231 Programming Languages Spring 2017

Exam # 1 Practice Problems

1. DFA, NFA, Regular Expressions

   (a) Give a formal definition of a regular expression

   Answer:

   The set of regular expressions over alphabet Σ is defined as follows:

   Base Case:

   i. The symbol ϕ is a regular expression denoting the empty language (the empty set).

   ii. The symbol ϵ is a regular expression denoting the language containing the empty string.

   iii. The symbol a where $a \in \Sigma$ is a regular expression denoting the language $\{a\}$.

   General case:

   If R is a regular expression, we use the notation $L(R)$ to denote the language associated with R.

   i. If R_1 and R_2 are a regular expressions, then $(R_1 \mid R_2)$ is a regular expression, denoting the language $L(R_1) \cup L(R_2)$.

   ii. If R_1 and R_2 are a regular expressions, then $(R_1 \cdot R_2)$ is a regular expression, denoting the language $L(R_1) \cdot L(R_2)$.

   iii. If R is a regular expression, then (R^*) is a regular expression denoting the language $(L(R))^*$.

   (b) Give a formal definition of an NFA

   Answer:

   A Non-deterministic Finite Automaton is a 5-tuple:

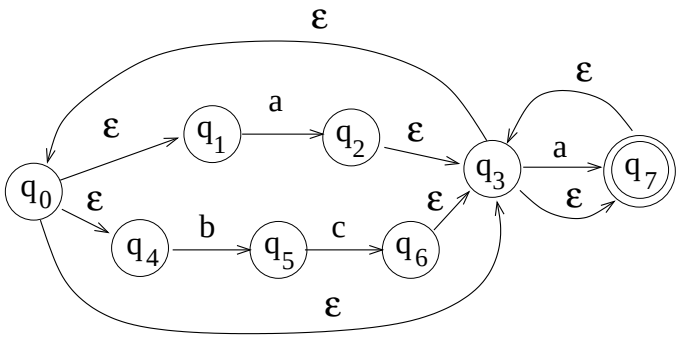
   $$N = (Q, \Sigma, \delta, q_0, F)$$

   where

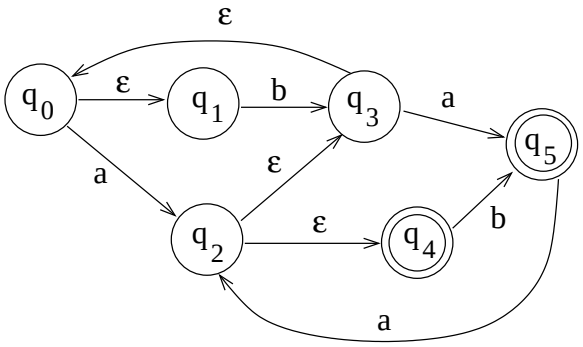
   Q is a finite set of states
   Σ is a finite alphabet (a set of symbols)
   δ is a transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$
   q_0 is the start state
   F is a set of final states (also called accepting states).

   (c) Draw an NFA for the regular expression $(a|bc)^*a^*$

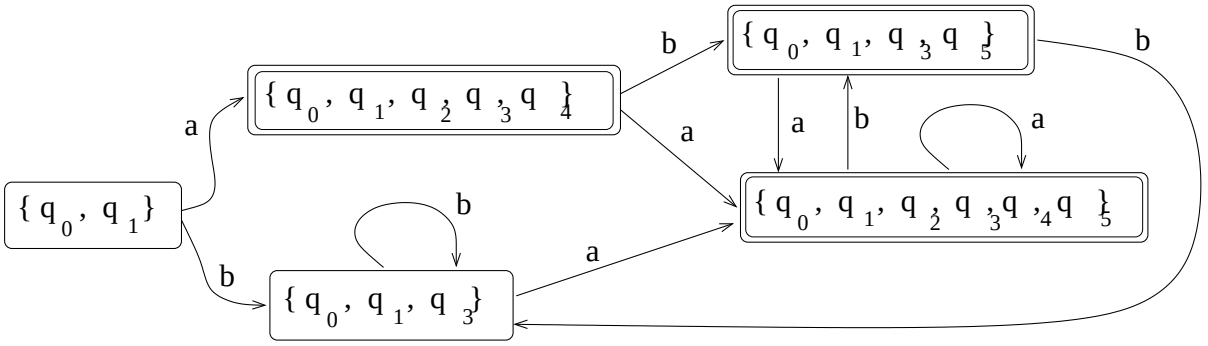
   Answer:



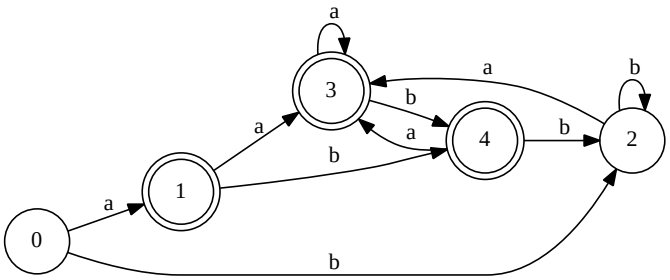
(d) Convert the following NFA to an equivalent DFA



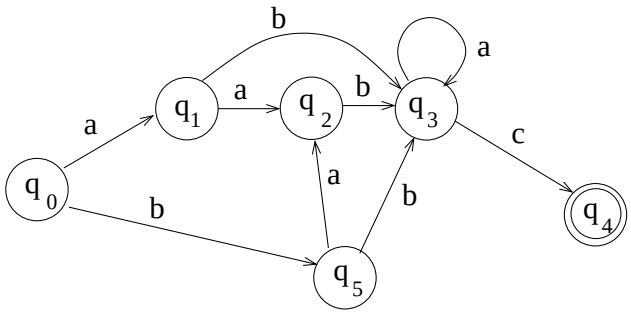
Answer:



Here is the same equivalent DFA drawn by graphvis

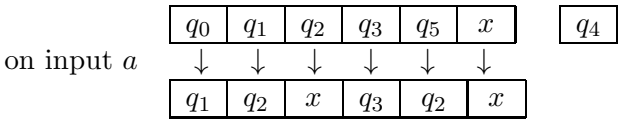


(e) Minimizing the number of states in the following DFA

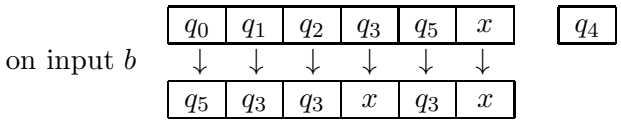


Answer: We begin by partitioning the set of states into two equivalence classes: accepting states and non-accepting states. We then consider transitions from each state on each symbol. Whenever a symbol leads states within an equivalence class to two or more equivalence classes, the equivalence class under consideration is split.

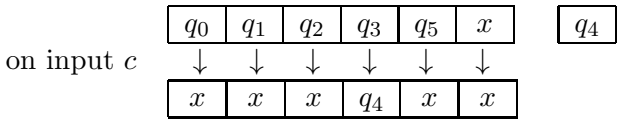
We begin with:



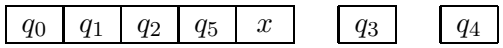
All of the destination states are in the same equivalence class, so no splitting occurs. We try again on input b.



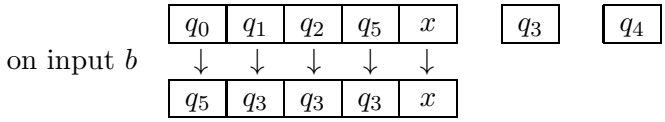
Again, no splitting occurs. We try again on input c.



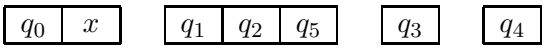
On this step, x and q_4 are in different equivalence classes. Therefore we split the current equivalence class $\{q_0, q_1, q_2, q_3, q_5\}$ splits into $\{q_0, q_1, q_2, q_5\}$ and $\{q_3\}$.



