Search and Decoding in Speech Recognition

Regular Expressions and Automata
Outline

◆ Introduction
◆ Regular Expressions
  ■ Basic Regular Expression Patterns
  ■ Disjunction, Grouping and Precedence
  ■ Examples
  ■ Advanced Operators
  ■ Regular Expression Substitution, Memory and ELIZA
◆ Finite-State Automata
  ■ Using an FSA to Recognize Sheeptalk
  ■ Formal Languages
  ■ Example
  ■ Non-Deterministic FSAs
  ■ Using an NFSA to Accept Strings
  ■ Recognition as Search
  ■ Relating Deterministic and Non-Deterministic Automata
◆ Regular Languages and FSAs
◆ Summary
Introduction

◆ Regular Expression (RE) – is a language for specifying text search strings.
  ▶ First developed by Kleene (1956)
  ▶ Requires a:
    ◆ Pattern – specification formula using a special language that specifies simple classes of strings.
    ◆ Corpus – a body of text to search through.
Introduction

Imagine that you have become a passionate fan of woodchucks. Desiring more information on this celebrated woodland creature, you turn to your favorite Web browser and type in woodchuck.

Your browser returns a few sites. You have a flash of inspiration and type in woodchucks. Instead of having to do this search twice, you would have rather typed one search command specifying something like woodchuck with an optional final s.

Or perhaps you might want to search for all the prices in some document; you might want to see all strings that look like $199 or $25 or $24.99.

In this chapter we introduce the regular expression, the standard notation for characterizing text sequences. The regular expression is used for specifying:

- text strings in situations like this Web-search example, and in other
- information retrieval applications, but also plays an important role in
- word-processing,
- computation of frequencies from corpora, and other such tasks.
Introduction

◆ Regular Expressions can be implemented via finite-state automaton.
◆ Finite-state automaton is one of the most significant tools of computational linguistics. Its variations:
  ■ Finite-state transducers
  ■ Hidden Markov Models, and
  ■ N-gram grammars

Important components of the Speech Recognition and Synthesis, spell-checking, and information-extraction applications that will be introduced in latter chapters.
Regular Expressions and Automata

Regular Expressions
Regular Expressions

- Formally, a regular expression is an algebraic notation for characterizing a set of strings.
  - Thus they can be used to specify search strings as well as to define a language in a formal way.
- Regular Expression requires
  - A pattern that we want to search for, and
  - A corpus of text to search through.

- Thus when we give a search pattern, we will assume that the search engine returns the line of the document returned. This is what the UNIX grep command does.
- We will underline the exact part of the pattern that matches the regular expression.
- A search can be designed to return all matches to a regular expression or only the first match. We will show only the first match.
Basic Regular Expression Patterns

The simplest kind of regular expression is a sequence of simple characters:

- `/woodchuck/`
- `/Buttercup/`
- `/!/`

<table>
<thead>
<tr>
<th>RE</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/woodchucks/</code></td>
<td>“interesting links to woodchucks and lemurs”</td>
</tr>
<tr>
<td><code>/a/</code></td>
<td>“Mary Ann stopped by Mona’s”</td>
</tr>
<tr>
<td><code>/Claire says,/</code></td>
<td>“Dagmar, my gift please,” Claire says,”</td>
</tr>
<tr>
<td><code>/song/</code></td>
<td>“all our pretty songs”</td>
</tr>
<tr>
<td><code>/!/</code></td>
<td>“You’ve left the burglar behind again!” said Nori</td>
</tr>
</tbody>
</table>
Basic Regular Expression Patterns

- Regular Expressions are **case sensitive**
  - `/s/`, is not the same as `/S/`
  - `/woodchucks/` will not match “Woodchucks”
- **Disjunction:** “[“ and “]”.

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/\[wW]oodchuck/</code></td>
<td>Woodchuck or woodchuck</td>
<td>“Woodchuck”</td>
</tr>
<tr>
<td><code>/\[abc]/</code></td>
<td>‘a’, ‘b’, or ‘c’</td>
<td>“In uomini, in soldati”</td>
</tr>
<tr>
<td><code>/\[1234567890]/</code></td>
<td>Any digit</td>
<td>“plenty of 7 to 5”</td>
</tr>
</tbody>
</table>
Basic Regular Expression Patterns

Specify **range** in Regular Expressions: “-”

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/\[A-Z]/</code></td>
<td>An uppercase letter</td>
<td>“we should call it ‘Drenched Blossoms’”</td>
</tr>
<tr>
<td><code>/\[a-z]/</code></td>
<td>A lower case letter</td>
<td>“my beans were impatient to be hoed!”</td>
</tr>
<tr>
<td><code>/\[0-9]/</code></td>
<td>A single digit</td>
<td>“Chapter 1: Down the Rabbit Hole”</td>
</tr>
</tbody>
</table>
Basic Regular Expression Patterns

- **Negative Specification** – *what pattern can not be*: “^”
  - If the first symbol after the open square brace “[“ is “^” the resulting pattern is negated.
  - Example /[^a]/ matches any single character (including special characters) except a.

