CPS 220 – Theory of Computation

Review - Regular Languages

RL - a simple class of languages that can be represented in two ways:

1 *Machine description*: Finite Automata are machines with a finite number of states and no extra memory, recognizing exactly the Regular Languages. Finite Automata can be Deterministic or Nondeterministic, and these two kinds are equivalent in the sense that they recognize the same languages, but it is interesting to note that Nondeterministic Finite Automata can provide much more concise (in the extreme case, exponentially shorter!) descriptions of the same language as Deterministic ones.

2 *Syntactic description*: Regular Expressions are a class of expressions built out of a given alphabet $\Sigma \cup \{ \varepsilon \}$, and with operations Union, Concatenation, and Star.

Some important properties of Regular Languages are:

1. If a language L is finite, then it is regular.
2. If a language L is regular, then the complement of L (Σ^* - L) is also regular. This closure under complementation is a very important and "rare" property.
3. If L is regular, then L^R, the language of the strings of L reversed, is regular.
4. If L_1 and L_2 are regular, then $L_1 \cup L_2$, L_1L_2, and $L_1 \cap L_2$ are regular. The closure under intersection follows from closure under complementation, because:

$$L_1 \bigcap L_2 = \overline{\overline{L_1} \bigcup \overline{L_2}}$$

[Note: stay tuned for project 1 - dealing with regular expressions]

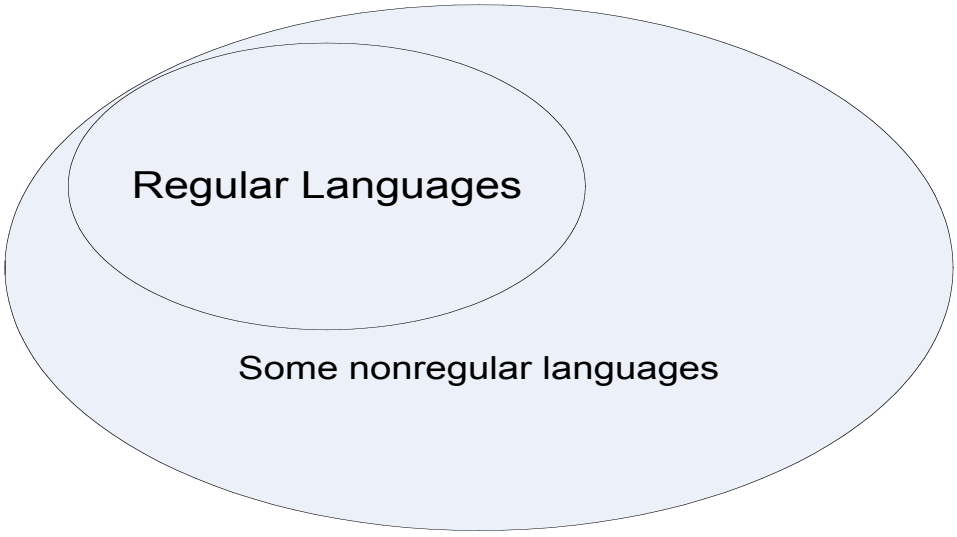
In order to prove that a language is not regular, a very useful tool is the Pumping Lemma

New Topic: Now we will start on a new topic, and we will describe a more powerful model of computation: the Context-Free Languages (CFL). This category of languages again has two ways to be described:

1 *Machine description*: CFLs can be recognized by Pushdown Automata (PDA). Those are basically Nondeterministic Finite Automata with an additional memory device, the stack. The stack is an unlimited size, First-In First-Out (FIFO) memory. One important thing to keep in mind is that the PDAs *need* nondeterminism in order to recognize the CFLs. So when the stack comes to play, the equivalence of the deterministic and nondeterministic machines breaks.

2 *Syntactic description*: CFLs can be recognized by a syntactic way of producing strings according to a finite set of rules, the Context-Free Grammars.

