CS 311-02 Formal Languages and Automata
Project #2
(Due: 1 PM, Tuesday, 2/21/2017)

A finite state automaton FSA, in general, is designed to recognize words of a predetermined language. However, in some application, it is also desirable to have a FSA that can recognize words it rejects before. For instance, consider the symbol table manager of a compiler: it initially recognizes only a set of reserved words, but during the compilation of a program, it recognizes also identifiers it has seen before. Such a FSA need the ability to modify its language dynamically, and hence be referred to a ***dynamic finite state automaton***.

Your job for this assignment is to implement the dynamic finite state automaton that recognizes none initially, but some after reading files Input1 and Input2 described as follows. Both files are now posted on Bb.

Input1.txt: Reserved words of Java

```
abstract
boolean  break  byte
case  catch  char  class  continue
default  do  double
else  enum  extends
final  finally  float  for
if  implements  import  instanceof  int  interface
long
native  new
package  private  protected  public
return
short  static  super  switch  synchronized
this  throw  throws  transient  try
void  volatile
while
```

Input2.txt: A Java program

During this process, your dynamic FSA should extract identifiers and expand its vocabulary whenever a new identifier is read. An identifier must begin with a letter, an underscore (_), or a dollar sign ($) followed by any combination of letters, digits, underscores, and dollar signs. Both cases of letter can be used, and Java is case-sensitive.

You should use three special markers for different types of vocabulary: * for the reserved words, ? for the new identifiers, and @ for the identifiers not shown at the first time. **Your program should ignore punctuations and numbers, and process identifiers only.**

For instance, the output for the following program

```
import java.util.Scanner;
public static void main()
{
   int hello = 1;
   Hello = there * there + hello;
}
```

should be

```
import* java? util? Scanner?
public* static* void* main?
int* hello?
Hello? there? there@ hello@
```

Clearly, a dynamic finite state automaton must be universal, which means, as I have explained in project #1, the transition information has to be stored in some general data structure to enable modification. Instead of using a transition table as in project #1, you should use a transition list this time that links transitions out of each state together. This approach saves space, and also makes the concept of state implicit since instead of finding the next state for a given symbol, you will find the first transition out of that state.

The transition list contains three arrays:

```
switch  : array [letters, _, $] of integer;
symbol : array [0..max_transition] of character;
next     : array [0..max_transition] of integer;
```

Transition Mechanism.

```
 1.  symbol = getNextSymbol();
 2.  ptr = transition.switch[symbol];
 3.  if ptr is undefined
 4.  then create(); // new identifier
 5.  else {
 6.        symbol = getNextSymbol();
 7.        exit = false;
 8.        while not exit {
 9.            if (transition.symbol[ptr] == symbol)
10.            then if symbol is not the endmarker
11.                   then { ptr = ptr + 1;
12.                          symbol = getNextSymbol(); }
13.                   else { print endmarker; exit = true; }
14.            else if transition.next[ptr] is defined
15.                 then ptr = transition.next[ptr];
16.                 else { create(); exit = true; } // new identifier
17.        } //while
18.  } //if
```

Submit on Bb and turn in the printout of (1) program code in Java or C++ with proper comments including your name and clear instructions to compile, link, and run your program (2) output of executions including both a list of all identifiers in input file #2 and the content of your transition list. Remember each identifier in the output should be immediately followed by one of the three markers *, ?, or @ depending on whether it is a reserved word, a new identifier, or an identifier which has shown before, respectively. The switch, symbol, and next arrays of your transition list must be printed with proper indices and appropriately aligned as shown below.

```
           A    B    C    D    E    F    G    H    I    J    K    L    M    N    O    P    Q    R    S    T
switch:   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1

           U    V    W    X    Y    Z    a    b    c    d    e    f    g    h    i    j    k    l    m    n
switch:   -1   -1   -1   -1   -1   -1    0    8   24   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1

           o    p    q    r    s    t    u    v    w    x    y    z    _    $
switch:   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1   -1

           0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18   19
symbol:    b    s    t    r    a    c    t    *    o    o    l    e    a    n    *    r    e    a    k    *
next:                                          15                                  20

          20   21   22   23   24   25   26   27   28   29   30   31   32   33   34   35   36   37   38   39
symbol:    y    t    e    *    a    s    e    *    t    c    h    *    h    a    r    *    l    a    s    s
next:                         32   28                               36                  41

...
```

Grading:
90% – correctness including turn-in materials and output format
10% – program structure and readability

Your project must be your own work. Copying programs or parts of programs will receive a zero grade and will be reported to the CS department and the University.

Absolutely no late project will be accepted. If you cannot complete the project by the due date, then submit whatever you have completed. Partial credit will be given for reasonable partial solutions.