<table>
<thead>
<tr>
<th>RE</th>
<th>Match (single characters)</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>/[^A-Z]/</td>
<td>Not an uppercase letter</td>
<td>“Oyfn pripetchik”</td>
</tr>
<tr>
<td>/[^Ss]/</td>
<td>Neither ‘S’ nor ‘s’</td>
<td>“I have no exquisite reason for ‘t’”</td>
</tr>
<tr>
<td>/[^\ ./]/</td>
<td>Not a period</td>
<td>“our resident Djinn”</td>
</tr>
<tr>
<td>/[^e]/</td>
<td>Either ‘e’ or ‘^’</td>
<td>“look up ^ now”</td>
</tr>
<tr>
<td>/a^b/</td>
<td>Pattern ‘a^b’</td>
<td>“look up a^b now”</td>
</tr>
</tbody>
</table>
Basic Regular Expression Patterns

- How do we specify both *woodchuck* and *woodchucks*?

- Optional character specification: `/?/`
- `/?/` means “the preceding character or nothing”.

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>/woodchucks?/</td>
<td>woodchuck or woodchucks</td>
<td>“woodchuck”</td>
</tr>
<tr>
<td>Colou?r</td>
<td>color or colour</td>
<td>“colour”</td>
</tr>
</tbody>
</table>
Basic Regular Expression Patterns

- Question-mark “?” can be thought of as “zero or one instance of the previous character”.
- It is a way to specify how many of something that we want.
- Sometimes we need to specify regular expressions that allow repetitions of things.
- For example, consider the language of (certain) sheep, which consists of strings that look like the following:
  - baa!
  - baaa?
  - baaaa?
  - baaaaa?
  - baaaaaa?
  - ...


Basic Regular Expression Patterns

◆ Any number of repetitions is specified by “*” which means “any string of 0 or more”.

◆ Examples:
  - /aa*/ - a followed by zero or more a’s
  - /[ab]*/ - zero or more a’s or b’s. This will match aaaa or abababa or bbbb
Basic Regular Expression Patterns

- We know enough to specify part of our regular expression for prices: multiple digits.
  - Regular expression for individual digit:
    - /\[0-9]/
  - Regular expression for an integer:
    - /\[0-9][0-9]*/
  - Why is not just /\[0-9]*/?
    - Because it is annoying to specify “at least once” RE since it involves repetition of the same pattern there is a special character that is used for “at least once”: “+”
  - Regular expression for an integer becomes then:
    - /\[0-9]+/
  - Regular expression for sheep language:
    - /baa*!/, or
    - /ba+!/

Basic Regular Expression Patterns

- One very important special character is the period: `/./`, a wildcard expression that matches any single character (except carriage return).
- Example: Find any line in which a particular word (for example Veton) appears twice:
  - `/Veton.*Veton/`

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/beg.n/</code></td>
<td>Any character between <code>beg</code> and <code>n</code></td>
<td><code>begin</code> <code>beg’n</code>, <code>begun</code></td>
</tr>
</tbody>
</table>
Anchors

Anchors are special characters that anchor regular expressions to particular places in a string.

- The most common anchors are:
  - “^” – matches the start of a line
  - “$” – matches the end of the line

Examples:
- `/^The/` - matches the word “The” only at the start of the line.
- Three uses of “^”:
  1. `/^xyz/` - Matches the start of the line
  2. `[^xyz]` – Negation
  3. `/^/` - Just to mean a caret
- `/\|$/` - “\” Stands for space “character”; matches a space at the end of line.
- `/^The\dog\.$/` - matches a line that contains only the phrase “The dog”.
Anchors

- \b - matches a word boundary
- \B - matches a non-boundary
- \bthe\b - matches the word “the” but not the word “other”.
- Word is defined as a any sequence of digits, underscores or letters.
- \b99\ - will match the string 99 in “There are 99 bottles of beer on the wall” but NOT “There are 299 bottles of beer on the wall” and it will match the string “$99” since 99 follows a “$” which is not a digit, underscore, or a letter.
Disjunction, Grouping and Precedence.

- Suppose we need to search for texts about pets; specifically we may be interested in cats and dogs. If we want to search for either “cat” or the string “dog” we can not use any of the constructs we have introduced so far (why not “[ ]”?).

- New operator that defines disjunction, also called the pipe symbol is “|”.

- /cat|dog/ - matches either cat or the string dog.
Grouping

◆ In many instances it is necessary to be able to group the sequence of characters to be treated as one set.