Context Free Grammars and Languages are commonly used in syntactic *parsers*, such as those seen in compilers or in the XML language.



Languages generated by CFGs

1. Context-Free Grammars - "more powerful method for describing languages"

Let's start with an example:

$A \rightarrow 0\ A\ 1$
$A \rightarrow B$
$B \rightarrow \varepsilon$

Start: A; $\Sigma = \{0, 1\}$

Definitions: substitution rules, variable, terminal, start variable, derivation

This is a Context-free Grammar (CFG).

Here is an example of using it to produce (or *derive*) a string:

$\mathbf{A \rightarrow 0\ A\ 1}$
$\mathbf{\rightarrow 0\ 0\ A\ 1\ 1}$
$\mathbf{\rightarrow 0\ 0\ 0\ A\ 1\ 1\ 1}$
.
.
.
$\mathbf{\rightarrow 0^n\ A\ 1^n}$
$\mathbf{\rightarrow 0^n\ B\ 1^n}$
$\mathbf{\rightarrow 0^n\ 1^n}$

L(G) = all strings which can be generated (language of the grammar)

L(G) is not a regular language. Why? FA does not have enough memory - however a new model
(PDA) has stack memory.

Definition: A context-free grammar (CFG) is a 4-tuple (V, T, P, S) such that:

1. V is a finite set of *variables*, or *nonterminals*.
2. Σ is the alphabet, here also called the set of *terminals*.
3. R is a set of derivation rules, or *productions*. Each rule is of the form: Variable $\rightarrow$ String of variables & terminals.
4. S is a designated *start symbol*. ($S \in V$)

2. Derivations

Definition: A Derivation.

1 We say that string u *yields* string v, denoted $u \Rightarrow v$, if u turns to v after one application of a derivation rule.
example: $0\ A\ 1 \Rightarrow 0\ 0\ A\ 1\ 1$

2 If u turns to v after many rule applications then we say that $u \Rightarrow^* v$.
example: $0\ A\ 1 \Rightarrow^* 0\ 0\ 0\ 0\ 0\ 0\ A\ 1\ 1\ 1\ 1\ 1\ 1$

3 The sequence $u \Rightarrow v_1 \Rightarrow v_2 \Rightarrow \ldots \Rightarrow v_k \Rightarrow v$ is called a derivation of v from u.

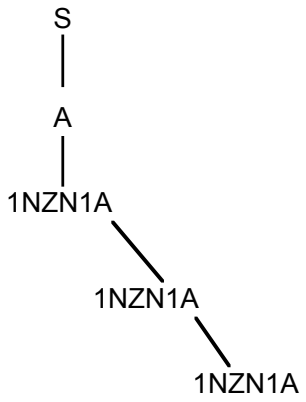
Definition: The language of a grammar G, $L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^* w \}$

Definition: A *Context-free language* (CFL) is a language generated by a CFG.

Practice Problem:
Let $\Sigma=\{0,1\}$ and let
L(G)={w|w contains an equal number of occurrences of the substrings 01 and 10}.

Solution: G=({S},{0,1},R,S)

Set of rules - R
$S \rightarrow A|B$
$A \rightarrow 1NZN1A|\varepsilon$
$B \rightarrow 0ZNZ0B|\varepsilon$
$Z \rightarrow 0Z|\varepsilon$
$N \rightarrow 1N|\varepsilon$
$Z \rightarrow \varepsilon$



1 ε 11 0 1 1ε 0ε1 ε
[parse tree]

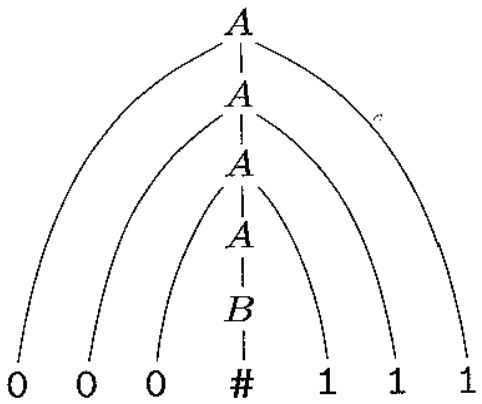
3. Parse Trees.

