Exercises for Chapter 3

Exercises for Section 3.1

3.1.1

Divide the following C++ program:

```
float limitedSquare(x){float x;
  /* returns x-squared, nut never more than 100 */
  return (x <= -10.0 || x >= 10.0) ? 100 : x*x;
}
```

into appropriate lexemes, using the discussion of Section 3.1.2 as a guide. Which lexemes should get associated lexical values? What should those values be?

Answer

```
<float> <id, limitedSquaare> <(> <id, x> <)> <{>
  <float> <id, x>
  <return> <(> <id, x> <op,"<="> <num, -10.0> <op, "||"> <id, x> <op, ">="> <num,
10.0> <)> <op, "?"> <num, 100> <op, ":"> <id, x> <op, "*"> <id, x>
<}>
```

3.1.2

Tagged languages like HTML or XML are different from conventional programming languages in that the punctuation (tags) are either very numerous (as in HTML) or a user-definable set (as in XML). Further, tags can often have parameters. Suggest how to divide the following HTML document:

```
Here is a photo of <b>my house</b>;
<p><img src="house.gif"/><br/>
see <a href="morePix.html">More Picture</a> if you
liked that one.</p>
```

into appropriate lexemes. Which lexemes should get associated lexical values, and what should those values be?

Answer

```
<text, "Here is a photo of"> <nodestart, b> <text, "my house"> <nodeend, b>
<nodestart, p> <selfendnode, img> <selfendnode, br>
<text, "see"> <nodestart, a> <text, "More Picture"> <nodeend, a>
<text, "if you liked that one."> <nodeend, p>
```

Exercises for Section 3.3

3.3.1

Consult the language reference manuals to determine

1. the sets of characters that form the input alphabet (excluding those that may only appear in character strings or comments)
2. the lexical form of numerical constants, and
3. the lexical form of identifiers,

for each of the following languages:

1. C
2. C++
3. C#
4. Fortran
5. Java
6. Lisp
7. SQL

3.3.2

Describe the languages denoted by the following regular expressions:

1. a(a|b)*a
2. ((ε|a)b*)*
3. (a|b)*a(a|b)(a|b)
4. a*ba*ba*ba*
5. !! (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*

Answer

1. String of a's and b's that start and end with a.
2. String of a's and b's.
3. String of a's and b's that the character third from the last is a.
4. String of a's and b's that only contains three b.
5. String of a's and b's that has a even number of a and b.

3.3.3

In a string of length n, how many of the following are there?

1. Prefixes.

2. Suffixes.
3. Proper prefixes.
4. ! Substrings.
5. ! Subsequences.

Answer

1. n + 1
2. n + 1
3. n - 1
4. C(n+1,2) + 1 (need to count epsilon in)
5. Σ(i=0,n) C(n, i)

3.3.4

Most languages are case sensitive, so keywords can be written only one way, and the regular expressions describing their lexeme is very simple. However, some languages, like SQL, are case insensitive, so a keyword can be written either in lowercase or in uppercase, or in any mixture of cases. Thus, the SQL keyword SELECT can also be written select, Select, or sElEcT, for instance. Show how to write a regular expression for a keyword in a case insensitive language. Illustrate the idea by writing the expression for "select" in SQL.

Answer

```
select -> [Ss][Ee][Ll][Ee][Cc][Tt]
```

3.3.5

! Write regular definitions for the following languages:

1. All strings of lowercase letters that contain the five vowels in order.
2. All strings of lowercase letters in which the letters are in ascending lexicographic order.
3. Comments, consisting of a string surrounded by / *and* /, without an intervening */, unless it is inside double-quotes (")
4. !! All strings of digits with no repeated digits. Hint: Try this problem first with a few digits, such as {O, 1, 2}.
5. !! All strings of digits with at most one repeated digit.
6. !! All strings of a's and b's with an even number of a's and an odd number of b's.
7. The set of Chess moves,in the informal notation,such as p-k4 or kbp*qn.
8. !! All strings of a's and b's that do not contain the substring abb.
9. All strings of a's and b's that do not contain the subsequence abb.

Answer

1、

```
want -> other* a (other|a)* e (other|e)* i (other|i)* o (other|o)* u (other|u)*
other -> [bcdfghjklmnpqrstvwxyz]
```

2、

```
a* b* ... z*
```

3、

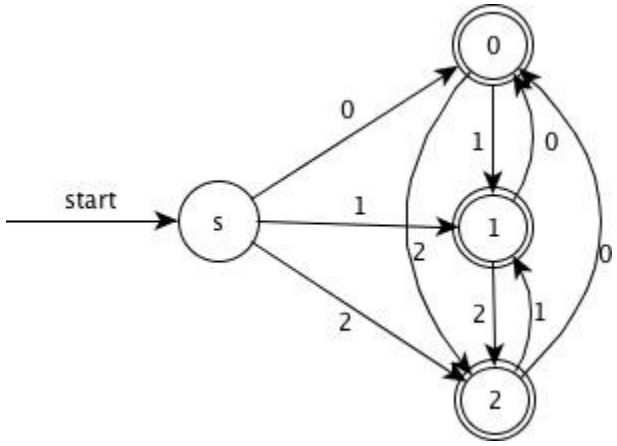
```
\/\*([^*"]*|".*"|\*+[^/])*\*\/
```

4、

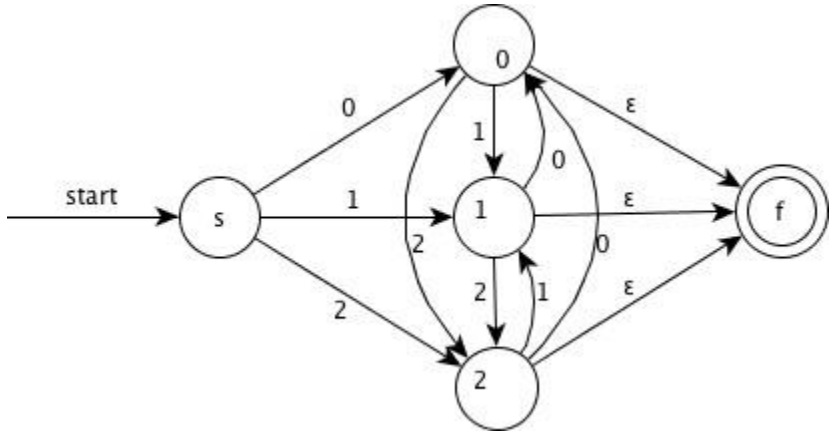
```
want -> 0|A?0?1(A0?1|01)*A?0?|A0?
A -> 0?2(02)*
```

Steps:

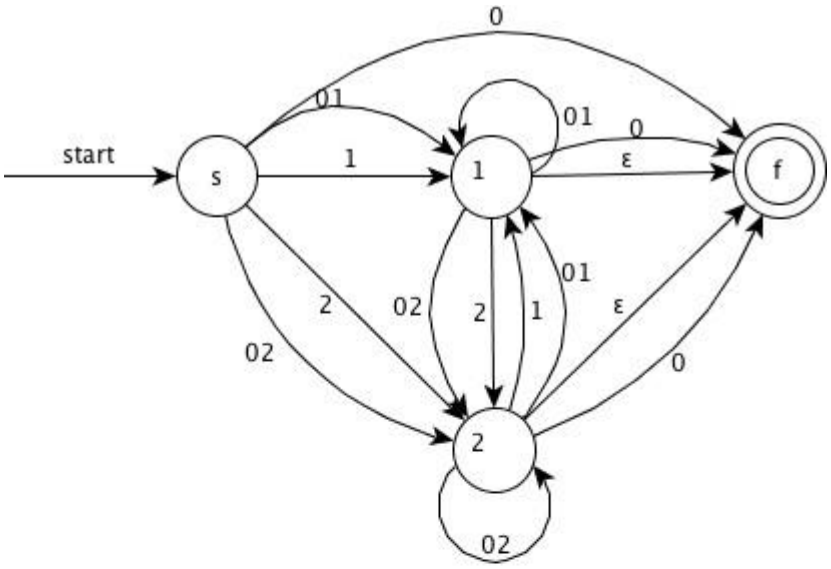
step1. Transition diagram

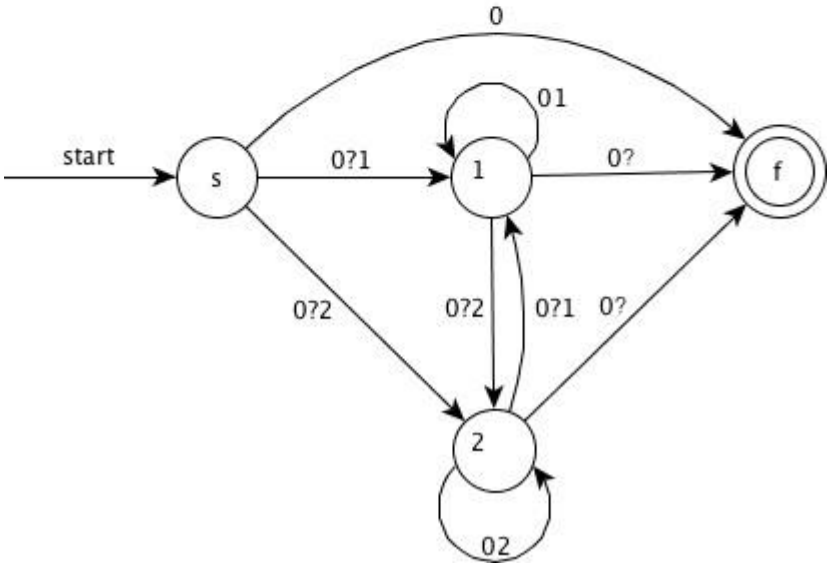


step2. GNFA

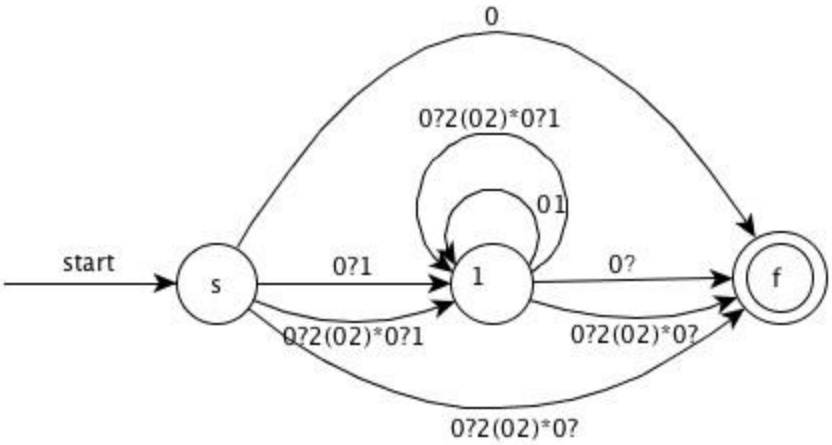


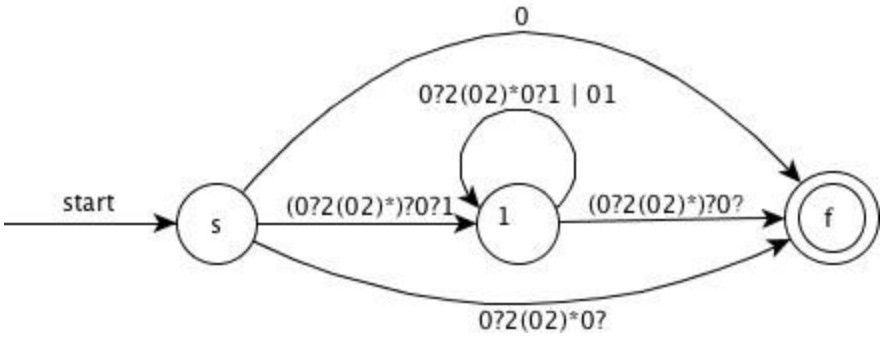
step3. Remove node 0 and simplify



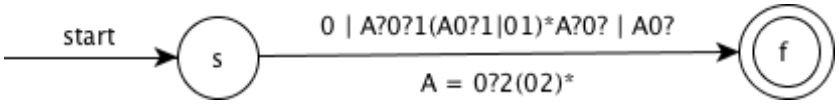


step4. Remove node 2 and simplify





step5. Remove node 1 and simplify

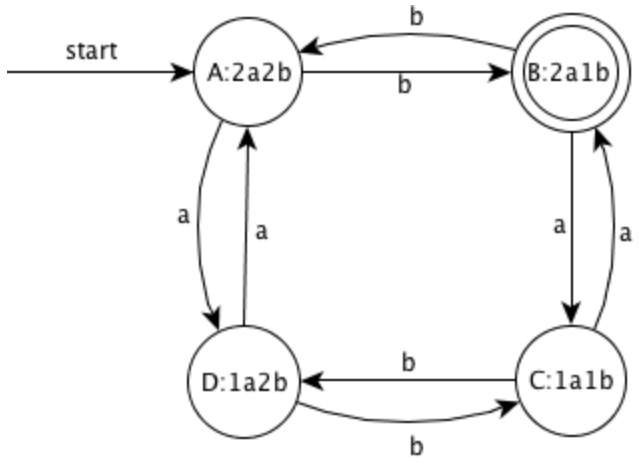


5、

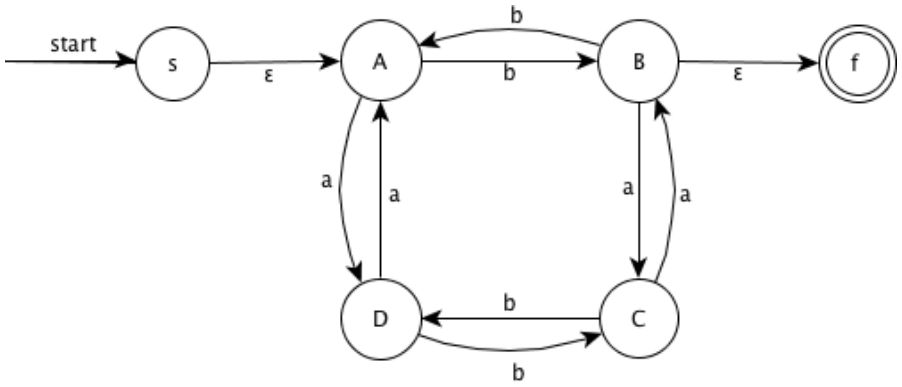
```
want -> (FE*G|(aa)*b)(E|FE*G)
E -> b(aa)*b
F -> a(aa)*b
G -> b(aa)*ab|a
F -> ba(aa)*b
```

Steps:

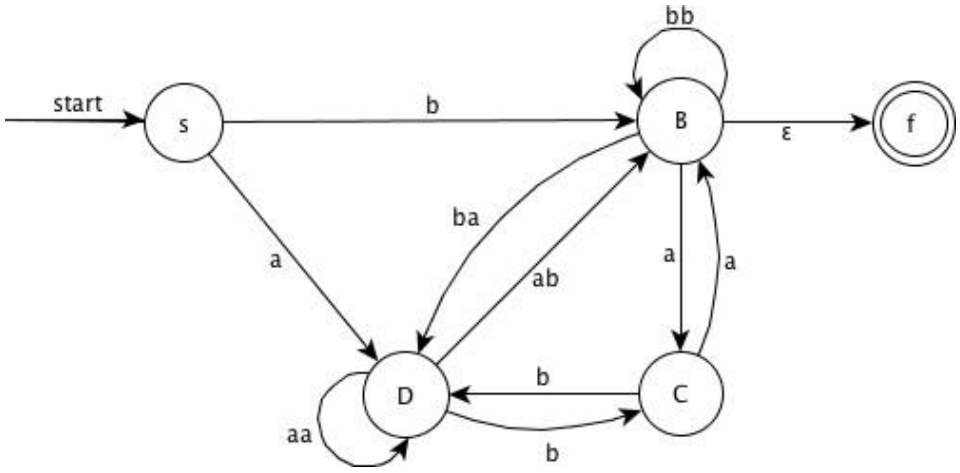
step1. Transition diagram



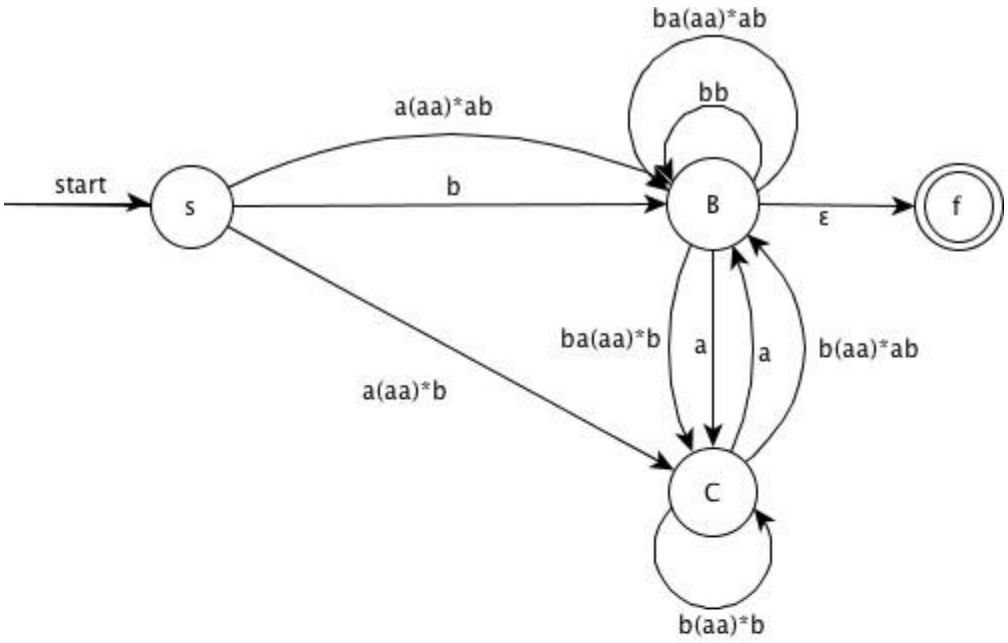
step2. GNFA

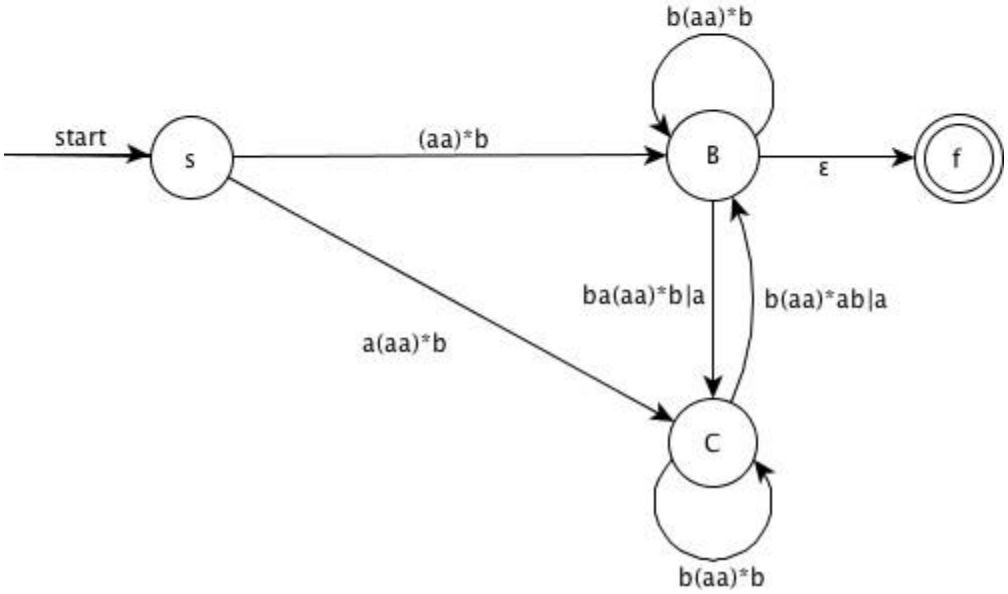


step3. Remove node A and simplify

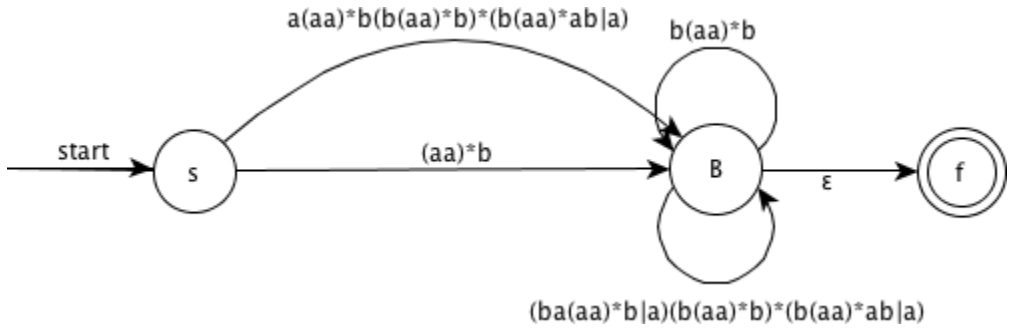


step4. Remove node D and simplify





step5. Remove node C and simplify



8、

```
b*(a+b?)*
```

9、

```
b* | b*a+ | b*a+ba*
```

3.3.6

Write character classes for the following sets of characters:

1. The first ten letters (up to "j") in either upper or lower case.
2. The lowercase consonants.
3. The "digits" in a hexadecimal number (choose either upper or lower case for the "digits" above 9).
4. The characters that can appear at the end of alegitimate English sentence (e.g. , exclamation point) .

Answer

1. [A-Ja-j]
2. [bcdfghjklmnpqrstvwxzy]
3. [0-9a-f]
4. [.?!]

3.3.7

Note that these regular expressions give all of the following symbols (operator characters) a special meaning:

```
\ " . ^ $ [ ] * + ? { } | /
```

Their special meaning must be turned off if they are needed to represent themselves in a character string. We can do so by quoting the character within a string of length one

or more; e.g., the regular expression "**" matches the string ** . We can also get the literal meaning of an operator character by preceding it by a backslash. Thus, the regular expression ** also matches the string **. Write a regular expression that matches the string "\.

