CSCI 2670, Fall 2012

Introduction to Theory of Computing

Department of Computer Science
University of Georgia
Athens, GA 30602

Instructor: Liming Cai
www.cs.uga.edu/∼cai

Lecture Note 2

Automata and Languages

II. Automata and Languages

Chapter 1. Regular Languages

We use languages to represent computation problems

Two kinds of computation problems:

(1) computing functions:
with a possibly long, desired answer – called *search problems*

(2) computing predicates:
with a short, yes/no answer – called *decision problems*

Decision problems are presented as languages;

A language is simply encoding of a decision problem.
I.e., It contains all strings, each encoding a problem instance having the 'yes' answer.

E.g.,

Decision problem SHORTEST DISTANCE

Input: graph G, vertices s, t, distance k;

Output: 'yes' iff the shortest distance between s and t is $\leq k$.

The corresponding language L_{sd}

$=\{ G\#s\#t\#k$: the shortest distance between s and t is $\leq k\}$

where $G\#s\#t\#k$ is an encoding of G, s, t, and k over the alphabet.

Your algorithm that can determine any given string

$G\#s\#t\#k$ belongs to L_{sd} or not

can also be used to solve the decision problem SHORTEST DISTANC.

We only study decision problems, thus languages. But why is this enough?

Because, algorithms for decision problems can be used to solve search problems, though indirectly!

Search problem SHORTEST PATH
Input: graph G, vertices s, t;
Output: a shortest path between s and t.

How an algorithm A for decision problem SHORTEST DISTANCE can help solve this search problem?
hint: construct a simple algorithm for SHORTEST PATH *that calls* A *as a subroutine.*

1.1 FINITE AUTOMATA

Now we begin to discuss a class of languages
  – called *regular languages*
and study them based on a weak computation model

Intuitively, a *finite automaton* is a machine consists of

(1) a finite number of states
(2) it reads one symbol at a time from the input (left to right)
(3) the currently read symbol determine the next state to transit
(4) the input is accepted if it ends at an "acceptance" state.
(5) otherwise, the input is "rejected".

An example of finite state machine: an automatic door

- it has two states: OPEN and CLOSED

- people standing at the door: *front, rear, both, neither*

draw a finite state machine for this.

State transition table:

	neither	*front*	*rear*	*both*
CLOSED	CLOSED	OPEN	CLOSED	CLOSED
OPEN	CLOSED	OPEN	OPEN	OPEN

a finite automaton (FA) has

- exactly one starting state (where computation starts)
- usually one acceptance state (where computation may end)
- transitions are labelled with symbols that make the transitions.

e.g, finite automaton called M_1 (Figure 1.4 on page 34).

Does it accept string 1001?

How about 10001?

10010? and 100100?

So it accepts a set of strings, i.e., the language *accepted by* the FA.

Formal definition of an FA

Definition 1.5 (page 35)

A *finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the *states*,
2. Σ is a finite set called the *alphabet*,
3. $\delta : Q \times \Sigma \to Q$ is teh *transition function*,
4. $q_0 \in Q$ is the *start state*, and
5. $F \subseteq Q$ is the *set of accept states*.

Formally describe M_1 (Figure 1.4 on page 34) as $(Q, \Sigma, \delta, q_0, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $Sigma = \{0, 1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state,
5. $F = \{q_2\}$.

What language does M_1 accept?

More FA examples

FA M_2 (Figure 1.8 page 37).

FA M_3 (Figure 1.10, page 38)

FA M_4 (Figure 1.12 page 38).

FA M_5 (Figure 1.13 page 39).

Example 1.15 page 40, when number of states is too large (or unspecific) to draw. Similar to M_5 but modulo i instead of 3.

Formal definition of computation

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton.
Let $w = w_1 w_2 \ldots w_n$ be a string where each symbol $w_i \in \Sigma$.
Then M *accepts* w if a sequence states $r_0, r_1, \ldots, r_n$ in Q exist with three conditions:

1. $r_0 = q_0$;
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, 1, \ldots, n-1$, and
3. $r_n \in F$.

We say M *recognizes language* L if $L = \{w : M \text{ accepts w}\}$.

Definition 1.16

A language is called a *regular language* if it can be recognized by some finite automaton.

