Converting regular expressions to state diagrams and vice versa

Up to now we have considered two language specification mechanisms: regular expressions and state diagrams. We show that regular expressions and state diagrams define exactly the same class of languages. This fact is remarkable because superficially state diagrams and regular expressions appear to be quite different.

To show that regular expressions and state diagrams are equivalent we need to do the conversion in both directions. This material is from Chapter 9 in the textbook.

Converting regular expressions to state diagrams

For an arbitrary regular expression we construct an equivalent nondeterministic state diagram with ε-transitions that satisfies the following conditions:

1. There is exactly one accepting state and it is distinct from the start state.
2. There are no transitions into the start state.
3. There are no outgoing transitions from the accepting state.

The recursive construction relies on the fact that the previously constructed state-transition diagrams satisfy the above conditions 1., 2., 3. (In general, an ε-NFA does not need to satisfy the above three conditions.)

The state diagrams for the base cases (i) $E = \emptyset$, (ii) $E = \varepsilon$, (iii) $E = a$, $(a \in \Sigma)$ are depicted in Figure 1.

Inductive step: Next suppose that E_i, $i = 1, 2$, are regular expressions and we have constructed a state diagram S_i that is equivalent to E_i and satisfies the conditions 1., 2., 3. An

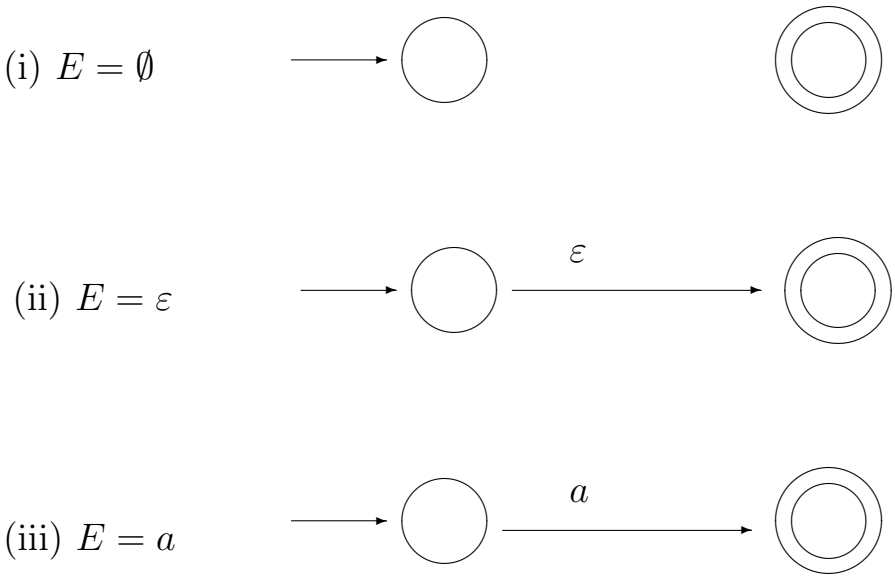

Figure 1: State diagrams for the base cases.

abstract representation of S_i is given in Figure 2: S_i has no incoming transtions to start state, has one accepting state, and no outgoing transitions from the accepting state.

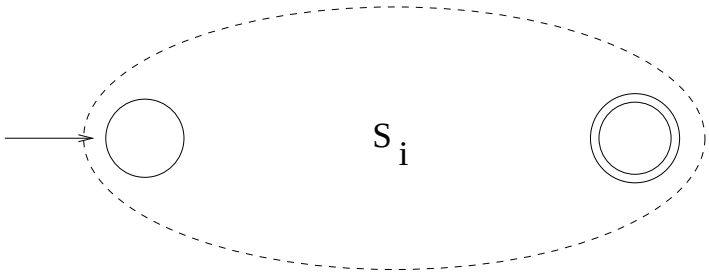

Figure 2: State diagram S_i that is constructed for the regular expression E_i, $i = 1, 2$.

For the inductive step we have to show how to construct state diagrams for the regular expressions $E_1 + E_2$, $E_1 \cdot E_2$ and E_1^*.

