CHAPTER 13
Modeling Computation

SECTION 13.1 Languages and Grammars

There is no magical way to come up with the grammars to generate a language described in English. In particular, Exercises 15 and 16 are challenging and very worthwhile. Exercise 21 shows how grammars can be combined. In constructing grammars, we observe the rule that every production must contain at least one nonterminal symbol on the left. This allows us to know when a derivation is completed—namely, when the string we have generated contains no nonterminal symbols.

1. The following sequences of lines show that each is a valid sentence.

a) sentence
noun phrase intransitive verb phrase
article adjective noun intransitive verb phrase
article adjective noun intransitive verb
the **adjective noun intransitive verb**
the happy **noun intransitive verb**
the happy hare **intransitive verb**
the happy hare runs

b) sentence
noun phrase intransitive verb phrase
article adjective noun intransitive verb phrase
article adjective noun intransitive verb adverb
the **adjective noun intransitive verb adverb**
the sleepy **noun intransitive verb adverb**
the sleepy tortoise **intransitive verb adverb**
the sleepy tortoise runs **adverb**
the sleepy tortoise runs quickly

c) sentence
noun phrase transitive verb phrase noun phrase
article noun transitive verb phrase noun phrase
article noun transitive verb noun phrase
article noun transitive verb article noun
the **noun transitive verb article noun**
the tortoise **transitive verb article noun**
the tortoise passes **article noun**
the tortoise passes the **noun**
the tortoise passes the hare

d) sentence
noun phrase transitive verb phrase noun phrase
article adjective noun transitive verb phrase noun phrase
article adjective noun transitive verb noun phrase

article adjective noun transitive verb article adjective noun
the **adjective noun transitive verb article adjective noun**
the sleepy **noun transitive verb article adjective noun**
the sleepy hare **transitive verb article adjective noun**
the sleepy hare passes **article adjective noun**
the sleepy hare passes the **adjective noun**
the sleepy hare passes the happy **noun**
the sleepy hare passes the happy tortoise

3. Since *runs* is only an **intransitive verb**, it can only occur in a sentence of the form **noun phrase intransitive verb phrase**. Such a sentence cannot have anything except an **adverb** after the **intransitive verb**, and *the sleepy tortoise* cannot be an **adverb**.

5. **a)** It suffices to give a derivation of this string. We write the derivation in the obvious way. $S \Rightarrow 1A \Rightarrow 10B \Rightarrow 101A \Rightarrow 1010B \Rightarrow 10101$.

b) This follows from our solution to part **(c)**, because 10110 has two 1's in a row and is not of the form discussed there.

c) Notice that the only production with A on the left is $A \rightarrow 0B$. Furthermore, the only productions with B on the left are $B \rightarrow 1A$ and $B \rightarrow 1$. Combining these, we see that we can eliminate B and replace these three rules by $A \rightarrow 01A$ and $A \rightarrow 01$. This tells us that every string in the language generated by G must end with some number of repetitions of 01 (at least one). Furthermore, because of the rules $S \rightarrow 0A$ and $S \rightarrow 1A$, the string must start with either a 0 or a 1 preceding the repetitions of 01. Therefore the strings in this language consist of a 0 or a 1 followed by one or more repetitions of 01. We can write this as $\{0(01)^n \mid n \geq 0\} \cup \{1(01)^n \mid n \geq 0\}$

7. We write the derivation in the obvious way. $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000111$. We used the rule $S \rightarrow 0S1$ in the first three steps and $S \rightarrow \lambda$ in the last step.

9. **a)** Using G_1, we can add 0's on the left or 1's on the right of S. Thus we have $S \Rightarrow 0S \Rightarrow 00S \Rightarrow 00S1 \Rightarrow 00S11 \Rightarrow 00S111 \Rightarrow 00S1111 \Rightarrow 001111$.

b) In this grammar we must add all the 0's first to S, then change to an A and add the 1's, again on the left. Thus we have $S \Rightarrow 0S \Rightarrow 00S \Rightarrow 001A \Rightarrow 0011A \Rightarrow 00111A \Rightarrow 001111$.

11. First we apply the first rule twice and the rule $S \rightarrow \lambda$ to get $00ABAB$. We can then apply the rule $BA \rightarrow AB$, to get $00AABB$. Now we can apply the rules $0A \rightarrow 01$ and $1A \rightarrow 11$ to get $0011BB$; and then the rules $1B \rightarrow 12$ and $2B \rightarrow 22$ to end up with 001122, as desired.

13. In each case we will list only the productions, because V and T will be obvious from the context, and S speaks for itself.

a) For this finite set of strings, we can simply have $S \rightarrow 0$, $S \rightarrow 1$, and $S \rightarrow 11$.

b) We assume that "only 1's" includes the case of no 1's. Thus we can take simply $S \rightarrow 1S$ and $S \rightarrow \lambda$.

c) The middle can be anything we like, and we will let A represent the middle. Then our productions are $S \rightarrow 0A1$, $A \rightarrow 1A$, $A \rightarrow 0A$, and $A \rightarrow \lambda$.

d) We will let A represent the pairs of 1's. Then our productions are $S \rightarrow 0A$, $A \rightarrow 11A$, and $A \rightarrow \lambda$.

15. **a)** We need to add the 0's two at a time. Thus we can take the rules $S \rightarrow S00$ and $S \rightarrow \lambda$.

b) We can use the same first rule as in part **(a)**, namely $S \rightarrow S00$, to increase the number of 0's. Since the string must begin 10, we simply adjoin to this the rule $S \rightarrow 10$.

c) We need to add 0's and 1's two at a time. Furthermore, we need to allow for 0's and 1's to change their order. Since we cannot have a rule $01 \rightarrow 10$ (there being no nonterminal symbol on the left), we make up nonterminal analogs of 0 and 1, calling them A and B, respectively. Thus our rules are as follows: $S \rightarrow AAS$, $S \rightarrow BBS$, $AB \rightarrow BA$, $BA \rightarrow AB$, $S \rightarrow \lambda$, $A \rightarrow 0$, and $B \rightarrow 1$. (There are also totally different ways to approach this problem, which are just as effective.)

d) This one is fairly simple: $S \rightarrow 0000000000A$, $A \rightarrow 0A$, $A \rightarrow \lambda$. This assures at least 10 0's and allows for any number of additional 0's.

e) We need to invoke the trick used in part **(c)** to allow 0's and 1's to change their order. Furthermore, since we need at least one extra 0, we use $S \rightarrow A$ as our vanishing condition, rather than $S \rightarrow \lambda$. Our solution, then, is $S \rightarrow AS$, $S \rightarrow ABS$, $S \rightarrow A$, $AB \rightarrow BA$, $BA \rightarrow AB$, $A \rightarrow 0$, and $B \rightarrow 1$.

f) This is identical to part **(e)**, except that the vanishing condition is $S \rightarrow \lambda$, rather than $S \rightarrow A$, and there is no rule $S \rightarrow AS$.

g) We just put together two copies of a solution to part **(e)**, one in which there are more 0's than 1's, and one in which there are more 1's than 0's. The rules are as follows: $S \rightarrow ABS$, $S \rightarrow T$, $S \rightarrow U$, $T \rightarrow AT$, $T \rightarrow A$, $U \rightarrow BU$, $U \rightarrow B$, $AB \rightarrow BA$, $BA \rightarrow AB$, $A \rightarrow 0$, and $B \rightarrow 1$.

17. In each case we will list only the productions, because V and T will be obvious from the context, and S speaks for itself.

a) It suffices to have $S \rightarrow 0S$ and $S \rightarrow \lambda$.

b) We let A represent the string of 1's. Thus we take $S \rightarrow A0$, $A \rightarrow 1A$, and $A \rightarrow \lambda$. Notice that $A \rightarrow A1$ works just as well $A \rightarrow 1A$ here, so either one is fine.

c) It suffices to have $S \rightarrow 000S$ and $S \rightarrow \lambda$.

19. a) This is a type 2 grammar, because the left-hand side of each production has a single nonterminal symbol. It is not a type 3 grammar, because the right-hand side of the productions are not of the required type.

b) This meets the definition of a type 3 grammar.

c) This is only a type 0 grammar; it does not fit the definition of type 1 because the right side of the second production does not maintain the context set by the left side.

d) This is a type 2 grammar, because the left-hand side of each production has a single nonterminal symbol. It is not a type 3 grammar, because the right-hand side of the productions are not of the required type.

e) This meets the definition of a type 2 grammar. It is not of type 3, because of the production $A \rightarrow B$.

f) This is only a type 0 grammar; it does not fit the definition of type 1 because the right side of the second production does not maintain the context set by the left side.

g) This meets the definition of a type 3 grammar. Note, however, that it does not meet the definition of a type 1 grammar because of $S \rightarrow \lambda$.

h) This is only a type 0 grammar; it does not fit the definition of type 1 because the right side of the third production does not maintain the context set by the left side.

i) This is a type 2 grammar because each left-hand side is a single nonterminal. It is not type 3 because of the production $B \rightarrow \lambda$.

j) This is a type 2 grammar because each left-hand side is a single nonterminal. It is not type 3; each of the productions violates the conditions imposed for a type 3 grammar.

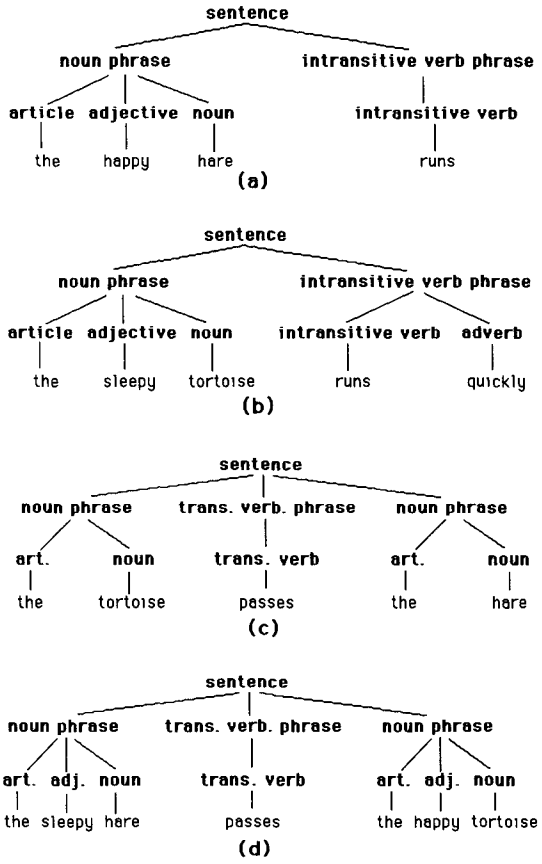
21. Let us assume that the nonterminal symbols of G_1 and G_2 are disjoint. (If they are not, we can give those in G_2, say, new names so that they will be; obviously this does not change the language that G_2 generates.) Call the start symbols S_1 and S_2. In each case we will define G by taking all the symbols and rules for G_1 and G_2, a new symbol S, which will be the start symbol for G, and the rules listed below.

a) Since we want strings that either G_1 or G_2 generate, we add the rules $S \rightarrow S_1$ and $S \rightarrow S_2$.

b) Since we want strings that consist of a string that G_1 generates followed by a string that G_2 generates, we add the rule $S \rightarrow S_1 S_2$.

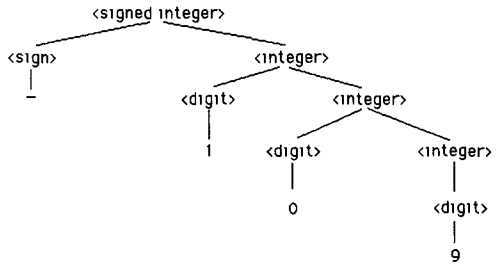
c) This time we add the rules $S \rightarrow S_1 S$ and $S \rightarrow \lambda$. This clearly gives us all strings that consist of the concatenation of any number of strings that G_1 generates.

23. We simply translate the derivations we gave in the solution to Exercise 1 to tree form, obtaining the following pictures.



25. We can assume that the derivation starts $S \Rightarrow AB \Rightarrow CaB \Rightarrow cbaB$, or $S \Rightarrow AB \Rightarrow CaB \Rightarrow baB$. This shows that neither the string in part **(b)** nor the string in part **(d)** is in the language, since they do not begin cba or ba. In order to derive the string in part **(a)**, we need to turn B into ba, and this is easy, using the rule $B \rightarrow Ba$ and then the rule $B \rightarrow b$. Finally, for part **(c)**, we again simply apply these two rules to change B into ba.

27. This is straightforward. The $-$ is the sign and the 109 is an integer, so the tree starts as shown. Then we decompose the integer 109 into the digit 1 and the integer 09, then in turn to the digit 0 and the integer (digit) 9.



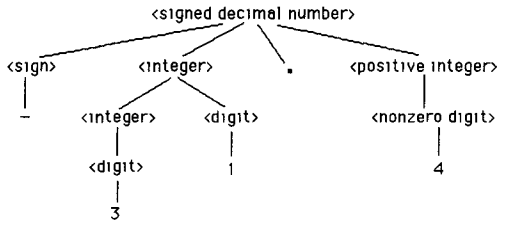
29. a) Note that a string such as "34." is not allowed by this definition, but a string such as -02.780 is. This is pretty straightforward using the following rules. As can be seen, we are using $\langle integer \rangle$ to stand for a nonnegative integer.

