Formal Languages, Regular Expressions, Automata, Transducers

Adam Meyers
New York University
1/30/2012
Outline

• Formal Languages in the Chomsky Hierarchy
• Regular Expressions
• Finite State Automata
• Finite State Transducers
• Some Sample CL tasks using Regexps
• Concluding Remarks
Formal Language = Set of Strings of Symbols

• All Combinations of the letters A and B
  – Strings: ABAB, AABB, AAAB, etc.

• Any number of As, followed by any number of Bs
  – Strings: AB, AABB, AB, AAAAAAAABBB, etc.

• Mathematical Equations:
  – Sample Strings: 1 + 2 = 5, 2 + 3 = 4 + 1, 6 = 6

• All the sentences of a simplified version of written English
  – 1 Sample String: My pet wombat is invisible.

• A sequence of musical notation (e.g., the notes in Beethoven's 9th Symphony)
  – 1 Sample String: A-sharp B-flat C G A-sharp
What is a Formal Grammar for?

• A formal grammar
  – set of rules
  – matches **all and only** instances of a **formal language**

• A formal grammar defines a formal language

• In Computer Science, formal grammars are used to both **generate** and to **recognize** formal languages.
  – Parsing a string of a language involves:
    • Recognizing the string and
    • Recording the analysis showing it is part of the language
  – A compiler translates from language X to language Y, e.g.,
    • This may include parsing language X and generating language Y
A Formal Grammar Consists of:

- **N**: a Finite set of nonterminal symbols
- **T**: a Finite set of terminal symbols
- **R**: a set of rewrite rules, e.g., \( XYZ \rightarrow abXzY \)
  - Replace the symbol sequence \( XYZ \) with \( abXzY \)
- **S**: A special nonterminal that is the start symbol
A Very Simple Formal Grammar

• Language_AB = 1 or more a, followed by 1 or more b, e.g., ab, aab, abb, aaaaaaabb, etc.
• N = \{A,B\}
• T={a,b}
• S=Σ
• R={A→a, A→Aa, B→b B→Bb, Σ→AB}
Generating a Sample String

- Start with $\Sigma$
- Apply $\Sigma \rightarrow AB$, Generate A B
- Apply $A \rightarrow Aa$, Generate A a B
- Apply $A \rightarrow Aa$, Generate A a a B
- Apply $A \rightarrow a$, Generate a a a B
- Apply $B \rightarrow b$, Generate a a a b
Derivation of $a a a b$

```
Σ → A B
  /    /
A → A a  B → b
 /     /
A → A a  a
/     /
A → a   a
```

Computational Linguistics
Lecture 2
2011-2012
Phrase Structure Tree for a a a b
The Chomsky Hierarchy: Type 0 and 1

• Type 0: No restrictions on rules
  – Equivalent to Turing Machine
    • General System capable of Simulating any Algorithm

• Type 1: Context-sensitive rules
  – $\alpha A \beta \rightarrow \alpha \gamma \beta$
    • Greek chars = 0 or more nonterms/terms
    • $A =$ nonterminal
    • $\gamma =$ 1 or more nonterms/terms
  – For example,
    • DUCK DUCK DUCK $\rightarrow$ DUCK DUCK GOOSE
    • Means convert DUCK to a GOOSE, if preceded by 2 DUCKS
Chomsky Hierarchy Type 2

- Context-free rules
- $A \rightarrow \alpha \gamma \beta$
- Like context-sensitive, except left-hand side can only contain exactly one nonterminal

- Example Rule from linguistics:
  - $NP \rightarrow POSSP \ n \ PP$
  - $NP \rightarrow Det \ n$
  - $NP \rightarrow n$
  - $POSSP \rightarrow NP \ 's$
  - $PP \rightarrow p \ NP$
  - $[NP \ [POSSP \ [NP \ [Det \ The] \ [n \ group]] \ 's]]$
    - $[n \ discussion]$
    - $[PP \ [p \ about] \ [NP \ [n \ food]]]]$

  - *The group's discussion about food*
Chomsky Hierarchy Type 3

- Regular (finite state) grammars
  - $A \rightarrow \beta a$ or $A \rightarrow \epsilon$ (left regular)
  - $A \rightarrow a\beta$, or $A \rightarrow \epsilon$ (right regular)

- Like Type 2, except non-terminals cannot occur on both sides and null string is allowed

- Example Rule from linguistics:
  - $NP \rightarrow POSSP \, n$
  - $NP \rightarrow n$
  - $NP \rightarrow det \, n$
  - $POSSP \rightarrow NP \, 's$