(iv) The state diagram for $E_1 + E_2$ is depicted in Figure 3. The resulting state diagram has all states of S_1 and S_2 with the exception that the start states (respectively, the accepting states) of S_1 and S_2 are merged together. The conditions 1., 2., 3. guarantee that the constructed state diagram accepts exactly the strings accepted by S_1 and the

strings accepted by S_2 (more details explained in class). Note that the resulting state diagram again satisfied the conditions 1., 2., and 3.

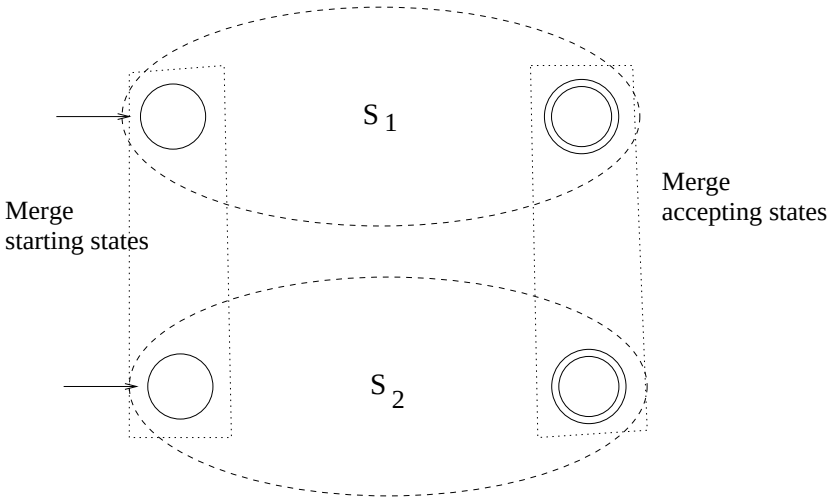

Figure 3: State diagram for regular expression $E_1 + E_2$.

(v) The state diagram for the regular expression $E_1 \cdot E_2$ is constructed by merging the accepting state of S_1 with the start state of S_2 and the merged state is not accepting. The construction is depicted in Figure 4. Again the conditions 1., 2., 3. guarantee that to reach the new accept state from the start state of S_1, the input has to first spell out a string accepted by S_1 followed by a suffix that is accepted by S_2. Note that since the merged state has no outgoing transitions in S_1 and no incoming transition in S_2, the resulting state diagram cannot have "mixed computations" that would first simulate S_1, then simulate S_2, and go back to simulating S_1. (Note that for correctness of this part of the construction only one of the conditions 2. or 3. would be sufficient.)

(vi) To conclude the recursive construction, we have to consider the closure operation. For the regular expression E_1^* we construct from S_1 a state diagram as follows: merge the accepting state of S_1 with the start state of S_1, the merged state is neither an

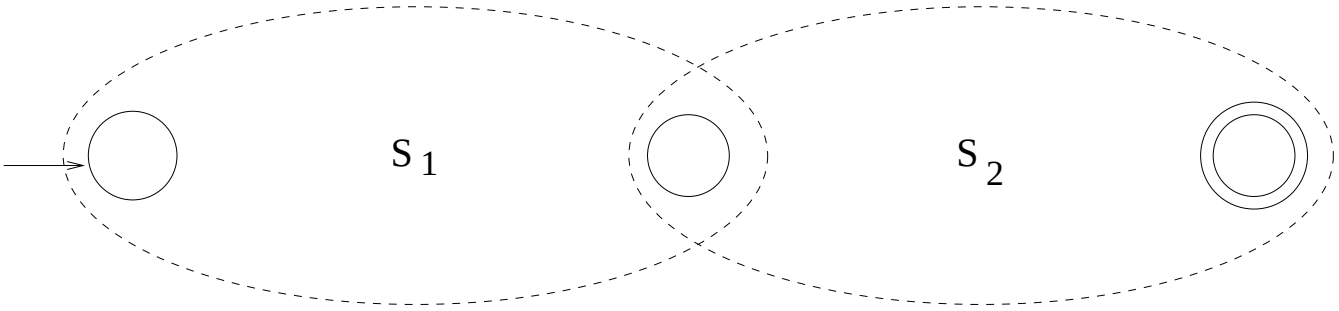

Figure 4: State diagram for the regular expression $E_1 \cdot E_2$.

accepting nor the start state. Then we introduce a new starting state and a new accepting state connected to the merged state by ε-transitions. The construction is depicted in Figure 5.