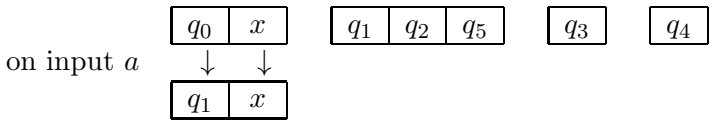
We now consider the set $\{q_0, q_1, q_2, q_5\}$. No splitting occurs on input a (diagram not shown), so we consider input b. On input b we have:



Some destination states are in different equivalence classes, so we split accordingly. We now have:

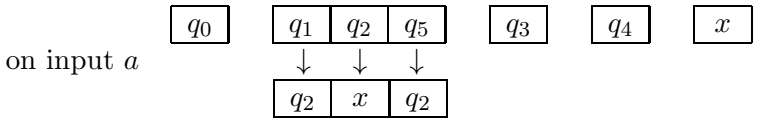


On input a, the set $\{q_0, x\}$ will split:

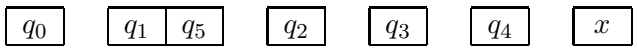


States q_1 and x are in different equivalence classes, so the set $\{q_0, x\}$ splits.

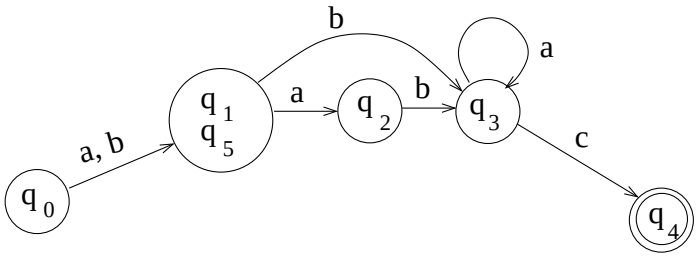
We now consider the set $\{q_1, q_2, q_5\}$ and input a:



The destination states q_2 and x are in different equivalence classes. Therefore, the set $\{q_1, q_2, q_5\}$ splits into $\{q_1, q_5\}$ and $\{q_2\}$. We now have:



Finally, we have now reached the set of equivalence classes where no more splitting is possible. I.e., the set $\{q_1, q_5\}$ does not split on any input. The minimized DFA is shown below:



2. Lexical Analyzers

   (a) What is the difference between a lexeme and a token ?

   Answer: A **token** is a basic linguistic unit of a programming language. A **lexeme** is a sequence of characters which are recognized as a specific token. For example, in **C++** the following two statements are both allowed:

```
if ( ! isdigit( ch ) ) {  // ....

if ( not isdigit( ch ) ) {  // ....
```

   The token of interest here is "**logical not**". There are two lexemes "**!**" and "**not**" that both correspond to the token "**logical not**".

   (b) A lexical analyzer may be required to handle a set of lexemes in which a lexeme for token T_1 may be a prefix of a lexeme for token T_2, where $T_1 \neq T_2$. Describe a technique to ensure that a lexical analyzer always recognizes the longest lexeme.

Answer: As each input character is read, the DFA is simulated by updating the current state according to the transition diagram. For each transition, we push the state and the character which led to that state on a stack. Continue running he DFA until the dead state is reached. Then, pop the stack until the most recent accepting state is found. As each stack frame is popped, put the associated character back into the input buffer. Those characters will need to be read again for the next token. If an accepting state is reached while popping the stack, the state number indicates the recognized token. If we reach the empty stack and no accepting states have been found, then we conclude there has been a lexical error in the input.

3. Grammars

(a) Context free grammars

i. Give a formal definition of a context free grammar.

Answer: A Context Free Grammar is a 4-tuple:

$$G = (V, T, R, S)$$

where

V is a finite set of variable symbols
T is a finite set of terminal symbols $V \cap T = \phi$
R is the set of rules. Each rule is of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup T)^*$
S is a start symbol. $S \in V$.

ii. Write a grammar that will recognize the regular language described in problem 1c

Answer:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \mid bcA \mid \epsilon \\ B &\rightarrow aB \mid \epsilon \end{aligned}$$

The grammar above is not suitable for an LL(1) parser. There is a parse table conflict:

```
Parse table conflict:
    Variable 'A' on input 'a' contains:
            A ->  a A
            A ->  ϵ
```

iii. Rewrite your grammar from part 3(a)ii so that it is suitable for an LL(1) predictive parser.

Answer: This problem is much easier to solve using the realization that $L((a|bc)^*a^*) = L((a|bc)^*)$. An equivalent LL(1) grammar is :

$$S \rightarrow aS \mid bcS \mid \epsilon$$

iv. What does it mean if we say a grammar is ambiguous ?

