TDDD14/TDDD85 Lecture 4: Closure Properties and Regular Expressions

Victor Lagerkvist (based on slides by Christer Bäckström and Gustav Nordh)

In this lecture we will describe yet another characterization of regular languages using *regular expressions*. We will prove that regular expressions turn out to be yet another characterization of regular languages, and in the process investigate closure properties of regular languages.

Background and Intuition

Regular expressions are an incredibly useful formalism for extracting and searching after data in text. For example, using the Unix utility `grep` we could issue the command `grep " *[0-9]" test.txt` to match all lines starting with blanks followed by a digit in the file `text.txt`. Broadly speaking, engines for regular expressions operate by "compiling" regular expressions to an intermediate data structure which is easier to work with than the original string, and the efficiency of the engine largely depends on the choice of this intermediate format. The preferred representation depends on the application in question, but being able to compile a regular expression to a simpler form which is easier to execute is advantageous for many types of regular expressions.

In fact, in this lecture we will prove that we already have a perfect machine for this purpose: the DFA. We will (1) prove that one for each regular expression can define an equivalent NFA, which by the subset construction admits an equivalent DFA, and (2) that there for each DFA exists a regular expression defining the same language. This not only leads to a new characterization of regular languages, but also to an important application of DFAs. However, before we begin with this task we investigate *closure properties* of regular languages, since these closure properties will make it much simpler to relate regular expressions to DFAs.

Closure Properties of Regular Languages

As promised, we begin by investigating closure properties of regular languages, before turning to regular expressions. The general idea is as follows: given regular languages $A_1, \ldots, A_k$, which operations can be performed on $A_1, \ldots, A_k$ while still guaranteeing that the result is a regular language?

Definition 1. *Let A be a set of languages. Say that A is* closed *under a function $f\colon A^k \to A$ if $f(A_1, \ldots, A_k) \in A$ for all $A_1, \ldots, A_k$.*

Note that set operations such as union intersection, and concatenation, are just binary functions which takes sets as arguments and returns new sets.

For example, the set of regular languages would be closed under $\cup$ (union) if $A_1 \cup A_2$ is a regular language whenever A_1 and A_2 are two regular languages. Which methods to we have available to prove that a language is regular? Our best chance at the moment is to attempt to construct an NFA for $A_1 \cup A_2$, and the most straightforward way of accomplishing this is to first take NFAs N_1 and N_2 accepting these A_1 and A_2, i.e., $L(N_1) = A_1$ and $L(N_2) = A_2$, and then trying to combine these machines in order to construct an NFA for $A_1 \cup A_2$. The two automata N_1 and N_2 are visualised in Figure 1 and Figure 2, where we for illustrative purposes have not bothered to write out the names of any states, or labelled the transitions with symbols. The two machines may then be combined as follows.

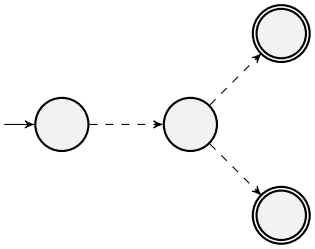


Figure 1: The NFA N_1.

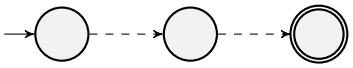
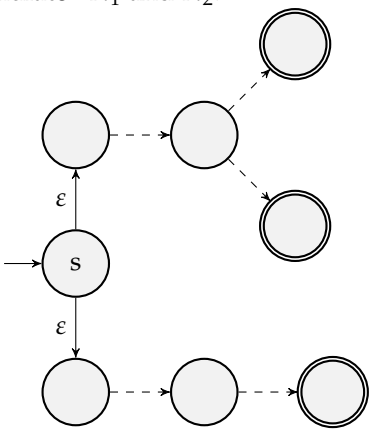


Figure 2: The NFA N_2.

Theorem 1. *The set of regular languages is closed under union.*

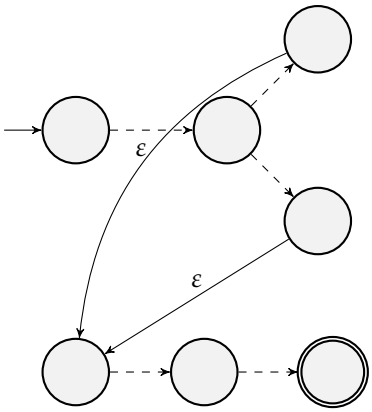
Proof. Assume that N_1 and N_2 are the NFAs in Figure 1 and Figure 2 and consider the following NFA which uses an ε-transition to "simulate" N_1 and N_2.