A derivation can be depicted in a parse tree.

Example: Σ = { 0, 1, # }; V = { A }

A → 0 A 1 | #

What would the parse tree look like for 000#111?



Reading the leaves of the tree from left to right gives the produced string.

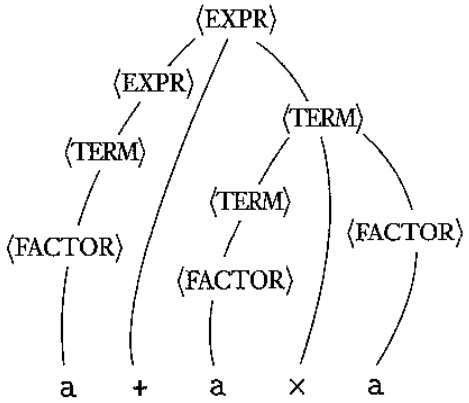
The language makes even more sense like this:
A → 0 A 1 | ε

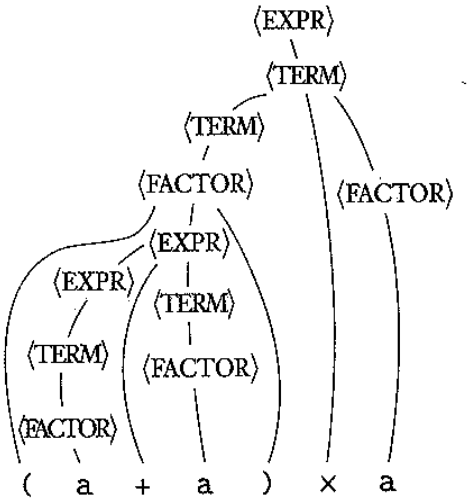
Example: A grammar of arithmetic expressions with parentheses.

$E \rightarrow E + T \mid T$
$T \rightarrow T \times F \mid F$
$F \rightarrow (E) \mid a$

Variables (nonterminals): { E, T, F }
Symbols (terminals): a + × ( )

Parse tree for strings a + a × a and (a + a) × a





What would the parse tree look like for result: (a + a) × (a + a)?

Note: This language "remembers" to close the right number of parentheses opened. A FA cannot do that because it has only a finite amount of "memory" hardwired in its states.

Example: $L = \{ 0^n 1^k 2^n \mid k \geq 0, n \geq 0 \}$

A grammar for L:

$A \rightarrow 0 A 2 \mid B$
$B \rightarrow 1 B \mid \varepsilon$

Example: $L = \{ w w^R \mid w \in \Sigma^* \}$

A grammar for L: $A \rightarrow 0 A 1 \mid 1 A 0 \mid \varepsilon$

Designing CFL

1. Divide and conquer

For example to get grammar: $\{ 0^n 1^n \mid n \geq 0 \} \cup \{ 0^n 1^n \mid n \geq 0 \}$

$$G1 \qquad S_1 \rightarrow 0S_1 1 \mid \varepsilon$$
$$G2 \qquad S_2 \rightarrow 1S_2 0 \mid \varepsilon$$

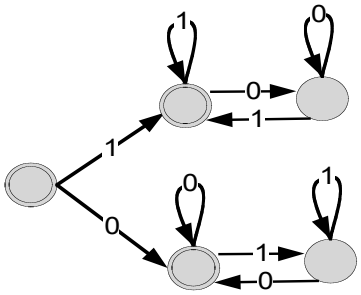
add this starting substitution rule

$$S \rightarrow S_1 \mid S_2$$

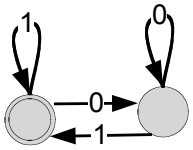
2. If the language is <u>regular</u> - create a DFA and then convert to CFG as follows:

a. Make a variable Ri for each state qi of the DFA.
b. Add the rule Ri →aRj to the CFG if there is a DFA transistion (with a) from state Ri to Rj.
c. Add the rule Ri → ε if qi is an accept state.
d. Make R0 the start variable - where q0 is the start state of DFA.

