Formal Languages, Regular Expressions, Automata, Transducers

Adam Meyers
New York University
2016
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: $AB$, $AABB$, $AB$, $AAAAAAAAABBBB$, 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
  - If all natural languages were formal languages, then Machine Translation systems would just be compilers
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=\Sigma
- R={A \rightarrow a, A \rightarrow Aa, B \rightarrow b B \rightarrow Bb, \Sigma \rightarrow 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 $a\ a\ a\ b$

Diagram:

```
  Σ → AB
   /\  \
  A → Aa  B → b
 /\  /\   /\  \\
A → Aa  a  A  b
 /\  /\  /\ \\
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:
  - \( NP \rightarrow \text{POSSP } n \ PP \)
  - \( NP \rightarrow \text{Det } n \)
  - \( NP \rightarrow n \)
  - \( \text{POSSP } \rightarrow \text{NP 's} \)
  - \( PP \rightarrow p \ NP \)
  - \( [NP [\text{POSSP } [NP [\text{Det } \text{The}] [n \text{ group}]] ')s] \)
    - \( \text{[n discussion]} \)
    - \( \text{[PP [p about][NP [n food]]]} \)
  - 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:
  - $NP \rightarrow POSSP \ n$
  - $NP \rightarrow n$
  - $NP \rightarrow det \ n$
  - $POSSP \rightarrow NP \ 's$
- $[NP \ [POSSP \ [NP \ [det \ The] \ [n \ group]] \ 's] \ [n \ discussion]]$
  - *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

- 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)
  – 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, ...
My T-Shirt

• My T-Shirt says:  /
\((\text{BB}|[^\text{B}]\{2\})/\)
  – 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 = sequence of one or more characters
  - $X$
  - $Y$
  - This sentence contains characters like &T^**%P
- Disjunction, 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]` – character ranges are equivalent to lists: a\|b\|c\|...\|A\|B\|...\|0\|1\|...\|9\]

- **Negation of character lists/sequences**
  - `^` inside 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., \( ABC\midD \) 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:
  – Period means any character, e.g., $A.*B$ – matches A and B and any characters between
  – Carrot (^) means the beginning of a line, e.g., ^ABC matches ABC at the beginning of a line
  – Dollar sign ($) means the end of a line, e.g., \[.?!] *$ matches final punctuation, zero or more spaces and the end of a 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:
  – find / |grep -i nltk
  – /Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/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 1\textsuperscript{st} item is matched against the input string, to produce an answer listed as the 2\textsuperscript{nd} item. The use of \%1 indicates repeated parts of the strings.
  – In util.py – note that the matching pattern for the 1\textsuperscript{st} item is created with \texttt{re.compile}, a method that turns a regular expression into a match-able pattern, although in the current examples (.*), a very simple (and general) regexp.
Regexps 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')
  – 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
• Your HW will include an expanded version of this problem (finding dollar amounts in text)
RegExp to Search for Common Types of Numeric Strings

• An XML (or html) tag
  – <[^>]+>

• 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 joint exercise
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 (*+?{ }...)
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)
- Interrogatives: 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 (include regexps)
- 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: \( A(ab)^{*}ABB? \)

(Red = Terminal, Black = Nonterminal)

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

- http://cs.nyu.edu/courses/fall16/CSCI-UA.0480-006/homework2.html