CS311: Computational Theory

Lecture 5: Regular Languages Applications

Dr. Manal Helal, Spring 2014. http://moodle.manalhelal.com

Lecture Learning Objectives

1. *Use Regular Languages in problem solving*

Decision Procedures

A decision procedure is an algorithm whose result is a Boolean value. It must:

- Halt
- Be correct

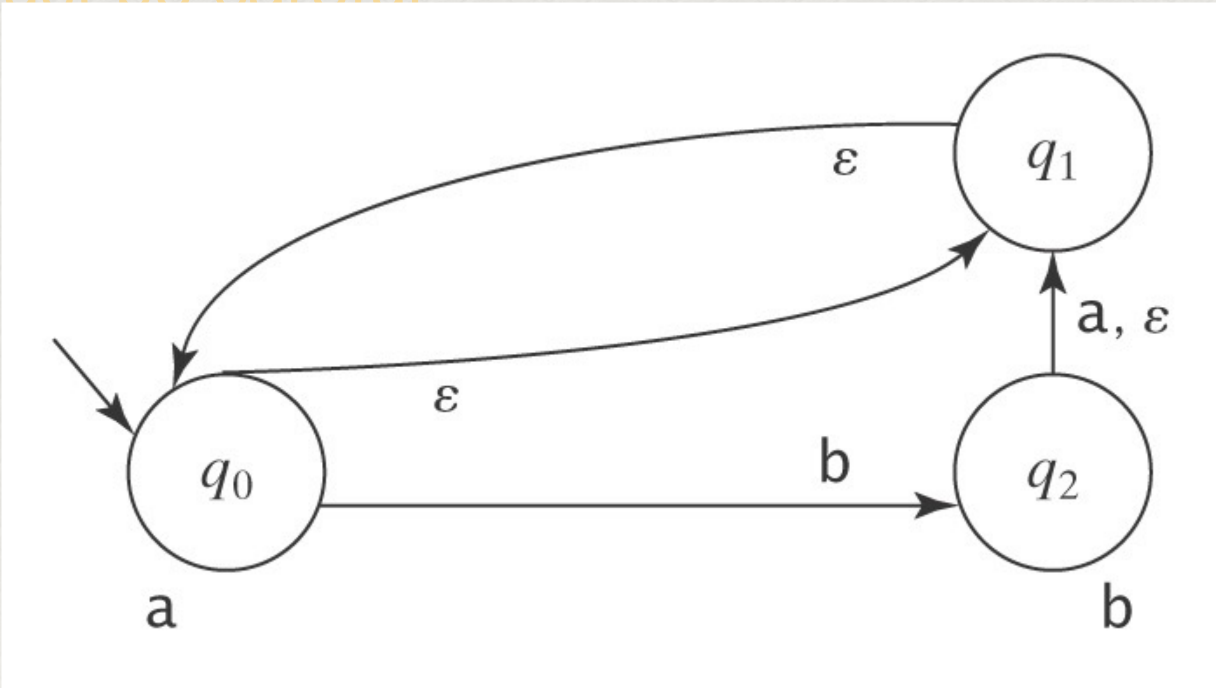
Important decision procedures exist for regular languages:

- Given an FSM M and a string s, does M accept s?
- Given a regular expression α and a string w, does α generate w?

Membership

We can answer the membership question by running an FSM.

But we must be careful:



Membership

decideFSM(M: FSM, w: string) =
If ndfsmsimulate(M, w) accepts then return True
else return False.

decideregex(α: regular expression, w: string) =
From α, use regextofsm to construct an FSM M such that $L(\alpha) = L(M)$.
Return decideFSM(M, w).

Emptiness and Finiteness

- Given an FSM M, is L(M) empty?
- Given an FSM M, is $L(M) = \Sigma_M{}^*$?
- Given an FSM M, is L(M) finite?
- Given an FSM M, is L(M) infinite?
- Given two FSMs M_1 and M_2, are they equivalent?

Emptiness

Given an FSM M, is L(M) empty?

- The graph analysis approach:

- The simulation approach:

Emptiness

Given an FSM M, is L(M) empty?

