

11-682/15-482:

Introduction to Human Language Technologies

Formal Language Theory

Readings:

Jurafsky and Martin,

“Speech and Language Processing”:

Required: Chapter 2 (all), 9.0-9.2, 9.11

Recommended: Chapter 9 (all)

Formal Languages

- A mathematical abstraction of “real” languages: Natural Languages and Computer Languages
- A language is no more than a set of items called “words” (the equivalent of sentences in a natural language)
- Languages can be defined declaratively, descriptively or computationally
- Formal Language Theory: The study of properties of the various types and classes of languages using formal mathematical proofs
- Fundamental problem - word membership:
Given a word w and a language L - is $w \in L$?
- what algorithm or computational device is necessary to answer this question depends on the class of the language

Basic Definitions

- Σ : the **alphabet** - a finite (non-empty) set of atomic symbols
 - each symbol σ in the set is a **letter**
 - letters will be denoted by lower case Latin letters a, b, c, \dots
- a **word** is a string of letters from a given alphabet Σ
- $|w|$ denotes the length of word w
- ϵ denotes the empty word: $|\epsilon| = 0$
- we will only consider words of finite length
- **Def:** a language L is a set (finite or infinite) of words constructed from a given alphabet Σ
- Note: the languages $L_1 = \{\epsilon\}$ and $L_2 = \emptyset$ are not the same!
- Note: Set Theory and operations apply to formal languages:
 - union, intersection, complementation, membership
 - $\bar{L} = \{w \in \Sigma^* \mid w \notin L\}$

Basic Operations on Words and Languages

- **Concatenation:** denoted by a decimal point $.$:
 - Example: $x = aa \quad y = bb \quad x.y = aabb$
 - the dot is omitted when understood
 - Language Concatenation: $L_1.L_2 = \{x.y \mid x \in L_1 \text{ and } y \in L_2\}$
 - Example: $L_1 = \{a, b\} \quad L_2 = \{c, d\} \quad L_1L_2 = \{ac, ad, bc, bd\}$
 - Note: $L_1.\emptyset = \emptyset \quad L_1.\{\epsilon\} = L_1$
- **The Power Operation:**
 - on letters: $a^1 = a \quad a^2 = a.a \quad a^n = a.a^{n-1}$
 - on words: $w = ab \quad w^2 = w.w = abab \quad w^n = w.w^{n-1}$
 - on languages: $L^0 = \{\epsilon\} \quad L^1 = L \quad L^2 = L.L \quad L^n = L.L^{n-1}$

Basic Operations on Words and Languages

- **The Kleene Star Operation:**

- on letters: $a^* = \bigcup_{i=0}^{\infty} a^i = \{\epsilon, a, aa, aaa, \dots\}$

- on words: $w^* = \bigcup_{i=0}^{\infty} w^i = \{\epsilon, w, w.w, w.w.w, \dots\}$

- on languages: $L^* = \bigcup_{i=0}^{\infty} L^i = \{\epsilon\} \cup L \cup L^2 \cup \dots$

- Important notation:

- Σ^* = set of all finite words over the alphabet Σ

- Σ^i = set of all words of length i over the alphabet Σ

Language Classes

- Sets of formal languages that can be defined using a particular descriptive definition or abstraction of a computational framework
- Examples:
 - The set of languages that can be described by Regular Expressions
 - The set of languages for which we can construct a Finite State Automaton
 - The set of languages that can be defined using a Context-free Grammar
- Knowing the class to which a language belongs will allow us to develop efficient algorithms for processing the language or deciding membership in the language

Regular Expressions

- A formal descriptive formalism for specifying regular languages
- **Definition:**
 1. \emptyset is a RE denoting the empty language
 2. ϵ is a RE denoting the language $\{\epsilon\}$
 3. for each letter $a \in \Sigma$, a is a RE denoting the language $\{a\}$
 4. If r and s are two REs, denoting the languages L_R and L_S then
 - $(r + s)$ denotes the language $L_R \cup L_S$
 - (rs) denotes the language $L_R.L_S$
 - r^* denotes the language L_R^*
- preferences on ops (* , $.$, $+$) allow us to omit parentheses where they are understood
- a convenient abbreviation: rr^* is denoted as r^+

Deterministic FSA

- a mathematical abstraction of a simple computational device
- used primarily to recognize membership in a language, or (with output) to transform the input into some appropriate output
- The **Deterministic Finite State Automaton** consists of:
 - an input tape (of arbitrary length)
 - a “read” head
 - a *finite* set of control states
 - a complete finite control transition table
- **Computation:** at each step, the machine reads the next symbol from the tape, and changes its state, based on the current state and the input symbol read
- The computation ends when the last input symbol is read
- FSA often easy to represent as a transition diagram