◆ Example: Search for guppy and guppies.
  ▪ /gupp(y|ies)/

◆ Useful in conjunction to “*” operator.
  ▪ /*/ - applies to single character and not to a whole sequence.

◆ Example: Match “Column 1 Column 2 Column 3 …”
  ▪ /Column\[0-9]+/*/ - will match “Column # …“
  ▪ /(Column\[0-9]+\)*/ - will match “Column 1 Column 2 Column 3 …”
# Operator Precedence Hierarchy

<table>
<thead>
<tr>
<th>Operator Class</th>
<th>Precedence from Highest to Lowest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parenthesis</td>
<td>()</td>
</tr>
<tr>
<td>Counters</td>
<td>* + ? {}</td>
</tr>
<tr>
<td>Sequences and anchors</td>
<td>^ $</td>
</tr>
<tr>
<td>Disjunction</td>
<td></td>
</tr>
</tbody>
</table>
Simple Example

Problem Statement: Want to write RE to find cases of the English article “the”.

1. /the/ - It will miss “The”
2. /\[tT\]he/ - It will match “amalthea”, “Bethesda”, “theology”, etc.
3. /\b[tT]he\b/ - Is the correct RE

Problem Statement: If we want to find “the” where it might also have undelines or numbers nearby ("The-", “the_” or “the25”) one needs to specify that we want instances in which there are no alphabetic letters on either side of “the”:

1. /([^a-zA-Z])[tT]he/([^a-zA-Z])/ - it will not find “the” if it begins the line.
2. /((^|[^a-zA-Z])*\[tT\]he/([^a-zA-Z])/
Simple Example (cont.)

◆ Refining RE by reduction of:
  ■ **false positives** *(false acceptance)*:
    ◆ Strings that are incorrectly matched.
  ■ **false negatives** *(false rejection)*:
    ◆ Strings that are incorrectly missed.
A More Complex Example

Problem Statement: Build an application to help a user by a computer on the Web.

- The user might want “any PC with more than 6 GHz and 256 GB of disk space for less than $1000
- To solve the problem must be able to match the expressions like 1000 MHz, 6 GHz and 256 GB as well as $999.99 etc.
Solution – Dollar Amounts

◆ Complete regular expression for prices of full dollar amounts:
  ■ /$[0-9]+/

◆ Adding fractions of dollars:
  ■ /$[0-9]+\.[0-9][0-9]/ or
  ■ /$[0-9]+\.[0-9] \{2}/

◆ Problem since this RE only will match “$199.99” and not “$199”. To solve this issue must make cents optional and make sure the $ amount is a word:
  ■ /\b$[0-9]+(\.[0-9][0-9])?\b/
Solution: Processor Speech

- Processor speech in megahertz = MHz or gigahertz = GHz)
  - `/\b[0-9]+\s*(MHz|[Mm]egahertz|GHz|[Gg]igahertz)\b/`
  - `\s*` is used to denote “zero or more spaces”.
Solution: Disk Space

◆ Dealing with disk space:
  ■ Gb = gigabytes

◆ Memory size:
  ■ Mb or MB = megabytes or
  ■ Gb or GB = gigabytes

◆ Must allow optional fractions:
  ■ /[0-9]+\*(M[Bb]|Mm)egabytes?\b/
  ■ /[0-9]+((\.[0-9]+)?\*(G[Bb]|Gg)igabytes?)\b/
Solution: Operating Systems and Vendors

- \b((Windows)+∪*(XP|Vista|7)?)\b/
- \b((Mac|Macintosh|Apple)\b/
# Advanced Operators

<table>
<thead>
<tr>
<th>RE</th>
<th>Expansion</th>
<th>Match</th>
<th>Example Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>[0-9]</td>
<td>Any digit</td>
<td>“Party of 5”</td>
</tr>
<tr>
<td>\D</td>
<td>[^0-9]</td>
<td>Any non-digit</td>
<td>“Blue moon”</td>
</tr>
<tr>
<td>\w</td>
<td>[a-zA-Z0-9 ]</td>
<td>Any alphanumeric or space</td>
<td>Daiyu</td>
</tr>
<tr>
<td>\W</td>
<td>[^\w]</td>
<td>A non-alphanumeric</td>
<td>!!!!</td>
</tr>
<tr>
<td>\s</td>
<td>[\t\n\f]</td>
<td>Whitespace (space, tab)</td>
<td>“ ”</td>
</tr>
<tr>
<td>\S</td>
<td>[^\s]</td>
<td>Non-whitespace</td>
<td>“in Concord”</td>
</tr>
</tbody>
</table>