Some basic questions:

1. Is $\{\epsilon\}$ regular?
2. Is Σ^* regular?
3. Is $\{\}$ regular?
4. Can two FA recognize the same languages?
5. Is L always regular, for any L that is finite?

Designing Finite Automata

- put yourself in the position of an automaton
- small amount of memory (which is a few states) to use
- scan the input from left to right

e.g., construct an FA to recognize the language consisting of all strings, each containing an odd number 1s.

- only need to pay attention to 1s.
- the current state is either even or odd
- switching state if additional 1 is encountered.

Figures 1.18, 1.19 (page 42), and 1.20 (page 43).

Another example, construct an FA to recognize all strings that contain 001 as a substring.

four possibilities:

1. have not see any symbols of the pattern 001
2. have seen just a 0
3. have seen 00
4. have seen 001

You may assign 4 states for these 4 possibilities.

Figure 1.22 (page 43)

The Regular Operations

There are some set operations to consider:

UNION
INTERSECTION
COMPLEMENT
CONCATENATION

e.g,

L_1 contains all strings with substring 11
L_2 contains all strings with substring 000

$L_1 \cup L_2$
$L_1 \cap L_2$
$\overline{L_1}$
$L_1 L_2$

Consider the recognitions by two FAs at the same time

$M_1 = (Q_1, \Sigma, \delta_1, q_0, F_1)$
$M_2 = (Q_2, \Sigma, \delta_2, p_0, F_2)$

for input: $b_1 b_2 \dots b_n$

$q_0 b_1 q_1 b_2 q_2 \dots b_n q_n$
$p_0 b_1 p_1 b_2 p_2 \dots b_n p_n$

In the UNION case, it suffices if either sequence holds

In the INTERSECTION case, both sequences should hold.

Technically, we can define a new FA, whose states are $[q, p]$

How do we define the transition function?

$[q, p]$ and symbol a, transit to $[q', p']$

where $q' = \delta_1(q, a), p' = \delta_2(p, a)$

For UNION, we need $q_n \in F_1$ or $p_n \in F_2$

For INTERSECTION, we need $q_n \in F_1$ AND $p_n \in F_2$

$L_1 = \{x : x \text{ contains substring } 1\}$
$L_2 = \{x : x \text{ contains substring } 00\}$

with the corresponding FA M_1 and M_2.

M_1 has start state q and accepting state q_1

	0	1
q	q	q_1
q_1	q_1	q_1

M_2 has start state p, state p_0 accepting state p_{00}

	0	1
p	p_0	p
p_0	p_{00}	p
p_{00}	p_{00}	p_{00}

Then the new machine M has 6 states:

$[q, p], [q, p_0], [q, p_{00}], [q_1, p], [q_1, p_0], [q_1, p_{00}]$ with the transition function:

	0	1
$[q, p]$	$[q, p_0]$	$[q_1, p]$
$[q, p_0]$	$[q, p_{00}]$	$[q_1, p_{00}]$
$[q, p_{00}]$	$[q, p_{00}]$	$[q_1, p_{00}]$
$[q_1, p]$	$[q_1, p_0]$	$[q_1, p]$
$[q_1, p_0]$	$[q_1, p_{00}]$	$[q_1, p]$
$[q_1, p_{00}]$	$[q_1, p_{00}]$	$[q_1, p_{00}]$

For accepting $L_1 \cap L_2$, $[q_1, p_{00}]$ is the only accepting state

For accepting $L_1 \cup L_2$, there are 4 states as accepting states.
what are they?

How to construct an FA for complement of L, if L is regular?

- flipping all states between accepting to rejecting, does it work?

How to construct an FA for concatenation of two regular languages?

- making the accepting state of the first FA the same as
  the start state of the second FA, does it work?

Need a notion of *nondeterminism* to make things easier.

1.2 NONDETERMINISM

What is deterministic computation?

The state to be transited to is completely determined by
- the current state and
- the current symbol

Figure 1.28 (page 49)

How about “nondeterministic computation’?

- there are possibly more than one state option to transit to
- more than one “computation path” for the same input
- the input is accepted if one of the paths accepts it.

Figure 1.20 (page 49)

"Signatures" of a nondeterministic FA

- for the same symbol, more than one transitions from a state
- a transition is labeled with the empty string/symbol ϵ

e.g., nondeterministic FA N_1 in Figure 1.27 page 48

- from q_1, symbol 1 allows to stay in q_1 or transit to q_2.
- what does the empty symbol on a transition mean?

