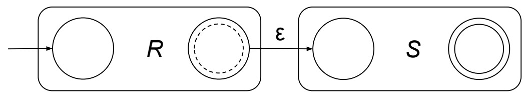


Now, we can compose the other regular expression operations as follows:

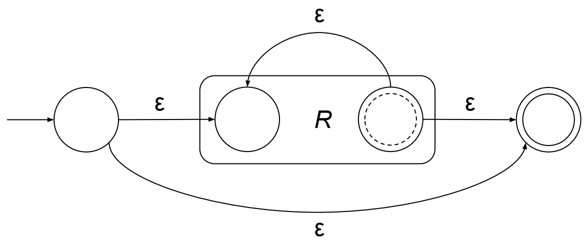
- $R + S$: Assuming black-box machines for R and S, connect a start state to R and S via ϵ-moves, then for every accepting state in R and S, connect all of them to a single new accepting state also via ϵ-moves.



- RS: Assuming black-box machines for R and S, the start state of R becomes the start state of the entire machine. Serially connect accepting states in R to the start state of S via ϵ-moves. The accepting states of S now become the accepting states of the entire RS machine.



- R^*: This one is harder to describe, and is best seen as a diagram. But the general idea is to create a machine with an ϵ move directly from the start to accepting state, with another path that has R in between, connected via ϵ moves from start to R, then from R to the accepting state, then, most importantly, from R back to itself.



- (R): This is the machine for R itself.

With these building blocks, an NFA can then be constructed from any regular expression by mapping the subexpressions within that regular expression into their corresponding submachines.

2.4 Right-Linear Grammars: Three Become Four (just like the Musketeers!)

There is one more approach for describing a regular language that is conceptually different from NFAs, DFAs, or REs: these are *right-linear grammars* (RLGs). Finite state automata take the perspective of *recognition* or *acceptance* of strings in a regular language. Regular expressions also constitute a form of matching, but in a more *declarative* way ("this is how the strings look" rather than "process a string to see if it gets accepted"). Right-linear grammars, and formal grammars in general, take the perspective of *production*: "this is how to build a string for this language."

The intuition for a grammar is straightforward: they consist of *production rules* that describe how to build up a string from ϵ. Each rule changes a string from one to the other, with some rules denoting that we should "stop." The string that we have at that point would then be a string that belongs to the language represented by the grammar.

The Σ that we have known so far becomes known as a set of *terminal* symbols: these are symbols that are "finished" ("terminated") in the sense that they cannot be replaced by a grammar's rules. Instead, we define a new alphabet, V, which constitute *non-terminal* or *variable* symbols. These symbols can be substituted with other symbols by the grammar's production rules.

Formally, then, a right-linear grammar G is a tuple (again!) $G = (V, \Sigma, P, S)$ where:

- V is the non-terminal or variable alphabet of the grammar
- Σ is the terminal alphabet of the grammar
- $V \cap \Sigma = \emptyset$
- $S \in V$ is the *start symbol*
- P is the set of *productions* or *rules*
- A production has the form $X \to wY$ or $X \to w$, where $X, Y \in V$ and $w \in \Sigma^*$

One can produce or generate a string by starting at a rule with S on the left, then substituting S with the expression on the right side of the rule. Every non-terminal in the new string can then be substituted with any rule that has the non-terminal on the left. Keep going until there are no more non-terminals in the string. Voila, you should now have a string that belongs to the grammar's corresponding regular language.

3 Important Skills

For this portion of the course, make sure you can do the following:

- Characterize or recognize strings that match a regular expression
- Construct a regular expression for a given description of some set of strings
- Use the pumping lemma to show that a given language isn't regular
- Use other properties of regular languages (e.g., closure under certain operations) to help show or disprove that a language is regular
- Generate strings by following the rules of a right-linear grammar
- Navigate between DFAs, NFAs, REs, and RLGs: recognize or construct/derive equivalents across each paradigm

4 Big Picture Points

You likely won't retain all of the skills in the list above for the long term, unless of course you end up writing parsers, compilers, or doing theoretical computer science as part of your career. Even without them, do remember or appreciate the following:

- DFAs $\equiv$ NFAs $\equiv$ REs $\equiv$ RLGs: That is, they all represent the exact same family of languages, the so-called *regular set*
- The *pumping lemma* is a very important characterizer of a regular language, and is a go-to tool for showing that a language is *not* regular
- Languages that are not regular represent a higher echelon of language types. These cannot be "handled" by DFAs, NFAs, REs, nor RLGs. Instead, new "powers," abilities, or models of computation are required in other to "reach" these higher levels. We will progress to those later in the course.