Formal Languages, Regular Expressions, Automata, Transducers

Adam Meyers
Montclair State University
9/15/2014 and 9/22/2014
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

• A Formal Language Can Model a Phenomenon, e.g., written English

• Examples
  – All Combinations of the letters A and B: $ABAB$, $AABB$, $AAAB$, etc.
  – Any number of As, followed by any number of Bs: $AB$, $AABB$, $AB$, $AAAAAAAABBB$, etc.
  – Mathematical Equations: $1 + 2 = 5$, $2 + 3 = 4 + 1$, $6 = 6$
  – All the sentences of a simplified version of written English, e.g., *My pet wombat is invisible.*
  – A sequence of musical notation (e.g., the notes in Beethoven's 9th Symphony), e.g., *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** 
  
  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
```
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 can either precede (left) or follow (right) terminals, but not both
  - null string is allowed

- Example Rule from linguistics:
  - $NP \rightarrow \text{POSSP } n$
  - $NP \rightarrow n$
  - $NP \rightarrow \text{det } n$
  - $\text{POSSP } \rightarrow \text{NP 's}$

$[NP \ [\text{POSSP} \ [NP \ [\text{det } \text{The}] \ [n \text{ group}] \ ]] \ 's]$

$[n \text{ discussion}]]$

- *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
- Complexity of recognizer for languages:
  - Type 0 = exponential; Type 1 = polynomial; Type 2 = $O(n^3)$; Type 3 = $O(n \log n)$
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/

• 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, ...

Computational Linguistics
Lecture 2
2014
My T-Shirt

• My T-Shirt says:  /(BB|[^[^B]{2})]/
  – The “/”, “(“ and “)” can be ignored for now
  – B represents the string “B”
  – “|” represents the operator 'inclusive or'
  – “^” represents the negative operator
  – [] represents a single character
  – {N}, where N is a number represents N repetitions of the preceding item

• What famous quote could this represent?
• What details are different from the quote?
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

1. 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

2. Disjunction
   - If X is a regexp and Y is a regexp, then \( X \mid Y \) is a regexp
   - Example: \( ABC \mid DEF \) will match either \( ABC \) or \( DEF \)

3. 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., \(a\mid b\mid c\ldots\mid A\mid B\ldots\mid 0\mid 1\ldots\mid 9\)
  - ^ 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
    - \(A(BC)\mid(DEF)\) means \(ABC\) or \(ADEF\)
    - Otherwise defaults apply, e.g., \(ABC\mid D\) means \(ABC\) or \(ABD\)

- ? signifies optionality
  - \(ABC?\) is equivalent to \((ABC)\mid(AB)\)

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

• Special Symbols:
  – \textbf{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 or
  – /usr/lib/pymodules/python2.7/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 1\textsuperscript{st} item is matched against the input string, to
    produce an answer listed as the 2\textsuperscript{nd} 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
Regexp with Unix tools

- grep -E '$[0-9\.,]+' all-OANC | less
- In the program less
  - \$[0-9.,]  
    - Highlights numeric instances
    - Note some of the problems with this regexp for characterizing money strings
RegExp to Search for Common Types of Numeric Strings

• Money
  – $[0-9\.,]+  
  – Would this match the string '$,,,,,'?
    • Maybe that doesn't matter?
  – How might we handle cases like “$4 million”?
  – What might be a better regexp for money?

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

• Time of Day – Let's Do this one as a lab
Time of Day

• Let's agree on the components of a time of day as printed
  – **** fill in here ****

• For 5 minutes, Everyone should attempt to write such an expression independently. You can test your regexp with Python or grep.

• Let's look at some of the proposed answers, test them and possibly combine aspects.
NLTK's Regexp Language for Chunking

- sentence = "The big grey dog with three heads was on my lap"
- tokens = nltk.word_tokenize(sentence)
- pos_tagged_items = nltk.pos_tag(tokens)
- chunk_grammar = nltk.RegexpParser(r""
  NG: {(<DT|JJ|NN|PRP\$>)*(<NN|NNS>)}
  VG: {<MD|VB|VBD|VBN|VBZ|VBP|VBG>*<VB|VBD|VBN|VBZ|VBP|VBG><RP>?}
"
- chunk_grammar.parse(pos_tagged_items)
- Structure:
  - 1 rule per line
  - Nonterminal: Regexp
  - Regexp = terminals, nonterminals & operators (*+?{ }...)

Computational Linguistics
Lecture 2
2014
NLTK's Regexp Language for Chunking

- `sentence = "The big grey dog with three heads was on my lap"
- `tokens = nltk.word_tokenize(sentence)
- `pos_tagged_items = nltk.pos_tag(tokens)
- `chunk_grammar = nltk.RegexpParser(r""
  NG: {(<DT|JJ|NN|PRP\$>)*(<NN|NNS>)}
  VG: {<MD|VB|VBD|VBN|VBZ|VBP|VBG>*<VB|VBD|VBN|VBZ|VBP|VBG><RP>?}
"
- `chunk_grammar.parse(pos_tagged_items)

- Structure:
  - 1 rule per line
  - Nonterminal: Regexp
  - Regexp = terminals, nonterminals & operators (*+?\{\}...)

Computational Linguistics
Lecture 2
2014
Chunking Rules with NonTerminal on Right Hand Side

- chunks2 = r''

  DTP: {<PDT><DT|CD>}
  NG: {(<DT|JJ|NN|DTP|PRP\$>)*(<NN|NNS>)}
  VG: {<MD|VB|VBD|VBN|VBZ|VBP|VBG>*<VB|VBD|VBN|VBZ|VBP|VBG><RP>??
  PP: {<IN|TO><NG>}
  VP: {<VG> <NG|PP>}

  """
The Penn Treebank II POS tagset

- **Verbs:** VB, VBP, VBZ, VBD, VBG, VBN
  - base, present-non-3rd, present-3rd, past, -ing, -en
- **Nouns:** NNP, NNPS, NN, NNS
  - proper/common, singular/plural (singular includes mass + generic)
- **Adjectives:** JJ, JJR, JJS (base, comparative, superlative)
- **Adverbs:** RB, RBR, RBS, RP (base, comparative, superlative, particle)
- **Pronouns:** PRP, PP$ (personal, possessive)
- **Interogatives:** WP, WP$, WDT, WRB (compare to: PRP, PP$, DT, RB)
- **Other Closed Class:** CC, CD, DT, PDT, IN, MD
- **Punctuation:** # $ . , : ( ) “ ” ' '
- **Weird Cases:** FW(*deja vu*), SYM (@), LS (1, 2, a, b), TO (to), POS('s, '), UH (no, OK, well), EX (it/there)
- **Newer tags:** HYPH, PU
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: $A(ab)^*ABB$?