$S \rightarrow \langle sign \rangle \langle integer \rangle$
$S \rightarrow \langle sign \rangle \langle integer \rangle . \langle positive\ integer \rangle$
$\langle sign \rangle \rightarrow +$
$\langle sign \rangle \rightarrow -$
$\langle integer \rangle \rightarrow \langle integer \rangle \langle digit \rangle$
$\langle integer \rangle \rightarrow \langle digit \rangle$
$\langle positive\ integer \rangle \rightarrow \langle integer \rangle \langle nonzero\ digit \rangle \langle integer \rangle$
$\langle positive\ integer \rangle \rightarrow \langle integer \rangle \langle nonzero\ digit \rangle$
$\langle positive\ integer \rangle \rightarrow \langle nonzero\ digit \rangle \langle integer \rangle$
$\langle positive\ integer \rangle \rightarrow \langle nonzero\ digit \rangle$
$\langle digit \rangle \rightarrow \langle nonzero\ digit \rangle$
$\langle digit \rangle \rightarrow 0$
$\langle nonzero\ digit \rangle \rightarrow 1$
$\langle nonzero\ digit \rangle \rightarrow 2$
$\langle nonzero\ digit \rangle \rightarrow 3$
$\langle nonzero\ digit \rangle \rightarrow 4$
$\langle nonzero\ digit \rangle \rightarrow 5$
$\langle nonzero\ digit \rangle \rightarrow 6$
$\langle nonzero\ digit \rangle \rightarrow 7$
$\langle nonzero\ digit \rangle \rightarrow 8$
$\langle nonzero\ digit \rangle \rightarrow 9$

b) We combine rows of the previous answer with the same left-hand side, and we change the notation to produce the answer to this part.

$\langle signed\ decimal\ number \rangle ::= \langle sign \rangle \langle integer \rangle \mid \langle sign \rangle \langle integer \rangle . \langle positive\ integer \rangle$
$\langle sign \rangle ::= + \mid -$
$\langle integer \rangle ::= \langle integer \rangle \langle digit \rangle \mid \langle digit \rangle$
$\langle positive\ integer \rangle ::= \langle integer \rangle \langle nonzero\ digit \rangle \langle integer \rangle \mid \langle integer \rangle \langle nonzero\ digit \rangle$
$\quad \mid \langle nonzero\ digit \rangle \langle integer \rangle \mid \langle nonzero\ digit \rangle$
$\langle digit \rangle ::= \langle nonzero\ digit \rangle \mid 0$
$\langle nonzero\ digit \rangle ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

c) We easily produce the following tree.



31. a) We can think of appending letters to the end at each stage:

$\langle identifier \rangle ::= \langle lcletter \rangle \mid \langle identifier \rangle \langle lcletter \rangle$
$\langle lcletter \rangle ::= a \mid b \mid c \mid \cdots \mid z$

b) We need to be more explicit here than in part **(a)** about how many letters are used:

$\langle identifier \rangle ::= \langle lcletter \rangle \langle lcletter \rangle \langle lcletter \rangle \mid \langle lcletter \rangle \langle lcletter \rangle \langle lcletter \rangle \langle lcletter \rangle \mid$
$\quad \langle lcletter \rangle \langle lcletter \rangle \langle lcletter \rangle \langle lcletter \rangle \langle lcletter \rangle \mid$

$\langle lcletter\rangle\langle lcletter\rangle\langle lcletter\rangle\langle lcletter\rangle\langle lcletter\rangle\langle lcletter\rangle$

$\langle lcletter\rangle ::= a \mid b \mid c \mid \cdots \mid z$

c) This is similar to the part **(b)**, allowing for two types of letters:

$\langle identifier\rangle ::= \langle ucletter\rangle \mid \langle ucletter\rangle\langle letter\rangle \mid \langle ucletter\rangle\langle letter\rangle\langle letter\rangle \mid$
$\langle ucletter\rangle\langle letter\rangle\langle letter\rangle\langle letter\rangle \mid \langle ucletter\rangle\langle letter\rangle\langle letter\rangle\langle letter\rangle\langle letter\rangle \mid$
$\langle ucletter\rangle\langle letter\rangle\langle letter\rangle\langle letter\rangle\langle letter\rangle\langle letter\rangle$

$\langle letter\rangle ::= \langle lcletter\rangle \mid \langle ucletter\rangle$

$\langle lcletter\rangle ::= a \mid b \mid c \mid \cdots \mid z$

$\langle ucletter\rangle ::= A \mid B \mid C \mid \cdots \mid Z$

d) This is again similar to previous parts. We need to invent a name for "digit or underscore."

$\langle identifier\rangle ::= \langle lcletter\rangle\langle digitorus\rangle\langle alphanumeric\rangle\langle alphanumeric\rangle\langle alphanumeric\rangle \mid$
$\langle lcletter\rangle\langle digitorus\rangle\langle alphanumeric\rangle\langle alphanumeric\rangle\langle alphanumeric\rangle\langle alphanumeric\rangle$

$\langle digitorus\rangle ::= \langle digit\rangle \mid \_$

$\langle alphanumeric\rangle ::= \langle letter\rangle \mid \langle digit\rangle$

$\langle letter\rangle ::= \langle lcletter\rangle \mid \langle ucletter\rangle$

$\langle lcletter\rangle ::= a \mid b \mid c \mid \cdots \mid z$

$\langle ucletter\rangle ::= A \mid B \mid C \mid \cdots \mid Z$

$\langle digit\rangle ::= 0 \mid 1 \mid 2 \mid \cdots \mid 9$

33. We create a name for "letter or underscore" and then define an identifier to consist of one of those, followed by any number of other allowed symbols. Note that an underscore by itself is a valid identifier, and there is no prohibition on consecutive underscores.

$\langle identifier\rangle ::= \langle letterorus\rangle \mid \langle identifier\rangle\langle symbol\rangle$

$\langle symbol\rangle ::= \langle letterorus\rangle \mid \langle digit\rangle$

$\langle letterorus\rangle ::= \langle letter\rangle \mid \_$

$\langle letter\rangle ::= \langle lcletter\rangle \mid \langle ucletter\rangle$

$\langle lcletter\rangle ::= a \mid b \mid c \mid \cdots \mid z$

$\langle ucletter\rangle ::= A \mid B \mid C \mid \cdots \mid Z$

$\langle digit\rangle ::= 0 \mid 1 \mid 2 \mid \cdots \mid 9$

35. We assume that leading 0's are not allowed in the whole number part, since the problem explicitly mentioned them only for the decimal part. Our rules have to allow the optional sign using the question mark, the integer part consisting of one or more digits, not beginning with a 0 unless 0 is the entire whole number part, and then either the decimal part or not. Note that the decimal part has a decimal point followed by zero or more digits.

$numeral ::= sign?\ nonzerodigit\ digit*\ decimal? \mid sign?\ 0\ decimal?$

$decimal ::= .digit*$

$digit ::= 0 \mid nonzerodigit$

$sign ::= + \mid -$

$nonzerodigit ::= 1 \mid 2 \mid \cdots \mid 9$

37. We can simplify the answer given in Exercise 33 using the asterisk for repeating optional elements.

$identifier ::= letterorus\ symbol*$

$symbol ::= letterorus \mid digit$

$letterorus ::= letter \mid \_$

$letter ::= lcletter \mid ucletter$

$lcletter ::= a \mid b \mid c \mid \cdots \mid z$

$ucletter ::= A \mid B \mid C \mid \cdots \mid Z$

$digit ::= 0 \mid 1 \mid 2 \mid \cdots \mid 9$

39. **a)** This string is generated by the grammar. The substring $bc*$ is a term, since it consists of the factor b followed by the factor c followed by the mulOperator $*$. Thus the entire expression consists of two terms followed by an addOperator. We can show the steps in the following sequence:

$\langle expression \rangle$
$\langle term \rangle \langle term \rangle \langle addOperator \rangle$
$\langle factor \rangle \langle factor \rangle \langle factor \rangle \langle mulOperator \rangle \langle addOperator \rangle$
$\langle identifier \rangle \langle identifier \rangle \langle identifier \rangle \langle mulOperator \rangle \langle addOperator \rangle$
$a\,b\,c\,*\,+$

b) This string is not generated by the grammar. The second plus sign needs two terms preceding it, and $xy+$ can only be deconstructed to be one term.

c) This string is generated by the grammar. The substring $xy-$ is a factor, since it is an expression, namely the term x followed by the term y followed by the addOperator $-$. Thus the entire expression consists of two factors followed by a mulOperator. We can show the steps in the following sequence:

$\langle expression \rangle$
$\langle term \rangle$
$\langle factor \rangle \langle factor \rangle \langle mulOperator \rangle$
$\langle expression \rangle \langle factor \rangle \langle mulOperator \rangle$
$\langle term \rangle \langle term \rangle \langle addOperator \rangle \langle factor \rangle \langle mulOperator \rangle$
$\langle factor \rangle \langle factor \rangle \langle addOperator \rangle \langle factor \rangle \langle mulOperator \rangle$
$\langle identifier \rangle \langle identifier \rangle \langle addOperator \rangle \langle identifier \rangle \langle mulOperator \rangle$
$x\,y\,-\,z\,*$

d) This is similar to part **(c)**. The entire expression consists of two factors followed by a mulOperator; the first of these factors is just w, and the second is the term $x\,y\,z\,-\,*$. That term, in turn, deconstructs as in previous parts. We can show the steps in the following sequence:

$\langle expression \rangle$
$\langle term \rangle$
$\langle factor \rangle \langle factor \rangle \langle mulOperator \rangle$
$\langle factor \rangle \langle expression \rangle \langle mulOperator \rangle$
$\langle factor \rangle \langle term \rangle \langle mulOperator \rangle$
$\langle factor \rangle \langle factor \rangle \langle factor \rangle \langle mulOperator \rangle \langle mulOperator \rangle$
$\langle factor \rangle \langle factor \rangle \langle expression \rangle \langle mulOperator \rangle \langle mulOperator \rangle$
$\langle factor \rangle \langle factor \rangle \langle term \rangle \langle term \rangle \langle addOperator \rangle \langle mulOperator \rangle \langle mulOperator \rangle$
$\langle factor \rangle \langle factor \rangle \langle factor \rangle \langle factor \rangle \langle addOperator \rangle \langle mulOperator \rangle \langle mulOperator \rangle$
$\langle identifier \rangle \langle identifier \rangle \langle identifier \rangle \langle identifier \rangle \langle addOperator \rangle \langle mulOperator \rangle \langle mulOperator \rangle$
$w\,x\,y\,z\,-\,*\,/$

e) This string is generated as follows (similar to previous parts of this exercise):

$\langle expression \rangle$
$\langle term \rangle$
$\langle factor \rangle \langle factor \rangle \langle mulOperator \rangle$
$\langle factor \rangle \langle expression \rangle \langle mulOperator \rangle$
$\langle factor \rangle \langle term \rangle \langle term \rangle \langle addOperator \rangle \langle mulOperator \rangle$
$\langle factor \rangle \langle factor \rangle \langle factor \rangle \langle addOperator \rangle \langle mulOperator \rangle$
$\langle identifier \rangle \langle identifier \rangle \langle identifier \rangle \langle addOperator \rangle \langle mulOperator \rangle$
$a\,d\,e\,-\,*$

41. The answers will depend on the grammar given as the solution to Exercise 40. We assume here that the answer to that exercise is very similar to the preamble to Exercise 39. The only difference is that the operators are

placed between their operands, rather than behind them, and parentheses are required in expressions used as factors.

a) This string is not generated by the grammar, because the addition operator can only be applied to two terms, and terms that are themselves expressions must be surrounded by parentheses.

b) This string is generated by the grammar. The substrings a/b and c/d are terms, so they can be combined to form the expression. We show the steps in the following sequence:

$\langle expression \rangle$
$\langle term \rangle \langle addOperator \rangle \langle term \rangle$
$\langle factor \rangle \langle mulOperator \rangle \langle factor \rangle \langle addOperator \rangle \langle factor \rangle \langle mulOperator \rangle \langle factor \rangle$
$\langle identifier \rangle \langle mulOperator \rangle \langle identifier \rangle \langle addOperator \rangle \langle identifier \rangle \langle mulOperator \rangle \langle identifier \rangle$
$a/b + c/d$

c) This string is generated by the grammar. The substring $(n + p)$ is a factor, since it is an expression surrounded by parentheses. We show the steps in the following sequence:

$\langle expression \rangle$
$\langle term \rangle$
$\langle factor \rangle \langle mulOperator \rangle \langle factor \rangle$
$\langle factor \rangle \langle mulOperator \rangle (\langle expression \rangle)$
$\langle factor \rangle \langle mulOperator \rangle (\langle term \rangle \langle addOperator \rangle \langle term \rangle)$
$\langle factor \rangle \langle mulOperator \rangle (\langle factor \rangle \langle addOperator \rangle \langle factor \rangle)$
$\langle identifier \rangle \langle mulOperator \rangle (\langle identifier \rangle \langle addOperator \rangle \langle identifier \rangle)$
$m * (n + p)$

d) There are several reasons that this string is not generated, among them the fact that it is impossible for an expression to start with an operator in this grammar.

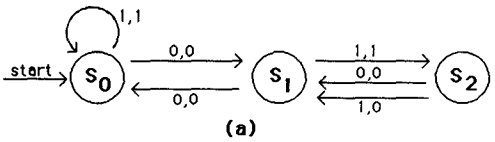
e) This is very similar to part **(c)**:

$\langle expression \rangle$
$\langle term \rangle$
$\langle factor \rangle \langle mulOperator \rangle \langle factor \rangle$
$(\langle expression \rangle) \langle mulOperator \rangle (\langle expression \rangle)$
$(\langle term \rangle \langle addOperator \rangle \langle term \rangle) \langle mulOperator \rangle (\langle term \rangle \langle addOperator \rangle \langle term \rangle)$
$(\langle factor \rangle \langle addOperator \rangle \langle factor \rangle) \langle mulOperator \rangle (\langle factor \rangle \langle addOperator \rangle \langle factor \rangle)$
$(\langle identifier \rangle \langle addOperator \rangle \langle identifier \rangle) \langle mulOperator \rangle (\langle identifier \rangle \langle addOperator \rangle \langle identifier \rangle)$
$(m + n) * (p - q)$

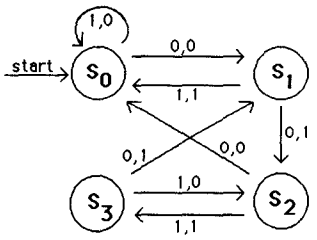
SECTION 13.2 Finite-State Machines with Output

Finding finite-state machines to do specific tasks is in essence computer programming. There is no set method for doing this. You have to think about the problem for awhile, ask yourself what it might be useful for the states to represent, and then very carefully proceed to construct the machine. Expect to have several false starts. "Bugs" in your machines are also very common. There are of course many machines that will accomplish the same task. The reader should look at Exercises 20–25 to see that it is also possible to build finite-state machines with the output associated with the states, rather than the transitions.