**Aliases for common sets of characters**
# Repetition Metacharacters

<table>
<thead>
<tr>
<th>RE</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Matches any number of occurrences of the previous character – zero or more</td>
<td>/ac*e/ - matches “ae”, “ace”, “acce”, “accce” as in “The aerial acceleration alerted the ace pilot”</td>
</tr>
<tr>
<td>?</td>
<td>Matches at most one occurrence of the previous characters – zero or one.</td>
<td>/ac?e/ - matches “ae” and “ace” as in “The aerial acceleration alerted the ace pilot”</td>
</tr>
<tr>
<td>+</td>
<td>Matches one or more occurrences of the previous characters</td>
<td>/ac+e/ - matches “ace”, “acce”, “accce” as in “The aerial acceleration alerted the ace pilot”</td>
</tr>
<tr>
<td>{n}</td>
<td>Matches exactly n occurrences of the previous characters.</td>
<td>/ac{2}e/ - matches “acce” as in “The aerial acceleration alerted the ace pilot”</td>
</tr>
<tr>
<td>{n,}</td>
<td>Matches n or more occurrences of the previous characters</td>
<td>/ac{2,}e/ - matches “acce”, “accce” etc., as in “The aerial acceleration alerted the ace pilot”</td>
</tr>
<tr>
<td>{n,m}</td>
<td>Matches from n to m occurrences of the previous characters.</td>
<td>/ac{2,4}e/ - matches “acce”, “accce” and “accce” as in “The aerial acceleration alerted the ace pilot”</td>
</tr>
<tr>
<td>.</td>
<td>Matches one occurrence of any characters of the alphabet except the new line character</td>
<td>/a.e/ matches aae, aAe, abe, aBe, a1e, etc., as in “The aerial acceleration alerted the ace pilot”</td>
</tr>
<tr>
<td>.*</td>
<td>Matches any string of characters and until it encounters a new line character</td>
<td></td>
</tr>
</tbody>
</table>
### Literal Matching of Special Characters & “\” Characters

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>An asterisk “*”</td>
<td>“K<em>A</em>P<em>L</em>A*N”</td>
</tr>
<tr>
<td>.</td>
<td>A period “.”</td>
<td>“Dr. Këpuska, I presume”</td>
</tr>
<tr>
<td>?</td>
<td>A question mark “?”</td>
<td>“Would you like to light my candle?”</td>
</tr>
<tr>
<td>\n</td>
<td>A newline</td>
<td></td>
</tr>
<tr>
<td>\t</td>
<td>A tab</td>
<td></td>
</tr>
<tr>
<td>\r</td>
<td>A carriage return character</td>
<td></td>
</tr>
</tbody>
</table>

Some characters that need to be back-slashèd “\”
Regular Expression Substitution, Memory, and ELIZA
Substitutions are an important use of regular expressions.

- `s/regexp1/regexp2/` - allows a string characterized by one regular expression (`regexp1`) to be replaced by a string characterized by a second regular expression (`regexp2`).
- `s/colour/color/`

It is also important to refer to a particular subpart of the string matching the first pattern.

- Example: replace
  - “the 35 boxes”, to
  - “the <35> boxes”

- `s/([0-9]+)/<\1>/` - “\1” refers to the first pattern matched by the first regular expression.
The parenthesis and number operators can also be used to specify that a certain string or expression must occur twice in the text.

Example:
- “the \textit{X}er they were, the \textit{X}er they will be”
- We want to constrain the two X’s to be the same string:
  - 
  - This RE will match:
    - “The bigger they were, the bigger they will be”, but not
    - “The bigger they were, the faster they will be”

The number operator can be used with other numbers: if you match two different sets of parenthesis, \textbackslash 2 means whatever matched the second set.

Example:
- 
  - This Re will match:
    - “The faster they ran, the faster we ran”, but not
    - “The faster they ran, the faster we ate”,
Registers

- Numbered memories are called registers:
  - \(1\) – register 1
  - \(2\) – register 2
  - \(3\) – register 3
Substitutions using memory are very useful in implementing simple natural-language understanding programs like ELIZA.

Here is example of dialog with ELIZA:

User1: Men are all alike.

ELIZA1: IN WHAT WAY

User2: They’re always bugging us about something or other.

ELIZA2: CAN YOU THINK OF A SPECIFIC EXAMPLE

User3: Well, my boyfriend made me come here.

ELIZA3: YOUR BOYFRIEND MADE YOU COME HERE

User4: He says I’m depressed much of the time.

ELIZA4: I AM SORRY TO HEAR YOU ARE DEPRESSED.
ELIZA

◆ Eliza worked by having a cascade of regular expression
substitutions that each matched some part of the input
lines and changed them.

