Regular expressions and Kleene's theorem

Informatics 2A: Lecture 5

John Longley

School of Informatics
University of Edinburgh
jrl@inf.ed.ac.uk

26 September 2017

1. More closure properties of regular languages
   - Operations on languages
   - ϵ-NFAs
   - Closure under concatenation and Kleene star
2. Regular expressions
   - Regular expressions
   - From regular expressions to regular languages
3. Kleene's theorem and Kleene algebra
   - Kleene's theorem
   - Kleene algebra
   - From DFAs to regular expressions

Concatenation

We write $L_1.L_2$ for the concatenation of languages L_1 and L_2, defined by:

$$L_1.L_2 \;=\; \{xy \mid x \in L_1, y \in L_2\}$$

For example, if $L_1 = \{aaa\}$ and $L_2 = \{b, c\}$ then $L_1.L_2$ is the language $\{aaab,\ aaac\}$.

Later we will prove the following closure property.

If L_1 and L_2 are regular languages then so is $L_1.L_2$.

Kleene star

We write L^* for the Kleene star of the language L, defined by:

$$L^* \;=\; \{\epsilon\} \cup L \cup L.L \cup L.L.L \cup \ldots$$

For example, if $L_3 = \{aaa, b\}$ then L_3^* contains strings like *aaaaaa*, *bbbbb*, *baaaaaabbaaa*, etc.

More precisely, L_3^* contains all strings over $\{a, b\}$ in which the letter a always appears in sequences of length some multiple of 3

Later we will prove the following closure property.

If L is a regular language then so is L^*.

Exercise

Consider the language over the alphabet $\{a, b, c\}$

$$L = \{x \mid x \text{ starts with } a \text{ and ends with } c\}$$

Which of the following strings are valid for the language $L.L$?

1. *abcabc*
2. *acacac*
3. *abcbcac*
4. *abcbacbc*

Exercise

Consider the language over the alphabet $\{a, b, c\}$

$$L = \{x \mid x \text{ starts with } a \text{ and ends with } c\}$$

Which of the following strings are valid for the language $L.L$?

1. $abcabc$
2. $acacac$
3. $abcbcac$
4. $abcbacbc$

Answer: 1,2,3 are valid, but 4 isn't. (To split the string into two L-strings, we'd need c followed by a.)

Another exercise

Consider the (same) language over the alphabet $\{a, b, c\}$

$$L = \{x \mid x \text{ starts with } a \text{ and ends with } c\}$$

Which of the following strings are valid for the language L^* ?

1. ϵ
2. *acaca*
3. *abcbc*
4. *acacacacac*

Another exercise

Consider the (same) language over the alphabet $\{a, b, c\}$

$$L = \{x \mid x \text{ starts with } a \text{ and ends with } c\}$$

Which of the following strings are valid for the language L^* ?

1. ϵ
2. $acaca$
3. $abcbc$
4. $acacacacac$

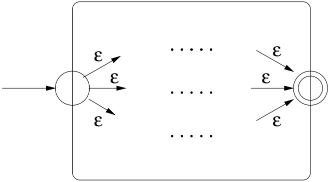
Answer: 1,3,4 are valid, but not 2. (In this particular case, it so happens that $L^* = L + \{\epsilon\}$, but this won't be true in general.)

NFAs with ϵ-transitions

We can vary the definition of NFA by also allowing transitions labelled with the special symbol ϵ (*not* a symbol in Σ).

The automaton may (but doesn't have to) perform a spontaneous ϵ-transition at any time, without reading an input symbol.

This is quite convenient: for instance, we can turn any NFA into an ϵ-NFA with just one start state and one accepting state:



(Add ϵ-transitions from new start state to each state in S, and from each state in F to new accepting state.)

Equivalence to ordinary NFAs

Allowing ϵ-transitions is just a convenience: it doesn't fundamentally change the power of NFAs.

If $N = (Q, \Delta, S, F)$ is an ϵ-NFA, we can convert N to an ordinary NFA with the same associated language, by simply 'expanding' Δ and S to allow for silent ϵ-transitions.

To achieve this, perform the following steps on N.

- For every pair of transitions $q \xrightarrow{a} q'$ (where $a \in \Sigma$) and $q' \xrightarrow{\epsilon} q''$, add a new transition $q \xrightarrow{a} q''$.
- For every transition $q \xrightarrow{\epsilon} q'$, where q is a start state, make q' a start state too.

Repeat the two steps above until no further new transitions or new start states can be added.

Finally, remove all ϵ-transitions from the ϵ-NFA resulting from the above process. This produces the desired NFA.