Answer

```
\"\\
```

3.3.9 !

The regular expression r{m, n} matches from m to n occurrences of the pattern r. For example, a [1 , 5] matches a string of one to five a's. Show that for every regular expression containing repetition operators of this form, there is an equivalent regular expression without repetition operators.

Answer

r{m,n} is equals to r.(m).r | r.(m + 1).r | ... | r.(n).r

3.3.10 !

The operator ^ matches the left end of a line, and $ matches the right end of a line. The operator ^ is also used to introduce complemented character classes, but the context always makes it clear which meaning is intended. For example, ^aeiou*$ matches any complete line that does not contain a lowercase vowel.

1. How do you tell which meaning of ^ is intended?
2. Can you always replace a regular expression using the ^ and $ operators by an equivalent expression that does not use either of these operators?

Answer

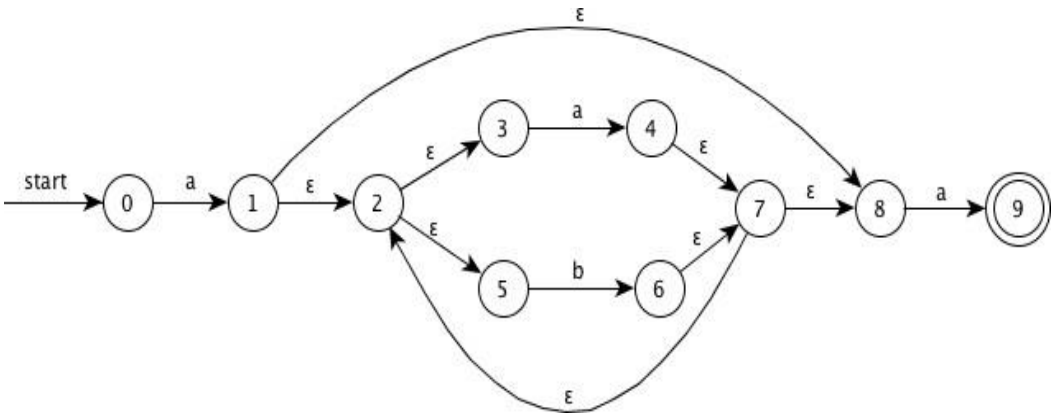
1. if ^ is in a pair of brakets, and it is the first letter, it means complemented classes, or it means the left end of a line.

3.4

3.4.1

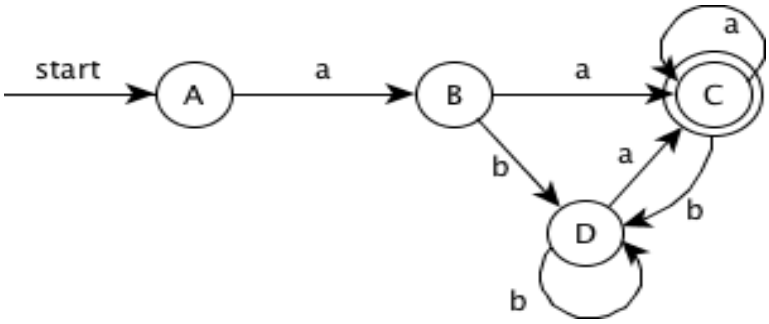
Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.2.a(a|b)*a

NFA:

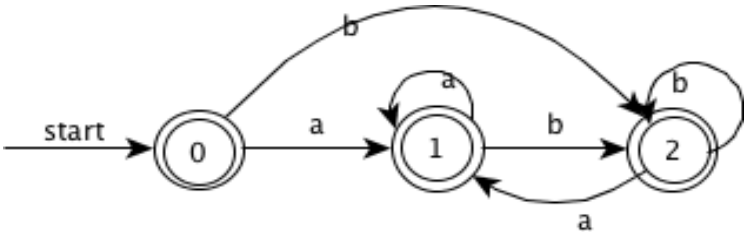


DFA:

NFA	DFA	a	b
{0}	A	B	
{1,2,3,5,8}	B	C	D
{2,3,4,5,7,8,**9**}	**C**	C	D
{2,3,5,6,7,8}	D	C	D

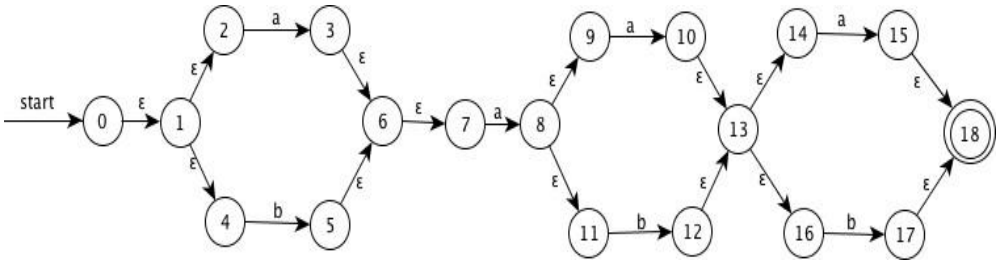


1. ((ε|a)b*)*



2. (a|b)*a(a|b)(a|b)

NFA:

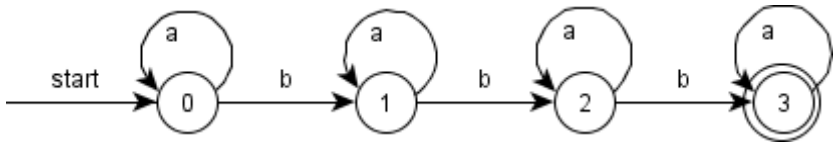


DFA:

```
<table>
    <thead>
        <tr>
            <th>NFA</th>
            <th>DFA</th>
            <th>a</th>
            <th>b</th>
        </tr>
    </thead>
    <tbody>
        <tr>
```

```
            <td>{0,1,2,4,7}</td>
            <td>A</td>
            <td>B</td>
            <td>C</td>
        </tr>
        <tr>
            <td>{1,2,3,4,6,7,8,9,11}</td>
            <td>B</td>
            <td>D</td>
            <td>E</td>
        </tr>
        <tr>
            <td>{1,2,4,5,6,7}</td>
            <td>C</td>
            <td>B</td>
            <td>C</td>
        </tr>
        <tr>
            <td>{1,2,3,4,6,7,8,9,10,11,13,14,16}</td>
            <td>D</td>
            <td><b>F</b></td>
            <td><b>G</b></td>
        </tr>
        <tr>
            <td>{1,2,4,5,6,7,12,13,14,16}</td>
            <td>E</td>
            <td><b>H</b></td>
            <td><b>I</b></td>
        </tr>
        <tr>
            <td>{1,2,3,4,6,7,8,9,10,11,13,14,15,16,<b>18</b>}</td>
            <td><b>F</b></td>
            <td><b>F</b></td>
            <td><b>G</b></td>
        </tr>
        <tr>
            <td>{1,2,4,5,6,7,12,13,14,16,17,<b>18</b>}</td>
            <td><b>G</b></td>
            <td><b>H</b></td>
            <td><b>I</b></td>
        </tr>
        <tr>
            <td>{1,2,3,4,6,7,8,9,11,15,<b>18</b>}</td>
            <td><b>H</b></td>
            <td>D</td>
            <td>E</td>
        </tr>
        <tr>
            <td>{1,2,4,5,6,7,17,<b>18</b>}</td>
            <td><b>I</b></td>
            <td>B</td>
            <td>C</td>
        </tr>
    </tbody>
</table>
```

1. a*ba*ba*ba*



3.4.2

Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.5.