Given a string s the automaton then begins in the new start state and is then able to proceed either to the part of the automaton corresponding to N_1, or to the part of the automaton corresponding to N_2. □

Theorem 2. *The set of regular languages is closed under concatenation.*

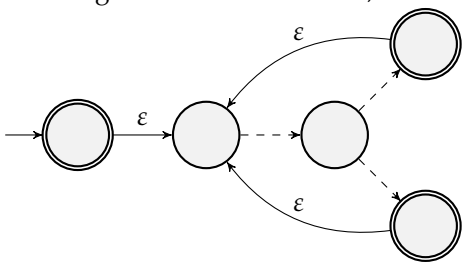
Proof. Similar to the previous proof we assume that A_1 and A_2 are two regular languages and that N_1 and N_2 be two NFAs accepting these languages (Figure 1 and Figure 2). Now consider the following NFA which uses an ε-transition to first proceed through N_1 and then immediately jump to N_2.



Given a string s the automaton then begins in N_1 and after having reached an original accepting state it jumps to N_2 with an ε-transition and simulates N_2 on the rest of the string. □

Theorem 3. *The set of regular languages is closed under the star operation.*

Proof. Let A_1 be a regular language and let N_1 be an NFA accepting this language (Figure 1). We will use ε-transitions so that we, when an accepting state is reach, have the possibility of jumping back to the start state. In addition we need to explicitly force the automaton to accept the empty string ε, which we can do by adding a new start state, make it an accepting state, and adding an ε-transition to the original start state. Hence, we obtain the following NFA.



Given a string s the automata then either accepts immediately (if $s = \varepsilon$) or begins simulating N_1. After having reached the original accepting state it then has the possibility of jumping back to the original start state. □

Regular Expressions

We now turn to the formal definition of regular expressions. There are several equal characterizations of regular expressions and we settle for the following definition which using as few constructs as necessary.

Definition 2. *Let Σ be an alphabet. We will simultaneously define a* regular expression *and the language that each construct defines, where $L(R)$ will denote the language described by the regular expression R. Then, R is a regular expression if R is*

1. *a for $a \in \Sigma$, $L(a) = \{a\}$,*
2. *ε, $L(\varepsilon) = \{\varepsilon\}$,*
3. *$\emptyset$, $L(\emptyset) = \emptyset$,*
4. *$R_1 + R_2$ where R_1 and R_2 are regular expressions, $L(R_1 + R_2) = L(R_1) \cup L(R_2)$[1],*
5. *$R_1 R_2$ where R_1 and R_2 are regular expressions, $L(R_1 R_2) = L(R_1)L(R_2)$, and*
6. *R_1^* where R_1 is a regular expression, $L(R_1^*) = L(R_1)^*$.*

[1] Some authors prefer to write $R_1 + R_2$ as $R_1 \cup R_2$ (or $R_1|R_2$), but we prefer the former to avoid overloading the usage of $\cup$.

The star operation has higher precedence than concatenation and $+$, and concatenation has higher precedence than $+$. However, if there is any risk of confusion we typically use parentheses to clarify the intended meaning.

Trivia: the operations defining regular expressions together with an alphabet forms as *Kleene algebra*, roughly meaning that $\emptyset$ behaves as 0, ε behaves as 1, $+$ behaves as addition, concatenation behaves as multiplication. The star operation does not have a direct counter part in arithmetic but may be thought of as a variant of iterated multiplication.

Example 1. *Let $R = (0+1)^*0$. What is the language of this regular expression? It consists of two parts, $(0+1)^*$, and 0, and if we can understand these two subexpressions, then understanding $(0+1)^*0$ is simple. However, since $(0+1)^*$ simpy describes all Boolean strings, the regular expression $(0+1)^*0$ must define the set of Boolean strings ending with 0.*

In this example we could immediately see the the language defined by the regular expression, but this is not always so simple if the expression is larger. In such a case it is good to be able to explicitly calculate the resulting language $L(R)$, which we can do by following the rules in Definition 2. In this particular example we get $L(R) = L((0+1)*0) = L((0+1)^*)L(0) = L((0+1)^*)\{0\} = L(0+1)^*\{0\} = (L(0) \cup L(1))^*\{0\} = (\{0\} \cup \{1\})^*\{0\} = \{0,1\}^*\{0\}$.