Deterministic FSA

- **Formal Definition of a DFSA:** $A = (Q, \Sigma, \delta, q_0, F)$ where:
 - Q is a finite set of states
 - Σ is a finite alphabet
 - $q_0 \in Q$ is an initial (start) state
 - $F \subseteq Q$ is a set of *final* states
 - $\delta : Q \times \Sigma \rightarrow Q$ is the complete transition function
- The language accepted by a DFSA A is defined to be:
 $L(A) = \{w \in \Sigma^* \mid \text{after computing on } w, A \text{ is in a state } q \in F\}$
- based on the function δ , we define $\hat{\delta}$, the function on words that models the computation of a DFSA recursively as follows:
 - $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$
 - $\hat{\delta}(q, \epsilon) = q \quad \forall q \in Q$
 - $\hat{\delta}(q, x\sigma) = \delta(\hat{\delta}(q, x), \sigma)$

Deterministic FSA

- on words of length 1, the $\hat{\delta}$ function is the same as the original δ function that is defined on single letters
- Formal definition of $L(A)$: $L(A) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$
- **Def: Regular Language:** a language $L \subseteq \Sigma^*$ is called *regular* if there exists some DFSA A such that $L = L(A)$
- Examples of regular languages:
 - $L = \Sigma^*$ (for any Σ)
 - $L = \emptyset$
 - $L = \{\epsilon\}$
 - $L = (aab)^*$

Non-Deterministic FSA

- a more general model where at any step the machine can transition into one of possibly several different states defined by the control transition function
- the only difference from a DFSA is in the definition of the δ function
- **Computation** with a NDFSA: at each step, the machine picks *one* of the possible new states. If none are defined, the machine halts.
- the $\hat{\delta}$ computation function for NDFSAs:
 - $\hat{\delta} : Q \times \Sigma^* \rightarrow 2^Q$
 - $\hat{\delta}(q, \epsilon) = \{q\} \quad \forall q \in Q$
 - $\hat{\delta}(q, x\sigma) = \bigcup_{q' \in \hat{\delta}(q, x)} \delta(q', \sigma)$

Non-Deterministic FSA

- a NDFSA *accepts* an input word w if there *exists* a computation path that ends in a final state
- **Def:** The language of a NDFSA A :
$$L(A) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Non-Deterministic FSA with ϵ -Moves

- a further extension of the NDFSA model - allowing transitions without reading the next input symbol: ϵ -transitions
- the choice of whether to do a regular transition or an ϵ -transition is also non-deterministic
- an input word is accepted if there exists an accepting computation path
- the formal definitions (of δ and $\hat{\delta}$) have to be modified to account for the possibility of ϵ -transitions

Equivalence of DFSA, NDFSA and RE

- NDFSA are not more powerful computationally - they can recognize only the set of *Regular* languages
- the formal proof is via a scheme for converting any NDFSA into a DFSA that accepts the exact same language
- All the FSA models can recognize only the set of languages that can be described using Regular Expressions
- Given a RE, we can construct a NDFSA with ϵ -moves that accepts the language described by the RE.
- Given a DFSA, we can construct a RE that describes the language accepted by the DFSA.

Context-Free Grammars

- A descriptive generative formalism for specifying the set of words in a language using production rules
- **Formal Definition:** a *context-free grammar* $G = (V, T, P, S)$
 - V is a finite set of variables
 - T is a finite set of terminal symbols (similar to Σ for FSAs)
 - P is a set of *context-free* production rules, each of the form $A \rightarrow \alpha$, where $\alpha \in (V \cup T)^*$
 - S is a start non-terminal ($S \in V$)
- Notations:
 - we denote elements of V by $S, A, B, C...$
 - we denote elements of T by $a, b, c...$
 - we denote strings over T^* by $w, x, y...$
 - we denote strings over $(T \cup V)^*$ by $\alpha, \beta, \gamma...$
 - we denote single variables or terminals by $X, Y, Z...$

Context-Free Grammars

- Example: $L = \{a^n b^n \mid n \geq 1\}$

G: S \rightarrow a S b

 S \rightarrow a b

- in this case the language $L(G)$ could be specified in a succinct mathematical form - often this is difficult or not possible

CFG Derivations

- derivations describe the process of using the context-free rules to derive a string of terminal symbols
- **Definition:** let $\varphi_1, \varphi_2 \in (V \cup T)^*$.
 φ_1 *directly derives* φ_2 , denoted by: $\varphi_1 \Longrightarrow_G \varphi_2$,
if $\varphi_1 = \alpha A \beta$, $\varphi_2 = \alpha \gamma \beta$ and $A \rightarrow \gamma$ is a rule in P_G
- φ_1 *derives* φ_2 , denoted by $\varphi_1 \xRightarrow{*}_G \varphi_2$,
if there exists a finite sequence of direct derivations such that
 $\varphi_1 \Longrightarrow_G \varphi'_1 \Longrightarrow_G \varphi'_2 \Longrightarrow_G \varphi'_3 \Longrightarrow_G \cdots \Longrightarrow_G \varphi_2$
- $\varphi_1 \xRightarrow{i}_G \varphi_2$ denotes that φ_1 derives φ_2 in exactly i derivation steps
- a *rightmost* derivation is a derivation in which at each step, the rightmost non-terminal in the string is picked for expansion
- similarly for a *leftmost* derivation