Answer: A grammar G is **ambiguous** if and only if there exists a string $w \in L(G)$ such that w has more than one leftmost derivation.
An alternate definition may be written as follows: "A grammar G is **ambiguous** if and only if there exists a string $w \in L(G)$ such that w has more than one parse tree."

(b) Using the grammar in problem 4, give a leftmost derivation of

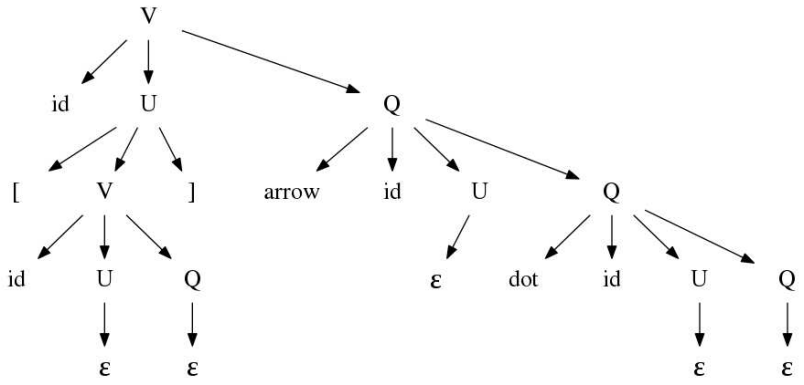
```
id [ id ] arrow id dot id
```

Answer:

$$
\begin{aligned}
V &\rightarrow \text{id } U \ Q \\
&\rightarrow \text{id } [\ V\ ]\ Q \\
&\rightarrow \text{id } [\ \text{id } U \ Q\ ]\ Q \\
&\rightarrow \text{id } [\ \text{id } Q\ ]\ Q \\
&\rightarrow \text{id } [\ \text{id }\ ]\ Q \\
&\rightarrow \text{id } [\ \text{id }\ ]\ \text{arrow id } U \ Q \\
&\rightarrow \text{id } [\ \text{id }\ ]\ \text{arrow id } Q \\
&\rightarrow \text{id } [\ \text{id }\ ]\ \text{arrow id dot id } U \ Q \\
&\rightarrow \text{id } [\ \text{id }\ ]\ \text{arrow id dot id } Q \\
&\rightarrow \text{id } [\ \text{id }\ ]\ \text{arrow id dot id}
\end{aligned}
$$

(c) Draw a parse tree for the derivation in problem 3b.

Answer:



(d) Left factor the following grammar

$$
\begin{aligned}
S &\rightarrow a\ b\ c\ B\ A \\
A &\rightarrow a\ b\ c\ B\ a \mid a\ b\ c\ A\ b \mid a\ b\ S\ d \\
B &\rightarrow a\ B \mid a\ A \mid \epsilon
\end{aligned} \tag{1}
$$

Answer:

Step 1: Replace production $A \rightarrow a\ b\ c\ B\ a \mid a\ b\ c\ A\ b \mid a\ b\ S\ d$ by:

$$
\begin{aligned}
A &\rightarrow a\ b\ c\ A' \mid a\ b\ S\ d \\
A' &\rightarrow B\ a \mid A\ b
\end{aligned} \tag{2}
$$

Step 2: Replace production $A \rightarrow a\ b\ c\ A' \mid a\ b\ S\ d$ by:

$$
\begin{aligned}
A &\rightarrow a\ b\ A'' \\
A'' &\rightarrow c\ A' \mid S\ d
\end{aligned} \tag{3}
$$

Step 3: Replace production $B \rightarrow a\ B \mid a\ A \mid \epsilon$
by:

$$\begin{array}{lcl} B & \rightarrow & a\ B' \mid \epsilon \\ B' & \rightarrow & B \mid A \end{array} \tag{4}$$

Combining the substitutions made above, the final grammar is:

$$\begin{array}{lcl} S & \rightarrow & a\,b\,c\,B\,A \\ A & \rightarrow & a\,b\,A'' \\ A'' & \rightarrow & c\,A' \mid S\,d \\ A' & \rightarrow & B\ a \mid A\ b \\ B & \rightarrow & a\ B' \mid \epsilon \\ B' & \rightarrow & B \mid A \end{array} \tag{5}$$

