Introduction to Automata Theory, Languages, and Computation

Solutions for Chapter 7

Revised 3/11/01.

Solutions for Section 7.1

Exercise 7.1.1

A and *C* are clearly generating, since they have productions with terminal bodies. Then we can discover *S* is generating because of the production *S->CA*, whose body consists of only symbols that are generating. However, *B* is not generating. Eliminating *B*, leaves the grammar

```
S -> CA
A -> a
C -> b
```

Since *S, A*, and *C* are each reachable from *S*, all the remaining symbols are useful, and the above grammar is the answer to the question.

Exercise 7.1.2

Revised 6/27/02.

a)

Only *S* is nullable, so we must choose, at each point where *S* occurs in a body, to eliminate it or not. Since there is no body that consists only of *S*'s, we do not have to invoke the rule about not eliminating an entire body. The resulting grammar:

```
S -> ASB | AB
A -> aAS | aA | a
B -> SbS | bS | Sb | b | A | bb
```

b)

The only unit production is *B -> A*. Thus, it suffices to replace this body *A* by the bodies of all the *A*-productions. The result:

```
S -> ASB | AB
A -> aAS | aA | a
B -> SbS | bS | Sb | b | aAS | aA | a | bb
```

c)

Observe that *A* and *B* each derive terminal strings, and therefore so does *S*. Thus, there are no useless symbols.

d)

Introduce variables and productions *C -> a* and *D -> b*, and use the new variables in all bodies that are not a single terminal:

```
S -> ASB | AB
A -> CAS | CA | a
```

```
B -> SDS | DS | SD | b | CAS | CA | a | DD
C -> a
D -> b
```

Finally, there are bodies of length 3; one, *CAS*, appears twice. Introduce new variables *E, F*, and *G* to split these bodies, yielding the CNF grammar:

```
S -> AE | AB
A -> CF | CA | a
B -> SG | DS | SD | b | CF | CA | a | DD
C -> a
D -> b
E -> SB
F -> AS
G -> DS
```

Exercise 7.1.10

It's not possible. The reason is that an easy induction on the number of steps in a derivation shows that every sentential form has odd length. Thus, it is not possible to find such a grammar for a language as simple as {00}.

To see why, suppose we begin with start symbol *S* and try to pick a first production. If we pick a production with a single terminal as body, we derive a string of length 1 and are done. If we pick a body with three variables, then, since there is no way for a variable to derive epsilon, we are forced to derive a string of length 3 or more.

Exercise 7.1.11(b)

The statement of the entire construction may be a bit tricky, since you need to use the construction of part (c) in (b), although we are not publishing the solution to (c). The construction for (b) is by induction on *i*, but it needs to be of the stronger statement that if an *A_i*-production has a body beginning with *A_j*, then *j* > *i* (i.e., we use part (c) to eliminate the possibility that *i=j*).

Basis: For *i* = 1 we simply apply the construction of (c) for *i = 1*.

Induction: If there is any production of the form *A_i -> A_1...*, use the construction of (a) to replace *A_1*. That gives us a situation where all *A_i* production bodies begin with at least *A_2* or a terminal. Similarly, replace initial *A_2*'s using (a), to make *A_3* the lowest possible variable beginning an *A_i*-production. In this manner, we eventually guarantee that the body of each *A_i*-production either begins with a terminal or with *A_j*, for some *j* >= *i*. A use of the construction from (c) eliminates the possibility that *i = j*.

Exercise 7.1.11(d)

As per the hint, we do a backwards induction on *i*, that the bodies of *A_i* productions can be made to begin with terminals.

Basis: For $i = k$, there is nothing to do, since there are no variables with index higher than k to begin the body.

Induction: Assume the statement for indexes greater than i. If an A_i-production begins with a variable, it must be A_j for some $j > i$. By the induction hypothesis, the A_j-productions all have bodies beginning with terminals now. Thus, we may use the construction (a) to replace the initial A_j, yielding only A_i-productions whose bodies begin with terminals.

After fixing all the A_i-productions for all i, it is time to work on the B_i-productions. Since these have bodies that begin with either terminals or A_j for some j, and the latter variables have only bodies that begin with terminals, application of construction (a) fixes the B_j's.

Solutions for Section 7.2

Exercise 7.2.1(a)

Let n be the pumping-lemma constant and consider string $z = a^nb^{n+1}c^{n+2}$. We may write $z = uvwxy$, where v and x, may be ``pumped," and $|vwx| <= n$. If vwx does not have c's, then uv^3wx^3y has at least $n+2$ a's or b's, and thus could not be in the language.

