Lecture 4:
Regular Expressions

Lecture Outline

- Definition of Regular Expressions
- Regular Expressions and Their Relation to Regular Languages
- Summary of Results on Finite Automata, Regular Grammars and Regular Expressions

Reading

Chapters 1,3 and 4 of Martin

Chapter 1 of Brookshear

Chapter 3 of Revesz

Part I, Chapter 4 of Cohen

Chapter 4 of Floyd & Beigel

Definition of Regular Expressions

- Finite automata and regular grammars offer two ways of describing formal languages.

- A third approach is to specify more complex languages in terms of operations on simpler languages – **regular expressions** provide a means of doing this.

- Given an alphabet Σ, simple languages are $\emptyset$, the empty language, and the languages $\{x\}$ for each symbol $x \in \Sigma$.

- From these simple languages more complex languages are built according to three operations:

  1. **union**: if L_1 and L_2 are languages then $L_1 \cup L_2$ is the language containing those strings which occur either in L_1 or in L_2.
     E.g. if $L_1 = \{x, zyz, yy\}$ and $L_2 = \{x, zz, yzy\}$ then

     $$L_1 \cup L_2 = \{x, zyz, yy, zz, yzy\}$$

  2. **concatenation**: if L_1 and L_2 are languages then $L_1 \circ L_2$ is the language obtained by concatenating a string from the L_1 with a string from L_2.
     E.g. if $L_1 = \{x, zyz\}$ and $L_2 = \{zz, yzy\}$ then

     $$L_1 \circ L_2 = \{xzz, xyzy, zyzzz, zyzyzy\}$$

  3. **Kleene star**: if L is a language then L^* is the language obtained by concatenating zero or more strings from L.
     E.g. if $L = \{x, zyz\}$ then

     $$\begin{aligned} L^* \;=\; & \{\lambda, x, xx, xxx, \ldots, zyz, zyzzyz, \ldots, xzyz, xxzyz, \\ & \ldots, zyzx, zyzxx, \ldots\} \end{aligned}$$

Definition of Regular Expressions (cont)

- Formally, a **regular expression over an alphabet** Σ is defined by:
  1. $\emptyset$ is a regular expression;
  2. each member of Σ is a regular expression;
  3. if p and q are regular expressions then so is $p \cup q$;
  4. if p and q are regular expressions then so is $p \circ q$;
  5. if p is a regular expression then so is p^*.

- A regular expression r is said to **represent** a language, denoted $L(r)$, as follows:
  1. if $r = \emptyset$ then $L(r) = \emptyset$;
  2. if $r \in \Sigma$ then $L(r) = \{r\}$;
  3. if $r = p \cup q$ then $L(r) = L(p) \cup L(q)$;
  4. if $r = p \circ q$ then $L(r) = L(p) \circ L(q)$;
  5. if $r = p^*$ then $L(r) = L(p)^*$.

  E.g. $((x \circ y)^* \cup z^*)$ represents the language consisting of strings of zero or more xy's together with strings of zero or more z's.

Relation Between Regular Languages and Regular Expressions

Proposition 0.1 *(Kleene's Theorem) Given an alphabet Σ, the regular languages over Σ are exactly the languages that are represented by regular expressions over Σ.*

Proof

The regular languages are just those that are accepted by deterministic finite automata. However, since we have already shown that the languages accepted by deterministic finite automata are the same as those accepted by nondeterministic finite automata, we may work with finite automata of either sort in the following without any loss of generality.

Thus, we need to show that

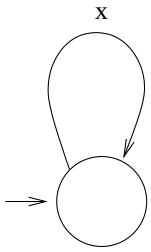
1. for any regular expression r there is a finite automaton M such that the language $L(r)$ represented by r is just the language $L(M)$ accepted by M; and
2. for any finite automaton M the language $L(M)$ accepted by M may be represented by a regular expression r.