```
<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q0</td>
<td>A</td>
</tr>
<tr>
<td>Q1</td>
<td>B</td>
</tr>
<tr>
<td>Q2</td>
<td>a</td>
</tr>
<tr>
<td>Q3</td>
<td>b</td>
</tr>
<tr>
<td>Q4</td>
<td>ε</td>
</tr>
</tbody>
</table>
```

Diagram:
- Q0 -> Q1 on A
- Q1 -> Q3 on A
- Q3 -> Q4 on B
- Q4 -> Q5 on ε
- Q5 -> Q6 on ε
- Q6 -> Q0 on ε
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: \( A(ab)^*ABB? \)
(Red = Terminal, Black = Nonterminal)

- \( Q \rightarrow ARS \)
- \( R \rightarrow \epsilon \)
- \( R \rightarrow abR \)
- \( S \rightarrow ABB \)
- \( S \rightarrow AB \)
Readings

• Jurafsky and Martin, Chapters 2 and 3
• NLTK Chapters 2 and 3
Homework #2: Content

Write regular expressions that search for the following and test them on the all-OANC.txt corpus available on the class website. It is expected that your regexp will overgenerate and undergeneate somewhat. It is a good regexp if at least 1/2 of the answers are real matches; it generates lots of examples; and (possibly) there is some variety of different cases covered.

1. Names Prefixed by “Dr.”
   - Keep in mind:
     - Periods can be variables meaning any character (but \. means a real period)
     - Which patterns of uppercase, lowercase, and other characters match names?

2. Strings consisting of a name, any form of the word “like” and a plural noun.
   - Keep in mind:
     - how might you approximate a name in English?
     - what are the forms of “like”
     - how do you approximate a plural noun?

3. Telephone numbers
   - Keep in mind variations on how telephone numbers are listed
HW #2: Make Your Regexp Testable

• Make one shell script for each regexp requested
  – Make a file in a text editor (emacs, vi, wordpad, ...)
  – There should be just one one and it should be a single command, substituting “$1” for the filename, e.g.,
    • grep -E '^[0-9,.]+$' $1
  – Make that file executable
    • chmod 750 find_the_money.script
  – Test the shell script:
    • find_the_money.script all-OANC |less

• Write a program in java, python or another standard language and make an executable script so that to test it all I need to do is type
  – your_program all-OANC.txt |less

• Do something else, but give me explicit instructions on how to test your program on a command line (no mouse clicks) – it should works in any platform – if you are not sure, ask.
Handing in HW

• Homework files should have names in the following format:
  – LastFirst_Assignment_X.extension
  – For example,
    • MeyersAdam_Assignment_1.zip

• Multiple files should be put into a single archive in a standard format such as .zip, .tgz, .tar.gz or .tar

• For example, in any Unix platform, you should be able to do the following command:
  – tar -zcf MeyersAdam_Assignment.tgz directory-containing-my-files

• I will demonstrate this in class
Optional HW

• Read through the Bots that are part of NLTK and use their libraries to make your own
• The current bots do not make much use of regexp. Introduce a new feature that uses regexps.