```
fsa
n101  see  textbk  p54
       0     1       ..
 q1    q1    q1,q2   ..
 q2    q3    ..      q3
 q3    ..    q4      ..
*q4    q4    q4      ..
```

Follow the **guidelines** of the previous homework for packaging, submission, and allowable use of resources. There is no paper submission for this homework. A *general guideline* is that you must not use resources that provide solutions to the focus of the assigned questions (e.g., NFA implementations, conversions) but may use those that help with peripheral concerns (e.g., parsing, debugging). Use the class newsgroup for clarifications. **Questions marked with a** ⋆ are optional, for extra credit. While highly recommended, these questions are also graded more strictly than the others. Therefore, you should work on the ⋆ questions only after you are satisfied with your work on the others. The **goal** of this assignment is to solidify our understanding of finite-state automata and their properties. We have studied these concepts abstractly in class and in readings, and worked out a few small concrete examples by hand as well. We will now study them in a more exhaustively concrete setting by implementing the algorithms and transformations described in proofs and elsewhere. As a vehicle for this study, we will extend the *Lexaard* language of the previous assignment.

1. (50 pts.) Extend Lexaard to support nondeterministic finite-state automata (NFAs). In particular, extend the `define`, `print`, and `run` statements to work with NFAs. The syntactic representation of NFAs in Lexaard is very similar to the representation used for deterministic automata (DFAs) earlier, as illustrated by the above representation of the NFA N_1 from page 54 of the textbook. The representation of NFAs extends the earlier representation of DFAs as follows:

   - The alphabet row may include the token `..` (two adjacent periods) to represent the empty string ϵ. Entries in each subsequent row in the column corresponding to this token denote the destinations of ϵ-transitions from the state in the first column of that row. In the example above, the last entry in the fifth row denotes an ϵ-transition from q_2 to q_3.
   - In the transition rows which follow the alphabet row, each entry denoting a transition's set of destination states represents those states as a comma-separated list. There is a single comma between adjacent states in this representation, with no whitespace or other delimiters. The empty set of states is represented by the token `..` (two adjacent periods). Although this token is identical to that used for ϵ in the alphabet row, the meaning is clear from the context (transition rows v. alphabet row). The states in each entry should be listed in lexicographic order by name, but Lexaard should accept lists in other orders too. In the example above, the third column of the fourth row denotes the transition $\delta(q_1, \texttt{1}) = \{q_1, q_2\}$ and the third column of the fifth row denotes $\delta(q_2, \texttt{1}) = \emptyset$.

   A nice feature of this representation is that an NFA that does not use any nondeterministic features is syntactically identical to the corresponding DFA.

2. (30 pts.) Extend Lexaard with a function `nfa2dfa` that converts NFAs to equivalent DFAs by implementing the algorithm described in the proof of Theorem 1.39 in the textbook. The function is invoked in the language by listing its name followed by its argument, which must be the name of an FSA defined earlier. Applying `nfa2dfa` to a DFA is not an error. For example, if `n4` is a name bound to the NFA N_4 on page 57 of the textbook then the following binds `n4dfa` to the DFA of Figure 1.43 on page 58:
`define n4dfa nfa2dfa n4`

3. (30 pts.) Extend Lexaard with a function `dfaUnion` that implements the algorithm described in the proof of Theorem 1.25 on page 45 of the textbook. This function is invoked by listing its name followed by the names of two DFAs. If `d1` and `d2` are bound to DFAs then the following statement binds `d1ord2` to a DFA that recognizes the union of the languages of `d1` and `d2`:
`define d1or2 dfaUnion d1 d2`

4. (30 pts.) Extend Lexaard with functions that implement the NFA union, concatenation, and star operations using the algorithms described in the proofs of Theorems 1.45, 1.46, and 1.47 in the textbook. These functions are named `nfaUnion`, `nfaConcat`, and `nfaStar`, respectively, and are invoked using a syntax similar to that used by functions `nfa2dfa` and `dfaUnion` above. For example, if `nn1` and `nn2` are bound to the NFAs N_1 and N_2 of Figure 1.48 on page 61 of the textbook then the following statement binds `nn1nn2` to the NFA N suggested by that figure:
`define nn1nn2 nfaConcat nn1 nn2`

5. (30 pts.) Extend Lexaard with a function `pruneFSA` that prunes automata by removing unreachable states. It is invoked following earlier conventions. For example, if `n4dfa` as in Question 2 then then the following statement binds `n4dfapruned` to the DFA of Figure 1.44 on page 58 of the textbook.
`define n4dfapruned prune n4dfa`

6. (30 pts.) Implement a function `fsaEquivP` that evaluates to `true` if its two arguments are equivalent FSAs and to `false` otherwise. Use the method described in the proof of Theorem 4.5 on page 197 of the textbook. (Although we have not covered that chapter yet, this portion is easy to understand based on what we have already studied.) For example, if `n4` and `n4dfa` are as in Question 2 then the following statement binds `eqvAB` to `true`.
`define eqvAB fsaEquivP n4 n4dfa`
In order to enable the above, we also need to extend Lexaard with boolean objects `true` and `false` with the conventional semantics. In particular, extend the `define` and `print` statements to booleans as suggested by the following input and output:

```
define b1 false
define b2 true
print b2
print b1
```

```
true
false
```

7. $\star$ (25 pts.) Extend Lexaard with a function `minDFA` that implements the DFA minimization algorithm described by Problem 7.42 in the textbook.