■ The first substitutions changed all instances of:
  ◆ “my” ⇨ “YOUR”
  ◆ “I’m” ⇨ “YOU ARE”

■ Next set of substitutions looked for relevant patterns in the
input and created an appropriate output;
  ◆ s/.* YOU ARE (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \\1/
  ◆ s/.* YOU ARE (depressed|sad) .*/WHY DO YOU THINK YOU ARE \\1/
  ◆ s/.* all .*/IN WHAT WAY/
  ◆ s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/
Since multiple substitutions could apply to a given input, substitutions were assigned a rank and were applied in order. Creation of such patterns is addressed in Exercise 2.2.
Finite State Automata
Finate State Automata

◆ The regular expression is more than just a convenient metalanguage for text searching.

1. A regular expression is one way of describing a finite-state-automaton (FSA).
   ▪ FSA – are the theoretical foundation of significant number of computational work described in the class.
   ▪ Any regular expression can be implemented as FSA (except regular expressions that use the memory feature).

2. Regular expression is one way of characterizing a particular kind of formal language called a regular language.
   ▪ Both FSA and RE can be used to describe regular languages.
FSA, RE and Regular Languages

Regular expressions

Finite automata

Regular Languages

Regular languages
Finite-state automaton for Regular Expressions

- Using FSA to Recognize Sheeptalk with RE: `/baa+/!`
The FSA can be used for recognizing (we also say accepting) strings in the following way. First, think of the input as being written on a long tape broken up into cells, with one symbol written in each cell of the tape, as figure below:
Recognition Process

- The machine starts in the start state \((q0)\), and iterates the following process:

1. **Check the next letter of the input.**
   a. If it matches the symbol on an arc leaving the current state, then
      i. cross that arc
      ii. move to the next state, also
      iii. advance one symbol in the input
   b. If we are in the accepting state \((q4)\) when we run out of input, the machine has successfully recognized an instance of sheeptalk.

2. **If the machine never gets to the final state,**
   a. either because it runs out of input, or
   b. it gets some input that doesn’t match an arc (as in Fig in previous slide), or
   c. if it just happens to get stuck in some non-final state, we say the machine **rejects** or fails to accept an input.
We’ve marked state 4 with a colon to indicate that it’s a final state (you can have as many final states as you want), and the Ø indicates an illegal or missing transition. We can read the first row as “if we’re in state 0 and we see the input b we must go to state 1. If we’re in state 0 and we see the input a or !, we fail”.

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td>a</td>
<td>!</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Ø</td>
<td>Ø</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Ø</td>
<td>2</td>
<td>Ø</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Ø</td>
<td>3</td>
<td>Ø</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Ø</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td></td>
</tr>
</tbody>
</table>
**Formal Definition of Automaton**

<table>
<thead>
<tr>
<th>Q = {q_0, q_1, ..., q_N}</th>
<th>A finite set of N states</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Sigma )</td>
<td>a finite <strong>input alphabet</strong> of symbols</td>
</tr>
<tr>
<td>( q_0 )</td>
<td>the <strong>start state</strong></td>
</tr>
<tr>
<td>( F )</td>
<td>the set of <strong>final states</strong>, ( F \subseteq Q )</td>
</tr>
<tr>
<td>( \delta(q, i) )</td>
<td>the <strong>transition function or transition matrix</strong> between states. Given a state ( q \in Q ) and an input symbol ( i \in \Sigma ), ( \delta(q, i) ) returns a new state ( q' \in Q ). ( \delta ) is thus a relation from ( Q \times \Sigma ) to ( Q );</td>
</tr>
</tbody>
</table>
FSA Example

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
- $\Sigma = \{a, b, !\}$,
- $F = \{q_4\}$, and
- $\delta(q, i)$

Start State

Final State

Transitions

24 August 2009

Veton Këpuska
Deterministic Algorithm for Recognizing a String

**function** D-RECOGNIZE(tape, machine) **returns** accept or reject

*index* ← Beginning of tape
*current-state* ← Initial state of machine

**loop**

  **if** End of input has been reached **then**
  
  **if** current-state is an accept state **then**
  
  **return** accept
  
  **else**
  
  **return** reject
  
  **elsif** transition-table[current-state, tape[index]] is empty **then**
  
  **return** reject
  
  **else**
  
  current-state ← transition-table[current-state, tape[index]]
  
  index ← index + 1

**end**
Tracing Execution for Some Sheep Talk

Start State: $q_0$

Transitions:
- $q_0 \xrightarrow{b} q_1$
- $q_1 \xrightarrow{a} q_2$
- $q_2 \xrightarrow{a} q_3$
- $q_3 \xrightarrow{!} q_4$

Final State: $q_4$

Input Table:

<table>
<thead>
<tr>
<th>State</th>
<th>b</th>
<th>a</th>
<th>!</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>1</td>
<td>Ø</td>
<td>2</td>
<td>Ø</td>
</tr>
<tr>
<td>2</td>
<td>Ø</td>
<td>3</td>
<td>Ø</td>
</tr>
<tr>
<td>3</td>
<td>Ø</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4:</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
</tbody>
</table>

Input: $b \ a \ a \ a \ !$
Before examining the beginning of the tape, the machine is in state \( q_0 \). Finding a \( b \) on input tape, it changes to state \( q_1 \) as indicated by the contents of transition-table[\( q_0,b \)] in Fig.