(e) Eliminate left recursion from the following grammar

$$\begin{array}{lcl} S & \rightarrow & a\,B\,A \\ A & \rightarrow & A\,b \mid A\,a\,c\,A \mid B\,d \\ B & \rightarrow & A\ d \mid x \end{array} \tag{6}$$

Answer: We begin by ordering the variables and choose the ordering S, A, B. The variable S is not involved in any left recursion so we do not have to re-write any of the S productions. We move on to the variable A. There are no productions of the form $A \rightarrow S\gamma$, so there is no "loop back" to an earlier variable in the list. We can proceed by eliminating immediate left recursion among the A-productions. We replace the productions:

$$A \rightarrow A\,b \mid A\,a\,c\,A \mid B\,d$$

by:

$$\begin{array}{lcl} A & \rightarrow & B\,d\,A' \\ A' & \rightarrow & b\,A' \mid a\,c\,A\,A' \mid \epsilon \end{array}$$

Before we move on to the next variable B, we review our current situation; our grammar is now:

$$\begin{array}{lcl} S & \rightarrow & a\,B\,A \\ A & \rightarrow & B\,d\,A' \\ A' & \rightarrow & b\,A' \mid a\,c\,A\,A' \mid \epsilon \\ B & \rightarrow & A\ d \mid x \end{array}$$

We then move on to the variable B. The production $B \rightarrow Ad$ "loops back" to a variable which occurs earlier in the ordering. This causes indirect left recursion. We replace the production:

$$B \rightarrow A\ d$$

by:

$$B \rightarrow B\,d\,A'\ d$$

We replaced the occurrence of A on the right hand side of "$B \rightarrow A\ d$" by all possible right hand sides for A in the current (modified) grammar. There is only one right hand side, i.e., $A \rightarrow B\ d\ A'$.

Our final task is to eliminate immediate left recursion from the B-productions. We replace the productions:

$$B \rightarrow B\ d\ A'\ d \mid x$$

by:

$$B \rightarrow x\ B'$$
$$B' \rightarrow d\ A'\ d\ B' \mid \epsilon$$

Our final grammar (free of left recursion) is shown below:

$$S \rightarrow a\ B\ A$$
$$A \rightarrow B\ d\ A'$$
$$A' \rightarrow b\ A' \mid a\ c\ A\ A' \mid \epsilon$$
$$B \rightarrow x\ B'$$
$$B' \rightarrow d\ A'\ d\ B' \mid \epsilon$$

4. The following is a grammar for C/C++ style variables, including notation for array subscripts, aggregate members, and pointer references. E.g., `a[ b.i ]->c.d.e[j]->f`

$$V \rightarrow \text{id}\ U\ Q$$
$$U \rightarrow [\ V\ ] \mid \epsilon \tag{7}$$
$$Q \rightarrow \text{dot id}\ U\ Q \mid \text{arrow id}\ U\ Q \mid \epsilon$$

Your tasks:

(a) Construct the FIRST and FOLLOW sets for this grammar

(b) Constructing a predictive parsing table

(c) Write (using pseudocode) a recursive descent parser for your predictive parsing table.

Answer:

```
V -> id U Q

U -> [ V ]    |
     epsilon

Q -> dot id U Q    |
     arrow id U Q    |
     epsilon

First sets:
     FIRST(V) = { id  }
     FIRST(U) = { epsilon [  }
     FIRST(Q) = { epsilon dot arrow  }
```

```
Follow sets:
     FOLLOW(V) = { eof ]  }
     FOLLOW(U) = { eof ] dot arrow  }
     FOLLOW(Q) = { eof ]  }

Parse Table:

Variable V :
   Input 'id':  V ->  id U Q

Variable U :
   Input 'eof':  U ->  epsilon
   Input '[':  U ->  [ V ]
   Input ']':  U ->  epsilon
   Input 'dot':  U ->  epsilon
   Input 'arrow':  U ->  epsilon

Variable Q :
   Input 'eof':  Q ->  epsilon
   Input ']':  Q ->  epsilon
   Input 'dot':  Q ->  dot id U Q
   Input 'arrow':  Q ->  arrow id U Q
```