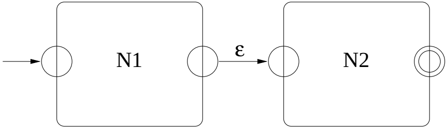
Closure under concatenation

We use ϵ-NFAs to show, as promised, that regular languages are closed under the concatenation operation:

$$L_1.L_2 \;=\; \{xy \mid x \in L_1, y \in L_2\}$$

If L_1, L_2 are any regular languages, choose ϵ-NFAs N_1, N_2 that define them. As noted earlier, we can pick N_1 and N_2 to have just one start state and one accepting state.

Now hook up N_1 and N_2 like this:



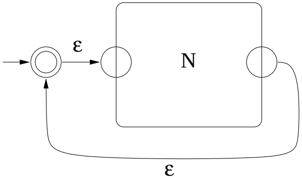
Clearly, this NFA corresponds to the language $L_1.L_2$.

Closure under Kleene star

Similarly, we can now show that regular languages are closed under the Kleene star operation:

$$L^* \ = \ \{\epsilon\} \cup L \cup L.L \cup L.L.L \cup \ldots$$

For suppose L is represented by an ϵ-NFA N with one start state and one accepting state. Consider the following ϵ-NFA:



Clearly, this ϵ-NFA corresponds to the language L^*.

Regular expressions

We've been looking at ways of specifying regular languages via machines (often presented as pictures). But it's very useful for applications to have more textual ways of defining languages.

A regular expression is a written mathematical expression that defines a language over a given alphabet Σ.

- The basic regular expressions are

$$\emptyset \qquad\qquad \epsilon \qquad\qquad a \quad (\text{for } a \in \Sigma)$$

- From these, more complicated regular expressions can be built up by (repeatedly) applying the two binary operations $+$, $.$ and the unary operation * . Example: $(a.b+\epsilon)^* + a$

We use brackets to indicate precedence. In the absence of brackets, * binds more tightly than ., which itself binds more tightly than $+$.

$$\text{So} \quad a + b.a^* \quad \text{means} \quad a + (b.(a^*))$$

Also the dot is often omitted: ab means $a.b$

How do regular expressions define languages?

A regular expression is itself just a written expression. However, every regular expression α over Σ can be seen as defining an actual language $\mathcal{L}(\alpha) \subseteq \Sigma^*$ in the following way.

- $\mathcal{L}(\boldsymbol{\emptyset}) = \emptyset, \quad \mathcal{L}(\boldsymbol{\epsilon}) = \{\epsilon\}, \quad \mathcal{L}(a) = \{a\}.$
- $\mathcal{L}(\alpha + \beta) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$
- $\mathcal{L}(\alpha.\beta) = \mathcal{L}(\alpha) \,.\, \mathcal{L}(\beta)$
- $\mathcal{L}(\alpha^*) = \mathcal{L}(\alpha)^*$

Example: $a + ba^*$ defines the language $\{a, b, ba, baa, baaa, \ldots\}$.

The languages defined by $\boldsymbol{\emptyset}, \boldsymbol{\epsilon}, a$ are obviously regular.

What's more, we've seen that regular languages are closed under union, concatenation and Kleene star.

This means every regular expression defines a regular language. (Formal proof by induction on the size of the regular expression.)

Exercises

Consider (again) the language

$$\{x \in \{0,1\}^* \mid x \text{ contains an even number of 0's}\}$$

Which of the following regular expressions define the above language?

1. $(1^*01^*01^*)^*$
2. $(1^*01^*0)^*1^*$
3. $1^*(01^*0)^*1^*$
4. $(1 + 01^*0)^*$

Exercises

Consider (again) the language

$$\{x \in \{0,1\}^* \mid x \text{ contains an even number of 0's}\}$$

Which of the following regular expressions define the above language?

1. $(1^*01^*01^*)^*$
2. $(1^*01^*0)^*1^*$
3. $1^*(01^*0)^*1^*$
4. $(1 + 01^*0)^*$

Answer: 2 and 4 define the required language. 1 doesn't: e.g. 11 doesn't match the expression. 3 doesn't: e.g. 00100 doesn't match the expression.

Kleene's theorem

We've seen that every regular expression defines a regular language.

Remarkably, the converse is also true: every regular language can be defined by a regular expression.

The equivalence between regular languages and expressions is:

Kleene's theorem

> *DFAs and regular expressions give rise to exactly the same class of languages (the regular languages).*

(For proof, see Kozen, Lecture 9.)
As we've already seen, NFAs (with or without ϵ-transitions) also give rise to this class of languages.

So the evidence is mounting that the class of regular languages is mathematically a very natural and well-behaved one.

Kleene algebra

Regular expressions give a textual way of specifying regular languages. This is useful e.g. for communicating regular languages to a computer.