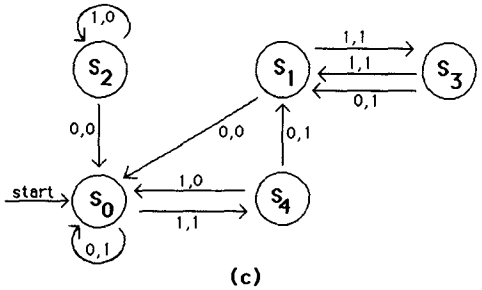
1. We draw the state diagrams by making a node for each state and a labeled arrow for each transition. In part **(a)**, for example, since under input 1 from state s_2 we are told that we move to state s_1 and output a 0, we draw an arrow from s_2 to s_1 and label it $1,0$. It is assumed that s_0 is always the start state.



(a)



(b)



(c)

3. **a)** The machine starts in state s_0. Since the first input symbol is 0, the machine moves to state s_1 and gives 0 as output. The next input symbol is 1, so the machine moves to state s_2 and gives 1 as output. The next input is 1, so the machine moves to state s_1 and gives 0 as output. The fourth input is 1, so the machine moves to state s_2 and gives 1 as output. The fifth input is 0, so the machine moves to state s_1 and gives 0 as output. Thus the output is 01010.

b) The machine starts in state s_0. Since the first input symbol is 0, the machine moves to state s_1 and gives 0 as output. The next input symbol is 1, so the machine moves to state s_0 and gives 1 as output. The next input is 1, so the machine stays in state s_0 and gives 0 as output. The fourth input is 1, so the machine again stays in state s_0 and gives 0 as output. The fifth input is 0, so the machine moves to state s_1 and gives 0 as output. Thus the output is 01000.

c) The machine starts in state s_0. Since the first input symbol is 0, the machine stays in state s_0 and gives 1 as output. The next input symbol is 1, so the machine moves to state s_4 and gives 1 as output. The next input is 1, so the machine moves to state s_0 and gives 0 as output. The fourth input is 1, so the machine moves to state s_4 and gives 1 as output. The fifth input is 0, so the machine moves to state s_1 and gives 1 as output. Thus the output is 11011.

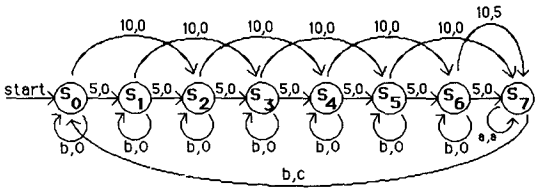
5. **a)** The machine starts in state s_0. Since the first input symbol is 0, the machine moves to state s_1 and gives 1 as output. (This is what the arrow from s_0 to s_1 with label $0,1$ means.) The next input symbol is 1. Because of the edge from s_1 to s_0, the machine moves to state s_0 and gives 1 as output. The next input is 1. Because of the loop at s_0, the machine stays in state s_0 and gives output 0. The same thing happens on the fourth input symbol. Therefore the output is 1100 (and the machine ends up in state s_0).

b) This is similar to part **(a)**. The first two symbols of input cause the machine to output two 0's and remain in state s_0. The third symbol causes an output of 1 as the machine moves into state s_1. The fourth input

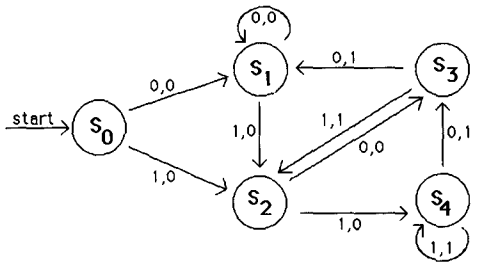
takes us back to state s_0 with output 1. The next four symbols of input cause the machine to give output 0110 as it goes to states s_0, s_1, s_0, and s_0, respectively. Therefore the output is 00110110.

c) This is similar to the other parts. The machine alternates between states s_0 and s_1, outputting 1 for each input. Thus the output is 11111111111.

7. We model this machine as follows. There are four possible inputs, which we denote by 5, 10, 25, and b, standing for a nickel, a dime, a quarter, and a button labeled by a kind of soda pop, respectively. (Actually the model is a bit more complicated, since there are three kinds of pop, but we will ignore that; to incorporate the kind of pop into the model, we would simply have three inputs in place of just b.) The output can either be an amount of money in cents—0, 5, 10, 15, 20, or 25—or can be a can of soda pop, which we denote c. There will be eight states. Intuitively, state s_i will represent the state in which the machine is indebted to the customer by $5i$ cents. Thus s_0, the start state, will represent that the machine owes the customer nothing; state s_1 will represent that the machine has accepted 5 cents from the customer, and so on. State s_7 will mean that the machine owes the customer 35 cents, which will be paid with a can of soda pop, at which time the machine will return to state s_0, owing nothing. The following picture is the state diagram of this machine, simplified even further in that we have eliminated quarters entirely for sake of readability. For example, the transition from state s_6 (30 cents credit) on input of a dime is to state s_7 (35 cents credit) with the return of 5 cents in change. We have also used a to stand for any monetary input: if you deposit any amount when the machine already has your 35 cents, then you get that same amount back. Thus the transition a, a really stands for three transitions: $5, 5$ and $10, 10$ and $25, 25$.

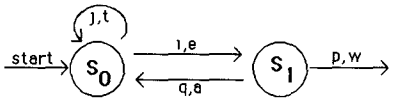


9. We draw the diagram for this machine. Intuitively, we need four states, corresponding to the four possibilities for what the last two bits have been. In our picture, state s_1 corresponds to the last two bits having been 00; state s_2 corresponds to the last two bits having been 01; state s_3 corresponds to the last two bits having been 10; state s_4 corresponds to the last two bits having been 11. We also need a state s_0 to get started, to account for the delay. Let us see why some of the transitions are what they are. If you are in state s_3, then the last two bits have been 10. If you now receive an input 0, then the last two bits will be 00, so we need to move to state s_1. Furthermore, since the bit received two pulses ago was a 1 (we know this from the fact that we are in state s_3), we need to output a 1. Also, since we are told to output 00 at the beginning, it is right to have transitions from s_0 as shown.

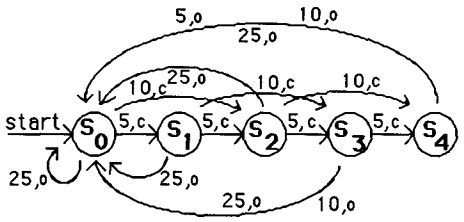


If we look at this machine, we observe that states s_0 and s_1 are equivalent, i.e., they cause exactly the same transitions and outputs. Therefore a simpler answer would be a machine like this one, but without state s_0, where state s_1 is the start state.

11. This machine is really only part of a machine; we are not told what happens after a successful log-on. Also, the machine is really much more complicated than we are indicating here, because we really need a separate state for each user. We assume that there is only one user. We also assume that an invalid user ID is rejected immediately, without a request for a password. (The alternate assumption is also reasonable, that the machine requests a password whether or not the ID is valid. In that case we obtain a different machine, of course.) We need only two states. The initial state waits for the valid user ID. We let i be the valid user ID, and we let j be any other input. If the input is valid, then we enter state s_1, outputting the message e: "enter your password." If the input is not valid, then we remain in state s_0, outputting the message t: "invalid ID; try again." From state s_1 there are only two relevant inputs: the valid password p and any other input q. If the input is valid, then we output the message w: "welcome" and proceed. If the input is invalid, then we output the message a: "invalid password; enter user ID again" and return to state s_0 to await another attempt at logging-on.



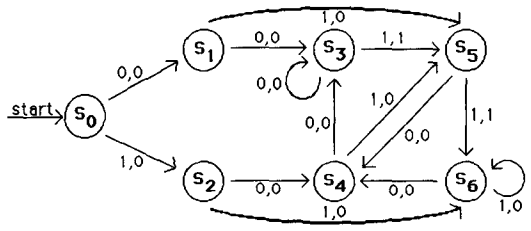
13. This exercise is similar to Exercise 7. We let state s_i for $i = 0, 1, 2, 3, 4$ represent the fact that $5i$ cents has been deposited. When at least 25 cents has been deposited, we return to state s_0 and open the gate. Nickels (input 5), dimes (input 10) and quarters (input 25) are available. We let o and c be the outputs: the gate is opened (for a limited time, of course), or remains closed. After the gate is opened, we return to state s_0. (We assume that the gate closes after the car has passed.)



15. The picture for this machine would be too complex to draw. Instead, we will describe the machine verbally, and even then we won't give every last gory detail. We assume that possible inputs are the ten digits. We will let s_0 be the start state and let s_1 be the state representing a successful call (so we will not list any outputs from s_1). From s_0, inputs of 2, 3, 4, 5, 6, 7, or 8 send the machine back to s_0 with output of an error message for the user. From s_0 an input of 0 sends the machine to state s_1, with the output being that the 0 is sent to the network. From s_0 an input of 9 sends the machine to state s_2 with no output; from there an input of 1 sends the machine to state s_3 with no output; from there an input of 1 sends the machine to state s_1 with the output being that the 911 is sent to the network. All other inputs while in states s_2 or s_3 send the machine back to s_0 with output of an error message for the user. From s_0 an input of 1 sends the machine to state s_4 with no output; from s_4 an input of 2 sends the machine to state s_5 with no output; and this path continues in a similar manner to the 911 path, looking next for 1, then 2, then any seven digits, at which point the machine goes to state s_1 with the output being that the ten-digit input is sent to the network. Any "incorrect" input while in states s_5 or s_6 (that is, anything except a 1 while in s_5 or a 2 while in s_6) sends the machine back to s_0 with output of an error message for the user. Similarly, from s_4 an input of 8 followed by appropriate successors drives us eventually to s_1, but inappropriate outputs drive us back to s_0 with an error message. Also, inputs while in state s_4 other than 2 or 8 send the machine back s_0 with output of an error message for the user.

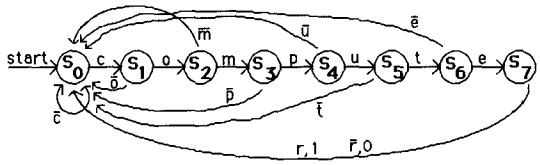
17. We interpret this problem as asking that a 1 be output if the conditions are met, and a 0 be output otherwise. For this machine, we need to keep track of what the last two inputs have been, and we need four states to

"store" this information. Let the states s_3, s_4, s_5, and s_6 be the states corresponding to the last two inputs having been 00, 10, 01, and 11, respectively. We also need some states to get started—to get us into one of these four states. There are only two cases in which the output is 1: if we are in states s_3 or s_5 (so that the last two inputs have been 00 or 01) and we receive a 1 as input. The transitions in our machine are the obvious ones. For example, if we are in state s_5, having just read 01, and receive a 0 as input, then the last two symbols read are now 10, so we move to state s_4.



As in Exercise 9, we can actually get by with a smaller machine. Note that here states s_1 and s_4 are equivalent, as are states s_2 and s_6. Thus we can merge each of these pairs into one state, producing a machine with only five states. At that point, furthermore, state s_0 is equivalent to the merged s_2 and s_6, so we can omit state s_0 and make this other state the start state. The reader is urged to draw the diagram for this simpler machine.

19. We need some notation to make our picture readable. The alphabet has 26 symbols in it. If α is a letter, then by $\overline{\alpha}$ we mean any letter other than α. Thus an arrow labeled $\overline{\alpha}$ really stands for 25 arrows. The output is to be 1 when we have just finished reading the word *computer*. Thus we need eight states, to stand for the various stages of having read part of that word. The picture below gives the details, except that we have omitted all the outputs except on inputs r and $\overline{r}$; all the omitted ones are intended to be 0. The reader might contemplate why this problem would have been harder if the word in question were something like *baboon*.



21. We construct the state table by having one row for each state. The arrows tell us what the values of the transition function are. For example, since there is an arrow from s_0 to s_1 labeled 0, the transition from s_0 on input 0 is to s_1. Similarly, the transition from s_0 on input 1 is to s_2. The output function values are shown next to each state. Thus the output for state s_0 is 1, the output for state s_1 is 1, and the output for state s_2 is 0. The table is therefore as shown here.

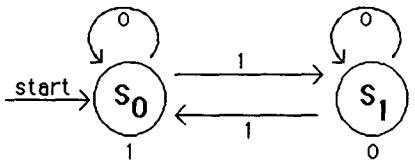
	Input		
State	0	1	Output
s_0	s_1	s_2	1
s_1	s_1	s_0	1
s_2	s_1	s_2	0

23. **a)** The input drives the machine successively to states s_1, s_0, s_1, and s_0. The output is the output of the start state, followed by the outputs of these four states, namely 11111.

b) The input drives the machine to state s_2, where it remains because of the loop. The output is the output of the start state, followed by the output at state s_2 six times, namely 1000000.

c) The states visited after the start state are, in order, s_2, s_2, s_2, s_1, s_0, s_2, s_2, s_1, s_0, s_2, and s_2. Therefore the output is 100011001100.