- $[NP \, [POSSP \, [NP \, \, \, [det \, The] \, [n \, group]] \, 's]]$
  - $[n \, discussion]]$
  - *The group's discussion*
Chomsky Hierarchy

- $Type_0 \supseteq Type_1 \supseteq Type_2 \supseteq Type_3$
- Type 3 grammars
  - Least expressive, Most efficient processors
- Processors for Type 0 grammars
  - Most expressive, Least efficient processors
CL mainly features Type 2 & 3 Grammars

- **Type 3 grammars**
  - Include regular expressions and finite state automata (aka, finite state machines)
  - The focal point of the rest of this talk
  - Also see Nooj CL tools: [www.nooj4nlp.net/](http://www.nooj4nlp.net/)

- **Type 2 grammars**
  - Commonly used for natural language parsers
  - Used to model syntactic structure in many linguistic theories (often supplemented by other mechanisms)
  - We will play a key roll in the next talk on parsing
Regular Expressions

• The language of *regular expressions* (regexps)
  – A standardized way of representing search strings
  – Kleene (1956)

• Computer Languages with regexp facilities:
  – Python, JAVA, Perl, Ruby, most scripting languages, …
  – If not officially supported, a library still may exist

• Many UNIX (linux, Apple, etc.) utilities
  – grep (grep -e regexp file), emacs, vi, ex, ...

• Other
  – Mysql, Microsoft Office, Open Office, ...
Regexp = formula specifying set of strings

- Regexp = $\emptyset$
  - The empty set
- Regexp = $\varepsilon$
  - The empty string
- Regexp = a sequence of one or more characters from the set of characters
  - $X$
  - $Y$
  - *This sentence contains characters like &T^**%P*
- Disjunctions, concatenation, and repetition of regexps yield new regexps
Concatenation, Disjunction, Repetition

- Concatenation
  - If X is a regexp and Y is a regexp, then XY is a regexp
  - Examples
    - If ABC and DEF are regexps, then ABCDEF is a regexp
    - If AB* and BC* are regexps, then AB*BC* is a regexp
      - Note: Kleene * is explained below

- Disjunction
  - If X is a regexp and Y is a regexp, then X | Y is a regexp
  - Example: ABC|DEF will match either ABC or DEF

- Repetition
  - If X is a regexp than a repetition of X will also be a regexp
    - The Kleene Star: A* means 0 or more instances of A
    - Regexp{number}: A{2} means exactly 2 instances of A
Regexp Notation Slide 2

• Disjunction of characters
  – \([ABC]\) – means the same thing as \(A \mid B \mid C\)
  – \([a-zA-Z0-9]\) – ranges of characters equivalent to listing characters, e.g., \(a\mid b\mid c\mid...\mid A\mid B\mid...\mid 0\mid 1\mid...\mid 9\)
  – ^ inside of bracket means complement of disjunction, e.g., \[^a-z]\) means a character that is neither \(a\) nor \(b\) nor \(c\) … nor \(z\)

• Parentheses
  – Disambiguate scope of operators
    • \(A(BC)\)|(DEF) means ABC or ADEF
    • Otherwise defaults apply, e.g., \(ABC\mid D\) means \(ABC\) or \(ABD\)

• ? signifies optionality
  – \(ABC?\) is equivalent to \((ABC)\)|(AB)

• + indicates 1 or more
  – \(A(BC)^*\) is equivalent to \(A\mid(A(BC)^+)\)
Regexp Notation Slide 3

• Special Symbols:
  – A.*B – matches A and B and any characters between (period = any character)
  – ^ABC – matches ABC at beginning of line (^ represents beginning of line)
  – [\./?!]$ – matches sentence final punctuation ($ represents end of line)

• Python's Regexp Module
  – Searching
    • Groups and Group Numbers
  – Compiling
  – Substitution

• Similar Modules for: Java, Perl, etc.
Regexp in NLTK's Chatbot

• Running eliza
  – import nltk
  – from nltk.chat.eliza import *
  – eliza_chat()

• NLTK's chatbots: /usr/local/lib/python2.6/site-packages/nltk/chat
  – See util.py and eliza.py

• How it works
  – It creates a Chat object (defined in util.py) that includes a substitute method
  – The settings for this chat object are in eliza.py
  – For each pair in pairs, the 1st item is matched against the input string, to produce an answer listed as the 2nd item. Note the use of %1 to indicate the repeated parts of the strings.
Regexp in Python (2 and 3)

- import re                imports regexp package
- Example re functions
  - re.search(regexp,input_string)  creates a search object
  - re.sub (regexp,repl,string)
- search_object methods
  - start() and end() -- respectively output start and end position in the string
  - group(0) – outputs whole match
  - group(N) – outputs the nth group (item in parentheses)
- Patterns can be compiled
  - Pattern1 = re.compile(r'[Aa]Bc')
  - Efficient, can take re functions as methods
  - Methods takes additional parameters (e.g., starting position)
    - Pattern1.search('ABcaBc',2)
    - starts search at position 2 and finds 2\textsuperscript{nd} instance of pattern
RegExp to Search for Common Types of Numeric Strings

• Money
  – r'(\$[0-9,]+(\.[0-9][0-9])?)|([0-9]?[0-9]?\$)'
  – How could this be elaborated on?
  – Would this match the string '$,,,,,,'?