3.4.3

Construct the failure function for the strings。

1. abababaab
2. aaaaaa
3. abbaabb

1. [0, 0, 1, 2, 3, 4, 5, 1, 2]
2. [0, 1, 2, 3, 4, 5]
3. [0, 0, 0, 1, 1, 2, 3]

3.4.4 !

Prove, by induction on s, that the algorithm of Fig. **3.19** correctly computes the failure function

```
01  t = 0;
02  f(1) = 0;
03  for (s = 1; s < n; s ++){
04      while( t > 0 && b_s+1 != b_t+1) t = f(t);
05      if(b_s+1 == b_t+1){
06          t = t + 1;
07          f(s + 1) = t;
08      }else{
09          f(s + 1) = 0;
10      }
11  }
```

Exercises for Section 3.5

3.5.1

Describe how to make the following modifications to the Lex program of Fig. 3.23:

1. Add the keyword while.
2. Change the comparison operators to be the C operators of that kind.
3. Allow the underscore (_) as an additional letter.
4. ! Add a new pattern with token STRING. The pattern consists of a double quote (") , any string of characters and a final double-quote. However, if a double-quote appears in the string, it must be escaped by preceding it with a backslash () , and therefore a backslash in the string must be represented by two backslashes. The lexical value, which is the string without the surrounding double-quotes, and with backslashes used to es.,. cape a character removed. Strings are to be installed in a table of strings.

source

3.5.2

Write a Lex program that copies a file, replacing each non empty sequence of white space by a single blank

3.5.3

Write a Lex program that copies a C program, replacing each instance of the keyword f loat by double.。

3.5.4 !

Write a Lex program that converts a file to "Pig latin." Specifically, assume the file is a sequence of words (groups . of letters) separated by whitespace. Every time you encounter a word:

1. If the first letter is a consonant, move it to the end of the word and then add ay!
2. If the first letter is a vowel, just add ay to the end of the word.

All nonletters are copied int act to the output.

source

3.5.5 !

In SQL, keywords and identifiers are case-insensitive. Write a Lex program that recognizes the keywords SELECT, FROM, and WHERE (in any combination of capital and lower-case letters) , and token ID, which for the purposes of this exercise you may take to be any sequence of letters and digits, beginning with a letter. You need not install identifiers in a symbol table, but tell how the "install" function would differ from that described for case-sensitive identifiers as in Fig. 3.23.

source

Exercises for Section 3.6

3.6.1 !

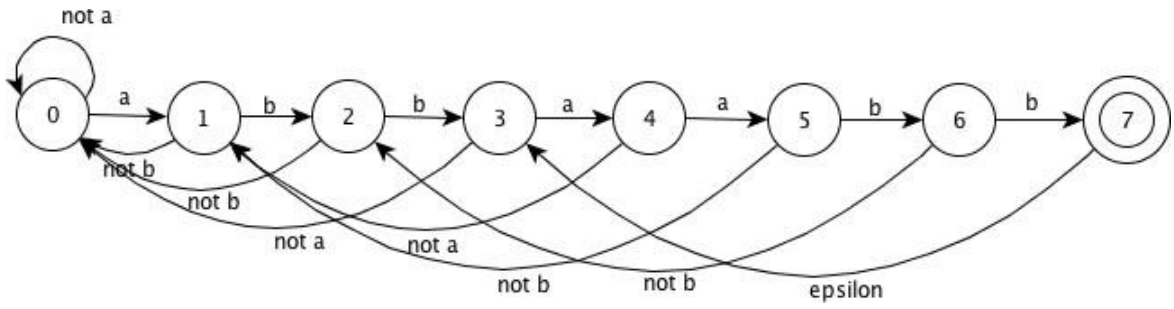
Figure 3.19 in the exercises of Section 3.4 computes the failure function for the KMP algorithm. Show how, given that failure function, we can construct, from a keyword b1b2...bn an n + 1-state DFA that recognizes .*b1b2...bn, where the dot stands for "any character." Moreover, this DFA can be constructed in O(n) time.

Answer

Take the string "abbaabb" in exercise 3.4.3-3 as example, the failure function is:

- n : 1, 2, 3, 4, 5, 6, 7
- f(n): 0, 0, 0, 1, 1, 2, 3

The DFA is：



Pseudocode of building the DFA：

```
for (i = 0; i< n; i ++) {
  move[s[i], c] = {
    if ( c == b1b2…bn[i] ) {
      goto s[i+1]
    } else {
      goto s[f(i)]
```

```
    }
  }
}
```

It is obviously that with the known f(n), this DFA can be constructed in O(n) time.

3.6.2

Design finite automata (deterministic or nondeterministic) for each of the languages of Exercise 3.3.5.

3.6.3

For the NFA of Fig. 3.29, indicate all the paths labeled aabb. Does the NFA accept aabb?

Answer

- (0) -a-> (1) -a-> (2) -b-> (2) -b-> ((3))
- (0) -a-> (0) -a-> (0) -b-> (0) -b-> (0)
- (0) -a-> (0) -a-> (1) -b-> (1) -b-> (1)
- (0) -a-> (1) -a-> (1) -b-> (1) -b-> (1)
- (0) -a-> (1) -a-> (2) -b-> (2) -b-> (2)
- (0) -a-> (1) -a-> (2) -b-> (2) -ε-> (0) -b-> (0)
- (0) -a-> (1) -a-> (2) -ε-> (0) -b-> (0) -b-> (0)

This NFA accepts "aabb"

3.6.4

Repeat Exercise 3.6.3 for the NFA of Fig. 3.30.

3.6.5

Give the transition tables for the NFA of:

1. Exercise 3.6.3.
2. Exercise 3.6.4.
3. Figure 3.26.

Answer

Table 1

state	a	b	ε
0	{0,1}	{0}	∅
1	{1,2}	{1}	∅
2	{2}	{2,3}	{0}
3	∅	∅	∅

Table 2

state	a	b	ε
0	{1}	∅	{3}
1	∅	{2}	{0}
2	∅	{3}	{1}
3	{0}	∅	{2}

Table 3

state	a	b	ε
0	∅	∅	{1,2}
1	{2}	∅	∅
2	{2}	∅	∅
3	∅	{4}	∅
4	∅	{4}	∅

Exercises for Section 3.7

3.7.1

Convert to DFA's the NFA's of:

1. Fig. 3.26.
2. Fig. 3.29.
3. Fig. 3.30.

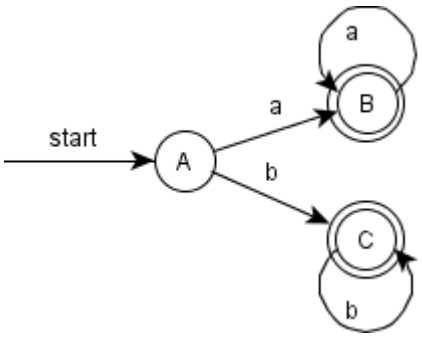
Answer

1、

Transition table

NFA State	DFA State	a	b
{0,1,3}	A	B	C
{2}	B	B	∅
{4}	C	∅	C

DFA

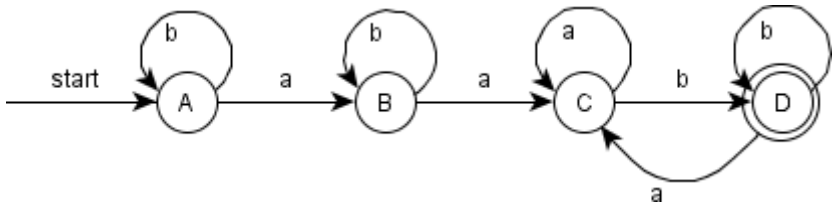


2、

Transition table

NFA State	DFA State	a	b
{0}	A	B	A
{0,1}	B	C	B
{0,1,2}	C	C	D
{0,2,3}	D	C	D

DFA

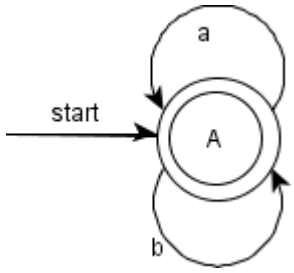


3、

Transition table

NFA State	DFA State	a	b
{0,1,2,3}	A	A	A

DFA



3.7.2

use Algorithm 3.22 to simulate the NFA's:

1. Fig. 3.29.
2. Fig. 3.30.

on input aabb.

Answer

1. -start->{0}-a->{0,1}-a->{0,1,2}-b->{0,2,3}-b->{0,2,3}
2. -start->{0,1,2,3}-a->{0,1,2,3}-a->{0,1,2,3}-b->{0,1,2,3}-b->{0,1,2,3}

3.7.3

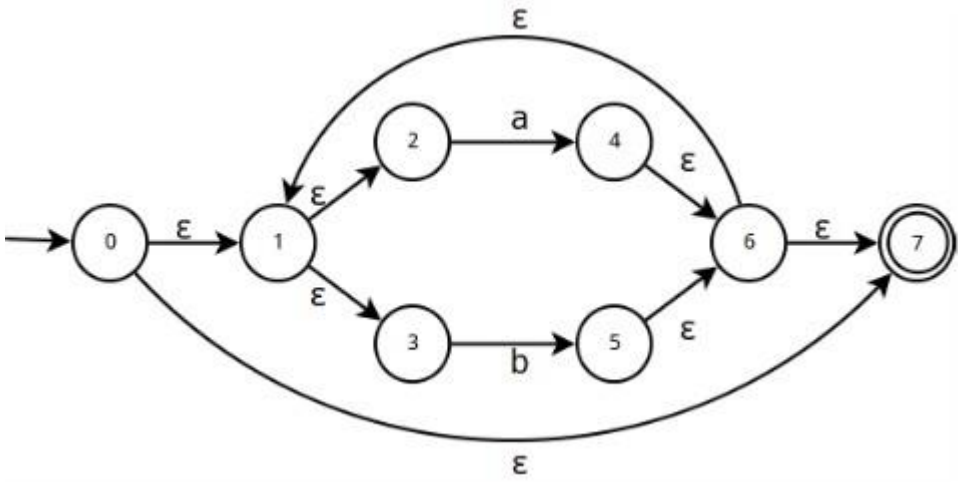
Convert the following regular expressions to deterministic finite automata, using algorithms 3.23 and 3.20:

1. (a|b)*
2. (a*|b*)*
3. ((ε|a)|b*)*
4. (a|b)*abb(a|b)*

Answer

1、

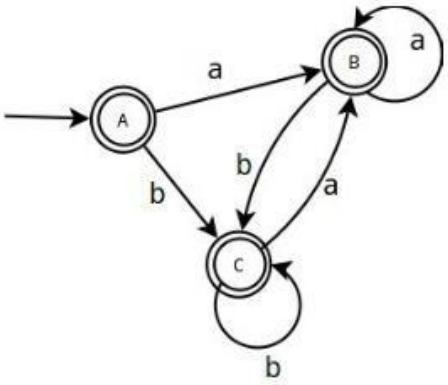
NFA



Transition table

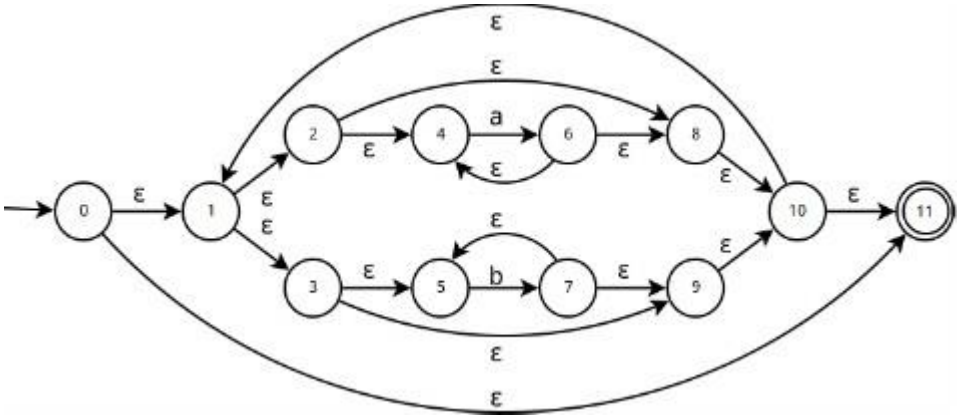
NFA State	DFA State	a	b
{0,1,2,3,7}	A	B	C
{1,2,3,4,6,7}	B	B	C
{1,2,3,5,6,7}	C	B	C

DFA



2、

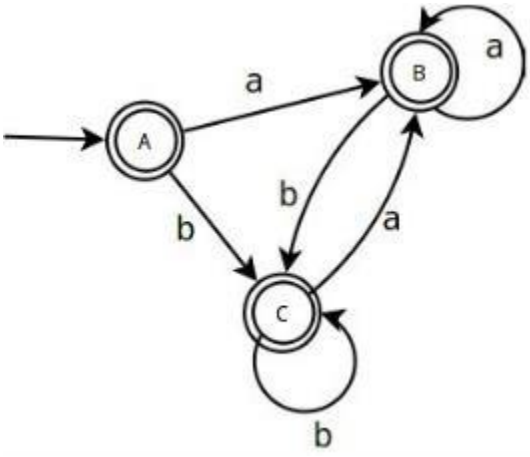
NFA



Transition table

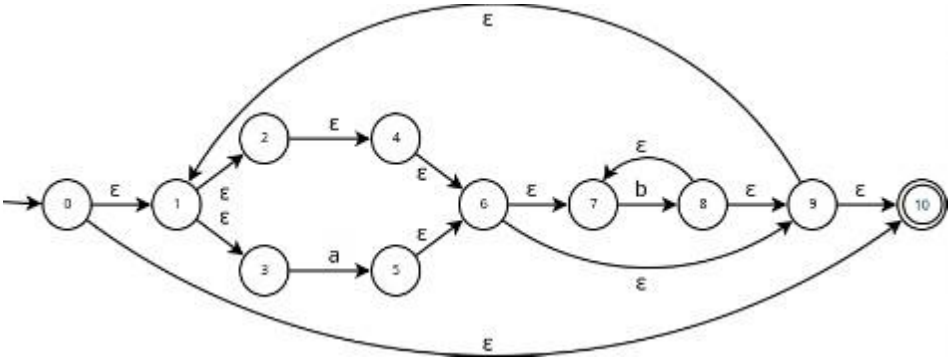
NFA State	DFA State	a	b
{0,1,2,3,4,5,8,9,10,11}	A	B	C
{1,2,3,4,5,6,8,9,10,11}	B	B	C
{1,2,3,4,5,7,8,9,10,11}	C	B	C