If vwx has a c, then it could not have an a, because its length is limited to n. Thus, uwy has n a's, but no more than $2n+2$ b's and c's in total. Thus, it is not possible that uwy has more b's than a's and also has more c's than b's. We conclude that uwy is not in the language, and now have a contradiction no matter how z is broken into $uvwxy$.

Exercise 7.2.1(d)

Let n be the pumping-lemma constant and consider $z = 0^n1^n^2$. We break $Z = uvwxy$ according to the pumping lemma. If vwx consists only of 0's, then uwy has n^2 1's and fewer than n 0's; it is not in the language. If vwx has only 1's, then we derive a contradiction similarly. If either v or x has both 0's and 1's, then uv^2wx^2y is not in $0*1*$, and thus could not be in the language.

Finally, consider the case where v consists of 0's only, say k 0's, and x consists of m 1's only, where k and m are both positive. Then for all i, $uv^{i+1}wx^{i+1}y$ consists of $(n+ik)^2 = n^2 + 2ink + i^2k^2$ 0's and $n^2 + im$ 1's. If the number of 1's is always to be the square of the number of 0's, we must have, for some positive k and m: $2ink + i^2k^2 = im$. But the left side grows quadratically in i, while the right side grows linearly, and so this equality for all i is impossible. We conclude that for at least some i, $uv^{i+1}wx^{i+1}y$ is not in the language and have thus derived a contradiction in all cases.

Exercise 7.2.2(b)

It could be that, when the adversary breaks *z = uvwxy*, *v = 0^k* and *x = 1^k*. Then, for all *i*, *uv^iwx^iy* is in the language.

Exercise 7.2.2(c)

The adversary could choose *z = uvwxy* so that *v* and *x* are single symbols, on either side of the center. That is, |*u*| = |*y*|, and *w* is either epsilon (if *z* is of even length) or the single, middle symbol (if *z* is of odd length). Since *z* is a palindrome, *v* and *x* will be the same symbol. Then *uv^iwx^iy* is always a palindrome.

Exercise 7.2.4

The hint turns out to be a bad one. The easiest way to prove this result starts with a string *z = 0^n1^n0^n1^n* where the middle two blocks are distinguished. Note that *vwx* cannot include 1's from the second block and also 1's from the fourth block, because then *vwx* would have all *n* distinguished 0's and thus at least *n+1* distinguished symbols. Likewise, it cannot have 0's from both blocks of 0's. Thus, when we pump *v* and *x*, we must get an imbalance between the blocks of 1's or the blocks of 0's, yielding a string not in the language.

Solutions for Section 7.3

Exercise 7.3.1(a)

For each variable *A* of the original grammar *G*, let *A′* be a new variable that generates *init* of what *A* generates. Thus, if *S* is the start symbol of *G*, we make *S′* the new start symbol.

If *A -> BC* is a production of *G*, then in the new grammar we have *A -> BC*, *A′ -> BC′*, and *A′ -> B′*. If *A -> a* is a production of *G*, then the new grammar has *A -> a*, *A′ -> a*, and *A′ -> epsilon*.

Exercise 7.3.1(b)

The construction is similar to that of part (a), but now *A′* must be designed to generate string *w* if and only if *A* generates *wa*. That is, *A″*s language is the result of applying */a* to *A*'s language.

If *G* has production *A -> BC*, then the new grammar has *A -> BC* and *A′ -> BC′*. If *G* has *A -> b* for some *b != a*, then the new grammar has *A -> b*, but we do not add any production for *A′*. If *G* has *A -> a*, then the new grammar has *A -> a* and *A′ -> epsilon*.

Exercise 7.3.3(a)

Consider the language *L = {a^ib^jc^k | i <= k* or *j <= k}*. *L* is easily seen to be a CFL; you can design a PDA that guesses whether to compare the *a*'s or *b*'s with the *c*'s.

However, *min(L) = {a^ib^jc^k | k = min(i,j)}*. It is also easy to show, using the pumping lemma, that this language is not a CFL. Let *n* be the pumping-lemma constant, and consider *Z = a^nb^nc^n*.

Exercise 7.3.4(b)

If we start with a string of the form *0^n1^n* and intersperse any number of 0's, we can obtain any string of 0's and 1's that begins with at least as many 0's as there are 1's in the entire string.

Exercise 7.3.4(c)

Given DFA's for *L_1* and *L_2*, we can construct an NFA for their shuffle by using the product of the two sets of states, that is, all states *[p,q]* such that *p* is a state of the automaton for *L_1*, and *q* is a state of the automaton for *L_2*. The start state of the automaton for the shuffle consists of the start states of the two automata, and its accepting states consist of pairs of accepting states, one from each DFA.