Examples:

Construct a DFA for all strings whose third position is a 1

relatively easy, but how?

Construct a DFA for all strings whose third position from the end is a 1

a little hard, why?

how about construct an NFA?

nondeterministic computation can do "guessing".
Figure 1.31 page 51.
compared to Figure 1.32 on the same page

More examples: Figure 1.34 (page 52) and Figure 1.36 (page 53)

Formal Definition of a nondeterministic FA

Definition 1.37 (page 35)

A *nondeterministic finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the *states*,
2. Σ is a finite set called the *alphabet*,
3. $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the *transition function*,
4. $q_0 \in Q$ is the *start state*, and
5. $F \subseteq Q$ is the *set of accept states*.

where

$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$

$\mathcal{P}(Q)$ is the power set of Q, also written as 2^Q

Examples of NFAs

Example 1.38 (page 54)

Formal definition of computation with NFA

Let $N = (Q, \Sigma_\epsilon, \delta, q_0, F)$ be a nondeterministic finite automaton.
Let $w = w_1 w_2 \ldots w_n$ be a string where each symbol $w_i \in \Sigma_\epsilon$.
Then N *accepts* w if a sequence states $r_0, r_1, \ldots, r_n$ in Q exist with three conditions:

1. $r_0 = q_0$;
2. $r_{i+1} \in \delta(r_i, w_{i+1})$, for $i = 0, 1, \ldots, n-1$, and
3. $r_n \in F$.

We say N *recognizes language* L if $L = \{w : N \text{ accepts } w\}$.

Equivalence of NFA and DFA

An DFA is an NFA. This is because

(1) $\Sigma \subseteq \Sigma_\epsilon$
(2) $\delta(q, a) = p$ can be redefined as $\delta(q, a) = \{p\}$.

To show NFA are not more powerful than DFA,
we only need to show

that for every NFA, there is a DFA accepting the same language.

Theorem 1.39 (page 55)

Every NFA has an equivalent DFA.

Proof idea

To use a DFA to keep tracking on all the transitions of the NFA and to "simulate the latter's computation"

- assume the NFA has k states
- given state q and symbol a, there may be k or less transitions
- that is a subset of the k states
- each such subset is remembered with one state in the DFA, so the DFA needs 2^k states

E.g., 3 states in the NFA,
$2^3 = 8$ states: $p_{000} \sim p_{111}$ are in the DFA

where p_{011} is to remember subset $\{q_2, q_3\}$

The transition is explained with the example:

$\delta'(p_{011}, a) = p_{101}$ because

$\delta(q_2, a) \subseteq \{q_1, q_3\}$ and
$\delta(q_3, a) \subseteq \{q_1, q_3\}$

Example: NFA N that has two states: start state q and accepting state q_1, with transition function δ as:

	0	1
q	$\{q\}$	$\{q, q_1\}$
q_1	$\{q\}$	$\{\}$

(What language does N recognize by the way?)

In the DFA M, there will be 2^2 states: $p_{00}, p_{01}, p_{10}, p_{11}$

	0	1
p_{00}		
p_{01}	p_{10}	p_{00}
p_{10}	p_{10}	p_{11}
p_{11}	p_{10}	p_{11}

with p_{01}, p_{11} as accepting states and p_{10}

Note:

- p_{01} is a unnecessary state since there is no in-coming transition
- p_{00} is a unnecessary state since there is no out-going transition
  and it is not an accepting state

- In M, a state is an accepting state if it represents the subset of states in N that contains an accepting state of N.

- In M, the state is designated as the start state if its represents the subset of states in N that contains exact the start state of N.

- **But in all these, we did not consider ϵ transitions.**

Proof of Theorem 1.39

Assume $N = (Q, \Sigma_\epsilon, \delta, q_0, F)$ be the NFA recognizing a language L.

We construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ to recognize L.