The constructed state diagram accepts exactly all strings that are the concatenation of a finite number of strings accepted by S_1. Due to the ε-path from the start state to the accepting state the finite number can be zero. More details will be discussed in class.

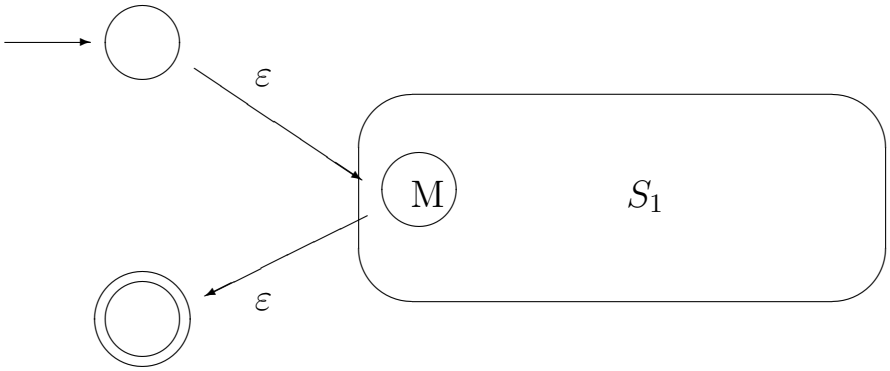

Figure 5: State diagram for the regular expression E_1^*.

It should be noted that in each of the above cases (iv), (v) and (vi) the resulting state diagram again satisfies the properties 1., 2., 3. This is needed for the correctness of the

recursive construction.

Using techniques from the previous chapter (last week) we can first eliminate ε-transitions and then convert the nondeterministic state diagram into an equivalent deterministic state diagram (subset construction).

Example. We apply the construction to the regular expression $(01 + 1)^*1$. To be done in class.

State diagrams to regular expressions (Section 9.2)

Next we consider the converse transformation. For a given state diagram we construct an equivalent regular expression using the so called *state-elimination procedure.* The intermediate stages in the construction are *generalized state-transition diagrams* where the edges can be labeled by any regular expressions (instead of individual alphabet symbols).

The input state diagram is assumed to have exactly one accepting state that is not the start state. A nondeterministic state diagram (NFA) can always be converted into this form. How?

The procedure removes (eliminates) states one at a time. The start state and the accepting state are not removed. The state-elimination step is illustrated in Figure 6. In the figure represents part of a generalized state-transition diagrams and the labels of the arrows (X, Y, Z, W) are regular expressions. If state C is eliminated, the transition from A to B in the modified diagram will be labeled by $X + YW^*Z$.

Important note: When a particular state (like C in the figure) is eliminated, we have to modify transitions between all pairs of states that are connected to the eliminated state as indicated in the figure. In particular, it is possible that A and B may be the same state.

Since the start state and the unique accepting state are not eliminated, at the end of the