- The graph analysis approach:

1. Mark all states that are reachable via some path from the start state of M.
2. If at least one marked state is an accepting state, return False. Else return True.

Emptiness

Given an FSM M, is L(M) empty?

- The simulation approach:

1. Let M' = ndfsmtodfsm(M).
2. For each string w in Σ^* such that $|w| < |K_{M'}|$ do:
   Run decideFSM(M', w).
3. If M' accepts at least one such string, return False. Else return True.

Totality

- Given an FSM M, is $L(M) = \Sigma_M{}^*$?

1. Construct M' to accept $\neg L(M)$.
2. Return emptyFSM(M').

Finiteness

Given an FSM M, is L(M) finite?

- The graph analysis approach:

Finiteness

Given an FSM M, is L(M) finite?

- The graph analysis approach:

The mere presence of a loop does not guarantee that L(M) is infinite. The loop might be:

- labeled only with ε,
- unreachable from the start state, or
- not on a path to an accepting state.

Finiteness

Given an FSM M, is L(M) finite?

- The graph analysis approach:

1. $M' = \text{ndfsmtodfsm}(M)$.
2. $M'' = \text{minDFSM}(M')$.
3. Mark all states in M'' that are on a path to an accepting state.
4. Considering only marked states, determine whether there are any cycles in M''.
5. If there are cycles, return True. Else return False.

Finiteness

Given an FSM M, is L(M) finite?

- The simulation approach:

1. $M' = \text{ndfsmtodfsm}(M)$.
2. For each string w in Σ^* such that $|K_{M'}| \leq w \leq 2 \cdot |K_{M'}| - 1$ do:

   Run decideFSM(M', w).
3. If M' accepts at least one such string, return False.

   Else return True.

Equivalence

- Given two FSMs M_1 and M_2, are they equivalent? In other words, is $L(M_1) = L(M_2)$? We can describe two different algorithms for answering this question.

Equivalence

- Given two FSMs M_1 and M_2, are they equivalent? In other words, is $L(M_1) = L(M_2)$?

equalFSMs$_1$(M_1: FSM, M_2: FSM) =

1. M_1' = buildFSMcanonicalform(M_1).
2. M_2' = buildFSMcanonicalform(M_2).
3. If M_1' and M_2' are equal, return True, else return False.

Equivalence

- Given two FSMs M_1 and M_2, are they equivalent? In other words, is $L(M_1) = L(M_2)$?

Observe that M_1 and M_2 are equivalent iff:

$$(L(M_1) - L(M_2)) \cup (L(M_2) - L(M_1)) = \varnothing.$$

equalFSMs$_2$(M_1: FSM, M_2: FSM) =

1. Construct M_A to accept $L(M_1) - L(M_2)$.
2. Construct M_B to accept $L(M_2) - L(M_1)$.
3. Construct M_C to accept $L(M_A) \cup L(M_B)$.
4. Return emptyFSM(M_C).

Minimality

- Given DFSM M, is M minimal?

Minimality

- Given DFSM M, is M minimal?

1. $M' = \text{minDFSM}(M)$.
2. If $|K_M| = |K_{M'}|$ return True; else return False.

Answering Specific Questions

Given two regular expressions α_1 and α_2, is:

$$(L(\alpha_1) \cap L(\alpha_2)) - \{\varepsilon\} \neq \varnothing?$$

Answering Specific Questions

Given two regular expressions α_1 and α_2, is:

$$(L(\alpha_1) \cap L(\alpha_2)) - \{\varepsilon\} \neq \varnothing?$$

1. From α_1, construct an FSM M_1 such that $L(\alpha_1) = L(M_1)$.
2. From α_2, construct an FSM M_2 such that $L(\alpha_2) = L(M_2)$.
3. Construct M' such that $L(M') = L(M_1) \cap L(M_2)$.
4. Construct M_ε such that $L(M_\varepsilon) = \{\varepsilon\}$.
5. Construct M'' such that $L(M'') = L(M') - L(M_\varepsilon)$.
6. If $L(M'')$ is empty return False; else return True.

