COMPUTATION: DAY 3

BURTON ROSENBERG
UNIVERSITY OF MIAMI

Contents

1. Free Algebras with Kleene star

Definition 1.1. Let Σ be a finite set. The *free algebra generate by* Σ are elements of $\mathcal{P}(\Sigma^*)$ and the operations of union and concatenation, with concatenation defined as

$$A \cdot B \;=\; \{\, a \cdot b \,|\, a \in A, b \in B \,\}.$$

Where union is commutative, concatenation is non-commutative. We also include in the algebra the *Kleene star*,

$$A^* = \bigcup_{i \geq 0} A^i.$$

Many usual rules apply. The empty set is the identity for union, and the set containing only the empty string is the identity for concatenation. The distributive

law applies.

$$\begin{aligned} A &= \emptyset \cup A = A \cup \emptyset \\ A &= \varepsilon \cdot A = A \cdot \varepsilon \\ \emptyset &= \emptyset \cdot A = A \cdot \emptyset \\ A \cdot (B \cup C) &= A \cdot B \cup A \cdot C \\ (B \cup C) \cdot A &= B \cdot A \cup C \cdot A \end{aligned}$$

Note that the Kleene star, while an infinite union, has the simplifying property that if $a \in A^*$ then for some i, $a \in A^i$. Compare this to an attempt to define,

$$A^\dagger = \bigcap_{i \geq 0} A^i$$

where $a \in A^\dagger$ would require that $a \in A^i$ for all i.

2. Regular Languages are a Free Algebra with Kleene Star

We need to show that the operations of union, concatenation and Kleene star on regular languages give a regular language. This is proved by constructing and NFA which recognizes the resulting set from DFA's that recognize the component sets.

2.1. **Union.** Given regular languages $A, B \subseteq \Sigma^*$, there exist machines M_A and M_B such that $A = \mathcal{L}(M_A)$ and $B = \mathcal{L}(M_B)$. Relabel states if necessary so that state names are all distinct. Define the machine formed by the union of the states, final states and transition function. Define the a new initial state with ε-moves to the old initial states. Such machine accepts the language $A \cup B$.

2.2. **Concatenation.** Given regular languages $A, B \subseteq \Sigma^*$, there exist machines M_A and M_B such that $A = \mathcal{L}(M_A)$ and $B = \mathcal{L}(M_B)$. Relabel states if necessary so that state names are all distinct. Define the machine formed by the union of the states and transition functions. Add ε-moves from the final states of M_A to the initial state of M_B. Define the final states of the combined machine as just the final states of M_B. The initial state of the combined machine is the initial state of M_A. Such a machines accepts the language $A \cdot B$.

2.3. **Kleene star.** Given regular language $A \subseteq \Sigma^*$, there exist machines M_A such that $A = \mathcal{L}(M_A)$. Build a machine including M_A with a new initial state that is also a final state. Add an ε-move from this state to the old initial state. Add ε-moves from the final states to the old initial state. Such a machine accepts the language A^*.

N.B. The new initial state is necessary. Consider the machine for the empty language which is a single state with loops for all elements of Σ. The star is the set including only the empty string.

3. Regular Expressions

Formulas in the free algebra are called *Regular Expressions*. Parenthesis are used according to the normal precedence rules of exponents, product and addition. Hence,

$$A \cup B \cdot C^* = A \cup (B \cdot (C^*)).$$

Theorem 3.1. A regular expression describes a regular language.

Proof: We have already shown how, beginning with NFA's, the operations required by a regular expression can be implemented. To complete the proof we need to build three more machines.

(1) An NFA that accepts the empty set. A machine with a single state and no final states.
(2) An NFA that accepts only the empty string. A machine with a single state that is a final state.
(3) An NFA that accepts a $\sigma \in \Sigma$. A machine with two states, on initial the other final, with a transition between them on σ.

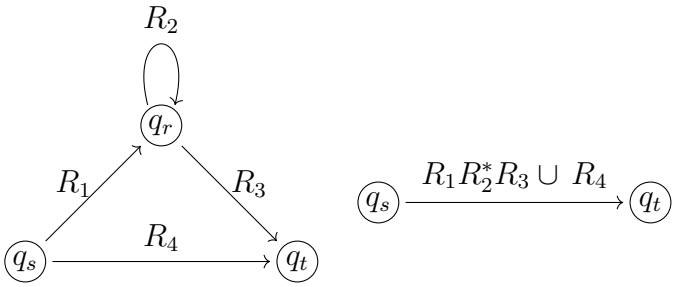
□

Regular expressions are part of many programming languages. The Python language has the `re` package that will use regular expressions to match against a string. In this case, the notation has been generalized from whether the entire string matches the Regular Expression to which substrings of a string match the Regular Expression. The notation is made a bit kinder for programming, with a number of useful extensions.

4. The Generalized NFA

We have so far shown that free algebras, and the Regular Expression formulas written in that algebra, define languages that are accepted by finite automata. However, any set accepted by a finite automata can be described by a Regular Expression. Hence, Regular Languages, Regular Expressions, and the sets of strings recognized by a finite automata are all the same concept.