Let us consider a few more examples.

Example 2. *Let $R = (0+1)^*00(0+1)^*$. Then each string defined by this regular expression can be "broken down" into three parts: an arbitrary string in $(0+1)^* = \{0,1\}^*$, the string of two consecutive zeroes 00, and again an arbitrary string in $(0+1)^* = \{0,1\}^*$. Hence, $L(R)$ is the set of all Boolean strings containing at least two consecutive zeroes.*

Let $R = (0+1+2+3+4+5+6+7+8+9)^*1234(0+1+2+3+4+5+6+7+8+9)^*$. Then $L(R)$ defines the same language as the PIN code automaton from lecture 1, but arguably does so in a more natural way.

Our aim is now to prove that regular expressions exactly coincide with regular languages.

Theorem 4. *A language is regular if and only if some regular expression describes it.*

Since this is an "if and only if" statement there are two statements that have to be proven. First, given a regular expression R, construct an NFA N such that $L(R) = L(N)$. Second, given a DFA D, construct a regular expression R such that $L(D) = L(R)$. We tackle these two problems in the following two sections.

Recall that NFAs and DFAs have the same strength. Hence, for the first statement we choose to construct an NFA since using an NFA gives us more flexibility, but for the second statement we choose to use a DFA as a starting point since it makes it easier to construct an equivalent regular expression.

Regular Expressions to Finite Automata

To prove the first condition in Theorem 4 we show that there for any regular expression R exists an NFA N where $L(R) = L(N)$.

Lemma 1. *If a language is described by a regular expression then it is recognized by an NFA.*

Proof. (Sketch) We describe the construction of the NFA N_R for a regular expression R recursively as follows.

1. If $R = R_1 + R_2$ then $L(R_1 + R_2) = L(R_1) \cup L(R_2)$ and we recursively compute N_{R_1} and N_{R_2} and then construct the NFA N_R for $L(R_1) \cup L(R_2)$ from Theorem 1.
2. If $R = R_1 R_2$ then $L(R_1 R_2) = L(R_1)L(R_2)$ and we recursively compute N_{R_1} and N_{R_2} and then construct the NFA N_R for $L(R_1)L(R_2)$ from Theorem 2.
3. If $R = R_1^*$ then $L(R_1^*) = L(R_1)^*$ and we recursively compute N_{R_1} and then construct the NFA N_R for $L(R_1)^*$ from Theorem 3.

We leave the three non-recursive base cases as an exercise (see Exercise 2 on p. 9). □

To obtain a DFA one may then, for example, use the subset construction from lecture 3.

Finite Automata to Regular Expressions

We now turn to the second condition in Theorem 4. Our aim is thus to prove the following lemma.

Lemma 2. *If a language is recognized by a DFA then it is described by a regular expression*

Proof. (Sketch) We provide a proof sketch based on the *GNFA-method*. For full details and a correctness proof, see Lemma 1.60 in Sipser [2]. The basic idea is to consider a *generalised NFA* (GNFA) where transitions may be marked with regular expressions and not only alphabet

[2] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013

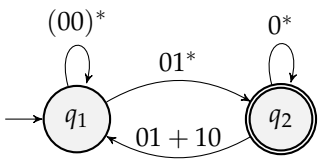


Figure 3: A GNFA.

symbols. For example, the automaton in Figure 3 is a GNFA where we may transition from q_1 to q_2 by the strings 0, 01, 011, and so on.

Given a DFA the plan is then to gradually convert this automaton to an equivalent GNFA which has a very simple form, consisting of a start state, an accept state, and a single transition from the start state to the accept state, labelled with the desired regular expression. Initially we perform the following three simple steps.

1. Add a new start state with an ε-transition to the old start state.
2. Add a new accept state with an ε-transitions from *all* old accept states
3. Replace transitions of the form a, b, c by $a + b + c$.

Clearly, these steps do not affect the recognised language, and the only purpose behind them is to get a few steps closer to the desired GNFA. See Figure 6 for an example. The crucial step is now to simplify the automaton by gradually *removing* states until only the new start state and the new accept state remain. However, it is clear that we cannot simply remove a state, since any transition leading to the state that was removed would no longer be valid. In general, assume that we have the GNFA in Figure 4, where R_1, R_2, R_3, R_4 are regular expressions, and that we want to remove the state s_e.