25. We can use a machine with just two states, one to indicate that there is an even number of 1's in the input string, the other to indicate that there is an odd number of 1's in the string. Since the empty string has an even number of 1's, we make s_0 (the start state) the state for an even number of 1's. The output for this state will be 1, as directed. The output from state s_1 will be 0 to indicate an odd number of 1's. The input 1 will drive the machine from one state to the other, while the input 0 will keep the machine in its current state. The diagram below gives the desired machine.



SECTION 13.3 Finite-State Machines with No Output

As in the previous section, many of these exercises are really exercises in programming. There is no magical way to become a good programmer, but experience helps. The converse problem is also hard—finding a good verbal description of the set recognized by a given finite-state automaton.

1. **a)** This is the set of all strings ab, where $a \in A$ and $b \in B$. Thus it contains precisely 000, 001, 1100, and 1101.

b) This is the set of all strings ba, where $a \in A$ and $b \in B$. Thus it contains precisely 000, 0011, 010, and 0111.

c) This is the set of all strings a_1a_2, where $a_1 \in A$ and $a_2 \in A$. Thus it contains precisely 00, 011, 110, and 1111.

d) This is the set of all strings $b_1b_2b_3$, where each $b_i \in B$. Thus it contains precisely 000000, 000001, 000100, 000101, 010000, 010001, 010100 and 010101.

3. Two possibilities are of course to let A be this entire set and let $B = \{\lambda\}$, and to let B be this entire set and let $A = \{\lambda\}$. Let us find more. With a little experimentation we see that $A = \{\lambda, 10\}$ and $B = \{10, 111, 1000\}$ also works, and it can be argued that there are no other solutions in which λ appears in either set. Finally, there is the solution $A = \{1, 101\}$ and $B = \{0, 11, 000\}$. It can be argued that there are no more. (Here is how the first of these arguments goes. If $\lambda \in A$, then necessarily $\lambda \notin B$. Hence the shortest string in B has length at least 2, from which it follows that $10 \in B$. Now since the only other string in AB that ends with 10 is 1010, the only possible other string in A is 10. This leads to the third solution mentioned above. On the other hand, if $\lambda \in B$, then $\lambda \notin A$, so it must be that the shortest string in A is 10. This forces 111 to be in A, and now there can be no other strings in B. The second argument is similar.)

5. **a)** One way to write this answer is $\{(10)^n \mid n = 0, 1, 2, \dots\}$. It is the concatenation of zero or more copies of the string 10.

b) This is like part **(a)**. This set consists of all copies of zero or more concatenations of the string 111. In other words, it is the set of all strings of 1's of length a multiple of 3. In symbols, it is $\{(111)^n \mid n = 0, 1, 2, \dots\} = \{1^{3n} \mid n = 0, 1, 2, \dots\}$.

c) A little thought will show that this consists of all bit strings in which every 1 is immediately preceded by a 0. No other restrictions are imposed, since $0 \in A$.

d) Because the 0 appears only in 101, the strings formed here have the property that there are at least two 1's between every pair of 0's in the string, and the string begins and ends with a 1. All strings satisfying this property are in A^*.

7. This follows directly from the definition. Every string w in A^* consists of the concatenation of one or more strings from A. Since $A \subseteq B$, all of these strings are also in B, so w is the concatenation of one or more strings from B, i.e., is in B^*.

9. **a)** This set contains all bit strings, so of course the answer is yes.

b) This set contains all strings consisting of any number of 1's, followed by any number of 0's, followed by any number of 1's. Since 11101 is such a string, the answer is yes.

c) Any string belonging to this set must start 110, and 11101 does not, so the answer is no.

d) All the strings in this set must in particular have even length. The given string has odd length, so the answer is no.

e) The answer is yes. Just take one copy of each of the strings 111 and 0, together with the required string 1.

f) The answer is yes again. Just take 11 from the first set and 101 from the second.

11. In each case we will list the states in the order that they are visited, starting with the initial state. All we need to do then is to note whether the place we end up is a final state (s_0 or s_3) or a nonfinal state. (It is interesting to note that there are no transitions to s_3, so this state can never be reached.)

a) We encounter $s_0s_1s_2s_0$, so this string is accepted.

b) We encounter $s_0s_0s_0s_1s_2$, so this string is not accepted.

c) We encounter $s_0s_1s_0s_1s_0s_1s_2s_0$, so this string is accepted.

d) We encounter $s_0s_0s_1s_2s_0s_1s_2s_0s_1s_2$, so this string is not accepted.

13. **a)** The set in question is the set of all strings of zero or more 0's. Since the machine in Figure 1 has s_0 as a final state, and since there is a transition from s_0 to itself on input 0, every string of zero or more 0's will leave the machine in state s_0 and will therefore be accepted. Therefore the answer is yes.

b) Since this set is a subset of the set in part **(a)**, the answer must be yes.

c) One string in this set is the string 1. Since an input of 1 drives the machine to the nonfinal state s_1, not every string in this set is accepted. Therefore the answer is no.

d) One string in this set is the string 01. Since an input of 01 drives the machine to the nonfinal state s_1, not every string in this set is accepted. Therefore the answer is no.

e) The answer here is no for exactly the same reason as in part **(d)**.

f) The answer here is no for exactly the same reason as in part **(c)**.

15. We use structural induction on the input string y. The basis step is $y = \lambda$, and for the inductive step we write $y = wa$, where $w \in I^*$ and $a \in I$. For the basis step, we have $xy = x$, so we must show that $f(s,x) = f(f(s,x),\lambda)$. But part (i) of the definition of the extended transition function says that this is true. We then assume the inductive hypothesis that the equation holds for shorter strings and try to prove that $f(s,xwa) = f(f(s,x),wa)$. By part (ii) of the definition, the left-hand side of this equation equals $f(f(s,xw),a)$. By the inductive hypothesis (because w is shorter than y), $f(s,xw) = f(f(s,x),w)$, so $f(f(s,xw),a) = f(f(f(s,x),w),a)$. On the other hand, the right-hand side of our desired equality is, by part (ii) of the definition, equal to $f(f(f(s,x),w),a)$. We have shown that the two sides are equal, and our proof is complete.

17. The only final state is s_2, so we need to determine which strings drive the machine to state s_2. Clearly the strings 0, 10, and 11 do so, as well as any of these strings followed by anything else. Thus we can write the answer as $\{0,10,11\}\{0,1\}^*$.

19. A string is accepted if and only if it drives this machine to state s_1. Thus the string must consist of zero or more 0's, followed by a 1, followed by zero or more 1's. In short, the answer is $\{\, 0^m1^n \mid m \geq 0 \wedge n \geq 1 \,\}$.

21. Because s_0 is final, the empty string is accepted. The strings that drive the machine to final state s_3 are precisely $\{0\}\{1\}^*\{0\}$. There are three ways to get to final state s_4, and once we get there, we stay there. The path through s_2 tells us that strings in $\{10,11\}\{0,1\}^*$ are accepted. The path $s_0s_1s_3s_4$ tells us that strings in $\{0\}\{1\}^*\{01\}\{0,1\}^*$ are accepted. And the path $s_0s_1s_3s_5s_4$ tells us that strings in $\{0\}\{1\}^*\{00\}\{0\}^*\{1\}\{0,1\}^*$ are accepted. Thus the language recognized by this machine is $\{\lambda\}\cup\{0\}\{1\}^*\{0\}\cup\{10,11\}\{0,1\}^*\cup\{0\}\{1\}^*\{01\}\{0,1\}^*\cup\{0\}\{1\}^*\{00\}\{0\}^*\{1\}\{0,1\}^*$.

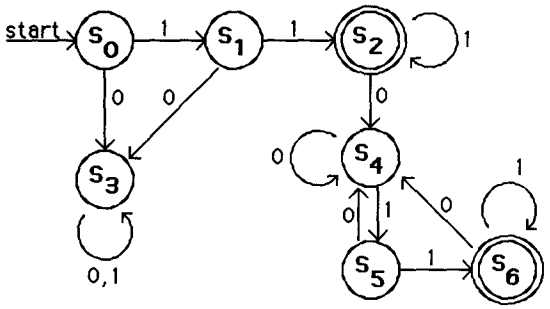
23. We want to accept only the strings that begin 01. Let s_2 be the only final state, and put transitions from s_2 to itself on either input. We want to reach s_2 after encountering 01, so put a transition from the start state s_0 to s_1 on input 0, and a transition from s_1 to s_2 on input 1. Finally make a "graveyard" state s_3, and have the other transitions from s_0 and s_1 (as well as both transitions from s_3) lead to s_3.

25. We can have a sequence of three states to record the appearance of 101. State s_1 will signify that we have just seen a 1; state s_2 will signify that we have just seen a 1 followed by a 0; state s_3 will be the only final state and will signify that we have seen the string 101. Put transitions from s_3 to itself on either input (it doesn't matter what follows the appearance of 101). Put a transition from the start state s_0 to itself on input 0, because we are still waiting for a 1. Put a transition from s_0 to s_1 on input 1 (because we have just seen a 1). From s_1 on input 0 we want to go to state s_2, but on input 1 we stay at s_1 because we have still just seen a 1. Finally, from s_2, put a transition on input 1 to the final state s_3 (success!), but on input 0 we have to start over looking for 101, so this transition must be back to s_0.

27. We can let state s_i, for $i = 0,1,2,3$ represent that exactly i 0's have been seen, and state s_4 will represent that four or more 0's have been seen. Only s_3 will be final. For $i = 0,1,2,3$, we transition from s_i to itself on input 1 and to s_{i+1} on input 0. Both transitions from s_4 are to itself.

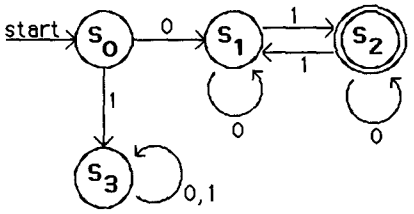
29. We can let state s_i, for $i = 0,1,2,3$ represent that i consecutive 1's have been seen. Only s_3 will be final. For $i = 0,1,2$, we transition from s_i to s_{i+1} on input 1 but back to s_0 on input 0. Both transitions from s_3 are to itself.

31. This is a little tricky. We want states at the start that prevent us from accepting a string if it does not start with 11. Once we have seen the first two 1's, we can accept the string if we do not encounter a 0 (after all, the strings 11 and 111 do satisfy the condition). We can also accept the string if it has anything whatsoever in the middle, as long as it ends 11. The machine shown below accomplishes all this. Note that s_3 is a graveyard state, and state s_4 is where we "start over" looking for the final 11.



33. We need just two states, s_0 to represent having seen an even number of 0's (this will be the start state, because to begin we have seen no 0's), and s_1 to represent having seen an odd number of 0's (this will be the only final state). The transitions are from each state to itself on input 1, and from each state to the other on input 0.

35. This is similar to Exercise 33, except that we need to look for the initial 0. Note that s_3 is the graveyard.



37. We prove this by contradiction. Suppose that such a machine exists, with start state s_0 and other state s_1. Because the empty string is not in the language but some strings are accepted, we must have s_1 as the only final state, with at least one transition from s_0 to s_1. Because the string 0 is not in the language, any transition from s_0 on input 0 must be to itself, so there must be a transition from s_0 to s_1 on input 1. But this cannot happen, because the string 1 is not in the language. Having obtained a contradiction, we conclude that no such finite-state automaton exists.

39. We want the new machine to accept exactly those strings that the original machine rejects, and vice versa. So we simply change each final state to a nonfinal state and change each nonfinal state to a final state.

41. We use exactly the same machine as in Exercise 25, but make s_0, s_1, and s_2 the final states and make s_3 nonfinal.

43. First some general comments on Exercises 43–49: In general it is quite hard to describe succinctly languages recognized by machines. An ad hoc approach is usually best. In this exercise there is only one final state, s_2, and only three ways to get there, namely on input 0, 01, or 11. Therefore the language recognized by this machine is $\{0, 01, 11\}$.

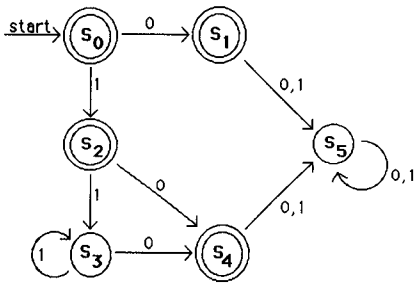
45. Clearly the empty string is accepted. There are essentially two ways to get to the final state s_2. We can go through state s_1, and every string of the form 0^n1^m, where n and m are positive integers, will take us through state s_1 on to s_2. We can also bypass state s_1, and every string of the form 01^m for $m \geq 0$ will take us directly to s_2. Thus our answer is $\{\lambda\} \cup \{\, 0^n1^m \mid n, m \geq 1 \,\} \cup \{\, 01^m \mid m \geq 0 \,\}$. Note that this can also be written as $\{\lambda, 0\} \cup \{\, 0^n1^m \mid n, m \geq 1 \,\}$.

47. First it is easy to see that all strings of the form 10^n for $n \geq 0$ can drive the machine to the final state s_1. Next we see that all strings of the form 10^n10^m for $n, m \geq 0$ can drive the machine to state s_3. No other strings can drive the machine to a final state. Therefore the answer is $\{\, 10^n \mid n \geq 0 \,\} \cup \{\, 10^n10^m \mid n, m \geq 0 \,\}$.

49. We notice first that state s_2 is a final state, that once we get there, we can stay there, and that any string that starts with a 0 can lead us there. Therefore all strings that start with a 0 are in the language. If the string starts with a 1, then we must go first to state s_1. If we ever leave state s_1, then the string will not be accepted, because there are no paths out of s_1 that lead to a final state. Therefore the only other strings that are in the language are the empty string (because s_0 is final) and those strings that can drive the machine to state s_1, namely strings consisting of all 1's (we've already included those of the form 01^*). Therefore the language accepted by this machine is the union of the set of all strings that start with a 0 and the set of all strings that have no 0's.