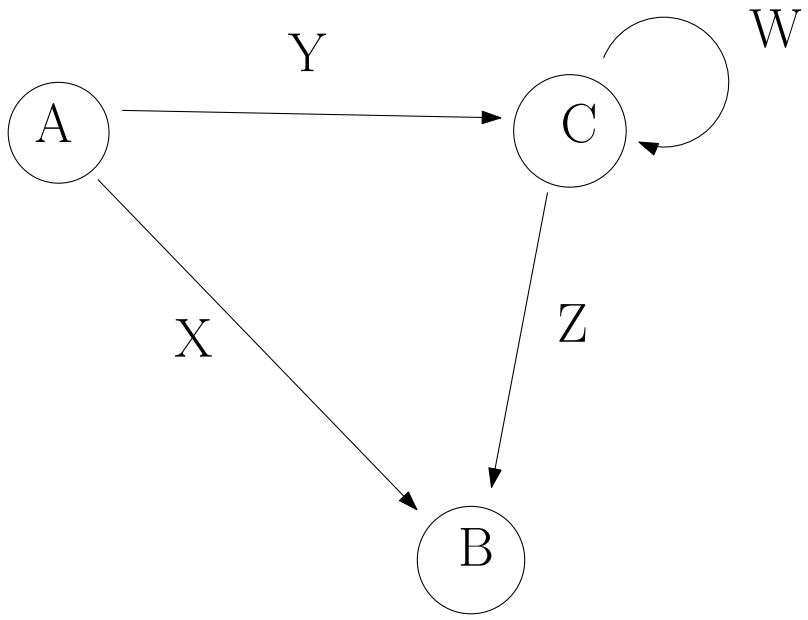

Figure 6: State elimination step: state C is removed and in the new diagram the transition from A to B will be labeled by $X + YW^*Z$.

process we have a two state diagram as depicted in Figure 7. From this diagram we can read the resulting regular expression to be:

$$U^*R(V + SU^*R)^*$$

Explained in class. Note that in the two-state diagram obtained at the end of the process, some of the transitions may be missing, that is, some of the regular expressions R, S, U, V may be $\emptyset$. In this context it is useful to remember what is the result of applying closure to $\emptyset$ or the result of concatenating a language and the empty set.

The state elimination algorithm is described in more detail on pp. 196–197 in the textbook. Below we consider an example.

Example. We consider the state diagram given in Figure 8.

In the first stage of the construction we modify the state diagram so that it has only one accepting state. The resulting state diagram is given in Figure 9, in the figure e stands for

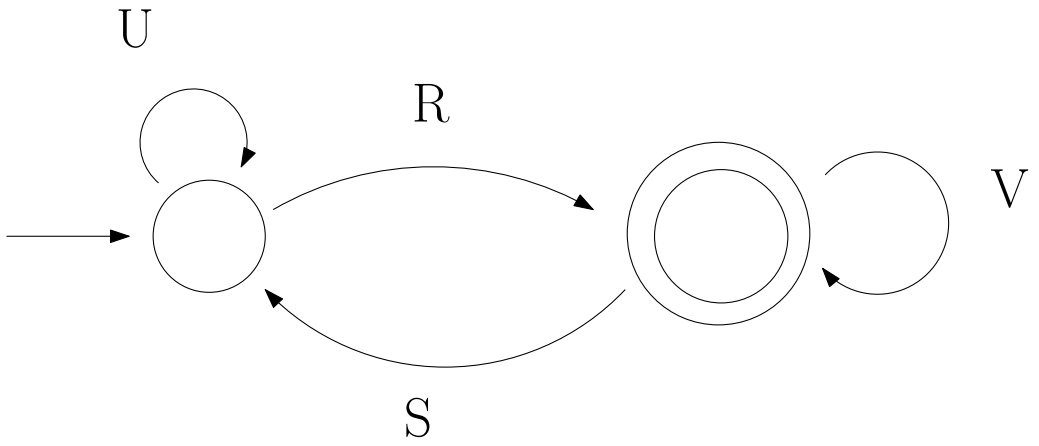

Figure 7: The state diagram (with two states) after all states except the start state and the accepting state have been eliminated.

