Regular Expressions and Text Searching

• Everybody does it
  ◆ emacs, vi, grep, sed, Perl, Python, Ruby, Java etc.

• Regular expressions are a compact textual representation of a set of strings representing a language.

• Example web page:
  ◆ http://rubular.com/
Example

• Regular expression search requires a **pattern** that we want to search for and a **corpus** of text to search through.
• Find all the instances of the word “the” in a text.
  - `/the/`
  - `/[tT]he/`
  - `\b[tT]he\b/`
Errors

• The process we just went through was based on **fixing two kinds of errors**
  - Matching strings that we should not have matched (**there, then, other**)
    - False positives (Type I)
  - Not matching things that we should have matched (**The**)
    - False negatives (Type II)
Errors

• We’ll be telling the same story for many tasks, all semester. Reducing the error rate for an application often involves two antagonistic efforts:
  - Increasing accuracy, or precision, (minimizing false positives)
  - Increasing coverage, or recall, (minimizing false negatives).
Range, negation and optionality

- `/^[A-Z]/` an upper case letter
- `/^[a-z]/` a lower case letter
- `/^[0-9]/` a single digit
- `/^[^A-Z]/` not an upper case letter
- `/^[^\./]/` not a period
- `/colou?r/` color or colour
Kleene * and +

- `/s+/` one or more occurrences of s
- `/\[0-9\]+/` a sequence of digits
- `/s*/` zero or more occurrences of s
- `/\[0-9\]\[0-9\]*` a sequence of digits
Anchors

• Special characters that anchor regular expressions to particular places in a string
  • /^/ matches the start of a line
  • /$/ matches the end of a line
  • /^T/ matches what?
  • /\./ matches what?
Disjunction and Grouping

- Disjunction operator
  - `/cat|dog/` matches cat or dog
- Grouping
  - `/gupp(y|ies)/`
    - Matches guppy or guppies
Advanced operators

- `\d/  = /\[0-9]/`
- `\D/  = /[^0-9]/`
- `\w/  = /\[a-zA-Z0-9_\]/`
- `\W/  = /[^\w]/`
- `\s/  = [ \r\t\n\f]  (white space)`
- `\S/  = /[^\s]/`
Finite State Automata

- Regular expressions can be viewed as a textual way of specifying the structure of finite-state automata (FSA).
- Regular expressions can be implemented with FSAs.
- FSAs and their probabilistic relatives are at the core of much of what we’ll be doing all semester.
- They also capture significant aspects of what linguists say we need for morphology and parts of syntax.
FSAs as Graphs

• Let’s start with the sheep language from Chapter 2
  ♦ /baa+!/
• We can say the following things about this machine
  ▶ It has 5 states
  ▶ b, a, and ! are in its alphabet
  ▶ q₀ is the start state
  ▶ q₄ is an accept state
  ▶ It has 5 transitions
But Note

- There are other machines that correspond to this same language

- More on this one later
More Formally

- We can specify an FSA by enumerating the following things.
  - The set of states: $Q$
  - A finite alphabet: $\Sigma$
  - A start state
  - A set $F$ of accept/final states
  - A transition function that maps $Q \times \Sigma$ to $Q$
The sheeptalk automaton

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

<table>
<thead>
<tr>
<th>State/ Input</th>
<th>b</th>
<th>a</th>
<th>!</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>1</td>
<td>( \emptyset )</td>
<td>2</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>2</td>
<td>( \emptyset )</td>
<td>3</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>3</td>
<td>( \emptyset )</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
About Alphabets

• Don’t take term *alphabet* word too narrowly; it just means we need a finite set of symbols in the input.

• These symbols can and will stand for bigger objects that can have internal structure.
Dollars and Cents
Recognition

• Recognition is the process of determining if a string should be accepted by a machine
• Or... it’s the process of determining if a string is in the language we’re defining with the machine
• Or... it’s the process of determining if a regular expression matches a string
• Those all amount the same thing in the end
Recognition

• Simply a process of starting in the start state
• Examining the current input
• Consulting the table
• Going to a new state and updating the input pointer.
• Until you run out of input.
Deterministic (Finite-state) Automaton (DFA)

• The behavior during recognition is fully **determined** by the state it is in and the symbol it is looking at.
Deterministic recognition

- **Input**: a string $x$ ending with EOF. DFA, $D$, with start state $q_0$ and a set, $F$, of final states.
- **Output**: true if $D$ recognizes $x$, otherwise false.

$q = q_0$
$c = \text{nextchar}()$
while ($c \neq \text{EOF}$) {
    $q = \text{move}(q, c)$; // returns the state to which the
    // automaton moves
    // from state $q$ on input $c$
    $c = \text{nextchar}()$
} if ($q \in F$) then return true
else return false;
Key Points

• Deterministic means that at each point in processing there is always one unique thing to do (no choices).