Answering Specific Questions

Given two regular expressions α_1 and α_2, are there at least 3 strings that are generated by both of them?

Summary of Algorithms

- Operate on FSMs without altering the language that is accepted:
  - Ndfsmtodfs
  - MinDFSM

Summary of Algorithms

- Compute functions of languages defined as FSMs:
  - Given FSMs M_1 and M_2, construct a FSM M_3 such that $L(M_3) = L(M_2) \cup L(M_1)$.
  - Given FSMs M_1 and M_2, construct a new FSM M_3 such that $L(M_3) = L(M_2)\, L(M_1)$.
  - Given FSM M, construct an FSM M* such that $L(M^*) = (L(M))^*$.
  - Given a DFSM M, construct an FSM M* such that $L(M^*) = \neg L(M)$.
  - Given two FSMs M_1 and M_2, construct an FSM M_3 such that $L(M_3) = L(M_2) \cap L(M_1)$.
  - Given two FSMs M_1 and M_2, construct an FSM M_3 such that $L(M_3) = L(M_2) - L(M_1)$.
  - Given an FSM M, construct an FSM M* such that $L(M^*) = (L(M))^R$, (i.e., the reverse of L(M)).
  - Given an FSM M, construct an FSM M* that accepts letsub(L(M)), where letsub is a letter substitution function.

Algorithms, Continued

- Converting between FSMs and regular expressions:
  - Given a regular expression α, construct an FSM M such that:

    $$L(\alpha) = L(M)$$

  - Given an FSM M, construct a regular expression α such that:

    $$L(\alpha) = L(M)$$

- Algorithms that implement operations on languages defined by regular expressions: any operation that can be performed on languages defined by FSMs can be implemented by converting all regular expressions to equivalent FSMs and then executing the appropriate FSM algorithm.

Algorithms: Decision Procedures

- Decision procedures that answer questions about languages defined by FSMs:
  - Given an FSM M and a string s, decide whether s is accepted by M.
  - Given an FSM M, decide whether L(M) is empty.
  - Given an FSM M, decide whether L(M) is finite.
  - Given two FSMs, M_1 and M_2, decide whether $L(M_1) = L(M_2)$.
  - Given an FSM M, is M minimal?
- Decision procedures that answer questions about languages defined by regular expressions: Again, convert the regular expressions to FSMs and apply the FSM algorithms.

A Special Case of Pattern Matching

Suppose that we want to match a pattern that is composed of a set of keywords. Then we can write a regular expression of the form:

$(\Sigma^* (k_1 \cup k_2 \cup \ldots \cup k_n)\ \Sigma^*)^+$

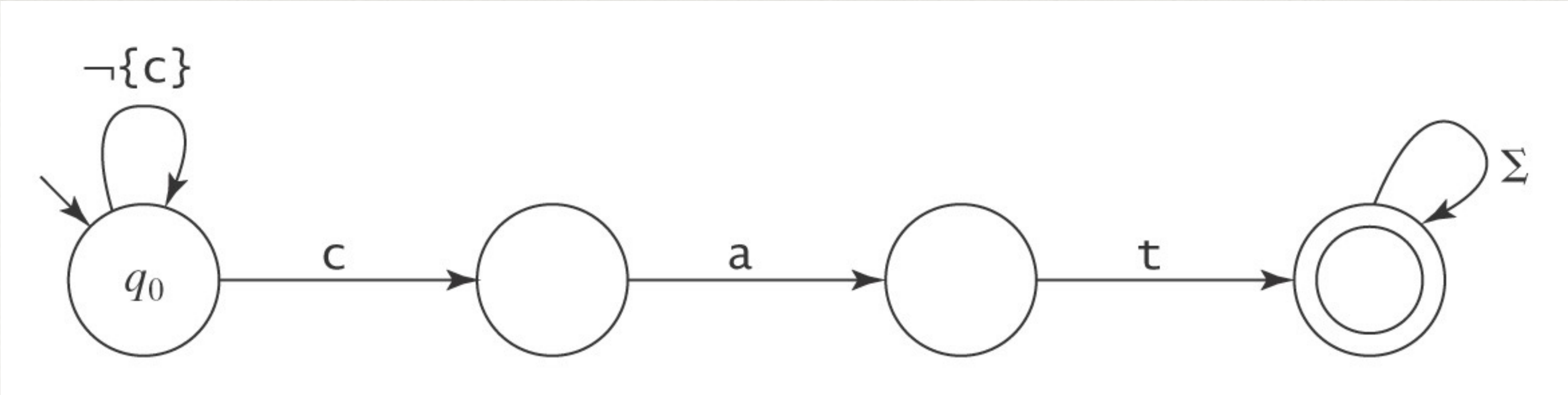
For example, suppose we want to match:

Σ^* `finite state machine` $\cup$
`FSM` $\cup$ `finite state automaton`Σ^*

We can use regextofsm to build an FSM. But …
We can instead use buildkeywordFSM.

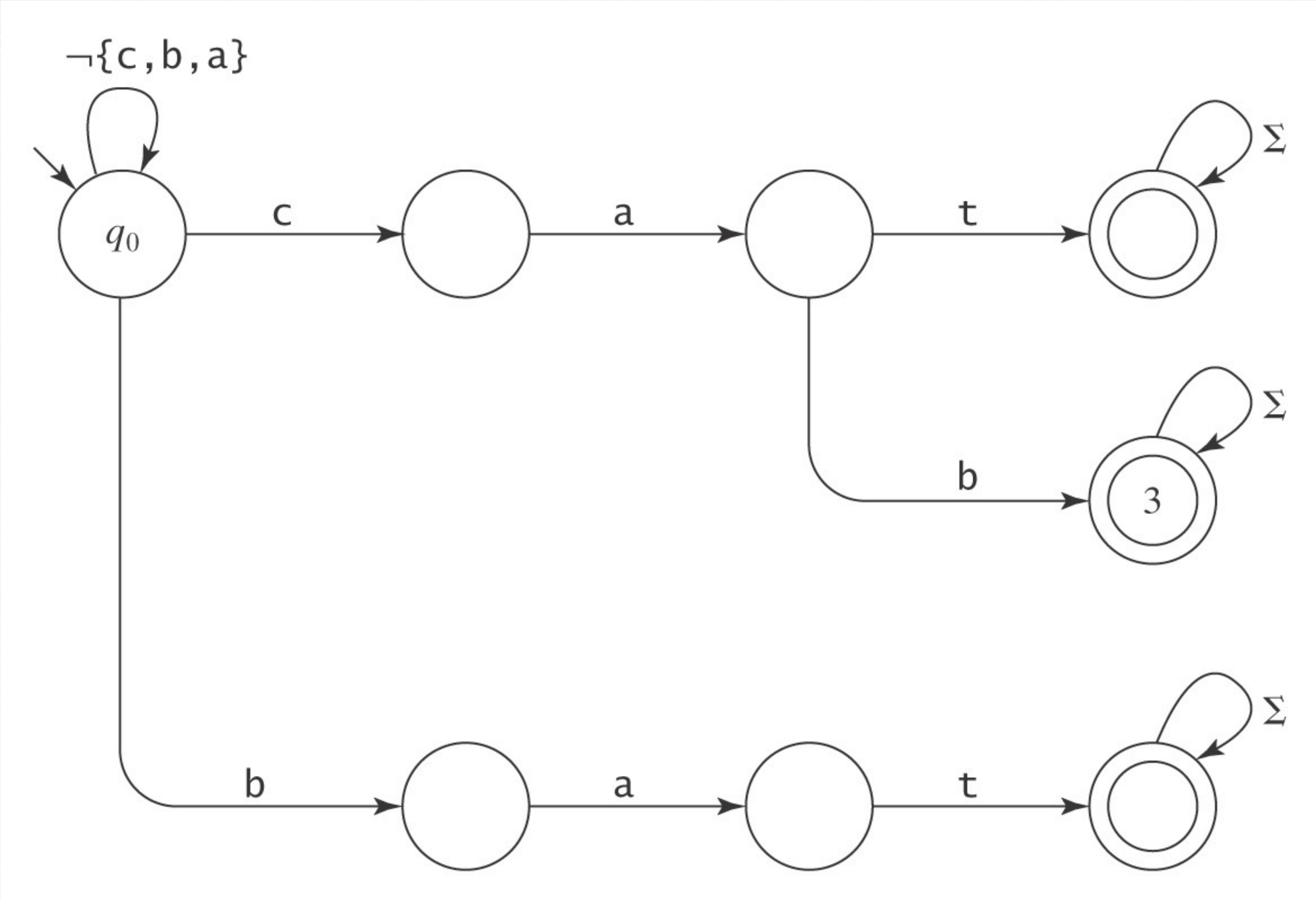
{cat, bat, cab}

The single keyword `cat`:



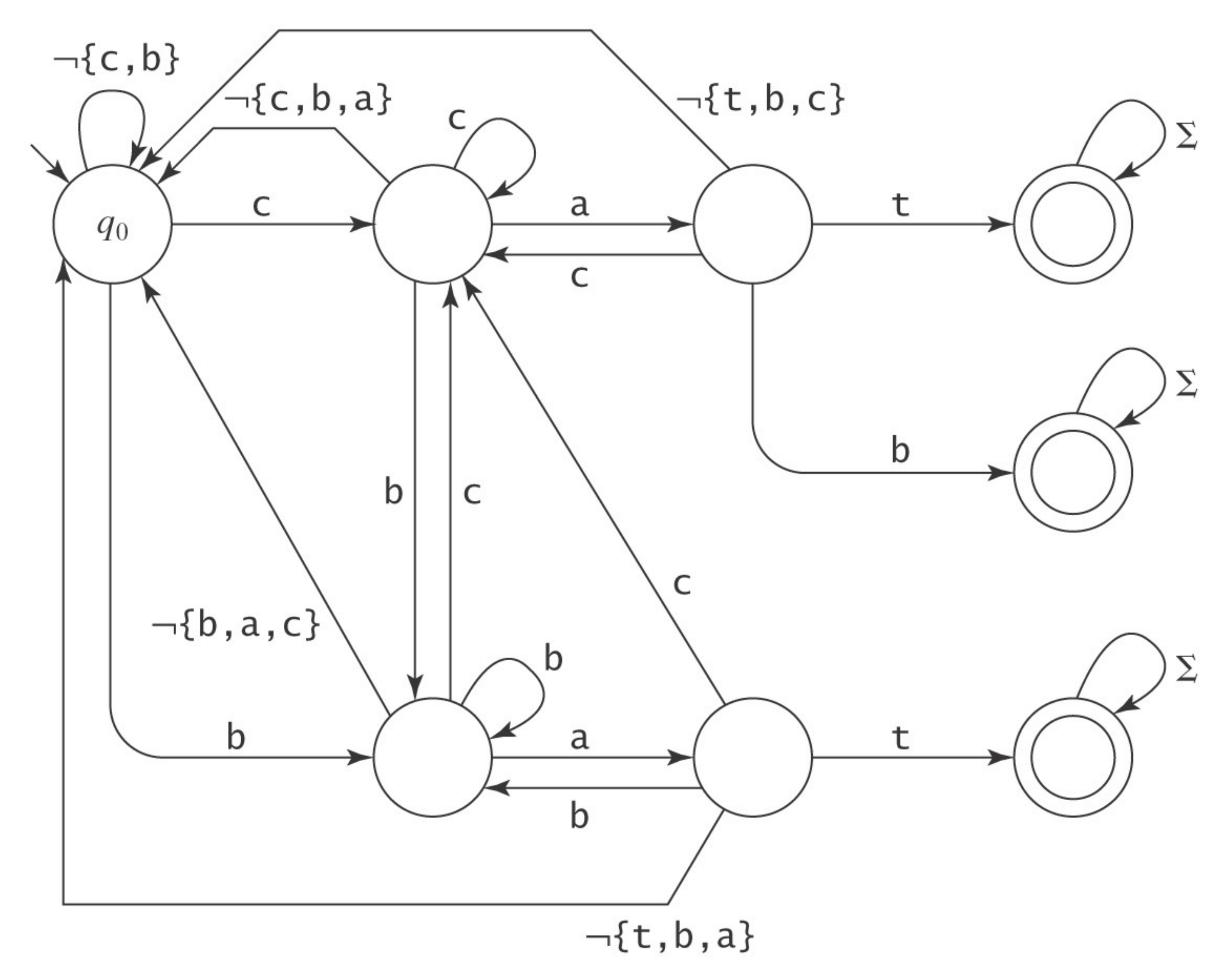
{cat, bat, cab}

Adding `bat`:



{cat, bat, cab}

Adding `cab`:



A Biology Example – BLAST

Given a protein or DNA sequence, find others that are likely to be evolutionarily close to it.

<u>ESGH</u>D<u>TTTYY</u>NKNRY<u>PAGWNN</u>HHDQM<u>FFW</u>V

Build a DFSM that can examine thousands of other sequences and find those that match any of the selected patterns.

Regular Expressions in Perl

<table>
<tr><th>Syntax</th><th>Name</th><th>Description</th></tr>
<tr><td>abc</td><td>Concatenation</td><td>Matches a, then b, then c, where a, b, and c are any regexs</td></tr>
<tr><td>a | b | c</td><td>Union (Or)</td><td>Matches a or b or c, where a, b, and c are any regexs</td></tr>
<tr><td>a*</td><td>Kleene star</td><td>Matches 0 or more a’s, where a is any regex</td></tr>
<tr><td>a+</td><td>At least one</td><td>Matches 1 or more a’s, where a is any regex</td></tr>
<tr><td>a?</td><td></td><td>Matches 0 or 1 a’s, where a is any regex</td></tr>
<tr><td>a{n, m}</td><td>Replication</td><td>Matches at least n but no more than m a’s, where a is any regex</td></tr>
<tr><td>a*?</td><td>Parsimonious</td><td>Turns off greedy matching so the shortest match is selected</td></tr>
<tr><td>a+?</td><td>″</td><td>″</td></tr>
<tr><td>.</td><td>Wild card</td><td>Matches any character except newline</td></tr>
<tr><td>^</td><td>Left anchor</td><td>Anchors the match to the beginning of a line or string</td></tr>
<tr><td>$</td><td>Right anchor</td><td>Anchors the match to the end of a line or string</td></tr>
<tr><td>[a-z]</td><td></td><td>Assuming a collating sequence, matches any single character in range</td></tr>