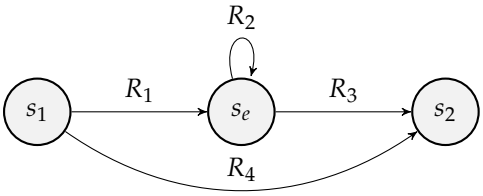


Figure 4: The $R_1R_2^*R_3 + R_4$ rule: before.

We claim that this is correctly accomplished by the the GNFA in Figure 5.

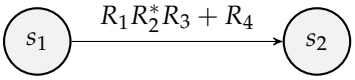


Figure 5: The $R_1R_2^*R_3 + R_4$ rule: after.

Why? Before removing s_e we had the choice of transitioning from s_1 to s_e using R_1, looping in s_e by reading R_2, and finally transitioning to s_2 by reading R_3. But this sequence of transitions precisely

corresponds to the regular expression $R_1R_2^*R_3$, and to ensure that the old transition from s_1 to s_2 using R_4 is still valid we end up with the regular expression $R_1R_2^*R_3 + R_4$. See Figure 7 and Figure 8 for a concrete example, where we have removed the state q_2. In order to apply the $R_1R_2^*R_3 + R_4$ rule we have to ensure that we apply it for *all* states in the automaton matching the configuration in Figure 4, and update the transition between these two states accordingly.

1. First, consider $s_1 = q_1$ and $s_2 = q_f$. We then want to update the transition between q_1 and q_f. We have that $R_1 = 1$, $R_2 = 1$, $R_3 = \varepsilon$, and that $R_4 = \emptyset$ (since there is no transition from q_1 to q_2 in Figure 7). This gives the expression $R_1R_2^*R_3 + R_4 = 11^*\varepsilon + \emptyset$, equivalent to 11^*, which we use to update the transition from q_1 to q_f.
2. Second, consider $s_1 = s_2 = q_1$. This case is needed since we in Figure 7 have a transition from q_2 to q_1 which needs to be taken into account when q_2 is removed, since we previously had the choice of going back and forth between q_1 and q_2. Then $R_1 = 1$ (q_1 to q_2), $R_2 = 1$ (q_2 to q_2), $R_3 = 0$ (q_2 to q_1) and $R_4 = 0$ (q_1 to q_1). This gives the expression $R_1R_2^*R_3 + R_4 = 11^*0 + 0$, and we therefore update the transition from q_1 to q_1 to reflect this (Figure 8).

We repeat the state removal step for each state different from the start and accept state. Hence, in Figure 8 we should now attempt to remove q_1. Thus, $s_1 = q_s$, $s_e = q_1$, and $s_2 = q_f$, giving $R_1 = \varepsilon$, $R_2 = 11^*0 + 0$, $R_3 = 11^* + \emptyset$, and $R_4 = \emptyset$. Hence, $R_1R_2^*R_3 + R_4 = \varepsilon(11^*0 + 0)^*(11^* + \emptyset) + \emptyset$ that we simplify to $(11^*0 + 0)^*(11^*)$ and becomes the new transition between q_s and q_f (Figure 9). Since only the start state q_s and accept state q_f remain with the transition labelled $R = (11^*0 + 0)^*(11^*)$, the regular expression corresponding to the original automaton in Figure 6 is R. □

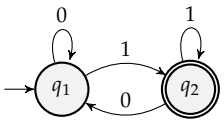


Figure 6: A DFA for the language $\{x1 \mid x \in \{0,1\}^*\}$.

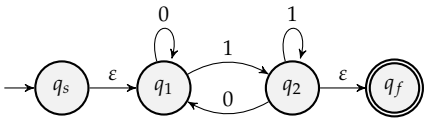


Figure 7: The NFA resulting from adding a new start and accept state.

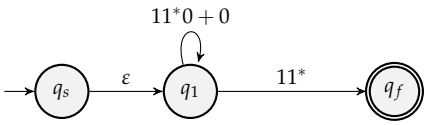


Figure 8: The GNFA resulting from eliminating q_2 and simplifying $11^*\varepsilon + \emptyset$.

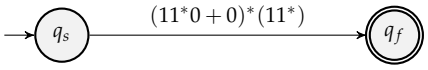


Figure 9: The GNFA resulting from eliminating q_1 and simplifying $\varepsilon(11^*0 + 0)^*(11^*) + \emptyset$.