Regular Languages / Regular Expressions (cont)

Part 1 Consider the first case. Suppose r is a regular expression. Then r has one of the five forms indicated above. We proceed as follows:

1. If $r = \emptyset$ then $L(r) = \emptyset$.

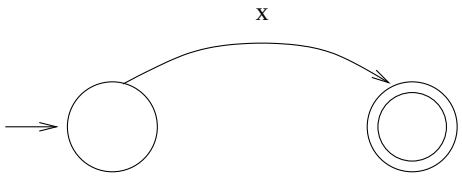
   We must construct a FA M such that $L(M) = \emptyset$.

   This automaton has no accept state and therefore accepts the language $\emptyset$.

2. If $r = x$ for some $x \in \Sigma$ then $L(r) = \{x\}$.

   We must construct a FA M such that $L(M) = \{x\}$.

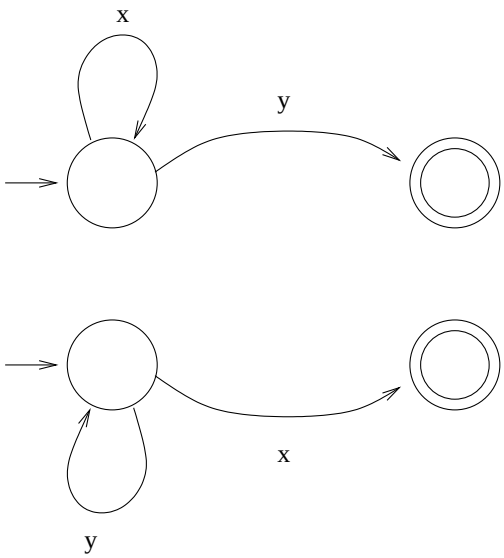
   This automaton accepts the language $\{x\}$.

3. If $r = p \cup q$ for some regular expressions p and q then $L(r) = L(p) \cup L(q)$.

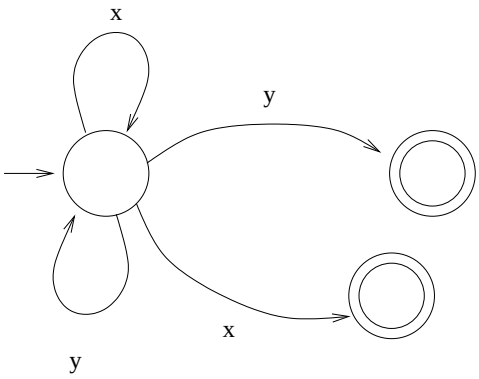
   Given two FA's M_p and M_q such that $L(M_p) = L(p)$ and $L(M_q) = L(q)$ we must construct a FA M such that $L(M) = L(M_p) \cup L(M_q)$.

   To take an example suppose M_p and M_q are as follows:

Regular Languages / Regular Expressions (cont)



To find an automaton that accepts all and only the strings accepted by each of these individually, it will not do to simply merge initial states. E.g.



This automaton accepts xyx which neither of the original two do.

Regular Languages / Regular Expressions (cont)

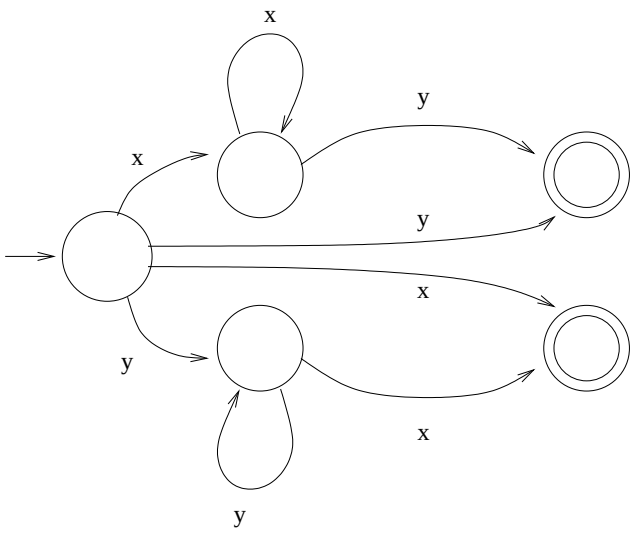
Instead:

(a) create a new initial state and cancel initial status of existing initial states;

(b) if and only if either of the previous initial states was an accept state, make the new initial state an accept state;

(c) to each state which is the destination of an arc from a previous initial state, draw an arc from the new initial state with the same label.