The NFA for the shuffle guesses, at each input, whether it is from *L_1* or *L_2*. More formally, *delta([p,q],a) = {[delta_1(p,a),q], [p,delta_2(q,a)]}*, where *delta_i* is the transition function for the DFA for *L_i* (*i* = 1 or 2). It is then an easy induction on the length of *w* that *delta-hat([p_0,q_0],w)* contains *[p,q]* if and only if *w* is the shuffle of some *x* and *y*, where *delta_1-hat(p_0,x) = p* and *delta_2-hat(q_0,y) = q*.

Exercise 7.3.5

a)

Consider the language of regular expression **(0l)***. Its permutations consist of all strings with an equal number of 0's and 1's, which is easily shown not regular. In proof, use the pumping lemma for regular languages, let *n* be the pumping-lemma constant, and consider string *0^n1^n*.

b)

The language of **(012)*** serves. Its permutations are all strings with an equal number of 0's 1's, and 2's. We can prove this language not to be a CFL by using the pumping lemma on *0^n1^n2^n*, where *n* is the pumping-lemma constant.

c)

Assume the alphabet of regular language *L* is {0,1}. We can design a PDA *P* to recognize *perm(L)*, as follows. *P* simulates the DFA *A* for *L* on an input string that it guesses. However, *P* must also check that its own input is a permutation of the guessed string. Thus, each time *P* guesses an input for *A*, it also reads one of its own symbols. *P* uses its stack to remember whether it has seen more 0's than it has guessed, or seen more 1's than it has guessed. It does so by keeping a stack string with a bottom-of-stack marker and either as many more 0's as it has seen than guessed, or as many more 1's as it has seen than guessed.

For instance, if *P* guesses 0 as an input for *A* but sees a 1 on its own input, then *P*:

1. If 0 is the top stack symbol, then push another 0.
2. If 1 is the top stack symbol, then pop the stack.
3. If *Z_0*, the bottom-of-stack marker is on top, push a 0.

In addition, if *P* exposes the bottom-of-stack marker, then it has guessed, as input to *A*, a permutation of the input *P* has seen. Thus, if *A* is in an accepting state, *P* has a choice of move to pop its stack on epsilon input, thus accepting by empty stack.

Solutions for Section 7.4

Exercise 7.4.1(a)

If there is any string at all that can be ``pumped," then the language is infinite. Thus, let *n* be the pumping-lemma constant. If there are no strings as long as *n*, then surely the language is finite. However, how do we tell if there is some string of length *n* or more? If we had to consider all such strings, we'd never get done, and that would not give us a decision algorithm.

The trick is to realize that if there is any string of length *n* or more, then there will be one whose length is in the range *n* through *2n-1*, inclusive. For suppose not. Let *z* be a string that is as short as possible, subject to the constraint that $|z| >= n$. If $|z| < 2n$, we are done; we have found a string in the desired length range. If $|z| >= 2n$, use the pumping lemma to write $z = uvwxy$. We know uwy is also in the language, but because $|vwx| <= n$, we know $|z| > |uwy| >= n$. That contradicts our assumption that *z* was as short as possible among strings of length *n* or more in the language.

We conclude that $|z| < 2n$. Thus, our algorithm to test finiteness is to test membership of all strings of length between *n* and *2n-1*. If we find one, the language is infinite, and if not, then the language is finite.

Exercise 7.4.3(a)

Here is the table:

```
   {S,A,C}
   {B}      {B}
   {B}      {S,C}    {B}
   {S,C}    {S,A}    {S,C}    {S,A}
   {A,C}    {B}      {A,C}    {B}      {A,C}
----------------------------------------
     a        b        a        b        a
```

Since *S* appears in the upper-left corner, *ababa* is in the language.

Exercise 7.4.4

The proof is an induction on n that if $A \Rightarrow^* w$, for any variable A, and $|w| = n$, then all parse trees with A at the root and yield w have $2n-1$ interior nodes.

Basis: $n = 1$. The parse tree must have a root with variable A and a leaf with one terminal. This tree has $2n-1 = 1$ interior node.

Induction: Assume the statement for strings of length less than n, and let $n > 1$. Then the parse tree begins with A at the root and two children labeled by variables B and C. Then we can write $w = xy$, where $B \Rightarrow^* x$ and $C \Rightarrow^* y$. Also, x and y are each shorter than length n, so the inductive hypothesis applies to them, and we know that the parse trees for these derivations have, respectively, $2|x|-1$ and $2|y|-1$ interior nodes.

Thus, the parse tree for $A \Rightarrow^* w$ has one (for the root) plus the sum of these two quantities number of interior nodes, or $2(|x|+|y|-1)$ interior nodes. Since $|x|+|y| = |w| = n$, we are done; the parse tree for $A \Rightarrow^* w$ has $2n-1$ interior nodes.