COMS W4115 Solutions to Homework #1
Homework assigned Monday, February 4, 2013
Answers due February 13, 2013

1. Lex programs.

a. Lex program to find the first longest lowercase English word in a dictionary that can be made up using only the letters in "alfre":

The Lex regular expression `^[alfre]+$` matches words that can be made up using only the letters in "alfre".

Lex program using a semantic action to record the first longest word:

```
%{
            #include <stdio.h>
            #include <string.h>
            int maxlen = 0;
            char maxstr[100];
%}
%%
^[alfre]+$  { if (yyleng > maxlen){
              maxlen = yyleng;
              strcpy(maxstr, yytext); }}
\n|.        ;
%%
int main()
{
  yylex();
  if (maxlen == 0)
    printf("no word was found\n");
  else
    printf("longest word is %s\n", maxstr);
}
```

First longest word in /usr/share/dict/words is `referral`.

b. Lex program to find first lowercase English word that is an anagram of "alfre".

This Lex program works by finding a five-letter word that can be made up of letters only in "alfre" using the Lex pattern `^[alfre]{5}$`. It then uses the semantic action to sort the letters in the word alphabetically to see if the sorted word matches the string "aeflr". If it does, it has found an anagram. This program finds the first anagram but can be easily extended to find all anagrams.


```
%{
              #include <stdio.h>
              #include <string.h>
              int found = 0;
              char word[5];
              char anagram[5];
              char target[] = "aeflr";
%}
%%
^[alfre]{5}$  { if (found == 0) {
                  strcpy(word, yytext);
                  sort(word);
                  if (strcmp(word, target) == 0 ) {
                    strcpy(anagram, yytext);
                    found = 1;
                  }
                }
              }
\n|.          ;
%%
int main()
{
  yylex();
  if (found == 0)
    printf("no anagram was found\n");
  else
    printf("first anagram is %s\n", anagram);
}

void sort(char s[])
{
  char temp;
  int i, j;
  for (i = 0; i < 4; i++) {
    for (j = i+1; j < 5; j++) {
      if (s[i] > s[j]) {
        temp = s[i];
        s[i] = s[j];
        s[j] = temp;
      }
    }
  }
}
```

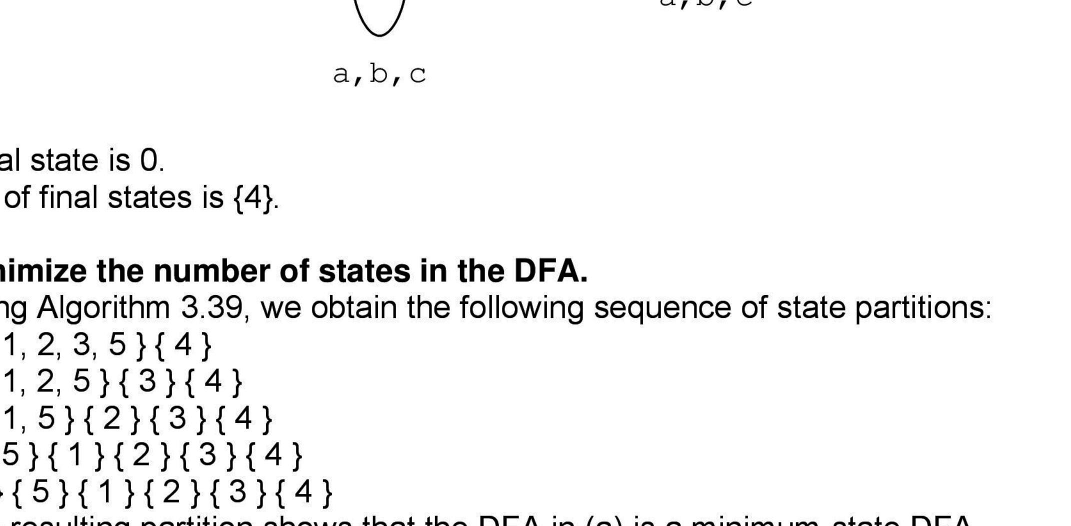
First anagram in /usr/share/dict/words is `feral` (`flare` is another).

A much easier way of finding anagrams is to use the Linux command pipe:

```
egrep '^[alfre]{5}$' | egrep -v 'a.*a|l.*l|f.*f|r.*r|e.*e'
```

2. Let *L* be the language consisting of all strings of *a*'s, *b*'s, and *c*'s such that each string is of the form *abxba* where *x* does not contain *ba* as a substring.

a. Write down a DFA for *L*.
Set of states Q is {0, 1, 2, 3, 4}.
Input alphabet Σ is {a, b, c}.
Transition function δ is given by the following transition diagram:



Initial state is 0.
Set of final states is {4}.

b. Minimize the number of states in the DFA.
Using Algorithm 3.39, we obtain the following sequence of state partitions:
{ 0, 1, 2, 3, 5 } { 4 }
{ 0, 1, 2, 5 } { 3 } { 4 }
{ 0, 1, 5 } { 2 } { 3 } { 4 }
{ 0, 5 } { 1 } { 2 } { 3 } { 4 }
{ 0 } { 5 } { 1 } { 2 } { 3 } { 4 }
The resulting partition shows that the DFA in (a) is a minimum-state DFA.

c. Explain what property of the scanned input each state of the minimized DFA recognizes.
State 0 recognizes the empty string.
State 1 recognizes the string *a*.
State 2 recognizes any string in $ab(a|b^*c)^*$.
State 3 recognizes any string in $ab(a|b^*c)^*bb^*$.
State 4 recognizes any string in $ab(a|b^*c)^*bb^*a$.
State 5 recognizes any string in
$a(a|b)(a|b|c)^* \mid (b|c)(a|b|c)^* \mid ab(a|b^*c)^*bb^*a(a|b|c)(a|b|c)^*$.

d. Write down a regular expression for *L*.
$ab(a|b^*c)^*bb^*a$

3. Let *L* be the language consisting of balanced parentheses.

a. Write down a recursive definition for *L*.
Basis: The empty string is a string of balanced parentheses.
Induction: If *x* and *y* are strings of balanced parenthesis, then (*x*)*y* is a string of balanced parentheses.
See Example 4.12 in ALSU for an inductive proof that this definition generates all strings of balanced parentheses and only such strings.

b. Use the pumping lemma for regular languages to show that *L* cannot be specified by a regular expression.
Assume L can be specified by a regular expression. This means L must be a regular language and so the pumping lemma applies to L. Let n be the constant that the pumping lemma associates with L and consider the string $w = (^n)^n$ that is in L. The pumping lemma states that w can be written as xyz where y is not empty, $|xy| \leq n$, and for all $k \geq 0$, xy^kz is in L. Setting $k = 0$ implies that $xz = (^p)^n$ must be in L where $p < n$, which cannot be true. We must therefore conclude that L cannot be a regular language and that L cannot be specified by a regular expression.

4. Let *R* be a regular expression of length *m* and let *w* be an input string of length *n*. Briefly discuss in terms of *m* and *n* the time-space complexity of the McNaughton-Yamada-Thompson algorithm to determine whether *w* is in *L*(R).

The MYT algorithm (Algorithm 3.23) converts the regular expression R to an epsilon-NFA N with at most $2m$ states. The conversion algorithm can be implemented in $O(m)$ time and $O(m)$ space. The NFA N has one start state with no incoming transitions and one final state with outgoing transitions. The NFA can be simulated with the two-stack algorithm of Section 3.7.3 on the input string w in $O(m \times n)$ time and $O(m)$ space.