Applying this to our example we get:



This automaton accepts all and only the strings accepted by either of the initial two automata.

Regular Languages / Regular Expressions (cont)

4. If $r = p \circ q$ for some regular expressions p and q then $L(r) = L(p) \circ L(q)$.

   Given two FA's M_p and M_q such that $L(M_p) = L(p)$ and $L(M_q) = L(q)$ we must construct a FA M such that $L(M) = L(M_p) \circ L(M_q)$.

   Proceed as follows:

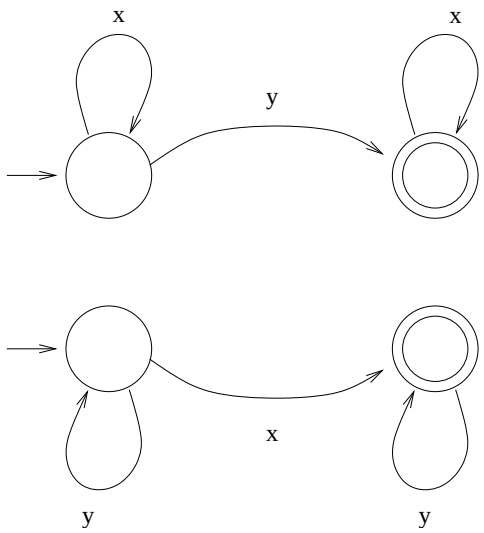
   (a) from each accept state in M_p draw an arc to each state in M_q to which an arc extends from the initial state of M_q and label these arcs with the labels of the arcs in M_q;

   (b) remove the accept state designation from accept states in M_p;

   (c) allow accept states in M_p to remain accepts states in M if and only if the initial state in M_q was an accept state;

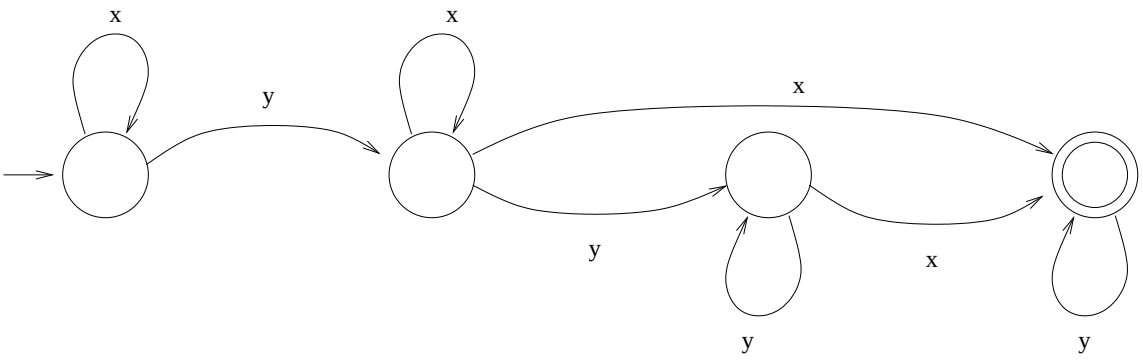
   (d) remove the initial state designation from the initial state of M_q.

   For example from the two automatons:



   the automaton accepting the concatenation of all strings from the first with strings from the second is:

Regular Languages / Regular Expressions (cont)



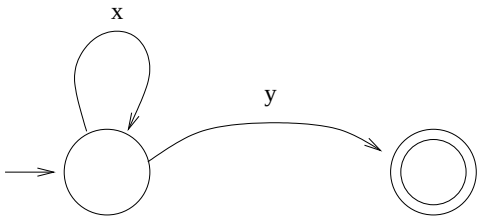
5. If $r = p^*$ for some regular expression p then $L(r) = L(p)^*$.

   Given a FA M_p such that $L(M_p) = L(p)$ we must construct a FA M such that $L(M) = L(M_p)^*$.