Let us briefly summarise the GNFA method for converting a DFA to a regular expression from the proof sketch of Lemma 2.

1. Add a new start state with an ε-transition to the old start state.
2. Add a new accept state with an ε-transitions from *all* old accept states.
3. Replace transitions of the form a, b, c by $a + b + c$.
4. (a) Pick a state q distinct from the start and accept state.
   (b) For *all* states q_1 and q_2, including the case when $q_1 = q_2$, when the $R_1R_2^*R_3 + R_4$ rule is applicable, update the transition from q_1 to q_2 with $R_1R_2^*R_3 + R_4$ (simplify if possible).

(c) Remove q and go back to step 4.(a).

5. When only the start and accept state remain with transition R then output R.

How do I Know that my Answer/Solution is Correct?

The GNFA method for converting a DFA to a regular expression is more complicated than the other procedures and methods that we have encountered thus far. While the idea of gradually removing states is not so difficult, even minor mistakes can propagate and result in an erroneous answer.

How Can I Verify my Answer?

Even if we follow the GNFA method and simplify it is possible to end up with a large regular expression and it can be difficult to verify its correctness just by "looking" at it. For example, if we forget to simplify even for the simple automaton in Figure 6 we obtained $(11^*0 + 0)^*(11^*)$ even though $(0 + 1)^*1$ describes the same language. The easiest course of action is to simply test your regular expression, similarly to how you would test a function/procedure which has just been implemented in a programming language.

1. Define a handful of input strings and begin by simulating the NFA on these strings. This can be done either by hand, or by using one of the numerous tools available online. This will not take more than a couple of minutes once you are used to the procedure.

2. Compare the results with your regular expression. Either by hand, with a programming language which supports regular expressions, or by using a tool such as `grep`. Using `grep` you could e.g. have a text file where each line consists of a string, and only output the strings matching a given regular expression.

How Can I Verify my Solution?

Assume that you have verified your answer by the method outlined in the preceding section. This is not a strict guarantee that the solution is also correct since you might for example have introduced errors which have cancelled each other out, or made invalid assumptions.

First, attempt to verify your solution by going through each step, from the beginning to the end, and check its soundness. Did you really check all transitions involved when removing a state, and did you correctly compute $R_1R_2^*R_3 + R_4$? This might appear to be

tiresome but it is in general much quicker to verify a solution than to generate it from scratch. Some more general guidelines to check for are given below.

1. Do *not* try to be clever and remove several states in one iteration. Simply pick one state and update all involved transitions accordingly.
2. You should simplify regular expressions but you have to be absolutely certain that they are equivalent. For example, $L(R + \emptyset) = L(R)$, but $L(R + \varepsilon)$ is in general *not* equal to $L(R)$. However, $L(R\emptyset)$ is in general *not* equal to $L(R)$, while $L(R\varepsilon) = L(R)$. Making one small mistake in the beginning of the procedure, e.g., by confusing ε with $\emptyset$, could make the rest of the solution wrong.
3. When picking a state to remove you have to update *all* affected transitions, including loops.
4. The resulting regular expression depends on the order which you have picked states. Hence, your solution may be perfectly correct even if the answer differs from a given answer.

Summary

We defined regular expressions, a class of expressions which not only have an abundance of applications, but also turned out to be yet another description of regular languages. Crucially, in the process we have also proved closure properties of regular languages and described how regular expressions can be converted to finite automata, and how finite automata can be converted to regular expressions. In the next lecture we will return to finite automata and describe a useful *minimisation* algorithm for DFAs.

Food for Thought

1. The constructions in the proofs of the closure properties in Theorem 1, Theorem 2, and Theorem 3 were for simplicity presented in a visual form. Given $N_1 = (Q, \Sigma, \delta, q_0, F)$ and $N_2 = (Q', \Sigma', \delta', q_0', F')$, can you think of a way to explicitly define the NFA for the language $L(N_1) \cup L(N_2)$?
2. In the proof of Lemma 1 we did not cover every type of regular expression. Which cases are missing from Definition 2, and for each such regular expression, how can you define the corresponding NFA?

3. The main selling point in this lecture is the realisation that DFAs, NFAs, and regular expressions turned out be equivalent characterizations of the class of regular languages. However, certain languages are much more succinctly represented by regular expressions than by DFAs, and vice versa. Can you come up with a regular expression where the equivalent DFA is much larger?

References

M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.