It then finds an \( a \) and switches to state \( q_2 \), another \( a \) puts it in state \( q_3 \), a third \( a \) leaves it in state \( q_3 \), where it reads the “!”, and switches to state \( q_4 \). Since there is no more input, the End of input condition at the beginning of the loop is satisfied for the first time and the machine halts in \( q_4 \).

State \( q_4 \) is an accepting state, and so the machine has accepted the string \( baaa! \) as a sentence in the sheep language.
The algorithm will fail whenever there is no legal transition for a given combination of state and input. The input $abc$ will fail to be recognized since there is no legal transition out of state $q_0$ on the input $a$, (i.e., this entry of the transition table has a $\emptyset$).

Even if the automaton had allowed an initial $a$ it would have certainly failed on $c$, since $c$ isn’t even in the sheepstalk alphabet! We can think of these “empty” elements in the table as if they all pointed at one “empty” state, which we might call the fail state or sink state.

In a sense then, we could FAIL STATE view any machine with empty transitions as if we had augmented it with a fail state, and drawn in all the extra arcs, so we always had somewhere to go from any state on any possible input. Just for completeness, next Fig. shows the FSA from previous Figure with the fail state $q_F$ filled in.
Adding a Fail State to FSA
Formal Languages
Formal Languages

Key Concept #1. **Formal Language**:
- A model which can both generate and recognize all and only the strings of a formal language acts as a definition of the formal language.

- A **formal language** is a set of strings, each string composed of symbols from a finite symbol-set called an **alphabet** (the same alphabet used above for defining an automaton!).

- The alphabet for a “sheep” language is the set $\Sigma = \{a, b, !\}$.
- Given a model $m$ (such as FSA) we can use $L(m)$ to mean “the formal language characterized by $m$”.
- $L(m)=\{baa!, baaa!, baaaa!, baaaaaa!, \ldots\}$
Example 2

◆ Alphabet consisting of words. Must build an FSA that models the sub-part of English language that deals with amounts of money:
  ■ Ten cents,
  ■ Three dollars,
  ■ One dollar thirty-five cents, ...

◆ Such a formal language would model the subset of English that consists of phrases like ten cents, three dollars, one dollar thirty-five cents, etc.

1. Solve the problem of building FSA for numbers 1-99 with which will model cents.

2. Model dollar amounts by adding cents to it.
FSA for the words for English numbers 1-99
FSA for the simple Dollars and Cents
Homework #1

Problem 1. Complete the FSA for English money expressions in Fig. 2.16 (of the pdf: http://www.cs.colorado.edu/~martin/SLP/Updates/2.pdf) as suggested in the text following the figure. You should handle amounts up to $100,000, and make sure that “cent” and “dollar” have the proper plural endings when appropriate.
Non-Deterministic FSAs
Non-Deterministic FSAs

Deterministic FSA

Non-Deterministic FSA
Deterministic vs Non-deterministic FSA

- Deterministic FSA is one whose behavior during recognition is fully determined by the state it is in and the symbol it is looking at.

- The FSA in the previous slide when FSA is at the state q2 and the input symbol is a we do not know whether to remain in state 2 (self-loop transition) or state 3. Clearly the decision dependents on the next input symbols.
Another NFSA for “sheep” language

- $\varepsilon$ - **transition** defines the arc that cases transition without an input symbol. Thus when in state $q_3$ transition to state $q_2$ is allowed without looking at the input symbol or advancing input pointer.

- This example is another kind of non-deterministic behavior – we might not know whether to follow the $\varepsilon$ - **transition** or the $!$ arc.
Using NFSA to Accept Strings

- There is a problem of (wrong) choice in non-deterministic FSA. There are three standard solutions to the problem of non-determinism:

  - **Backup:** Whenever we come to a choice point, we could put a marker to mark where we were in the input, and what state the automaton was in. Then if it turns out that we took the wrong choice, we could back up and try another path.

  - **Look-ahead:** We could look ahead in the input to help us decide which path to take.

  - **Parallelism:** Whenever we come to a choice point, we could look at every alternative path in parallel.

- We will focus here on the backup approach and defer discussion of the look-ahead and parallelism approaches to later chapters.
Back-up Approach for NFSA Recognizer

◆ The backup approach suggests that we should make choices that might lead to dead-ends, knowing that we can always return to unexplored alternative choices.