• Time
  – Let's do this one in class

• Others
  – Dates, Roman Numerals, Social Security, Telephone Numbers, Zip Codes, Library Call Numbers, etc.
Finite State Automata

• Devices for recognizing finite state grammars (including regular expressions)

• Two types
  – Deterministic Finite State Automata (DFSA)
    • Rules are unambiguous
  – NonDeterministic FSA (NDFSA)
    • Rules are ambiguous
      – Sometimes more than one sequence of rules must be attempted to determine if a string matches the grammar
        » Backtracking
        » Parallel Processing
        » Look Ahead

  – Any NDFSA can be mapped into an equivalent (but larger) DFSA
DFSA for Regexp: $A(ab)^*ABB$?
DFSA algorithm

- D-Recognize(tape, machine)
  
  pointer ← beginning of tape
  current state ← initial state Q0
  repeat until the end of the input is reached
    look up (current state, input symbol) in transition table
    if found: set current state as per table look up
    advance pointer to next position on tape
    else: reject string and exit function
  if current state is a final state: accept the string
  else: reject the string
NDFSA for Regexp: $A(ab)^*ABB$?
NDFSA algorithm

• ND-Recognize(tape, machine)
  agenda ← {(initial state, start of tape)}
  current state ← next(agenda)
  repeat until accept(current state) or agenda is empty
    agenda ← Union(agenda, look_up_in_table(current state, next_symbol))
    current state ← next(agenda)
    if accept(current state): return(True)
    else: false
• Accept if at the end of the tape and current state is a final state
• Next defined differently for different types of search
  – Choose most recently added state first (depth first)
  – Chose least recently added state first (breadth first)
  – Etc.
A Right Regular Grammar Equivalent to: $A(ab)^*ABB$?
(\textcolor{red}{Red} = Terminal, \textcolor{black}{Black} = Nonterminal)

- $Q \rightarrow A \textcolor{red}{RS}$
- $R \rightarrow \epsilon$
- $R \rightarrow abR$
- $S \rightarrow ABB$
- $S \rightarrow AB$
Nondeterministic Finite State Transducer to Generate Phrase Structure for $A(ab)^*ABB?$ (w/imperfection)

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>Q0</td>
<td>Q1</td>
</tr>
<tr>
<td>Q1</td>
<td>Q3</td>
</tr>
<tr>
<td>Q2</td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td>Q4</td>
</tr>
<tr>
<td>Q4</td>
<td>Q5</td>
</tr>
</tbody>
</table>
Readings

• This talk approximately corresponds to:
  – Jurafsky and Martin, Chapters 2 and 3
• Optional: NLTK, Chapter 3
Homework # 1: Slide 1

• Create One or More Regular Expressions to recognize dollar amounts and package them up so Ang can test them.
  – Test them yourself on some files before submitting them

• Program can be a shell script based on grep
  – grep -E regexp file-identifier (as demonstrated in class)
  – grep -E '(\$[0-9,]+(\.\[0-9-][0-9]\[0-9\])\?\|\[0-9\]??[0-9]¢)' FILENAME

• Program can be in any standard programming language

• Output format: insert brackets around money expressions
  – The Picasso print costs [$5 billion dollars and 50 cents] on Ebay.

• However, the program should be self-contained and include instructions for running it. So Ang can run it as follows:
  – Program INPUT_FILE OUTPUT_FILE_FILE
  – Or some minor variation.

• More Details on Next Slide
Homework #1: Slide 2

• This sample regexp '($[0-9,]+(\.[0-9][0-9])?|([0-9]?[0-9]¢)' is flawed, e.g., it doesn't account for number words (billion) or instances of “dollars” or “cents” in a corpus.
  – It doesn't handle “$53 billion dollars” correctly.

• You should train and test your system:
  – 1. You can use the MASC/OANC files from the class website
  – 2. Or any other files you can find on the web

• Ang will test your system on a set of examples that he collected and measure your system's precision, recall and F-score
Homework #2 (Optional)

• Read through the Bots that are part of NLTK and use their libraries to make your own