Context Free Languages (CFLs)

- **Formal Definition:** the language of a CFG G is defined as:
$$L(G) = \{w \in T^* \mid S \xRightarrow{*}_G w\}$$
- a language L is *context-free* if there exists a grammar G such that
 $L = L(G)$
- the set of all such languages is called the set of context-free languages (CFLs)
- two grammars G_1 and G_2 are called *equivalent* if $L(G_1) = L(G_2)$

Parse Trees

- a Parse Tree is a graphical representation of a derivation
- the leaves (yield) of the tree correspond to a terminal string in $L(G)$
- the tree does not represent the derivation order of the non-terminals
- the tree does reflect the structure of the input string - what rules were used to derive the various substrings of the input
- a parse tree constitutes a proof that a given input string is in $L(G)$
- a grammar G is called *ambiguous* if there exists a word $w \in L(G)$ that has two or more different parse trees
- There exist CFLs that are inherently ambiguous

Normal Forms of CFGs

- restricted forms of CFGs that are computationally or mathematically useful
- **Chomsky Normal Form (CNF):**
productions are all of the following form
 $A \rightarrow B C$
 $A \rightarrow a$
- **Greibach Normal Form (GNF):**
productions are of the following form
 $A \rightarrow a \alpha$, where $\alpha \in V^*$
- Every CFG for which $\epsilon \notin L(G)$ can be converted into an equivalent grammar in CNF and in GNF

Regular Grammars

- a restricted form of CFGs that can only generate regular languages
- productions are all of the following forms:
 - $A \rightarrow a B$
 - $A \rightarrow a$
 - $A \rightarrow \epsilon$
- there are algorithms for constructing FSAs and REs that define/accept the same language defined by a regular grammar

Pushdown Automata

- An extension of a FSA that is powerful enough to accept CFLs
- The FSA is augmented with a memory storage device in the form of a stack
- **Formal Definition:** a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:
 - Q, Σ, q_0, F are similar to those of a FSA
 - Γ is a finite set of stack symbols
 - Z_0 is a start stack symbol
 - $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$
 $\delta(q, \sigma, Z) = \{(q_1, \gamma_1), (q_2, \gamma_2), \dots, (q_m, \gamma_m)\}$
- Note that a PDA can make ϵ -moves on the input, can replace the top element of the stack, can “push” an element onto the stack, can “pop” an element from the stack

Pushdown Automata

- Example: $L = \{w c w^R \mid w \in \{0, 1\}^*\}$
- a grammar for L : $S \rightarrow 0S0 \mid 1S1 \mid c$
- a PDA for recognizing L :
 - state q_1 for pushing letters onto the stack
 - state q_2 for popping letters from the stack and matching them to the input
 - state q_3 final accepting state
 - when we see the c , transition from q_1 to q_2

Equivalence of CFLs and PDAs

- for a given grammar G in GNF, we can construct a PDA that simulates leftmost derivations in G and thus accepts the exact same language
- since any CFG can be converted to GNF, this shows that PDAs can recognize all CFLs
- for a given PDA M , we can also construct a grammar G such that every leftmost derivation in G corresponds to a valid computation of the PDA that accepts the input, thus showing that PDAs can only recognize CFLs

Recognition and Parsing of CFLs

- the recognition problem:
given a grammar G and a word w , is $w \in L(G)$
- the parsing problem:
given a grammar G and a word w , if $w \in L(G)$, find a parse tree (or all possible parse trees) for w
- there exist a variety of algorithms for parsing CFLs and their variants
- we will discuss one particular algorithm in class

Context Sensitive and Unrestricted Grammars

- CFGs are called *context-free* because the form of the grammar rules allows them to be used in a derivation regardless of the context in which a non-terminal appears
- there exist less restricted forms of grammars:
- **Context Sensitive** grammars are grammars where the rules have the form $\alpha \rightarrow \beta$, with the restriction that $|\alpha| \leq |\beta|$
- in order to be applied in a derivation, the entire left-hand side of the rule must match a substring of the current derived string
- **Unrestricted** grammars are grammars where the rules are unrestricted in form - $\alpha \rightarrow \beta$, where α contains one or more grammar symbols, and β contains zero or more grammar symbols
- more powerful computation devices are required in order to recognize the languages defined by these types of grammars

The Chomsky Hierarchy

- Chomsky was one of the pioneers in identifying the correspondence between the different types of grammars and the formal computational models that are required to recognize them:
- **Type-0 Grammars** are unrestricted grammars, correspond to recursively enumerable languages, require Turing Machines to recognize them
- **Type-1 Grammars** are context-sensitive grammars, correspond to context-sensitive languages and require a type of automata called *linear-bounded automata* to recognize them
- **Type-2 Grammars** are context-free grammars, correspond to CFLs and require PDAs to recognize them
- **Type-3 Grammars** are regular grammars, correspond to regular languages and require FSAs to recognize them
- The syntax of natural languages is often described by phrase structure rules that are “extended” CFGs. Algorithms for parsing them are often based on extensions of parsers for CFGs.