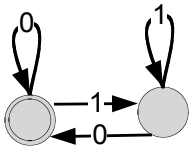
example:



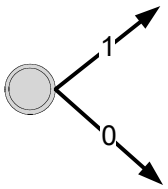
let's start with:



R1 --> 0R2|1R1|ε
R2 --> 1R1|0R2



R3 --> 1R4|0R3|ε
R4 --> 0R3|1R4



R0 --> R1|R3|ε

Test out the resulting grammar.

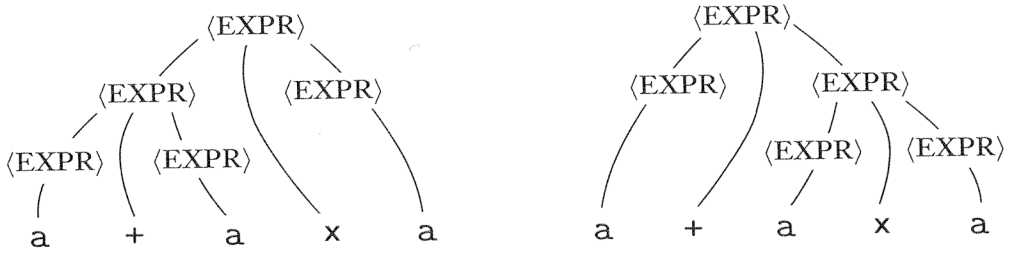
c. use this substitution rule R-->uRv

Example: 000111 S-->0S1|ε

==========

Ambiguous grammar - grammar generates the same string in multiple ways (has several different parse trees)

E → E+E | E x E | (E) | a



Two different parse tree for same string a+a x a

E-->ExE-->E+ExE-->a+ExE-->a+axE-->a+axa
E-->E+E-->a+E-->a+ExE-->a+axE-->a+axa

Generated ambiguously - 2 different parse trees, not 2 different derivations (same derivated string)

Leftmost derivation - leftmost variable is the one replaced
<<some CFL can be generated only my ambiguous grammars (inherently ambiguous)>>

==========
==

The Chomsky Normal Form

The Chomsky Normal Form (CNF) allows only the following two kinds of productions:

$A \rightarrow BC$, where B,C are nonterminals, and
$A \rightarrow a$, where a is a terminal.

Two more details:
1. The start symbol, S, cannot be at the rhs of any production.
2. We permit the special rule $S \rightarrow \varepsilon$.

This is a very useful form when designing algorithms on CFGs, and it is very simple to study mathematically.

Algorithm for converting a CFG into Chomsky Normal Form:

1. Create a new start symbol, S_0, and add the rule $S_0 \rightarrow S$. Add the rule $S_0 \rightarrow \varepsilon$ if ε could be produced by the grammar.
2. Remove ε-productions, except the possible one from S_0.

   To do so, whenever $R \rightarrow u_0 A_1 u_1 A_2 \ldots u_{k-1} A_k u_k$ is a rule, and A yields ε (in any number of steps!), add the $2^k - 1$ rules:

   $$R \rightarrow u_0 u_1 \ldots u_k$$
   $$R \rightarrow u_0 A_1 u_1 u_2 \ldots u_k$$
   $$\ldots$$
   $$R \rightarrow u_0 A_1 u_1 \ldots u_{k-2} A_{k-1} u_{k-1} u_k$$

   Example:

   $$A \rightarrow BaBaD$$
   $$B \rightarrow b \mid \varepsilon$$
   $$C \rightarrow c \mid \varepsilon$$