1. Q' contains $2^{|Q|}$ states, one for each subset of Q
2. Let $p_R \in Q'$, where $R \subseteq Q$ representing a subset of Q.
   define $\delta'(p_R, a) = p_S$, where
   $$S = \{q : q \in \delta(r, a), \ \text{for some } r \in R\}$$
3. $q_0' = p_{\{q_0\}}$
4. $F' = \{p_R : R \text{ contains an accepting state of } N\}$

When ϵ transitions are involved

2. Let $p_R \in Q'$, where $R \subseteq Q$ representing a subset of Q.
   define $\delta'(p_R, a) = p_S$, where
   $$S = \{q : q \in \mathbf{E}(\delta(r, a)), \text{ for some } r \in R\}$$
3. $q_0' = p_{\mathbf{E}(\{q_0\})}$

where $\mathbf{E}(S)$ includes all the states in S and those reachable through the ϵ transitions from the states in S.

Apparently M simulates the computation of N on the input and accepts it iff N accepts it.

Corollary 1.40 (page 56)

A language is regular if and only if some NFA recognizes it.

Example: 1.41 (page 57), Figure 1.42.

constructing a DFA for an NFA:

- determine the states for the DFA
- determine the start and accepting states
- determine the transition function
- simplify the DFA by removing unnecessary states

Regular Operations (revisited)

Theorem 1.45 (page 59)

The class of regular languages is closed under the union operation.

(What does it mean that a class is closed under certain operation?)

Proof: construct an NFA to recognize $L_1 \cup L_2$.

Figure 1.46 (page 59)

Theorem 1.46 (page 60)

The class of regular languages is closed under the concatenation operation.

(concatenation of two languages L_1L_2 or $L_1 \circ L_2$)

Proof: construct an NFA to recognize L_1L_2.

Figure 1.48 (page 61)

Start operation * (transitive closure):

$L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$

Theorem 1.49 (page 62)

The class of regular languages is closed under the start operation.

Proof construct an NFA to recognize L^*.

Figure 1.50 page 62.

Various issues arising from the first homework assignment

empty language $\phi = \{\}$, size (i.e., cardinality) $|\phi| = 0$

language $E = \{\epsilon\}$, size $|E| = 1$

ϵ is a string of length 0, i.e., $|\epsilon| = 0$.

‘ ’ is a string of length 1, i.e., $|\text{‘ ’}| = 1$

FA for ϕ, and

FA for E.

1.3 REGULAR EXPRESSIONS

Used to bridge between regular languages and finite automata.

- sometimes we may describe a regular language vaguely,
- but want to get a precise definition
- without resorting to building an FA

e.g.,

a language in which each string contains either substring `11` or `001`

expressed as:

"anything" followed by either `00` or `001` followed by "anything"

But how to express “anything”?

note for this we should represent a set of “all things”

1. Use 0 to represent the set of single element 0, likewise, 1 for $\{1\}$.

2. Use * to represent “repeats”, e.g., 1^* is the set of strings, each consisting of k many 1's, for some $k \geq 0$. That is

$$1^* = \{\epsilon, 1, 11, 111, \ldots\}$$

3. Use $\cup$ for 'OR', e.g., $1 \cup 0$ represents $\{0, 1\}$

4. Use $\circ$ for 'concatenation',
e.g., $0 \circ 1^*$ represents $\{0, 01, 011, 0111, \ldots\}$, or
simply with $\circ$ removed: 01^*

5. Use '(', ')' whenever needed (as in arithmetic expressions).

More examples:

$(1 \cup 0) \circ (1 \cup 0)$ represents $\{00, 01, 10, 11\}$

$(0 \cup 1)^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\} = \Sigma^*$

$(0 \cup 1)^* \circ 1 \circ 1 \circ 1 \circ (1 \cup 0)^*$, or simply $\Sigma^* 111 \Sigma^*$

$0\Sigma^*0 \cup 1\Sigma^*1 \cup 1 \cup 0$

$(0 \cup \epsilon)1^* = 01^* \cup 1^*$

$(0 \cup \epsilon)(1 \cup \epsilon) = \{\epsilon, 1, 0, 01\}$

$1 * \phi = \phi$

ϕ^*

Formal Definition of a Regular Expression

Definition 1.52 (page 64)

R is a *regular expression* if R is one of the followings:

1. a symbol $q \in \Sigma$ the alphabet
2. ϵ
3. ϕ
4. $(R_1 \cup R_2)$, where R_1, R_2 are regular expressions
5. $(R_1 \circ R_2)$, where R_1, R_2 are regular expressions

6 (R_1^*), where R_1 is a regular expression.