◆ There are two keys to this approach:
  1. Must know ALL alternatives for each choice point.
  2. Store sufficient information about each alternative so that we can return to it when necessary.
Back-up Approach for NFSA Recognizer

◆ When a backup algorithm reaches a point in its processing where no progress can be made:
  ■ Runs out of input, or
  ■ Has no legal transitions,

It returns to a previous choice point and selects one of the unexplored alternatives and continues from there.

◆ To apply this notion to current definition of FSA we need only to store two things for each choice point:
  ■ The State (or node)
  ■ Corresponding position on the tape.
Search State

◆ Combination of the node and the position specifies the search state of the recognition algorithm.

◆ To avoid confusion, the state of automaton is called a node or machine-state.

◆ Two changes are necessary in transition table:

1. To represent nodes that have ε - transitions we need to add ε - column,

2. Accommodate multiple transitions to different nodes from the same input symbol. Each cell entry consists of the list of destination nodes rather then a single node.
The Transition table from NFSA

<table>
<thead>
<tr>
<th>State</th>
<th>b</th>
<th>a</th>
<th>!</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>1</td>
<td>Ø</td>
<td>2</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>2</td>
<td>Ø</td>
<td>2,3</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>3</td>
<td>Ø</td>
<td>Ø</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4:</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
</tbody>
</table>

**Diagram:**

- **Start State:** $q_0$
- **Final State:** $q_4$
- Transition labels:
  - $q_0 \rightarrow q_1$: $b$
  - $q_1 \rightarrow q_2$: $a$
  - $q_2 \rightarrow q_3$: $a$
  - $q_3 \rightarrow q_4$: $!$
  - $q_3 \rightarrow q_3$: $\varepsilon$
An Algorithm for NFSA Recognition

function ND-RECOGNIZE(tape, machine) returns accept or reject

agenda ← {(Initial state of machine, beginning of tape)}
current-search-state ← NEXT(agenda)

loop
  if ACCEPT-STATE?(current-search-state) returns true
    then
      return accept
    else
      agenda ← U GENERATE-NEW-STATES(current-search-state)

  if agenda is empty
    then
      return reject
    else
      current-search-state ← NEXT(agenda)
  end
An Algorithm for NFSA Recognition (cont.)

function GENERATE-NEW-STATES(current-state) returns a set of search-states

current-node ← the node the current search-state is in
index ← the point on the tape the current search-state is looking at

return a list of search states from transition table as follows:
(transition-table[current-node, ε], index)
∪
(transition-table[current-node, tape[index]], index + 1)

function ACCEPT-STATE?(search-state) returns true or false

current-node ← the node search-state is in
index ← the point on the tape search-state is looking at

if index is at the end of the tape and current-node is an accept state of machine
    then
        return true
    else
        return false
Possible execution of ND-RECOGNIZE

Start State

<table>
<thead>
<tr>
<th>Input</th>
<th>b</th>
<th>a</th>
<th>!</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>1</td>
<td>∅</td>
<td>2</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>2</td>
<td>∅</td>
<td>2,3</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>3</td>
<td>∅</td>
<td>∅</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4:</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>
Recognition as Search

- ND-RECOGNIZE accomplishes the task of recognizing strings in a regular language by providing a way to systematically explore all the possible paths through a machine.
- This kind of solutions are known as state-space search algorithms.
- The key to the effectiveness of such programs is often the order which the states in the space are considered. A poor ordering of states may lead to the examination of a large number of unfruitful states before a successful solution is discovered.
  - Unfortunately typically it is not possible to tell a good choice from a bad one, and often the best we can do is to insure that each possible solution is eventually considered.
  - Node that the ordering of states is left unspecified in ND-RECOGNIZE (NEXT function).
  - Thus critical to the performance of the algorithm is the implementation of NEXT function.
Depth-First-Search

- Depth-First-Search or Last-In-First-Out (LIFO).
- Next return the state at the front of the agenda.
- Pitfall: Under certain circumstances they can enter an infinite loop.
Depth-First Search of ND-RECOGNIZE

A depth-first trace of FSA on some sheeptalk

<table>
<thead>
<tr>
<th>State</th>
<th>b</th>
<th>a</th>
<th>!</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>1</td>
<td>∅</td>
<td>2</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>2</td>
<td>∅</td>
<td>2,3</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>3</td>
<td>∅</td>
<td>∅</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>
Breadth-First Search

- Breadth-First Search or First In First Out (FIFO) strategy.
  - All possible choices explored at once.

- Pitfalls:
  - As with depth-first if the state-space is infinite, the search may never terminate.
  - More importantly due to growth in the size of the agenda if the state-space is even moderately large, the search may require an impractically large amount of memory.