Another benefit: regular expressions can be manipulated using algebraic laws (Kleene algebra). For example:

$$
\begin{array}{rclcrcl}
\alpha + (\beta + \gamma) & = & (\alpha + \beta) + \gamma & \qquad & \alpha + \beta & = & \beta + \alpha \\
\alpha + \emptyset & = & \alpha & & \alpha + \alpha & = & \alpha \\
\alpha(\beta\gamma) & = & (\alpha\beta)\gamma & & \epsilon\alpha & = & \alpha\epsilon \ = \ \alpha \\
\alpha(\beta + \gamma) & = & \alpha\beta + \alpha\gamma & & (\alpha + \beta)\gamma & = & \alpha\gamma + \beta\gamma \\
\emptyset\alpha & = & \alpha\emptyset \ = \ \emptyset & & \epsilon + \alpha\alpha^* & = & \epsilon + \alpha^*\alpha = \alpha^*
\end{array}
$$

Often these can be used to simplify regular expressions down to more pleasant ones.

Other reasoning principles

Let's write $\alpha \leq \beta$ to mean $\mathcal{L}(\alpha) \subseteq \mathcal{L}(\beta)$ (or equivalently $\alpha + \beta = \beta$). Then

$$\begin{array}{rcl} \alpha\gamma + \beta \leq \gamma & \Rightarrow & \alpha^*\beta \leq \gamma \\ \beta + \gamma\alpha \leq \gamma & \Rightarrow & \beta\alpha^* \leq \gamma \end{array}$$

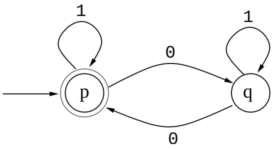
Arden's rule: Given an equation of the form $X = \alpha X + \beta$, its smallest solution is $X = \alpha^*\beta$.

What's more, if $\epsilon \notin \mathcal{L}(\alpha)$, this is the *only* solution.

Beautiful fact: The rules on this slide and the last form a complete set of reasoning principles, in the sense that if $\mathcal{L}(\alpha) = \mathcal{L}(\beta)$, then '$\alpha = \beta$' is provable using these rules. (Beyond scope of Inf2A.)

DFAs to regular expressions

We use an example to show how to convert a DFA to an equivalent regular expression.



For each state r, let the variable X_r stand for the set of strings that take us from r to an accepting state. Then we can write some simultaneous equations:

$$\begin{aligned} X_p &= 1X_p + 0X_q + \epsilon \\ X_q &= 1X_q + 0X_p \end{aligned}$$

Where do the equations come from?

Consider:

$$X_p = 1X_p + 0X_q + \epsilon$$

This asserts the following.

Any string that takes us from p to an accepting state is:

- a 1 followed by a string that takes us from p to an accepting state; or
- a 0 followed by a string that takes us from q to an accepting state; or
- the empty string.

Note that the empty string is included because p is an accepting state.

Solving the equations

We solve the equations by eliminating one variable at a time:

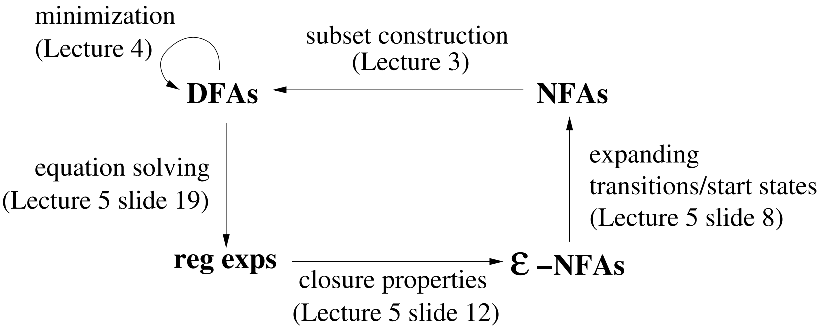
$$\begin{array}{rrcll} & X_q & = & 1^*0X_p & \text{by Arden's rule} \\ \text{So} & X_p & = & 1X_p + 01^*0X_p + \epsilon & \\ & & = & (1+01^*0)X_p + \epsilon & \\ \text{So} & X_p & = & (1+01^*0)^* & \text{by Arden's rule} \end{array}$$

Since the start state is p, the resulting regular expression for X_p is the one we are seeking. Thus the language recognised by the automaton is:

$$(1+01^*0)^*$$

The method we have illustrated here, in fact, works for arbitrary NFAs (without ϵ-transitions).

Theory of regular languages: overview



Reading

Relevant reading:

- Regular expressions: Kozen chapters 7,8; J & M chapter 2.1. (Both texts actually discuss more general 'patterns' — see next lecture.)
- From regular expressions to NFAs: Kozen chapter 8; J & M chapter 2.3.
- Kleene algebra: Kozen chapter 9.
- From NFAs to regular expressions: Kozen chapter 9.

Next two lectures: Some applications of all this theory.

- String and pattern matching
- Lexical analysis
- Model checking