Without the parentheses, evaluation of a regular expression is done in the precedence order: star, then concatenation, then union.

Also we use R^+ for RR^*, for example, $1^+ = \{1, 11, 111, \ldots\}$

Additional special cases:

$R \cup \epsilon =?$

$R \circ \epsilon =?$

$R \cup \phi =?$

$R \circ \phi =?$

Equivalence with Finite Automata

Theorem 1.54 (page 66)

A language is regular if and only some regular expression describe it.

It has two directions (stated in the following lemmas.)

Lemma 1.55 (page 67)

If a language is described by a regular expression, then it is regular.

Lemma 1.60 (page 69)

If a language is regular, then it is described by a regular expression.

Lemma 1.55 (page 67)

If a language is described by a regular expression, then it is regular.

proof idea

Converting a regular expression R into an NFA by using the constructing rules of regular expressions and identifying the atomic components of R.

Example:

building an NFA for regular expression $(ab \cup a)^*$ (Figure 1.57, page 68).

Lemma 1.55 (page 67)

If a language is described by a regular expression, then it is regular.

Proof: [*Using structural induction*]

We consider the six cases that R may have been constructed:

1. $R = a$ for some $a \in \Sigma$. We have an FA for R (page 67).
2. $R = \epsilon$, the set with single string ϵ. We also have an FA for R
3. $R = \phi$. the empty set. We have an FA for R.

4 $R = R_1 \cup R_2$

5. $R = R_1 \circ R_2$
6. R_1^*

The last three cases are proved using the proofs that the class of regular languages is closed under union, concatenation, and star operations. (page 67)

Another example: Building an NFA for regular expression $(a \cup b)^*aba$ (Figure 1.59, page 69)

Lemma 1.60 (page 69)

If a language is regular, it is described by a regular expression.

Proof:

The idea is to convert the DFA for the regular language
- to a *generalized* NFA
- then to a 2-state GNFA, which is a regular expression

A GNFA is an NFA whose transition edges have regular expressions instead of just symbols from the alphabet.

E.g., Figure 1.61 (page 70)

We require GNFA to meet the following condition:

- no incoming edge to the start state; (how to satisfy this?)

- only one accepting state; (how to satisfy this?)

- between every pair of states, there are two edges; (how to satisfy this?)

- every state has a self-loop edge; (how to satisfy this?)

Now constructing such a GNFA, using the following steps:

(1) Let DFA M be the one to recognize the regular language L; make it a GNFA G

(2) If G has only two states, then they are start and accept states. The regular expression on the edge is the desired expression. Stop.

(3) Remove one state q_{rip} from N and repair so the new G recognizes the same language:

- q_{rip} is neither the start state nor accepting state.

- assume q_i and q_j to be two states connecting to q_{rip}, i may be the same as j

- re-designate regular expressions over edges between q_i and q_j by considering the paths between q_i and q_j through q_{rip}

Proof:

Need a formal definition for GNFA. The transition function is slightly different now:

$$\delta : (Q - \{q_{accept}\}) \times (Q - \{Q_{start}\}) \rightarrow \mathcal{R}$$

which designates one regular expression to every directed edge, where $\mathcal{R}$ is the set of all regular expressions.

Definition 1.64 (page 73)

A *generalized NFA* is a 5-tuple $(Q, \Sigma, \delta, q_{start}, q_{accept})$, where

1. Q is the finite set of states;
2. Σ is the input alphabet;
3. δ is transition function: $(Q - \{q_{accept}\}) \times (Q - \{Q_{start}\}) \rightarrow \mathcal{R}$;
4. q_{start} is the start state;
5. $_{accept}$ is the accepting state.

The GNFA accepts a string $w = w_1 w_2 \ldots w_k$, where $w_i \in \Sigma^*$ if there is a sequence of states $q_0 q_1 \ldots q_k$ such that

(1) $q_0 = q_{start}$

(2) $q_k = q_{accept}$

(3) $w_i \in$ set of strings represented by regular expression R_i, where $R_i = \delta(q_{i-1}, q_i)$, for all $i = 1, 2, \ldots, k$.

THe proof consists of the following steps to convert M, the DFA for the original regular language, to a GNFA.