51. One way to do Exercises 50–54 is to construct a machine following the proof of Theorem 1. Rather than do that, we construct the machines in an ad hoc way, using the answers obtained in Exercises 43–47. Since λ, 0, and 1 are accepted by the nondeterministic automaton in Exercise 44, we make states s_0, s_1, and s_2 in the

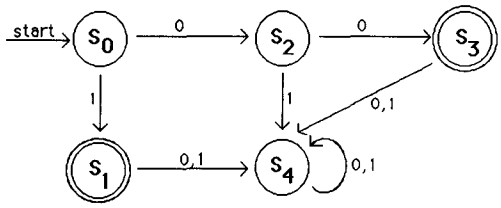
following diagram final. States s_3 and s_4 provide for the acceptance of strings of the form 1^n0 for all $n \geq 1$. State s_5, the graveyard state, assures that no other strings are accepted.



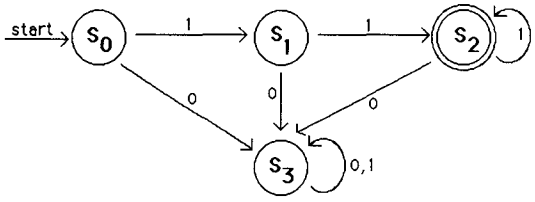
53. This machine is practically deterministic already, since there are no cases of ambiguous transitions (a given input allowing transition to more than one state). All that keeps this machine from being deterministic is that there are no transitions from certain states on certain inputs. Therefore to make this machine deterministic, we just need to add a "graveyard" state, s_3, with transitions from s_0 on input 0 and from s_1 on input 1 to this graveyard state, and transitions from s_3 to itself on input 0 or 1. The graveyard state is not final, of course.

55. **a)** We want to accept only the string 0. Let s_1 be the only final state, where we reach s_1 on input 0 from the start state s_0. Make a "graveyard" state s_2, and have all other transitions (there are five of them in all) lead there.

b) This uses the same idea as in part **(a)**, but we need a few more states. The graveyard state is s_4. See the picture for details.



c) In the picture of our machine, we show a transition to the graveyard state whenever we encounter a 0. The only final state is s_2, which we reach after 11 and remain at as long as the input consists just of 1's.



57. Intuitively, the reason that a finite-state automaton cannot recognize the set of bit strings containing an equal number of 0's and 1's is that there is not enough "memory" in the machine to keep track of how many extra 0's or 1's the machine has read so far. Of course, this intuition does not constitute a proof—maybe we are just not being clever enough to see how a machine could do this with a finite number of states. Instead, we must give a proof of this assertion. See Exercises 22–25 of Section 13.4 for a development of what are called "pumping lemmas" to handle various problems like this. (See also Example 6 in Section 13.4.)

The natural way to prove a negative statement such as this is by contradiction. So let us suppose that we do have a finite-state automaton M that accepts precisely the set of bit strings containing an equal number of 0's and 1's. We will derive a contradiction by showing that the machine must accept some illegal strings. The

idea behind the proof is that since there are only finitely many states, the machine must repeat some states as it computes. In this way, it can get into arbitrarily long loops, and this will lead us to a contradiction. To be specific, suppose that M has n states. Consider the string $0^{n+1}1^{n+1}$. As the machine processes this string, it must encounter the same state more than once as it reads the first $n+1$ 0's (by the pigeonhole principle). Say that it hits state s twice. Then some positive number, say k, of 0's in the input drives M from state s back to state s. But then the machine will end up at exactly the same place after reading $0^{n+1+k}1^{n+1}$ as it will after reading $0^{n+1}1^{n+1}$, since those extra k 0's simply drove it in a loop. Therefore since M accepts $0^{n+1}1^{n+1}$, it also accepts $0^{n+1+k}1^{n+1}$. But this is a contradiction, since this latter string does not have the same number of 0's as 1's.

59. We know from Exercise 58d that the equivalence classes of R_k are a refinement of the equivalence classes of R_{k-1} for each positive integer k. The equivalence classes are finite sets, and finite sets cannot be refined indefinitely (the most refined they can be is for each equivalence class to contain just one state). Therefore this sequence of refinements must stabilize and remain unchanged from some point onward. It remains to show that as soon as we have $R_n = R_{n+1}$, then $R_n = R_m$ for all $m > n$, from which it follows that $R_n = R_*$, and so the equivalence classes for these two relations will be the same. By induction, it suffices to show that if $R_n = R_{n+1}$, then $R_{n+1} = R_{n+2}$. By way of contradiction, suppose that $R_{n+1} \neq R_{n+2}$. This means that there are states s and t that are $(n+1)$-equivalent but not $(n+2)$-equivalent. Thus there is a string x of length $n+2$ such that, say, $f(s,x)$ is final but $f(t,x)$ is nonfinal. Write $x = aw$, where $a \in I$. Then $f(s,a)$ and $f(t,a)$ are not $(n+1)$-equivalent, because w drives the first to a final state and the second to a nonfinal state. But $f(s,a)$ and $f(t,a)$ are n-equivalent, because s and t are $(n+1)$-equivalent. This contradicts the fact that $R_n = R_{n+1}$, and our proof is complete.

61. **a)** By the way the machine $\overline{M}$ was constructed, a string will drive M from the start state to a final state if and only if that string drives $\overline{M}$ from the start state to a final state.

b) For a proof of this theorem, see a source such as *Introduction to Automata Theory, Languages, and Computation* (2nd Edition) by John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman (Addison Wesley, 2000).

SECTION 13.4 Language Recognition

Finding good verbal descriptions of the set of strings generated by a regular expression is not easy; neither is finding a good regular expression for a given verbal description. What Kleene's theorem says is that these problems of "programming" in regular expressions are really the same as the programming problems for machines discussed in the previous section. The **pumping lemma**, *discussed in Exercise 22 and the three exercises that follow it, is an important technique for proving that certain sets are not regular.*

1. **a)** This regular expression generates all strings consisting of zero or more 1's, followed by a lone 0.

b) This regular expression generates all strings consisting of zero or more 1's, followed by one or more 0's.

c) This set has only two elements, 111 and 001.

d) This set contains all strings in which the 0's come in pairs.

e) This set consists of all strings in which every 1 is preceded by at least one 0, with the proviso that the string ends in a 1 if it is not the empty string.

f) This gives us all strings of length at least 3 that end 00.

3. In each case we try to view 0101 as fitting the regular expression description.

a) The strings described by this regular expression have at most three "blocks" of different digits—a 0, then some 1's, then some 0's. Thus we cannot get the string 0101, which has four blocks.

b) The 1's that might come between the first and second 0 in any string described by this regular expression must come in pairs (because of the $(11)^*$). Therefore we cannot get 0101. Alternatively, note that every string described by this regular expression must have odd length.

c) We can get this string as $0(10)^1 1^1$.

d) We can get this string as $0^1 10(1)$, where the final 1 is one of the choices in $(0 \cup 1)$.

e) We can get this string as $(01)^2(11)^0$.

f) We cannot get this string, because every string with any 1's at all described by this regular expression must end with 10 or 11.

g) We cannot get this string, because every string described by this regular expression must end with 11.

h) We can get this string as $01(01)1^0$, where the second 01 is one of the choices in $(01 \cup 0)$.

5. **a)** We just need to take a union: $\mathbf{0} \cup \mathbf{11} \cup \mathbf{010}$.

b) More simply put, this is the set of strings of five or more 0's, so the regular expression is $\mathbf{000000^*}$.

c) We can use $(\mathbf{0} \cup \mathbf{1})$ to represent any symbol and $(\mathbf{00} \cup \mathbf{01} \cup \mathbf{10} \cup \mathbf{11})$ to represent any string of even length. We need one symbol followed by any string of even length, so we can take $(\mathbf{0} \cup \mathbf{1})(\mathbf{00} \cup \mathbf{01} \cup \mathbf{10} \cup \mathbf{11})^*$.

d) The one 1 can be preceded and/or followed by any number of 0's, so we have $\mathbf{0^*10^*}$.

e) This one is a little harder. In order to prevent 000 from appearing, we must have every group of one or two 0's followed by a 1 (if we note that the entire string ends with a 1 as well). Thus we can break our string down into groups of 1, 01, or 001, and we get $(\mathbf{1} \cup \mathbf{01} \cup \mathbf{001})^*$ as our regular expression.

7. **a)** We can translate "one or more 0's" into $\mathbf{00^*}$. Therefore the answer is $\mathbf{00^*1}$.

b) We can translate "two or more symbols" into $(\mathbf{0} \cup \mathbf{1})(\mathbf{0} \cup \mathbf{1})(\mathbf{0} \cup \mathbf{1})^*$. Therefore the answer is $(\mathbf{0} \cup \mathbf{1})(\mathbf{0} \cup \mathbf{1})(\mathbf{0} \cup \mathbf{1})^*\mathbf{0000^*}$.

c) A little thought tells us that we want all strings in which all the 0's come before all the 1's or all the 1's come before all the 0's. Thus the answer is $\mathbf{0^*1^*} \cup \mathbf{1^*0^*}$.

d) The string of 1's can be represented by $\mathbf{11(111)^*}$; the string of 0's, by $\mathbf{(00)^*}$. Thus the answer is $\mathbf{11(111)^*(00)^*}$.

9. **a)** The simplest solution here is to have just the start state s_0, nonfinal, with no transitions.

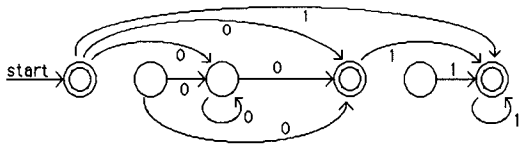
b) The simplest solution here is to have just the start state s_0, final, with no transitions.

c) The simplest solution here is to have just two states—the nonfinal start state s_0 (since we do not want to accept the empty string) and a final state s_1—and just the one transition from s_0 to s_1 on input a.

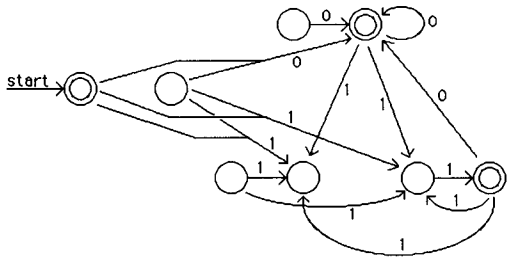
11. We can prove this by induction on the length of a regular expression for A. If this expression has length 1, then it is either $\emptyset$ or λ or x (where x is some symbol in the alphabet). In each case A is its own reversal, so there is nothing to prove. There are three inductive steps. If the regular expression for A is $\mathbf{BC}$, then $A = BC$, where B is the set generated by $\mathbf{B}$ and C is the set generated by $\mathbf{C}$. By the inductive hypothesis, we know that there are regular expressions $\mathbf{B'}$ and $\mathbf{C'}$ that generate B^R and C^R, respectively. Now $A^R = (BC)^R = (C^R)(B^R)$. Therefore a regular expression for A^R is $\mathbf{C'B'}$. The case of union is handled similarly. Let the regular expression for A be $\mathbf{B} \cup \mathbf{C}$, with B, C, $\mathbf{B'}$, and $\mathbf{C'}$ as before. Then a regular expression for A^R is $\mathbf{B'} \cup \mathbf{C'}$, since clearly $(B \cup C)^R = (B^R) \cup (C^R)$. Finally, if the regular expression for A is $\mathbf{B^*}$, then, with the same notation as before, it is easy to see that $(\mathbf{B'})^*$ is a regular expression for A^R.

13. **a)** We can build machines to recognize $\mathbf{0^*}$ and $\mathbf{1^*}$ as shown in the second row of Figure 3. Next we need to put these together to make a machine that recognizes $\mathbf{0^*1^*}$. We place the first machine on the left and

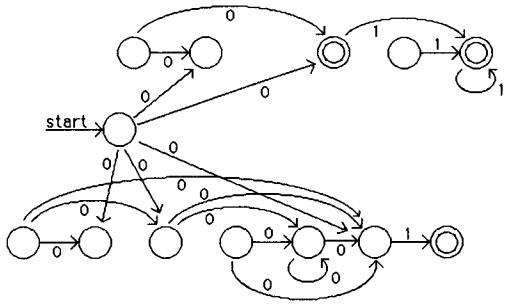
the second machine on the right. We make each final state in the first machine nonfinal (except for the start state, since $\lambda \in \mathbf{0^*1^*}$), but leave the final states in the second machine final. Next we copy each transition to a state that was formerly final in the first machine into a transition (on the same input) to the start state of the second machine. Lastly, since $\lambda \in \mathbf{0^*}$, we add the transition from the start state to the state to which there is a transition from the start state of the machine for $\mathbf{1^*}$. The result is as shown. (In all parts of this exercise we have not put names on the states in our state diagrams.)



b) This machine is quite messy. The upper portion is for $\mathbf{0}$, and the lower portion is for $\mathbf{11}$. They are combined to give a machine for $\mathbf{0 \cup 11}$. Finally, to incorporate the Kleene star, we added a new start state (on the far left), and adjusted the transitions according to the procedure shown in Figure 2.



c) This is similar to the other parts. We grouped the expression as $\mathbf{01^* \cup (00^*)1}$. The answer is as shown.