- For larger problems, more complex search techniques such as dynamic programming or A* must be used.
Breadth-First Search of ND-RECOGNIZE

A breadth-first trace of FSA on some sheeptalk

Input

<table>
<thead>
<tr>
<th>State</th>
<th>b</th>
<th>a</th>
<th>!</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>1</td>
<td>Ø</td>
<td>2</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>2</td>
<td>Ø</td>
<td>2,3</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>3</td>
<td>Ø</td>
<td>Ø</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4:</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
</tbody>
</table>
Regular Languages and FSA
Regular Languages and FSA

◆ The class of languages that is definable by Regular Expressions is exactly the same as the class of languages that are characterizable by finite-state automata:

Those languages are called Regular Languages.
Formal Definition of Regular Languages

- $\Sigma$ - alphabet = set of symbols in a language.
- $\varepsilon$ - empty string
- $\emptyset$ – empty set.

The regular languages (or regular sets) over $\Sigma$ is then formally defined as follows:

1. $\emptyset$ is a regular language
2. $\forall a \in \Sigma \cup \varepsilon$, $\{a\}$ is a regular language
3. If $L_1$ and $L_2$ are regular languages, and so are:
   a) $L_1 \cdot L_2 = \{xy | x \in L_1, y \in L_2\}$, the concatenation of $L_1$ and $L_2$
   b) $L_1 \cup L_2$, the union or disjunction of $L_1$ and $L_2$
   c) $L_1^*$, the * closure of $L_1$.

All and only the sets of languages which meet the above properties are regular languages.
Regular Languages and FSAs

- All regular languages can be implemented by the three operations which define regular languages:
  - Concatenation
  - Disjunction | Union (also called “|”),
  - * closure.

- Example:
  - (*,+,\{n,m\}) are just a special case of repetition plus * closure.
  - All the anchors can be thought of as individual special symbols.
  - The square braces [ ] are a kind of disjunction:
    - [ab] means “a or b”, or
    - The disjunction of a and b.
Regular Languages and FSAs

- Regular languages are also closed under the following operations:
  - **Intersection**: if $L_1$ and $L_2$ are regular languages, then so is $L_1 \cap L_2$, the language consisting of the set of strings that are in both $L_1$ and $L_2$.
  - **Difference**: if $L_1$ and $L_2$ are regular languages, then so is $L_1 - L_2$, the language consisting of the set of strings that are in $L_1$ but not $L_2$.
  - **Complementation**: if $L_1$ and $L_2$ are regular languages, then so is $\Sigma^* - L_1$, the set of all possible strings that are not in $L_1$.
  - **Reversal**: if $L_1$ is a regular language, then so is $L_1^R$, the language consisting of the set of reversals of the strings that are in $L_1$. 
Regular Expressions and FSA

- The regular expressions are equivalent to finite-state automaton (Proof: Hopcroft and Ullman 1979).
- Proof is inductive. Each primitive operations of a regular expression (concatenation, union, closure) is shown as part of inductive step of the proof:

![Automata diagrams](image)

(a) $r=\varepsilon$
(a) $r=\emptyset$
(a) $r=a$

Automata for the base case (no operators) for the induction showing that any regular expression can be turned into an equivalent automaton.
Concatenation

- FSAs next to each other by connecting all the final states of FSA1 to the initial state of FSA2 by an $\varepsilon$-transition
Closure

- Repetition: All final states of the FSA back to the initial states by $\varepsilon$-transition
- Zero occurrences case: Direct link from the initial state to final state

Closure of an FSA
Add a single new initial state $q_0$, and add new $\varepsilon$-transitions from it to the former initial states of the two machines to be joined.

The union ($|$) of two FSAs
Summary

This chapter introduced the most important fundamental concept in language processing, the finite automaton, and the practical tool based on automaton, the regular expression. Here’s a summary of the main points we covered about these ideas:

◆ The regular expression language is a powerful tool for pattern-matching.

◆ Basic operations in regular expressions include
  ■ concatenation of symbols,
  ■ disjunction of symbols ([, |, and .),
  ■ counters (*, +, and {n,m}),
  ■ anchors (^, $) and precedence operators ((,)).

◆ Any regular expression can be realized as a finite state automaton (FSA).
Memory (\1 together with (\)) is an advanced operation that is often considered part of regular expressions, but which cannot be realized as a finite automaton.

An automaton implicitly defines a **formal language** as the set of strings the automaton accepts.

An automaton can use any set of symbols for its vocabulary, including letters, words, or even graphic images.
The behavior of a **deterministic** automaton (DFSA) is fully determined by the state it is in.

A **non-deterministic** automaton (NFSA) sometimes has to make a choice between multiple paths to take given the same current state and next input.

Any NFSA can be converted to a DFSA.

The order in which a NFSA chooses the next state to explore on the agenda defines its **search strategy**.

- The **depth-first search** or LIFO strategy corresponds to the agenda-as-stack;
- The **breadth-first search** or FIFO strategy corresponds to the agenda-as-queue.

Any regular expression can be automatically compiled into a NFSA and hence into FSA.