This has a certain surprise, as intersection and complementation are not directly expressible by Regular Expressions. However, by the equivalence, they are expressible. This often results in a very large, inconvenient regular expression. For instance, while $(aaa)^*$ is the language of all strings of a's of length a multiple of 3, and $(aaaaa)^*$



(A) The path through q_r (B) q_r removed

FIGURE 1. Joining path through q_r into the path q_s to q_t

those a multiple of 5, the expression for strings of a's with length not a multiple of 3 or 5 is,

$$(aaaaaaaaaaaaaaa)^*(a \cdot (\varepsilon \mid a \cdot (\varepsilon \mid aa \cdot (\varepsilon \mid aaa \cdot (\varepsilon \mid a \cdot (\varepsilon \mid aaa \cdot (\varepsilon \mid aa \cdot (\varepsilon \mid a)))))))$$

To show that any language recognized by an NFA can be written as a Regular Expression, we generalize the NFA so that the edges can be regular expressions. To follow an edge with a regular expression is consume the length of input that matches the regular expression. There will also be a rule for a Generalized NFA (GNFA) that there is only one accept state, and that between all states there are transitions, including edges from a state back to itself, with the exceptions that there are no transitions to the start state and no transitions from the final state.

With these preparations we are set to start to remove states, except the initial and final state, according to the formula in Figure 1. To remove q_r, all other nodes q_s and q_t are considered, including $q_r = q_t$, and both the forward and reverse directions between q_s and q_t. Removing q_r would risk loosing paths from q_s to q_t. In order that this does not happen, the regular expression for q_s to q_t via q_r is added to paths directly between q_s and q_t. The operations of regular expressions are sufficient to do this.

This removal continues until only the start and final state remain, with the regular expression on the arrow between them being the regular expression for the language recognized by the GNFA.

5. Non-regular languages

Consider the language

$$L_o = \{\, a^i\, b^i \mid i \geq 1 \,\}$$

It cannot be regular.

Consider a very long string in L_o. Fix a machine M which claims to recognize L_o. It has a certain number of states p. Consider the string $s = a^i\, b^i$ with $i >> p$. The string s in the language L_o. As the machine processes successive a's, it must at some point repeat a state. Then the string s can be written as

$$a^i a^j a^k b^{i+j+k}$$

where after a^i the machine is in state q and returns to state q again after the next portion of the string a^j. In that case, without changing the outcome, we can remove the a^j, and therefore,

$$a^i a^k b^{i+j+k} \in \mathcal{L}(M)$$

is accepted by the machine, but that string is not in the language. In fact, for all t,

$$a^i a^{tj} a^k b^{i+j+k} \in \mathcal{L}(M)$$

but only for $t = 1$ is this string in L_o. Therefore $\mathcal{L}(M) \neq L_o$. The argument is general, so no DFA can recognize L_o, hence the language is not regular.

6. The Pumping Lemma

The argument above demonstrates a property of infinite regular languages. If a language can be recognized by a FA, all sufficiently long strings must cause the machine to repeat states, and the section of the string that causes the state sequence to loop can be pumped, that is, removed or repeated. And if the language is regular, and this is the proper machine recognizing the language, then all of those pumped strings must be in the language.

This is formalized in the pumping lemma which essentially says: *If A is an infinite regular language, every sufficiently long string in A can be pumped.*

Theorem 6.1 (The pumping lemma)**.** If A is an infinite regular language, then there exists an integer p, such that for every $s \in A$ with $|s| \geq p$, the string s can be written as

$$s = xyz$$

where $|xy| \leq p$, and $|y| > 1$ and for all $i \geq 0$, $xy^i z \in A$.

To use the pumping lemma to show a language is not regular,

(1) Cleverly pick a long enough s in the language.
(2) Consider every possible way that $s = xyz$, according to the lemma.
(3) For each such xyz, show an i such that $xy^i z$ is not in the language.

Then we have demonstrated that the language does not have a property that all regular languages must have, and therefore it is not regular.

Beware of coming to the conclusion that just because a language can be pumped that it is regular. The theorem begins with "if A is an infinite regular language",

```
def recognize_0i_1i(s):
    count, saw_one = 0, False
    for c in s:
        if c=='1':
            saw_one = True
            count -= 1
        else:
            if saw_one:
                return False
            count += 1
    return count==0
```

FIGURE 2. Python program recgnizing L_o

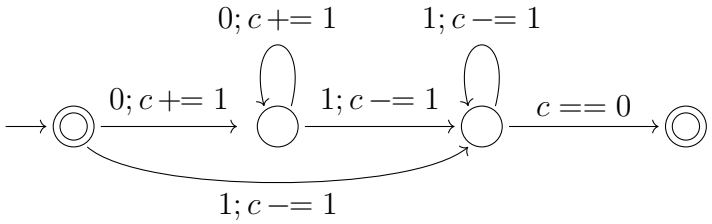


FIGURE 3. Counter machine for 0^i1^i

and therefore if it is not, nothing about the pumping lemma applies. For instance, the language,

$$\{\, a^i b^j c^k \mid j = k \text{ if } a > 0 \,\}$$

is not regular, but can be pumped.

7. A Counter Machine for a^ib^i

What kind of machine can recognize L_o? In Figure 2 is a Python program recognizing this language. I will attempt to transcribe it into a finite state machine but I will need an additional capability. I will add a *counter* to the DFA. The counter holds an integer, is initially zero, can be incremented, decremented and tested for zero. This is all the Python program does that is beyond what a DFA can do. The program structure, including the looping, the conditional execution are all expressible by states and state transitions.