15. We choose as the nonterminal symbols corresponding to states s_0, s_1, and s_2 the symbols S, A, and B, respectively. Thus S is our start symbol. The terminal symbols are of course 0 and 1. We construct the rules for our grammars by following the procedure described in the proof of the second half of Theorem 2: putting in rules of the form $X \to aY$ for each transition from the state corresponding to X to the state corresponding to Y, on input a, and putting in a rule of the form $X \to a$ for a transition from the state corresponding to X to the final state, on input a. Specifically, since there is a transition from s_0 to s_1 on input 0, we include the rule $S \to 0A$. Similarly, the other transitions give us the rules $S \to 1B$, $A \to 0B$, $A \to 1B$, $B \to 0B$, and $B \to 1B$. Also, the transition to the final state from S on input 0 gives rise to the rule $S \to 0$. Thus our grammar contains these seven rules.

17. This is similar to Exercise 15—see the discussion there for the approach. We let C correspond to state s_3. The set of rules contains $S \to 0C$, $S \to 1A$, $A \to 1A$, $A \to 0C$, $B \to 0B$, $B \to 1B$, $C \to 0C$, and $C \to 1B$ (for the transitions from a state to another state on a given input), as well as $S \to 1$, $A \to 1$, $B \to 0$, $B \to 1$, and $C \to 1$ (for the transitions to the final states that can end the computation).

19. This is clear, since the operation of the machine is exactly mimicked by the grammar. If the current string in the derivation in the grammar is $v_1v_2 \ldots v_kA_s$, then the machine has seen input $v_1v_2 \ldots v_k$ and is currently in state s. If the current string in the derivation in the grammar is $v_1v_2 \ldots v_k$, then the machine has seen input $v_1v_2 \ldots v_k$ and is currently in some final state. Hence the machine accepts precisely those strings that the grammar generates. (The empty string does not fit this discussion, but it is handled separately—and correctly—since we take $S \rightarrow \lambda$ as a production if and only if we are supposed to.)

21. First suppose that the language recognized by M is infinite. Then the length of the words recognized by M must be unbounded, since there are only a finite number of symbols. Thus $l(x)$ is greater than the finite number $|S|$ for some word $x \in L(M)$.

Conversely, let x be such a word, and let $s_0, s_{i_1}, s_{i_2}, \ldots, s_{i_n}$ be the sequence of states that the machine goes through on input x, where $n = l(x)$ and s_{i_n} is a final state. By the pigeonhole principle, some state occurs twice in this sequence, i.e., there is a loop from this state back to itself during the computation. Let y be the substring of x that causes the loop, so that $x = uyv$. Then for every nonnegative integer k, the string uy^kv is accepted by the machine M (i.e., is in $L(M)$), since the computation is the same as the computation on input x, except that the loop is traversed k times. Thus $L(M)$ is infinite.

23. We apply the pumping lemma in a proof by contradiction. Suppose that this set were regular. Clearly it contains arbitrarily long strings. Thus the pumping lemma tells us that for some strings u, $v \neq \lambda$, and w, the string uv^iw is in our set for every i. Now if v contains both 0's and 1's, then uv^2w cannot be in the set, since it would have a 0 following a 1, which no string in our set has. On the other hand, if v contains only 0's (or only 1's), then for large enough i, it is clear that uv^iw has more than (or less than) twice as many 0's as 1's, again contradicting the definition of our set. Thus the set cannot be regular.

25. We will give a proof by contradiction, using the pumping lemma. Following the hint, let x be the palindrome 0^N10^N, for some fixed $N > |S|$, where S is the set of states in a machine that recognizes palindromes. By the lemma, we can write $x = uvw$, with $l(uv) \leq |S|$ and $l(v) \geq 1$, so that for all i, uv^iw is a palindrome. Now since $0^N10^N = uvw$ and $l(uv) \leq |S| < N$, it must be the case that v is a string consisting solely of 0's, with the 1 lying in w. Then uv^2w cannot be a palindrome, since it has more 0's before its sole 1 than it has 0's following the 1.

27. It helps to think of L/x in words—it is the set of "ends" of strings in L that start with the string x; in other words, it is the set of strings obtained from strings in L by stripping away an initial piece x. To show that 11 and 10 are distinguishable, we need to find a string z such that $11z \in L$ and $10z \notin L$ or vice versa. A little thought and trial and error shows us that $z = 1$ works: $111 \notin L$ but $101 \in L$. To see that 1 and 11 are indistinguishable, note that the only way for $1z$ to be in L is for z to end with 01, and that is also the only way for $11z$ to be in L.

29. By Exercise 28, if two strings are distinguishable, then they drive the machine from the start state to different states. Therefore, if $x_1, x_2, \ldots, x_n$ are all distinguishable, the states $f(s_0, x_1), f(s_0, x_2), \ldots, f(s_0, x_n)$ are all different, so the the machine has at least n states.

31. We claim that any two distinct strings of the same length are distinguishable with respect to the language P of all palindromes. Indeed, if x and y are distinct strings of length n, let $z = x^R$ (the reverse of string x). Then $xz \in P$ but $yz \notin P$. Note that there are 2^n different strings of length n. By Exercise 29, this tells us that any deterministic finite-state automaton for recognizing palindromes must have at least 2^n states. Because n is arbitrary (we want our machine to recognize *all* palindromes), this tells us that no finite-state machine can recognize P.

SECTION 13.5 Turing Machines

In this final section of the textbook, we have studied a machine that has all the computing capabilities possible (if one believes the Church–Turing thesis). Most of these exercises are really programming assignments, and the programming language you are stuck with is not a nice, high-level, structured language like Java or C, nor even a nice assembly language, but something much messier and less efficient. One point of the exercises is to convince you that even in this horrible setting you can, with enough time and patience, instruct the computer—the Turing machine—to do whatever you wish computationally. Keep in mind that in many senses, a Turing machine is just as powerful as any computer running programs written in any language. One reason for talking about Turing machines at all, rather than just using high-level languages, is that their simplicity makes it feasible to prove some very interesting things about them (and therefore about computers in general). For example, one can prove that computers cannot solve the halting problem (see also Section 3.1), and one can prove that a large class of problems have efficient algorithmic solutions if and only if certain very specific problems, such as a decision version of the traveling salesman problem, do (the NP-complete problems—see also Section 3.3). This is part of what makes Turing machines so important in theoretical computer science, and time spent becoming acquainted with them will not go unrewarded as you progress in this field.

1. We will indicate the configuration of the Turing machine using a notation such as $0[s_2]1B1$. This string of symbols means that the tape is blank except for a portion which reads $01B1$ from left to right; that the machine is currently in state s_2; and that the tape head is reading the left 1 (the currently scanned symbol will always be the one following the bracketed state information).

a) The initial configuration is $[s_0]0011$. Because of the five-tuple $(s_0, 0, s_1, 1, R)$ and the fact that the machine is in state s_0 and the tape head is looking at a 0, the machine changes the 0 to a 1 (i.e., writes a 1 in that square), moves to the right, and enters state s_1. Therefore the configuration at the end of one step of the computation is $1[s_1]011$. Next the transition given by the five-tuple $(s_1, 0, s_2, 1, L)$ occurs, and we reach the configuration $[s_2]1111$. There are no five-tuples starting with s_2, so the machine halts at this point. The nonblank portion of the tape contains 1111.

b) The initial configuration is $[s_0]101$. Because of the five-tuple $(s_0, 1, s_1, 0, R)$ and the fact that the machine is in state s_0 and the tape head is looking at a 1, the machine changes the 1 to a 0, moves to the right, and enters state s_1. Therefore the configuration at the end of one step of the computation is $0[s_1]01$. At this time transition $(s_1, 0, s_2, 1, L)$ kicks in, resulting in configuration $[s_2]011$, and the machine halts, with 011 on its tape.

c) We seem to have the idea from the first two parts, so let us just list the configurations here, using the notation "$\rightarrow$" to show the progression from one to the next. $[s_0]11B01 \rightarrow 0[s_1]1B01 \rightarrow 00[s_1]B01 \rightarrow 0[s_2]0001$. Therefore the final output is 00001.

d) $[s_0]B \rightarrow 0[s_1]B \rightarrow [s_2]00$. So the final tape reads 00.

3. Note that all motion is from left to right.

a) The machine starts in state s_0 and sees the first 1. Therefore using the second five-tuple, it replaces the 1 by a 0, moves to the right, and enters state s_1. Now it sees the second 1, so, using the fifth five-tuple, it replaces the 1 by a 1 (i.e., leaves it unchanged), moves to the right, and enters state s_0. The third five-tuple now tells it to leave the blank it sees alone, move to the right, and enter state s_2, which is a final (accepting) state (because it is not the first state in any five-tuple). Since there are no five-tuples telling the machine what to do in state s_2, it halts. Note that 01 is on the tape, and the input was accepted.

b) When in state s_0 the machine skips over 0's, ignoring them, until it comes to a 1. When (and if) this happens, the machine changes this 1 to a 0 and enters state s_1. Note also that if the machine hits a blank (B) while in state s_0 or s_1, then it enters the final (accepting) state s_2. Next note that s_1 plays a role similar to that played by s_0, causing the machine to skip over 0's, but causing it to go back into state s_0 if and when it encounters a 1. In state s_1, however, the machine does not change the 1 it sees to a 0. Thus

the machine will alternate between states s_0 and s_1 as it encounters 1's in the input string, changing half of these 1's to 0's. To summarize, if the machine is given a bit string as input, it scans it from left to right, changing every other occurrence of a 1, if any, starting with the first, to a 0, and otherwise leaving the string unchanged; it halts (and accepts) when it comes to the end of the string.

5. **a)** The machine starts in state s_0 and sees the first 1. Therefore using the first five-tuple, it replaces the 1 by a 0, moves to the right, and enters state s_1. Now it sees the second 1, so, using the second five-tuple, it replaces the 1 by a 1 (i.e., leaves it unchanged), moves to the right, and stays in state s_1. Since there are no five-tuples telling the machine what to do in state s_1 when reading a blank, it halts. Note that 01 is on the tape, and the input was not accepted, because s_1 is not a final state; in fact, there are no final states (states that begin no 5-tuples).
b) This is essentially the same as part **(a)**. The first 1 (if any) is changed to a 0 and the others are left alone. The input is not accepted.

7. The machine needs to search for the first 0 and when (and if) it finds it, replace it with a 1. So let's have the machine stay in its initial state (s_0) as long as it reads 1's, constantly moving to the right. If it ever reads a 0 it will enter state s_1 while changing the 0 to a 1. No further action is required. Thus we can get by with just the following two five-tuples: $(s_0, 0, s_1, 1, R)$ and $(s_0, 1, s_0, 1, R)$. Note that if the input string consists of just 1's, then the machine eventually sees the terminating blank and halts.

9. The machine should scan the tape, leaving it alone until it has encountered the first 1. At that point, it needs to enter a phase in which it changes all the 1's to 0's, until it reaches the end of the input. So we'll have tuples $(s_0, 0, s_0, 0, R)$ and $(s_0, 1, s_1, 1, R)$ to complete the first phase, and then have tuples $(s_1, 0, s_1, 0, R)$ and $(s_1, 1, s_1, 0, R)$ to complete the second phase. When the machine encounters the end of the input (a blank on the tape) it halts, since there are no transitions given with a blank as the scanned symbol.

11. We can have the machine scan the input tape until it reaches the first blank, "remembering" what the last symbol was that it read. Let us use state s_0 to represent that last symbol's being a 1, and s_1 to represent its being a 0. It doesn't matter what gets written, so we'll just leave the tape unchanged as we move from left to right. Thus our first few five-tuples are $(s_0, 0, s_1, 0, R)$, $(s_0, 1, s_0, 1, R)$, $(s_1, 0, s_1, 0, R)$, $(s_1, 1, s_0, 1, R)$. Now suppose the machine encounters the end of the input, namely the blank at the end of the input string. If it is in state s_0, then the last symbol read was not a 0, so we want to not accept the string. If it is in state s_1, then the last symbol read was a 0, so we want to accept the string. Recall that the convention presented in this section was that acceptance is indicated by halting in a final state, i.e., one with no transitions out of it. So let's add the five-tuple (s_1, B, s_2, B, R) for accepting when we should. To make sure we don't accept when we shouldn't, we need do nothing else, because the machine will halt in the nonfinal state s_0 in this case.

An alternative approach to this problem is to have the machine scan to the right until it reaches the end of the tape, then back up, "look" at the last symbol, and take the appropriate action.

13. This is very similar to Exercise 11. We want the machine to "remember" whether it has seen an even number of 1's or not. We'll let s_0 be the state representing that an even number of 1's have been seen (which is of course true at the start of the computation), and let s_1 be the state representing that an odd number of 1's have been seen. So we put in the following tuples: $(s_0, 0, s_0, 0, R)$, $(s_0, 1, s_1, 1, R)$, $(s_1, 0, s_1, 0, R)$, and $(s_1, 1, s_0, 1, R)$. When the machine encounters the terminating blank, we want it to accept if it is in state s_0, so we add the tuple (s_0, B, s_2, B, R). Thus the machine will halt in final state s_2 if the input string has an even number of 0's, and it will halt in nonfinal state s_1 otherwise.

15. You need to play with this machine to get a feel for what is going on. After doing so, you will understand that it operates as follows. If the input string is blank or starts with a 1, then the machine halts in state s_0, which is not final, and therefore every such string is not accepted (which is a good thing, since it is not in the set to be recognized). Otherwise the initial 0 is changed to an M, and the machine skips past all the intervening 0's and 1's until it either comes to the end of the input string or else comes to an M (which, as we will see, has been written over the right-most remaining bit). At this point it backs up (moves left) one square and is in state s_2. Since the acceptable strings must have a 1 at the right for each 0 at the left, there had better be a 1 here if the string is acceptable. Therefore the only transition out of state s_2 occurs when this square contains a 1. If it does, then the machine replaces it with an M, and makes its way back to the left. (If this square does not contain a 1, then the machine halts in the nonfinal state s_2, as appropriate.) On its way back, it stays in state s_3 as long as it sees 1's, then stays in s_4 as long as it sees 0's. Eventually either it encounters a 1 while in state s_4, at which point it (appropriately) halts without accepting (since the string had a 0 to the right of a 1); or else it reaches the right-most M that had been written over a 0 near the beginning of the string. If it is in state s_3 when this happens, then there are no more 0's in the string, so it had better be the case (if we want to accept this string) that there are no more 1's either; this is accomplished by the transitions (s_3, M, s_5, M, R) and (s_5, M, s_6, M, R), and s_6 is a final state. Otherwise, the machine halts in nonfinal state s_5. If it is in state s_4 when this M is encountered, then we need to start all over again, except that now the string will have had its left-most remaining 0 and its right-most remaining 1 replaced by M's. So the machine moves (staying in state s_4) to the left-most remaining 0 and goes back into state s_0 to repeat the process.