   In building an automaton M to accept the Kleene star of a regular language the basic intuition is to build an automaton which concatenates the original back on itself.

   However M must also accept the empty string λ (definition of Kleene star). To do this we cannot just designate the initial state an accept state since there may be arcs originating and ending in the initial state.

   E.g. in this diagram, making the initial state an accept state would allow x to be accepted – this is not in the Kleene star of the regular language accepted by the automaton.



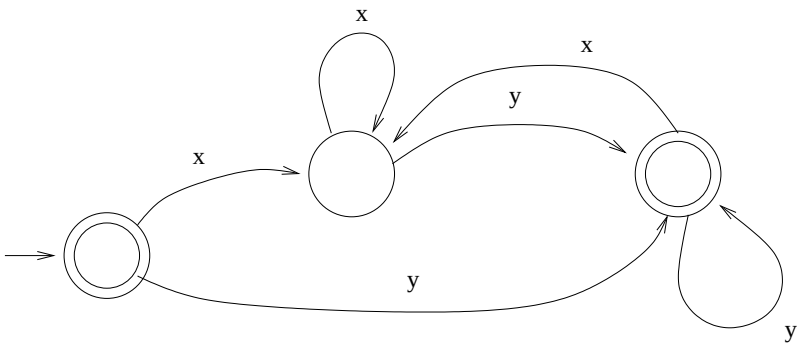
Regular Languages / Regular Expressions (cont)

Thus:

(a) create a new initial state and cancel initial status of existing initial states; make this new initial state an accept state;

(b) to each state which is the destination of an arc from the previous initial state, draw an arc from the new initial state with the same label;

(c) from each accept state draw an arc to each state which is the destination of an arc from the previous initial state with the same label as the corresponding arc from the old initial state.



Thus we have shown that for the regular expressions $\emptyset$ and the members of Σ, the languages they represent are regular (are accepted by a finite automaton). And we have shown that if the languages represented by two regular expressions p and q are regular then the languages represented by $p \cup q$, $p \circ q$ and p^* are regular too.

Regular Languages / Regular Expressions (cont)

Part 2 Now consider the second part of the proposition: for any finite automata M the language $L(M)$ accepted by M may be represented by a regular expression r.

The proof here proceeds by induction on the number of states in the transition diagram T for M.

There are two preliminaries:

1. The proof employs transition diagrams which allow arcs to be labelled with regular expressions where we understand these to mean the automaton must read a pattern compatible with the expression on arc in order to traverse it.

   E.g. if an arc is labelled with the expression $(x \circ y)^* \cup z$ then to traverse the arc a pattern matching z or $xy, xyxy, xyxyxy, \ldots$ must be read.

   Since these diagrams are more general than conventional ones, if our proof holds for these it will hold for the simpler diagrams; i.e. if languages accepted by these new diagrams can be represented by regular expressions then languages accepted by the older form of diagram will also be representable by regular expressions.

2. We assume T has only one accept state. Otherwise we could make a copy of T for each accept state with only that one accept state, find a regular expression which represented the language accepted by the copy and then form the union of all these regular expressions to obtain a regular expression for T as a whole.

Regular Languages / Regular Expressions (cont)

Base Step Assume T is a generalised transition diagram with only one accept state and assume that every state in T is either an initial state or an accept state.

There are two possibilities.

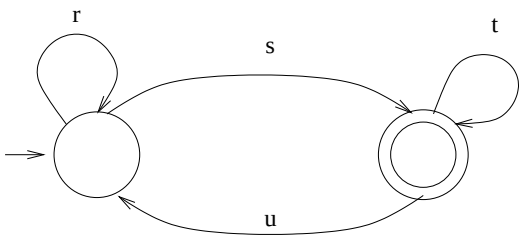
1. There is just one state which is both an accept state and an initial state.

   In this case the regular expression representing the language accepted by this transition diagram is the Kleene star of the union of the regular expressions occurring on the arcs of T.

2. There are two states one an initial state, the other an accept state.

   If there are multiple arcs from either state back to itself then replace these with one arc whose label is the union of the regular expressions occurring on the original arcs.