- add new start and accepting states to M, let the GNFA be G
- call the procedure CONVERT(G) recursively.

CONVERT(G):

1. Let k be the number of states in G;
2. If $k = 2$, then a single edge connecting from q_{statr} to q_{accept} with label R, return R and stop.
3. Select q_{rip}, for every pair of states q_i, q_j that are not q_{rip}, or start, or accept state, set
   $\delta'(q_i, q_j) = R_1 R_2 * R_3 \cup R_4$,
   where $R_1 = \delta(q_i, q_{rip})$, $R_2 = \delta(q_{rip}, q_{rip})$,
   $R_3 = \delta(q_{rip}, q_j)$, $R_4 = \delta(q_i, q_j)$
4. Let $G' = (Q - \{q_{rip}, \Sigma, \delta', q_{start}, q_{accept})$
5. Call CONVERT(G').

- prove that for any G, CONVERT(G) is equivalent to G.

We prove this claim by induction on k.

basis: $k = 2$. The claim is true since the regular expression on the only edge described the same lang.

assumption: the claim is true for $k-1$ states.

induction: k states. We show removing state q_{rip} still allows the recognition of the same language.

Let w be accepted by the k state GNFA, with path $q_{start}, q_1, q_2, \ldots, q_{accept}$

Case 1. q_{rip} does not occur in it.

Case 2. q_{rip} occurred in it.

CONVERT(G) step 3 guarantees the claim true.

Examples:

Figure 1.67 Converting a 2-state DFA to an equivalent regular expression (page 75)

Figure 1.69 Converting a 3-state DFA to an equivalent regular expression (page 76)

1.4 NONREGULAR LANGUAGES

To answer the following related questions:

- How powerful is a finite state machine?
- What problems may (not) be defined as regular languages?
- What makes a finite state (not) powerful?
- what languages cannot be recognized by FA?

Revisit the case of matching parentheses in arithmetic expressions

It is all because the small memory FA are limited to have!

Examples:

$L_1 = \{w : w \text{ contains the same number of 1 and 0}\}$.

$L_2 = \{0^k 1^k : k \geq 0\}$

$L_3 = \{ww^{-1} : w \in \Sigma\}$

Why these languages may not be regular?

Consider L_2, when $x = 0^k 1^k$ is long enough, say

$|x| = 2k = l > |Q|$

Then the path leading to x's acceptance would go through the same state **at least twice**, i.e., the path is NOT a simple path

- there is a circular subpath on this path
- it accommodates "non-empty string"
- it is not "too long"
- it makes the acceptance of "abnormal" strings possible

The Pumping Lemma For Regular Languages

Theorem 1.70 (Pumping lemma) (page 78)

If A is a regular language, then there is a number p (the pumping length) where, for any string $s \in A$ of length at least p, s can be divided into three pieces, i.e., $s = xyz$, such that

1. *for each $i \geq 0$, $xy^iz \in A$,*
2. *$|y| > 0$, and*
3. *$|xy| \leq p$*

proof idea

- $p = |Q|$
- pigeonhole principle to get repetition of a state in the path
- the string between repetition of the state is not empty
- the first $p + 1$ states must contain a repetition.

Proof

Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA that recognizes A and $p = |Q|$.

Let $s = s_1 s_2 \ldots s_n$, $n \geq p$.
Let $r_1, r_2, \ldots, r_{n+1}$ be the states on the path to accept s
Among the first $p + 1$ states, two must be the same: $r_j = r_l$, $j \neq l$
Let $x = s_1 \ldots s_{j-1}$, $y = s_j \ldots s_{l-1}$, and $z = s_l \ldots s_n$.

Assume r_{n+1} an accept state.

Then M must accept $xy^i z$ for any $i \geq 0$.

$|y| > 0$ because $j \neq l$.

$|xy| \leq p$ because $l \leq p + 1$.

Using the Pumping lemma to prove certain languages are not regular

Example 1.73 (page 80)

Show that $L_2 = \{0^k 1^k : k \geq 0\}$ is not regular.

Example 1.74

Show that $L_1 = \{w : w$ contains the same number of 1 and 0$\}$. is not regular [Note the use of condition 3]

Example 1.75 (page 81)

Show that $L_3 = \{ww : w \in \Sigma\}$ is not regular