17. This will be similar to the machine in Example 3, in that we will change the digits one at a time to a new symbol M. We can't work from the outside in as we did there, however, so we'll replace all three digits from left to right. Furthermore, we'll put a new symbol, E, at the left end of the input in order to tell more easily when we have arrived back at the starting point. Here is our plan for the states and the transitions that will accomplish our goal. State s_9 is our (accepting) final state. States s_0 and s_1 will write an E to the left of the initial input and return to the first input square, entering state s_2. (If, however, the tape is blank, then the machine will accept immediately, and if the first symbol is not a 0, then it will reject immediately.) The five-tuples are (s_0, B, s_9, B, L), $(s_0, 0, s_1, 0, L)$, and (s_1, B, s_2, E, R). State s_2 will skip past any M's until it finds the first 0, change it to an M, and enter state s_3. The transitions are (s_2, M, s_2, M, R) and $(s_2, 0, s_3, M, R)$. Similarly, state s_3 will skip past any remaining 0's and any M's until it finds the first 1, change it to an M, and enter state s_4. The transitions are $(s_3, 0, s_3, 0, R)$, (s_3, M, s_3, M, R) and $(s_3, 1, s_4, M, R)$. State s_4 will do the same for the first 2 (skipping past remaining 1's and M's, and ending in state s_5), with transitions $(s_4, 1, s_4, 1, R)$, (s_4, M, s_4, M, R) and $(s_4, 2, s_5, M, R)$. State s_5 then will skip over any remaining 2's and (if there is any chance of accepting this string) encounter the terminating blank. The transitions are $(s_5, 2, s_5, 2, R)$ and (s_5, B, s_6, B, L). Note that once this blank has been seen, we back up to the last symbol before it and enter state s_6. There are now two possibilities. If the scanned square is an M, then we should accept if and only if the entire string consists of M's at this point. We will enter state s_8 to check this, with the transition (s_6, M, s_8, M, L). Otherwise, there will be a 2 here, and we want to go back to the start of the string to begin the cycle all over; we'll use state s_7 to accomplish this, so we put in the five-tuple $(s_6, 2, s_7, 2, L)$. In this latter case, the machine should skip over everything until it sees the marker E that we put at the left end of the input, then move back to the initial input square, and start over in state s_2. The transitions $(s_7, 0, s_7, 0, L)$, $(s_7, 1, s_7, 1, L)$, $(s_7, 2, s_7, 2, L)$, (s_7, M, s_7, M, L), and (s_7, E, s_2, E, R) accomplish this. But if we entered state s_8, then we need to make sure that there is nothing but M's all the way back to the starting point; we add the five-tuples (s_8, M, s_8, M, L) and (s_8, E, s_9, E, L), and we're finished.

19. Recall that functions are computed in a funny way using unary notation. The string representing n is a string of $n + 1$ 1's. Thus we want our machine to erase three of these 1's (or all but one of them, if there are

fewer than four), and then halt. One way to accomplish this is as follows. If $n \geq 3$, then the five-tuples $(s_0, 1, s_1, B, R)$, $(s_1, 1, s_2, B, R)$, $(s_2, 1, s_3, B, R)$, and $(s_3, 1, s_4, 1, R)$ will do the trick (s_4 is just a halting state). To account for the possibilities that $n < 3$, we add transitions $(s_1, B, s_4, 1, R)$, $(s_2, B, s_4, 1, R)$, and $(s_3, B, s_4, 1, R)$. In each of these three cases, we needed to restore one 1 before halting (since the "answer" was to be 0).

21. The machine here first needs to "decide" whether $n \geq 5$. If it finds that $n \geq 5$, then it needs to leave exactly four 1's on the tape (according to our rules for representing numbers in unary); otherwise it needs to leave exactly one 1. We'll use states s_0 through s_6 for this task, with the following five-tuples, which erase the tape as they move from left to right through the input: $(s_0, 1, s_1, B, R)$, $(s_1, 1, s_2, B, R)$, (s_1, B, s_6, B, R), $(s_2, 1, s_3, B, R)$, (s_2, B, s_6, B, R), $(s_3, 1, s_4, B, R)$, (s_3, B, s_6, B, R), $(s_4, 1, s_5, B, R)$, (s_4, B, s_6, B, R). At this point, the machine is either in state s_5 (and $n \geq 5$), or in state s_6 with a blank tape (and $n < 5$). To finish in the latter case, we just write a 1 and halt: $(s_6, B, s_{10}, 1, R)$. For the former case, we erase the rest of the tape, write four 1's, and halt: $(s_5, 1, s_5, B, R)$, $(s_5, B, s_7, 1, R)$, $(s_7, B, s_8, 1, R)$, $(s_8, B, s_9, 1, R)$, and $(s_9, B, s_{10}, 1, R)$.

23. We start with a string of $n + 1$ 1's, and we want to end up with a string of $3n + 1$ 1's. Our idea will be to replace the last 1 with a 0, then for each 1 to the left of the 0, write a pair of new 1's to the right of the 0. To keep track of which 1's we have processed so far, we will change each left-side 1 to a 0 as we process it. At the end, we will change all the 0's back to 1's. Basically our states will mean the following ("first" means "first encountered"): s_0, scan right for last 1; s_1, change the last 1 to 0; s_2, scan left to first 1; s_3, scan right for end of input (having replaced the 1 where we started with a 0); s_3 and s_4, write the two more 1's; s_5, scan left to first 0; s_6, replace the remaining 0's with 1's; s_7, halt.

The needed five-tuples are as follows: $(s_0, 1, s_0, 1, R)$, (s_0, B, s_1, B, L), $(s_1, 1, s_2, 0, L)$, $(s_2, 0, s_2, 0, L)$, $(s_2, 1, s_3, 0, R)$, (s_2, B, s_6, B, R), $(s_3, 0, s_3, 0, R)$, $(s_3, 1, s_3, 1, R)$, $(s_3, B, s_4, 1, R)$, $(s_4, B, s_5, 1, L)$, $(s_5, 1, s_5, 1, L)$, $(s_5, 0, s_2, 0, L)$, $(s_6, 0, s_6, 1, R)$, $(s_6, 1, s_7, 1, R)$, (s_6, B, s_7, B, R).

25. The idea here is to match off the 1's in the two inputs (changing the 1's to 0's from the left, say, to keep track), until one of them is exhausted. At that point, we need to erase the larger input entirely (as well as the asterisk) and change the 0's back to 1's. Here is how we'll do it. In state s_0 we skip over any 0's until we come to either a 1 or the $*$. If it's the $*$, then we know that the second input (n_2) is at least as large as the first (n_1), so we enter a clean-up state s_5, which erases the asterisk and all the 0's and 1's to its right. The five-tuples for this much are $(s_0, 0, s_0, 0, R)$, $(s_0, *, s_5, B, R)$, $(s_5, 1, s_5, B, R)$, and $(s_5, 0, s_5, B, R)$. Once this erasing is finished, we need to go over to the part of the tape where the first input was and change all the 0's back to 1's; the following transitions accomplish this: (s_5, B, s_6, B, L), (s_6, B, s_6, B, L), $(s_6, 0, s_7, 1, L)$, and $(s_7, 0, s_7, 1, L)$. Eventually the machine halts in state s_7 when the blank preceding the original input is encountered.

The other possibility is that the machine encounters a 1 while in state s_0. We want to change this 1 to a 0, skip over any remaining 1's as well as the asterisk, skip over any 0's to the right of the asterisk (these represent parts of n_2 that have already been matched off against equal parts of n_1), and then either find a 1 in n_2 (which we change to a 0) or else come to the blank at the end of the input. Here are the transitions: $(s_0, 1, s_1, 0, R)$, $(s_1, 1, s_1, 1, R)$, $(s_1, *, s_2, *, R)$, $(s_2, 0, s_2, 0, R)$, $(s_2, 1, s_3, 0, L)$, (s_2, B, s_4, B, L). At this point we are either in state s_3, ready to go back for the next iteration, or in state s_4 ready for some cleanup. In the former case, we want to skip back over the nonblank symbols until we reach the start of the string, so we add five-tuples $(s_3, *, s_3, *, L)$, $(s_3, 0, s_3, 0, L)$, $(s_3, 1, s_3, 1, L)$, and (s_3, B, s_0, B, R). In the latter case, we know that the first string is longer than the second. Therefore we want to change the 0's in the second input string back to 1's and then erase the asterisk and remnants of the first input string. Here are the transitions: $(s_4, 0, s_4, 1, L)$, $(s_4, *, s_8, B, L)$, $(s_8, 0, s_8, B, L)$, $(s_8, 1, s_8, B, L)$.

27. The discussion in the preamble tells how to take the machines from Exercises 22 and 18 and create a new machine. The only catch is that the tape head needs to be back at the leftmost 1. Suppose that s_m, where m is the largest index, is the state in which the Turing machine for Exercise 22 halts after completing its work, and suppose that we have designed that machine so that when the machine halts the tape head is reading the leftmost 1 of the answer. Then we renumber each state in the machine for Exercise 18 by adding m to each subscript, and take the union of the two sets of five-tuples.

29. If the answer is yes/no, then the problem is a decision problem.

a) No, the answer here is a number, not yes or no.

b) Yes, the answer is either yes or no.

c) Yes, the answer is either yes or no.

d) Yes, the answer is either yes or no.

31. This is a fairly hard problem, which can be solved by patiently trying various combinations. The following five-tuples will do the trick: $(s_0, B, s_1, 1, L)$, $(s_0, 1, s_1, 1, R)$, $(s_1, B, s_0, 1, R)$.

GUIDE TO REVIEW QUESTIONS FOR CHAPTER 13

1. **a)** See p. 849. **b)** See p. 849.

2. **a)** See p. 850. **b)** $\{0^{3n}1 \mid n \geq 0\}$

c) The vocabulary is $\{S, 0, 1\}$; the terminals are $T = \{0, 1\}$; the start symbol is S; and the productions are $S \to S1$ and $S \to 0$.

3. **a)** See p. 851. **b)** a grammar that contains a production like $AB \to C$

c) See p. 851. **d)** a grammar that contains a production like $Sa \to Sbc$

e) See p. 851. **f)** a grammar that contains a production like $S \to SS$

4. **a)** See p. 851. **b)** See p. 851.

c) See Example 8 in Section 13.1.

5. **a)** See p. 854. **b)** See Example 14 in Section 13.1.

6. **a)** See p. 851 (machines with output) and p. 867 (machines without output, called finite-state automata). See also p. 863 for comments on other types of finite-state machines.

b) Have three states and only one input symbol, Q (quarter). The start state s_0 has a transition to state s_1 on input Q and outputs nothing; state s_1 has a transition to state s_2 on input Q and outputs nothing; state s_2 has a transition back to state s_1 on input Q and outputs a drink.

7. $1^* \cup 1^*00$

8. Have four states, with only s_2 final. From the start state s_0, go to a graveyard state s_1 on input 0, and go to state s_2 on input 1. From both states s_2 and s_3, go to s_2 on input 1 and to s_3 on input 0.

9. **a)** See p. 866.

b) the set of all strings in which all the maximal blocks of consecutive 1's (if any) have an even number of 1's

10. **a)** See p. 867. **b)** See p. 868.

11. **a)** See p. 873. **b)** See Theorem 1 in Section 13.3.

12. **a)** See p. 879. **b)** See p. 879.

13. See Theorem 1 in Section 13.4.

14. See the proof of Theorem 2 in Section 13.4.

15. See Example 6 in Section 13.4.

16. See p. 889.

17. See p. 891.

18. See p. 892.

19. See p. 895. The halting problem is unsolvable; see p. 895.

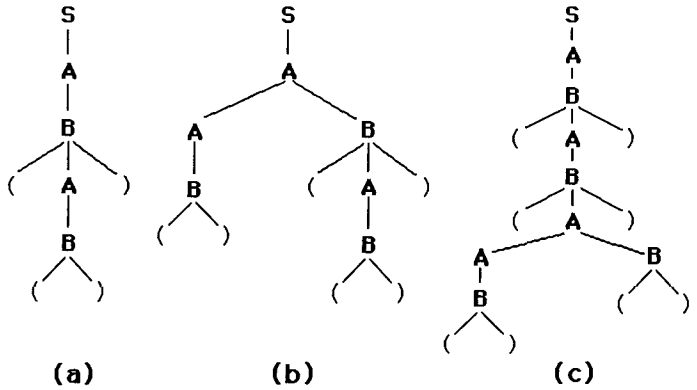
SUPPLEMENTARY EXERCISES FOR CHAPTER 13

1. **a)** We simply need to add two 0's on the left and three 1's on the right at the same time. Thus the rules can be $S \to 00S111$ and $S \to \lambda$.

b) We need to add two 0's for every 1 and also allow the symbols to change places at will. Following the trick in our solution to Exercise 15c in Section 13.1, we let A and B be nonterminal symbols representing 0 and 1, respectively. Our rules are $S \to AABS$, $AB \to BA$, $BA \to AB$, $A \to 0$, $B \to 1$, and $S \to \lambda$.

c) Our trick here is first to generate a string that looks like $Ew(w^R)$, with A in the place of 0, and B in the place of 1, in the second half. The rules $S \to ET$, $T \to 0TA$, $T \to 1TB$, and $T \to \lambda$ will accomplish this much. Then we force the A's and B's to march to the left, across all the 0's and 1's, until they bump into the left-hand wall (E), at which point they turn into their terminal counterparts. Finally, the wall disappears. The rules for doing this are $0A \to A0$, $1A \to A1$, $0B \to B0$, $1B \to B1$, $EA \to E0$, $EB \to E1$, and $E \to \lambda$.

