SAN JOSE STATE UNIVERSITY
DEPARTMENT OF ELECTRICAL ENGINEERING

CS 154 Formal Languages and Computability Spring 2012
Section 1 Room MH 225 Class 14: 03-12-12

T. Howell

Test 1 Discussion
Average : 59%
Show histogram
Estimated grades: 80-100 = A
65-80 = B
50-65 = C
40-50 = D
0-40 = F

Averages by problem:

1	7.9/15 = 52%	NFA to DFA
2	6.4/10 = 64%	RE to DFA
3	13.3/20 = 67%	T/F about Regular languages
4	9.4/20 = 47%	Pumping Lemma
5	7.5/15 = 50%	DFA to RE
6	10.4/15 = 69%	Product construction
7	10.4/15 = 69%	DFA minimization

Prob. 1 should have been easy. Many forgot that a DFA needs a transition from each state for each input.
Prob. 2 was harder than I intended it to be. I changed it to 10 points plus 5 extra credit. You got 10 points (full credit) for giving an NFA. Only a few were successful converting it to DFA.
Go over answers to Prob. 3 (T/F).
Pumping Lemma was toughest by a small margin over DFA to RE. We need to work on this. The CFL pumping lemma will be harder.
Problem for HW 4: Write out a careful proof that one of the languages in Prob. 4 is not regular. You may use the pumping lemma or you may adapt the proof of the pumping lemma, but you may not use any other languages known to be non-regular.
Prob. 5: Those who didn't use the algorithm as a guide generally floundered and did not get all the strings. Some of those who remembered the algorithm but hadn't practiced using it chose a poor state to drop and got confused. Those who knew how to use the algorithm and chose the middle state to drop could read off the answer almost by inspection. Some did.
Problems 6 and 7 were best. Maybe they were freshest in your minds. The product machine in Prob. 6 has 6 states (can be reduced to 5). The component machines have 2 and 3 states, respectively. The minimization algorithm in Prob. 7 removes one unreachable state (3) and combines (0, 4) and (2,5). The resulting machine has three states.

Last class
Context free Grammars and Languages

We defined grammars (CFG), productions, terminal symbols and variables, derivations, sentential forms, context-free languages. We looked at several examples.

Formal Definition:
$G = (N, \Sigma, P, S)$ is a context free grammar.

N is a finite set of nonterminals (variables).
Σ is a finite set of terminals (the alphabet.) $N \cap \Sigma = \varnothing$.
P is a finite subset of $N \times (N \cup \Sigma)^*$: *productions*.
$S \in N$ is the start symbol.

By convention, nonterminals (variables) are upper case letters, terminals are lower case letters, and strings (sentential forms) are lower case Greek letters in our examples. We write productions as $A \rightarrow \alpha$.

Roadmap of CFG/PDA unit

Grammars
Parsing and ambiguity
Push-down Automata: PDA/NPDA
Equivalence of grammars and PDA's
Chomsky Normal Form: CNF
Pumping Lemma
Decision Problems and CKY algorithm

Parsing and ambiguity
Parsing is the process of producing a tree illustrating the derivation for a given string. The order in which productions are applied in the derivation is often not important. The same parse tree corresponds to many derivations which differ only in the order in which productions are applied. Right-most and left-most derivations specify definite orders for the productions. Parse trees can be used to prove that a string is in the language and to learn about its structure. Parsing is very important for compilers and other language processing tasks.

Sections 5.2 and 5.3 of HMU cover parse trees and some applications. You should read these sections, but we will not cover them in class. Now we move on to ambiguity: some strings may have more than one different parse tree. This is usually an undesirable situation since different parse trees may imply different meanings for the same string.

Example:

Using the expression grammar from class 11:

$E \rightarrow I \mid E + E \mid E * E \mid (E)$
$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

The string a + a * a has more than one parse tree. (Draw trees)

$E \rightarrow E + E \rightarrow E + E * E \rightarrow a + a * a$ (meaning a + (a* a))
and
$E \rightarrow E * E \rightarrow E + E * E \rightarrow a + a * a$ (meaning (a + a) * a)

This kind of ambiguity can be fixed by modifying the grammar to include separate variables for additive and multiplicative expressions.

$E \rightarrow ME \mid E + ME \mid (E)$
$ME \rightarrow I \mid ME * I \mid (ME)$
$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Now the only tree for a + a * a is
$E \rightarrow ME + ME \rightarrow ME + ME * ME \rightarrow a + a * a$ (meaning a + (a* a))

This grammar fragment (borrowed from C) forces * to have precedence over + and for each operation to associate left-to-right.

A language is inherently ambiguous if all of its grammars are ambiguous. Our original grammar for expressions was ambiguous, but its language is unambiguous because it has at least one unambiguous grammar.

An example of an inherently ambiguous language (proof only sketched in HMU) is

$L = L_1 \cup L_2$, where $L_1 = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\}$ and $L_2 = \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$

A grammar for it is:

$S \rightarrow AB \mid C$
$A \rightarrow aAb \mid ab$
$B \rightarrow cBd \mid cd$
$C \rightarrow aCd \mid D$
$D \rightarrow bBc \mid bc$

AB generates strings in L_1, while C generates strings in L_2. Strings in $L_1 \cap L_2$ can be generated either way. There is no way around this as will be clearer when we have studied PDAs.