   These rules become:

   $$A \rightarrow aa \mid Baa \mid aBa \mid aaD \mid BaaD \mid BaBa \mid aBaD \mid BaBaD$$
   $$B \rightarrow b$$
   $$C \rightarrow c$$

3. Remove unit productions. Those are productions of the form $A \rightarrow B$ where A and B are nonterminals.

   To do so, find all nonterminal pairs X, Y such that $X \rightarrow B_1 \rightarrow \ldots \rightarrow B_k \rightarrow Y$ is a series of unit productions. For every such pair (X, Y), and for every production

Y→u other than the unit productions, add the production X→u to the grammar.

Example:

$S \rightarrow AB \mid A$
$A \rightarrow C \mid a \mid aa$
$B \rightarrow b$
$C \rightarrow cc$

Becomes:

$S \rightarrow AB \mid a \mid aa \mid cc$
$A \rightarrow a \mid aa \mid cc$
$B \rightarrow b$
$C \rightarrow cc$

Notice now that C is useless – we can remove it and its productions, if we wish.

4. Arrange all remaining productions $A \rightarrow u$ with $|u| \geq 2$, to contain only nonterminals.

   Example:

   $A \rightarrow cdAce$

   Generate new nonterminals C, D, and E, and change the above rule to:

   $A \rightarrow CDACE$
   $C \rightarrow c$
   $D \rightarrow d$
   $E \rightarrow e$

5. Now each production is of the form $A \rightarrow a$, where a is a terminal, or $A \rightarrow B_1 \ldots B_k$, where each B_i is a nonterminal. If $k > 2$, change the production to:

   $A \rightarrow B_1C_2$
   $C_2 \rightarrow B_2C_3$
   …
   $C_{k-1} \rightarrow B_{k-1}B_k$

Example:
Grammar G:
$S \rightarrow 0\ S\ 1 \mid A \mid \varepsilon$
$A \rightarrow 1\ A \mid \varepsilon$

1. New start symbol S_0

   New grammar:
   $S_0 \rightarrow S \mid \varepsilon$
   $S \rightarrow 0\ S\ 1 \mid A \mid \varepsilon$
   $A \rightarrow 1\ A \mid \varepsilon$

2. Eliminate ε productions (except from S_0):

   New grammar:
   $S_0 \rightarrow S \mid \varepsilon$
   $S \rightarrow 0\ S\ 1 \mid 0\ 1 \mid A$
   $A \rightarrow 1\ A \mid 1$
   Notice, now the new grammar does not produce ε (G did!)

2. Eliminate unit productions. We have to eliminate $S \rightarrow A$

   New grammar:
   $S_0 \rightarrow 0\ S\ 1 \mid 0\ 1 \mid 1\ A \mid 1 \mid \varepsilon$
   $S \rightarrow 0\ S\ 1 \mid 0\ 1 \mid 1\ A \mid 1$
   $A \rightarrow 1\ A \mid 1$

3. Arrange all remaining productions $X \rightarrow u$ where $|u| \geq 2$ to contain only nonterminals.

   The grammar becomes:
   $S_0 \rightarrow C\ S\ D \mid C\ D \mid D\ A \mid 1 \mid \varepsilon$
   $S \rightarrow C\ S\ D \mid C\ D \mid D\ A \mid 1$
   $A \rightarrow C\ A \mid 1$
   $C \rightarrow 0$
   $D \rightarrow 1$

4. Finally, arrange all $X \rightarrow u$ to have $|u| \leq 2$, by adding new nonterminals if needed.

   New grammar:
   $S_0 \rightarrow C\ S' \mid C\ D \mid D\ A \mid 1$
   $S \rightarrow C\ S' \mid C\ D \mid D\ A \mid 1$
   $S' \rightarrow S\ D$
   $A \rightarrow C\ A \mid 1$
   $C \rightarrow 0$
   $D \rightarrow 1$