DFA



3、

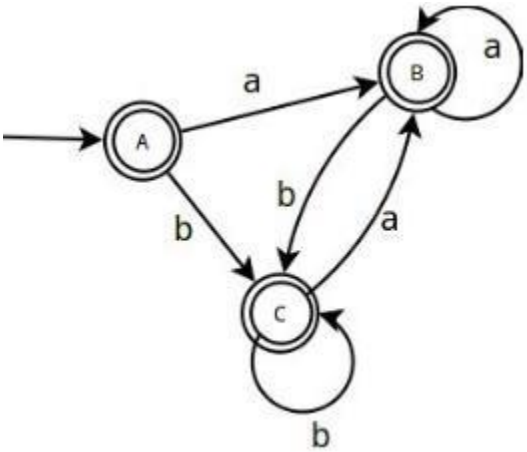
NFA



Transition table

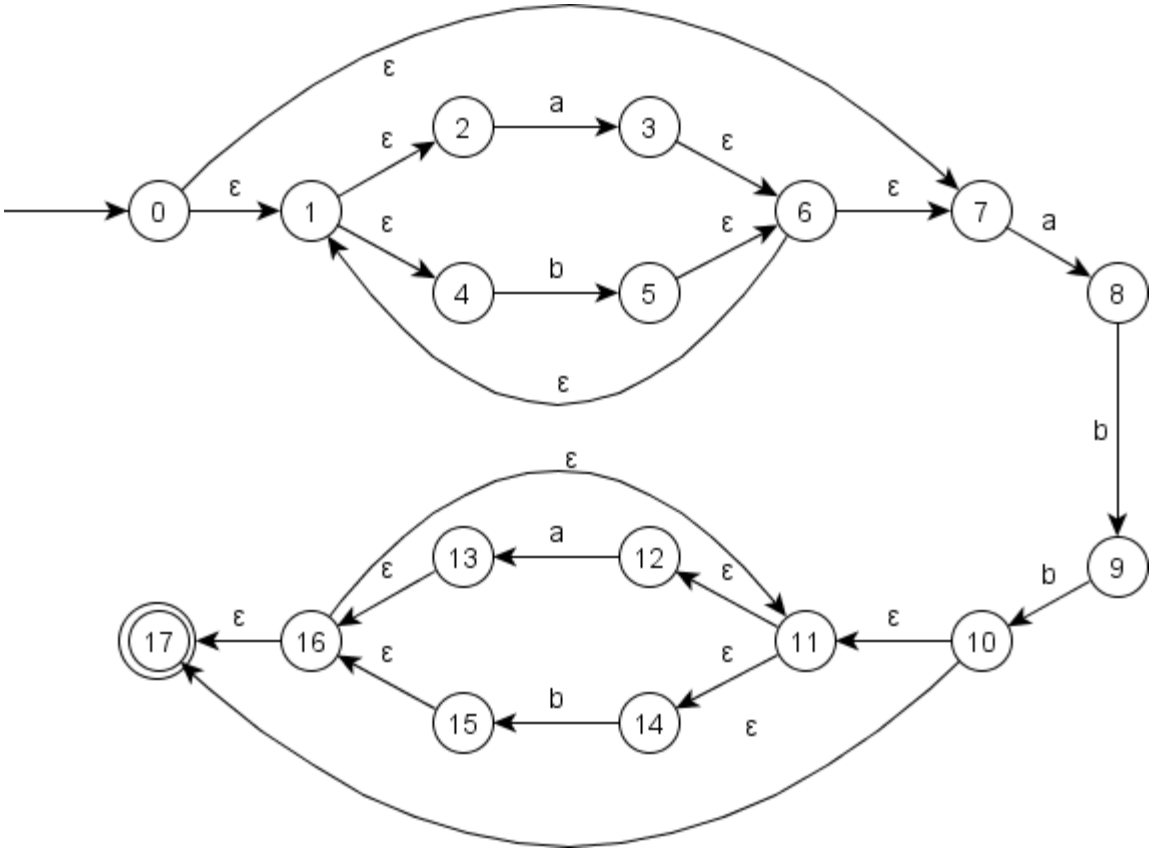
NFA State	DFA State	a	b
{0,1,2,3,4,6,7,9,10}	A	B	C
{1,2,3,4,5,6,7,9,10}	B	B	C
{1,2,3,4,6,7,8,9,10}	C	B	C

DFA



4、

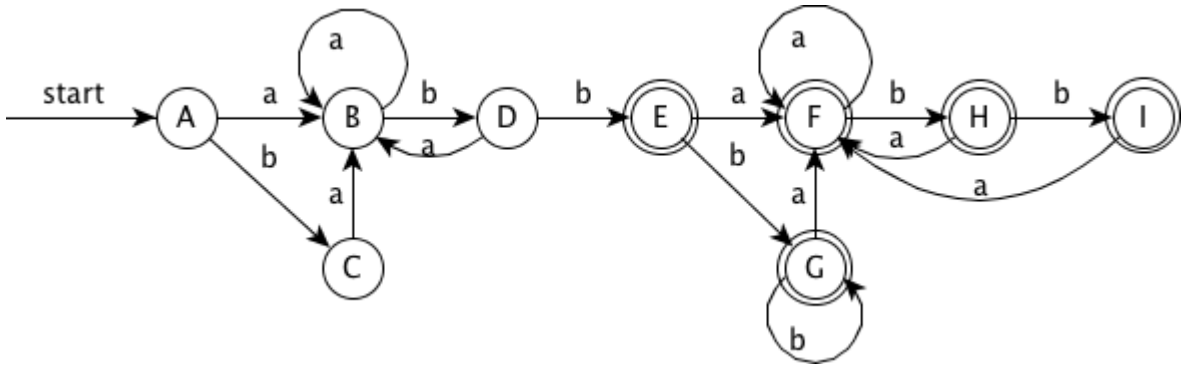
NFA



Transition table

NFA State	DFA State	a	b
{0,1,2,4,7}	A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,4,5,6,7,10,11,12,14,17}	E	F	G
{1,2,3,4,6,7,8,11,12,13,14,16,17}	F	F	H
{1,2,4,5,6,7,11,12,13,15,16,17}	G	F	G
{1,2,4,5,6,7,9,11,12,14,15,16,17}	H	F	I
{1,2,4,5,6,7,10,11,12,14,15,16,17}	I	F	G

DFA



Exercises for Section 3.8

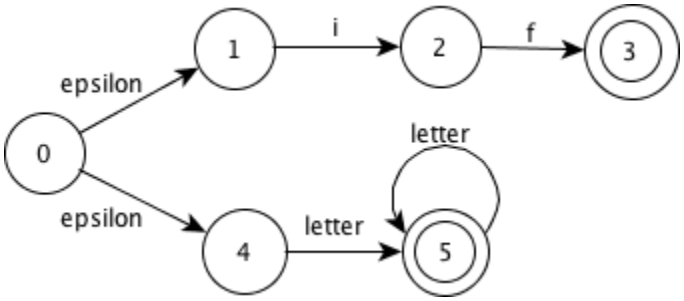
3.8.1

Suppose we have two tokens: (1) the keyword if, and (2) identifiers, which are strings of letters other than if. Show:

1. The NFA for these tokens, and
2. The DFA for these tokens.

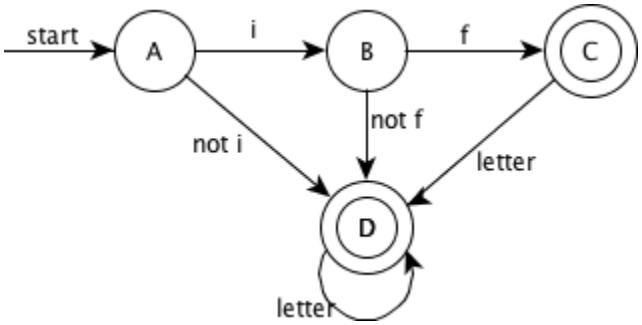
Answer

1. NFA



   NOTE: this NFA has potential conflict, we can decide the matched lexeme by 1. take the longest 2. take the first listed.