<tr><td>[^a-z]</td><td></td><td>Assuming a collating sequence, matches any single character not in range</td></tr>
<tr><td>\d</td><td>Digit</td><td>Matches any single digit, i.e., string in [0-9]</td></tr>
<tr><td>\D</td><td>Nondigit</td><td>Matches any single nondigit character, i.e., [^0-9]</td></tr>
<tr><td>\w</td><td>Alphanumeric</td><td>Matches any single “word” character, i.e., [a-zA-Z0-9]</td></tr>
<tr><td>\W</td><td>Nonalphanumeric</td><td>Matches any character in [^a-zA-Z0-9]</td></tr>
<tr><td>\s</td><td>White space</td><td>Matches any character in [space, tab, newline, etc.]</td></tr>
</table>

Regular Expressions in Perl

Syntax	Name	Description
\S	Nonwhite space	Matches any character not matched by \s
\n	Newline	Matches newline
\r	Return	Matches return
\t	Tab	Matches tab
\f	Formfeed	Matches formfeed
\b	Backspace	Matches backspace inside []
\b	Word boundary	Matches a word boundary outside []
\B	Nonword boundary	Matches a non-word boundary
\0	Null	Matches a null character
\nnn	Octal	Matches an ASCII character with octal value nnn
\xnn	Hexadecimal	Matches an ASCII character with hexadecimal value nn
\cX	Control	Matches an ASCII control character
\char	Quote	Matches char; used to quote symbols such as . and \
(a)	Store	Matches a, where a is any regex, and stores the matched string in the next variable
\1	Variable	Matches whatever the first parenthesized expression matched
\2		Matches whatever the second parenthesized expression matched
…		For all remaining variables

Using Regular Expressions in the Real World

Matching numbers:

-? ([0-9]+(\.[0-9]*)? | \.[0-9]+)

Matching ip addresses:

([0-9]{1,3} (\ . [0-9] {1,3}){3})

Finding doubled words:

([A-Za-z]+) \s+ \1

From Friedl, J., Mastering Regular Expressions, O'Reilly,1997.