• D(eterministic)-recognize is a simple table-driven interpreter

• The algorithm is universal for all unambiguous regular languages.
  ♦ To change the machine, you simply change the table.
Key Points

• Crudely therefore... matching strings with regular expressions (ala Perl, grep, vi, etc.) is a matter of
  ♦ translating the regular expression into a machine (a table) and
  ♦ passing the table and the string to an interpreter
Generative Formalisms

- **Formal Languages** are sets of strings composed of symbols from a finite set of symbols.
- Finite-state automata define formal languages (without having to enumerate all the strings in the language).
- The term *Generative* is based on the view that you can run the machine as a generator to get strings from the language.
Generative Formalisms

- FSAs can be viewed from two perspectives:
  - Acceptors that can tell you if a string is in the language
  - Generators to produce *all and only* the strings in the language
Non-Deterministic FSA (NFA)
Non-Determinism cont.

• Yet another technique
  - Epsilon transitions (\(\varepsilon\)-transitions)
  - Key point: these transitions do not examine or advance the input during recognition
Equivalence

- Non-deterministic machines can be converted to deterministic ones with a fairly simple construction
- That means that they have the same power; non-deterministic machines are not more powerful than deterministic ones in terms of the languages they can accept
NFA Recognition

- Two basic approaches (used in all major implementations of regular expressions, see Friedl 2006)
  1. Either take a NFA machine and convert it to a DFA machine and then do recognition with that.
  2. Or explicitly manage the process of recognition as a state-space search (leaving the machine as is).
Non-Deterministic Recognition: Search

- In an NFA there exists at least one path through the machine for a string that is in the language defined by the machine.
- But not all paths directed through the machine for an accept string lead to an accept state.
- No paths through the machine lead to an accept state for a string not in the language.
Non-Deterministic Recognition

- So **success** in non-deterministic recognition occurs when a path is found through the machine that ends in an accept.
- **Failure** occurs when all of the possible paths for a given string lead to failure.
Example

1

! b a a a !
Example
Example

1
\[
\begin{array}{c}
q_0 \\
q_2
\end{array}
\] 
\begin{array}{c}
\text{b} \\
\text{b}
\end{array}
\]

2
\[
\begin{array}{c}
q_0 \\
q_1
\end{array}
\] 
\begin{array}{c}
\text{a} \\
\text{a}
\end{array}
\]

3
\[
\begin{array}{c}
q_1 \\
q_2
\end{array}
\] 
\begin{array}{c}
\text{a} \\
\text{a}
\end{array}
\]
Example

1. $\ldots b a a a ! \ldots$

2. $\ldots b a a a ! \ldots$

3. $\ldots b a a a ! \ldots$

4. $\ldots b a a a ! \ldots$

Diagram:

$q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_4$

Transitions:

- $q_0 \rightarrow b$
- $q_2 \rightarrow a$
- $q_2 \rightarrow a$
- $q_3 \rightarrow !$
Example

1  b a a a !

2  b a a a !

3  b a a a !

4  b a a a !

5  b a a a !
Example

1

\[ baaa! \]

2

\[ baaa! \]

3

\[ baaa! \]

4

\[ baaa! \]

5

\[ baaa! \]

6

\[ baaa! \]

Diagram:

- \( q_0 \) to \( q_1 \) to \( q_2 \) to \( q_3 \) to \( q_4 \)
- Transition:
  - b
  - a a
  - a
  - !
Example
Example
Key Points

• States in the search space are pairings of input positions and states in the machine.

• By keeping track of as yet unexplored states, a recognizer can systematically explore all the paths through the machine given an input.
Why Bother?

- Non-determinism doesn’t get us more formal power and it causes headaches so why bother?
  - More natural (understandable) solutions
  - Regular expressions can (easily) be converted automatically to an NFA
Compositional Machines

- Formal languages are just sets of strings
- Therefore, we can talk about various set operations (intersection, union, concatenation)
- This turns out to be a useful exercise
Union

\[
\begin{array}{c}
FSA_1 \\
q_0 \rightarrow q_f \\
FSA_2 \\
q_0 \rightarrow q_f
\end{array}
\]

\[\epsilon \rightarrow q_0 \rightarrow q_f \]

\[\epsilon \rightarrow q_0 \rightarrow q_f \]
Concatenation
Negation

• Construct a machine $M_2$ to accept all strings not accepted by machine $M_1$ and reject all the strings accepted by $M_1$
  - Invert all the accept and not accept states in $M_1$

• Does that work for non-deterministic machines?