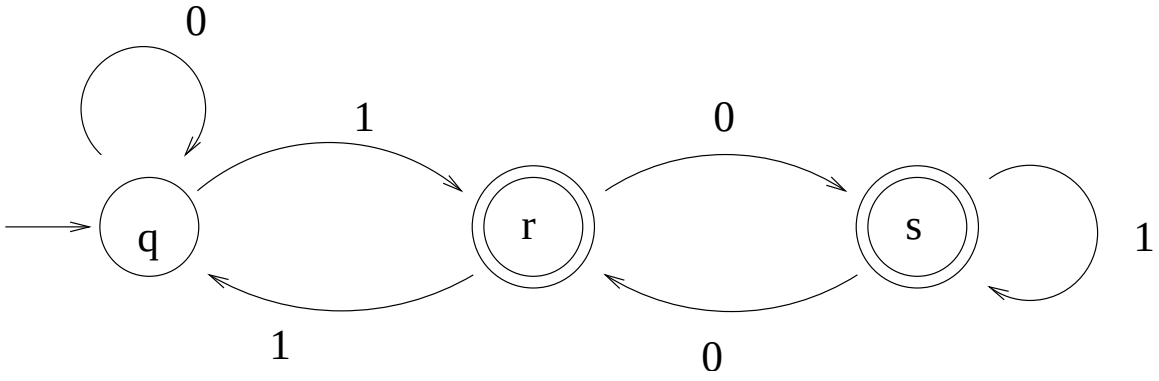

Figure 8: A state diagram to be used as input for the state-elimination algorithm.

the empty string.

Note: The high level description of the state-elimination algorithm does not specify the order in which state states should be eliminated. Thus when applying the procedure to the state diagram of Figure 9 we can first eliminate either state r or state s. Recall that the start state or the unique accepting state cannot be eliminated.

Assuming we apply the state elimination technique to the state diagram of Figure 9 by first eliminating the state r and then the state s, the resulting regular expression will be

$$(0 + 11 + (10(1 + 00)^*01))^*(1 + 10(1 + 00)^*(0 + \varepsilon))$$

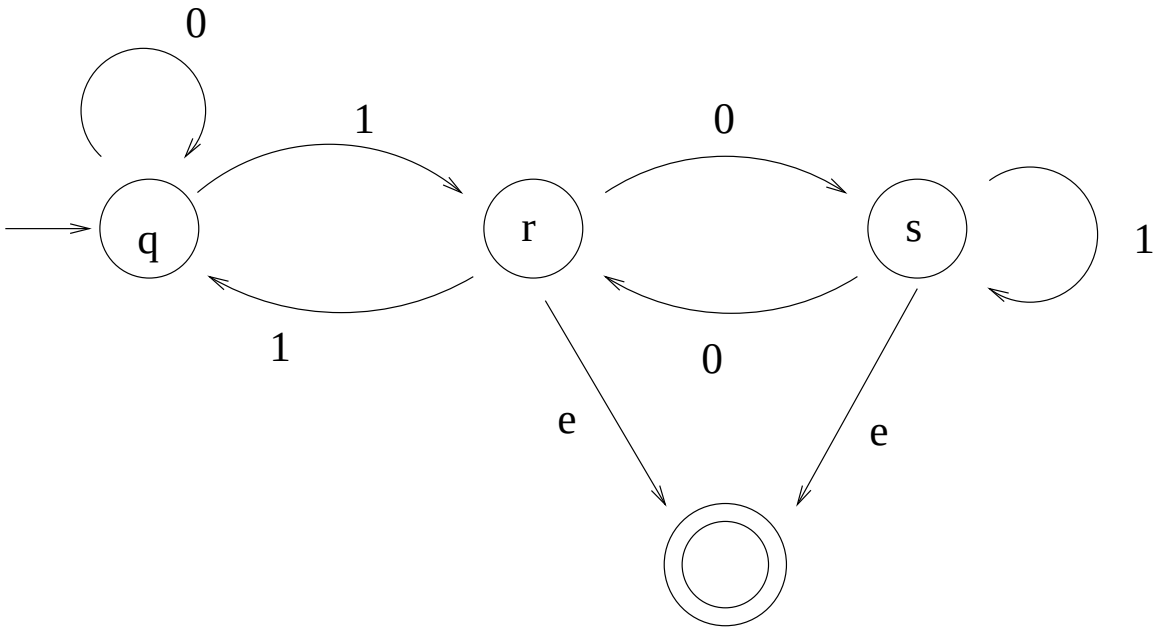

Figure 9: The modified state diagram that has only one accepting state. In the figure e denotes the empty string.

The details will be done in class.

Note: If, in the example, we would begin by eliminating the state s we will get a different regular expression. Generally, using different orders of state-elimination usually produces very different looking regular expressions, however they all denote the same language as the original state diagram.[1]

[1] Although this may not be easy to see just by comparing the regular expressions.