More Regular Expressions

Identifying spam:

```
\badv\(?ert\)?\b
```

Trawl for email addresses:

```
\b[A-Za-z0-9_%-]+@[A-Za-z0-9_%-]+ (\.[A-Za-z]+){1,4}\b
```

Using Substitution

Building a chatbot:

On input:

<phrase1> `is` <phrase2>

the chatbot will reply:

`Why is` <phrase1> <phrase2>`?`

Chatbot Example

<user> `The food there is awful`

<chatbot> `Why is the food there awful?`

Assume that the input text is stored in the variable `$text`:

```
$text =~
  s/^([A-Za-z]+)\sis\s([A-Za-z]+)\.?$/
    Why is \1 \2?/
;
```

Assignment 1

1. *Give state diagrams of DFAs recognizing the following languages. In all parts, the alphabet is {0,1}.*

a. ***{w|w begins with a 1 and ends with a 0}***
b. ***{w| w contains at least three 1s}***
c. ***{w| w contains the substring 0101 (i.e., w = x0101y for some x and y)}***
d. ***{w| w has length at least 3 and its third symbol is a 0}***

2. Give state diagrams of NFAs with the specified number of states recognizing each of the following languages. In all parts, the alphabet is {0,1}.

a. The language of Exercise 1.6c with five states:

- ***{w| w contains the substring 0101 (i.e., w = x0101y for some x and y)}***

b. The language of Exercise 1.6l with six states:

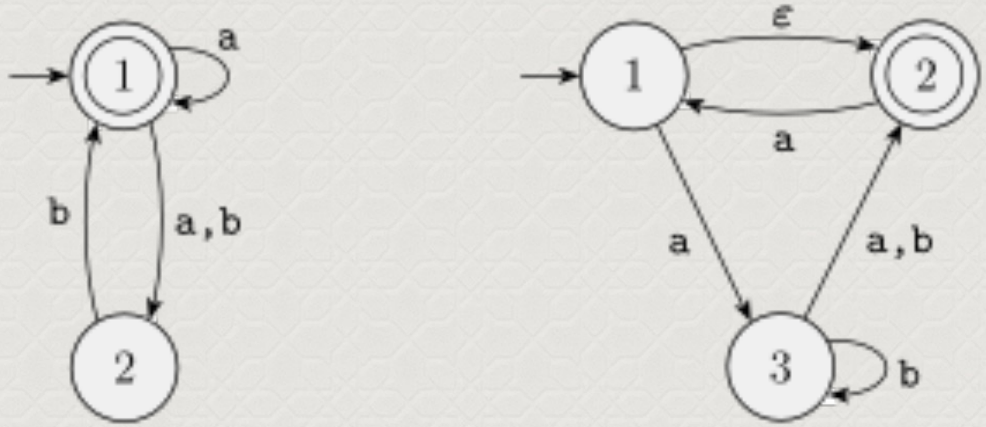
- ***{w|w contains an even number of 0s, or contains exactly two 1s}***

c. The language {0} with two states

d. The language $0^* 1^* 0^+$ with three states

Assignment 1 – Cont'd

3. Use the construction given in Theorem 1.39 to convert the following two nondeterministic finite automata to equivalent deterministic finite automata



(a) (b)

***4.** Use the procedure described in Lemma 1.55 to convert the following regular expressions to nondeterministic finite automata.*

***a.** $(0 \cup 1)^* 000 (0 \cup 1)^*$*

***b.** $(((00)^* (11)) \cup 01)^*$*

***c.** $\emptyset^*$*