   There are now only four possible arcs:



Regular Languages / Regular Expressions (cont)

The regular expression associated with T is arrived at as follows:

(a) If s is not present then the regular expression is $\emptyset$ since there is no way to get to the accept state.

(b) If s is present then

i. if u is not present then the regular expression is $((r^* \circ s) \circ t^*)$ where r and t are replaced by $\emptyset$ if they are not present (the expression can be read any number of r's followed by an s followed by any number of t's);

ii. if u is present then the regular expression is

$$(((r^* \circ s) \circ t^*) \circ (u \circ ((r^* \circ s) \circ t^*))^*)$$

where again r and t are replaced by $\emptyset$ if they are not present.

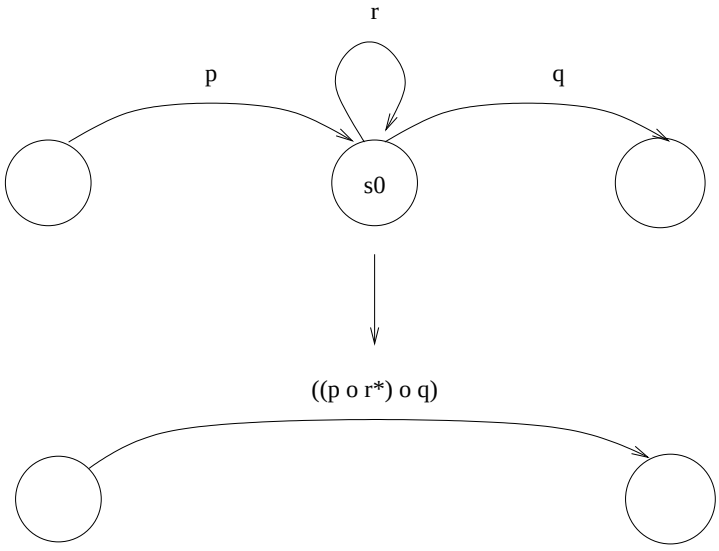
Regular Languages / Regular Expressions (cont)

Inductive Step Suppose that the language accepted by a generalised transition diagram T with no more than n states which are not initial or final states can be represented by a regular expression.

We must show that the language accepted by a generalised transition diagram T with no more than $n + 1$ states which are not initial or final states can be represented by a regular expression.

Proceed as follows:

1. select a state s_0 in T that is not an accept state or an initial state;
2. remove s_0 from T and remove all arcs starting or ending at s_0;
3. for each removed arc p which ended at s_0 and each removed arc q which started at s_0 add an arc from p's origin state to q's destination state with label $((p \circ r^*) \circ q)$ where r is the union of the labels on arcs starting and ending at s_0 or $\emptyset$ if there were no such arcs.



The resulting diagram is another generalised transition diagram which will accept the same language as T, but has only n states, and to which there therefore corresponds, by the hypothesis of induction, a regular expression representing the language it accepts. ∎

Summary of Results on Finite Automata, Regular Grammars and Regular Expressions

- **Finite automata** are abstract machines that can be used to recognise or to generate possibly infinite sets of strings.

- Finite automata come in two flavours: **deterministic** and **nondeterministic**.

  The languages (sets of strings) that can be accepted by these two types of finite automata are identical. These languages are called **regular** languages.

- Not all languages are regular languages, e.g $x^n y^n$.

- Languages can also be specified using abstract devices called **phrase structure grammars**. These specify allowable strings over an alphabet by means of **rewrite rules**.

  Phrase structure grammars whose rewrite rules are constrained to the two forms

  $$\begin{aligned} nonterminal &\rightarrow terminal \\ nonterminal &\rightarrow terminal\ nonterminal \end{aligned}$$

  are called **regular grammars**.

- The languages accepted by regular grammars are precisely the regular languages, i.e. those accepted by finite automata.

- A third way of specifying languages is by means of expressions built out of applications of the **union**, **concatenation** and **Kleene star** operators on simpler expressions, ultimately expressions representing symbols in the alphabet. Such expressions are called **regular expressions**.

- The languages represented by regular expressions are precisely the regular languages.