2. DFA

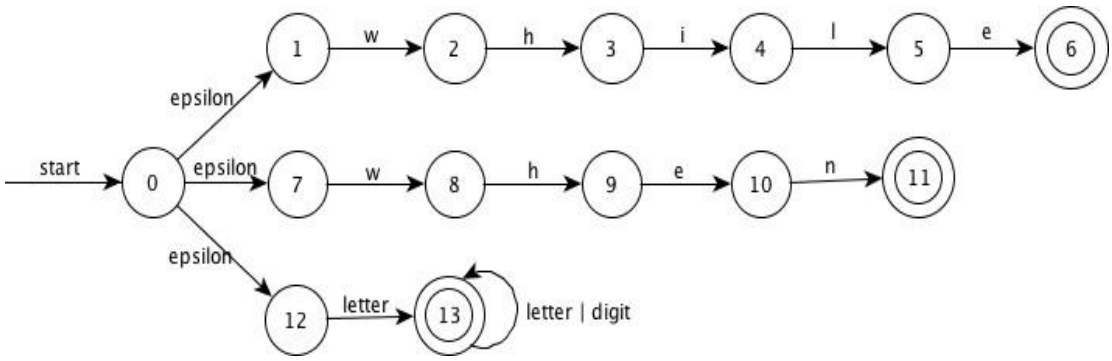


3.8.2

Repeat Exercise 3.8.1 for tokens consisting of (1) the keyword while, (2) the keyword when, and (3) identifiers consisting of strings of letters and digits, beginning with a letter.

Answer

1. NFA



2. DFA

   bother to paint

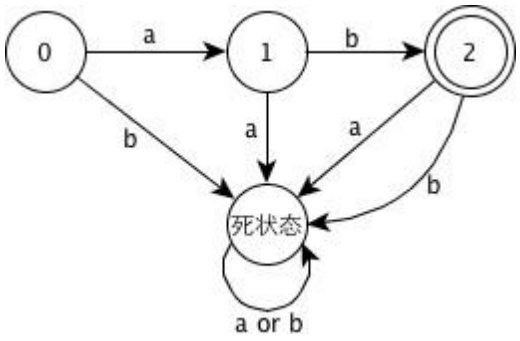
3.8.3 !

Suppose we were to revise the definition of a DFA to allow zero or one transition out of each state on each input symbol (rather than exactly one such transition, as in the standard DFA definition). Some regular expressions would then have smaller "DFA's" than they do under the standard definition of a DFA. Give an example of one such regular expression.

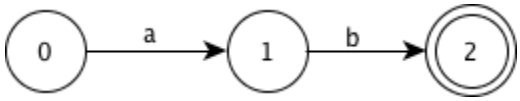
Answer

Take the language defined by regular expression "ab" as the example, assume that the set of input symbols is {a, b}

Standard DFA



Revised DFA



Obviously, the revised DFA is smaller than the standard DFA.

3.8.4 !!

Design an algorithm to recognize Lex-lookahead patterns of the form rl/r2, where rl and r2 are regular expressions. Show how your algorithm works on the following inputs:

1. (abcd|abc)/d
2. (a|ab)/ba
3. aa*/a*

Exercises for Section 3.9

3.9.1

Extend the table of Fig. 3.58 to include the operators

1. ?
2. +

Answer

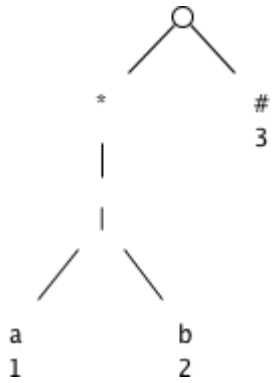
node n	nullable(n)	firstpos(n)
n = c_1 ?	true	firstpos(c_1)
n = c_1 +	nullable(c_1)	firstpos(c_1)

3.9.2

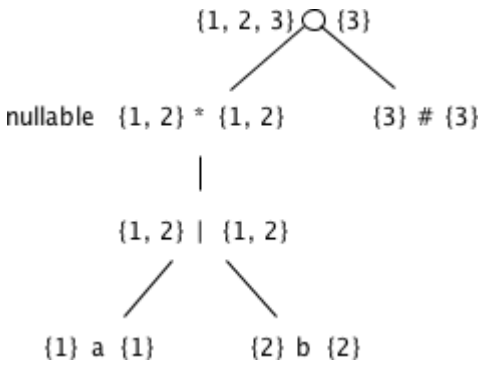
Use Algorithm 3.36 to convert the regular expressions of Exercise 3.7.3 directly to deterministic finite automata.

Answer

1. (a|b)*
    - Syntax tree



- firstpos and lastpos for nodes in the syntax tree



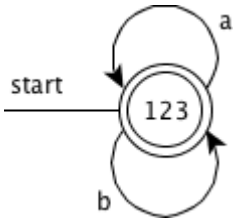

```
- The function followpos

    <table>
        <thead>
            <tr>
                <th>node n</th>
                <th>followpos(n)</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>1</td>
                <td>{1, 2, 3}</td>
            </tr>
            <tr>
                <td>2</td>
                <td>{1, 2, 3}</td>
            </tr>
            <tr>
                <td>3</td>
                <td>Ø</td>
            </tr>
        </tbody>
    </table>
```

- Steps

  The value of firstpos for the root of the tree is {1, 2, 3}, so this set is the start state of D. Call this set of states A. We compute Dtran[A, a] and Dtran[A, b]. Among the positions of A, 1 correspond to a, while 2 correspond to b. Thus Dtran[A, a] = followpos(1) = {1, 2, 3}, Dtran[A, b] = followpos(2) = {1, 2, 3}. Both the results are set A, so dose not have new state, end the computation.

- DFA



1. (a*|b*)*
2. ((ε|a)|b*)*
3. (a|b)*abb(a|b)*

3.9.3 !

We can prove that two regular expressions are equivalent by showing that their minimum-state DFA's are the same up to renaming of states. Show in this way that the following regular expressions: (a|b)*, (a*|b*)*, and ((ε|a)b*)* are all equivalent. Note: You may have constructed the DFA's for these expressions in response to Exercise 3.7.3.

Answer

Refer to the answers of 3.7.3 and 3.9.2-1

3.9.4 !

Construct the minimum-state DFA's for the following regular expressions:

1. (a|b)*a(a|b)
2. (a|b)*a(a|b)(a|b)
3. (a|b)*a(a|b)(a|b)(a|b)

Do you see a pattern?

3.9.5 !!

To make formal the informal claim of Example 3.25, show that any deterministic finite automaton for the regular expression

(a|b)*a(a|b)...(a|b)

where (a|b) appears n - 1 times at the end, must have at least 2n states. Hint: Observe the pattern in Exercise 3.9.4. What condition regarding the history of inputs does each state represent?