3. For part **(a)** note that (()) can come from (B), which in turn can come from (A), which can come from B, and we can start $S \Rightarrow A \Rightarrow B$. Thus the tree can be as shown in the first picture. For part **(b)** we need to use the rule $A \to AB$ early in the derivation, with the A turning into (), and the B turning into (()). The ideas in part **(c)** are similar.



5. The idea is that the rules enable us to add 0's to either the right or the left. Thus we can get three 0's in many ways, depending on which side we add the 0's on.

7. It is not true that $|AB|$ is always equal to $|A| \cdot |B|$, since a string in AB may be formed in more than one way. After a little experimentation, we might come up with the following example to show that $|AB|$ need not equal $|BA|$ and that $|AB|$ need not equal $|A| \cdot |B|$. Let $A = \{0, 00\}$, and let $B = \{01, 1\}$. Then $AB = \{01, 001, 0001\}$ (there are only 3 elements, not $2 \cdot 2 = 4$, since 001 can be formed in two ways), whereas $BA = \{010, 0100, 10, 100\}$ has 4 elements.

9. This is clearly not necessarily true. For example, we could take $A = V^*$ and $B = V$. Then A^* is again V^*, so it is true that $A^* \subseteq B^*$, but of course $A \not\subseteq B$ (for one thing, A is infinite and B is finite).

11. In each case we apply the definition to rewrite $h(\mathbf{E})$ in terms of h applied to the subexpressions of $\mathbf{E}$.

a) $h(\mathbf{0^*1}) = \max(h(\mathbf{0^*}), h(\mathbf{1})) = \max(h(\mathbf{0}) + 1, 0) = \max(0 + 1, 0) = 1$

b) $h(\mathbf{0^*1^*}) = \max(h(\mathbf{0^*}), h(\mathbf{1^*})) = \max(h(\mathbf{0}) + 1, h(\mathbf{1}) + 1) = \max(0 + 1, 0 + 1) = 1$

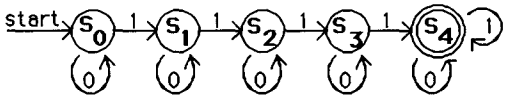
c) $h((\mathbf{0^*01})^*) = h(\mathbf{0^*01}) + 1 = 1 + 1 = 2$

d) This is similar to part **(c)**; the answer is 3.

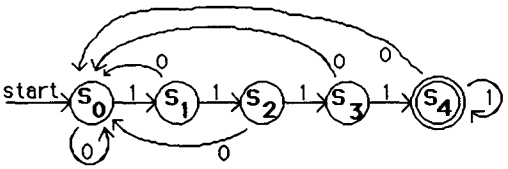
e) There are three "factors," and by the definition we need to find the maximum value that h takes on them. It is easy to compute that these values are 1, 2, and 2, respectively, so the answer is 2.

f) A calculation similar to that in part **(c)** shows that the answer is 4.

13. We need to have states to represent the number of 1's read in so far. Thus s_i, for $i = 0, 1, 2, 3$, will "mean" that we have seen exactly i 1's so far, and s_4 will signify that we have seen at least four 1's. We draw only the finite-state automaton; the machine with output is exactly the same, except that instead of a state being designated final, there is an output for each transition; all the outputs are 0 except for the outputs to our final state, and all of the outputs to this final state are 1.



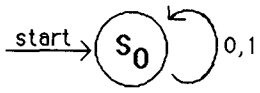
15. This is similar to Exercise 13, except that we need to return to the starting state whenever we encounter a 0, rather than merely remaining in the same state. As in Exercise 13, we draw only the automaton, since the machine with output is practically the same.



17. **a)** To specify a machine, we need to pick a start state (this can be done in n ways), and for each pair (state, input) (and there are nk such pairs), we need to choose a state and an output. By the product rule, therefore, the answer is $n \cdot n^{nk} \cdot m^{nk}$. (We are answering the question as it was asked. A much harder question is to determine how many "really" different machines there are, since two machines that really do the same thing and just have different names on the states should perhaps be considered the same. We will not pursue this question.)

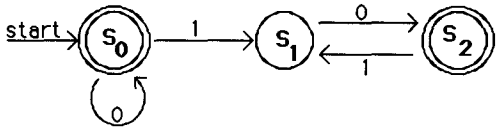
b) This is just like part **(a)**, except that we do not need to choose an output for each transition, only an output for each state. Thus the term m^{nk} needs to be replaced by m^n, and the answer is $n \cdot n^{nk} \cdot m^n$.

19. This machine has no final states. Therefore no strings are accepted. Any deterministic machine with no final states will be equivalent to this one. We show one such machine below.

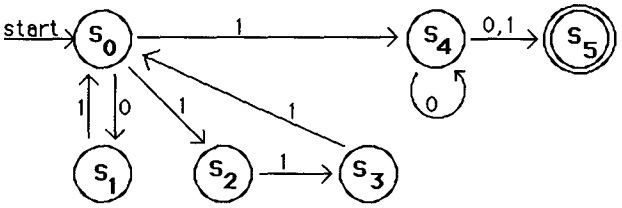


21. The answers are not unique, of course. There are two ways to approach this exercise. We could simply apply the algorithm inherent in the proof of Kleene's theorem, but that would lead to machines much more complicated than they need to be. Alternately, we just try to be clever and make the machines "do what the expressions say." This is essentially computer programming, and it takes experience to be able to do it well. In part **(a)**, for example, we want to accept every string of 0's, so we make the start state a final state, with returns to this state on input 0; and we want to accept every string that has this beginning and then consists of any number of copies of 10—which is precisely what the rest of our machine does. These pictures can either be viewed as nondeterministic machines, or else for all the missing transitions we assume a transition to a new state (the graveyard), which is not final and which has transitions to itself on both inputs. Also, as usual, having two labels on an edge is an abbreviation for two edges, one with each label.

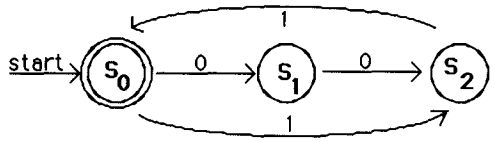
a) This one is pretty simple. State s_0 represents the condition that only 0's have been read so far; it is final. After we have read in as many 0's as desired, we still want to accept the string if we read in any number of copies of 10. This is accomplished with the other two states.



b) In this machine, we keep returning to s_0 as long as we are reading 01 or 111, corresponding to the first factor in our regular expression. Then we move to s_4 for the term $\mathbf{10^*}$, and finally to s_5 for the factor $(\mathbf{0 \cup 1})$.



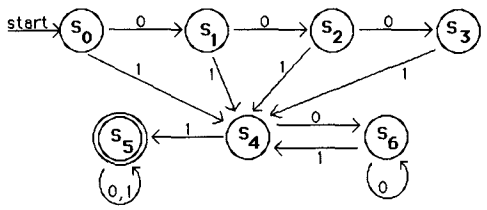
c) Note that the inner star in this regular expression is irrelevant; we get the same set whether it is there or not. Our machine returns us to s_0 after we have read either 001 or 11, so we accept every string consisting of any number of copies of these strings.



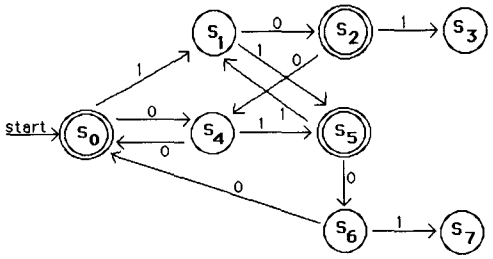
23. We invoke the power of Kleene's theorem here. If A is a regular set, then there is a deterministic finite automaton that accepts A. If we take the same machine but make all the final states nonfinal and all the nonfinal states final, then the result will accept precisely $\overline{A}$. Therefore $\overline{A}$ is regular.

25. See the comments for Exercise 21. Here the problem is even harder, since we are given just verbal descriptions of the sets. Thus there *is* no general algorithm we can invoke. We just have to be clever programmers. See the comments on the solution to Exercise 21 for how to interpret missing transitions.

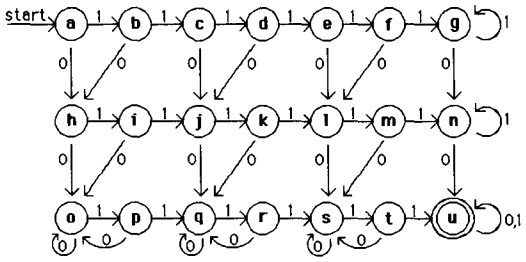
a) The top part of our machine (as drawn) takes us to a graveyard if there are more than three consecutive 0's at the beginning. The rest assures that there are at least two consecutive 1's.



b) This one is rather complicated. The states represent what has been seen recently in the input. For example, states s_2 and s_6 represent the condition in which the last two symbols have been 10. Thus if we encounter a 1 from either of these states, we move to a graveyard. (Note that we could have combined states s_3 and s_7 into one, or, under our conventions, we could have omitted them altogether; the answers to these exercises are by no means unique.) States s_0, s_2 and s_5 all represent conditions in which an even number of symbols have been read in, whereas s_1, s_4 and s_6 represent conditions in which an odd number of symbols have been read.



c) This one is really not as bad as it looks. The first row in our machine (as drawn) represents conditions before any 0's have been read; the second row after one 0, and the third row after two or more 0's. The horizontal direction takes care of looking for the blocks of 1's.



27. Suppose that $\{ 1^p \mid p \text{ is prime} \}$ is regular. Then by the pumping lemma, we can find a prime p such that $1^p = uvw$, with $l(v) \geq 1$, so that uv^iw is a string of a prime number of 1's for all i. If we let a be the number of 1's in uw and $b > 0$ the number of 1's in v, then this means that $a + bi$ is prime for all i. In other words, the gap between two consecutive primes (once we are looking at numbers greater than a) is at most b. This contradicts reality, however, since for every n, all the numbers from $n! + 2$ through $n! + n$ are not prime—in other words the gaps between primes can be arbitrarily large.

29. The idea here is to match off the 1's in the two inputs (changing the 1's to 0's from the left, say, to keep track), until one of them is exhausted. At that point, we need to erase the smaller input entirely (as well as the asterisk) and change the 0's back to 1's. Here is how we'll do it. In state s_0 we skip over any 0's until we come to either a 1 or the $*$. If it's the $*$, then we know that the second input (n_2) is at least as large as the first (n_1), so we enter a clean-up state s_5, which erases the asterisk and all the 0's and 1's to its left. The five-tuples for this much are $(s_0, 0, s_0, 0, R)$, $(s_0, *, s_5, B, L)$, and $(s_5, 0, s_5, B, L)$. Once this erasing is finished, we need to go over to the part of the tape where the second input was and change all the 0's back to 1's; the following transitions accomplish this: (s_5, B, s_6, B, R), (s_6, B, s_6, B, R), $(s_6, 0, s_7, 1, R)$, $(s_7, 0, s_7, 1, R)$,

and $(s_7, 1, s_7, 1, R)$. Eventually the machine halts in state s_7 when the blank following the original input is encountered.

The other possibility is that the machine encounters a 1 while in state s_0. We want to change this 1 to a 0, skip over any remaining 1's as well as the asterisk, skip over any 0's to the right of the asterisk (these represent parts of n_2 that have already been matched off against equal parts of n_1), and then either find a 1 in n_2 (which we change to a 0) or else come to the blank at the end of the input. Here are the transitions: $(s_0, 1, s_1, 0, R)$, $(s_1, 1, s_1, 1, R)$, $(s_1, *, s_2, *, R)$, $(s_2, 0, s_2, 0, R)$, $(s_2, 1, s_3, 0, L)$, (s_2, B, s_4, B, L). At this point we are either in state s_3, ready to go back for the next iteration, or in state s_4 ready for some cleanup. In the former case, we want to skip back over the nonblank symbols until we reach the start of the string, so we add five-tuples $(s_3, *, s_3, *, L)$, $(s_3, 0, s_3, 0, L)$, $(s_3, 1, s_3, 1, L)$, and (s_3, B, s_0, B, R). In the latter case, we know that the first string is longer than the second. Therefore we want to erase remnants of the second input string and the asterisk, and change the 0's in the first input string back to 1's. Here are the transitions: $(s_4, 0, s_4, B, L)$, $(s_4, *, s_8, B, L)$, $(s_8, 0, s_8, 1, L)$, $(s_8, 1, s_8, 1, L)$. The machine halts in state s_8 when the blank preceding the original input is encountered.

WRITING PROJECTS FOR CHAPTER 13

Books and articles indicated by bracketed symbols below are listed near the end of this manual. You should also read the general comments and advice you will find there about researching and writing these essays.

1. See the chapter on generative grammars in [De2]. Lindenmeyer systems are a special kind of generative grammar.

2. Most textbooks on programming languages should discuss this, as well as books on the specific languages mentioned. The call number for programming languages is QA 76.7. As usual, you can find websites with this information using a search engine; for example, search for the three keywords *backus*, *naur*, and *java*.

3. See [BeKa], for example.

4. One book on network protocols that discusses finite state machines is [Ho3].

5. Mehryar Mohri of the Department of Computer Science at the Courant Institute of Mathematical Sciences is an expert in this area. See his Web page.

6. The Wikipedia article on finite-state machines is a good place to start.

7. There are several textbooks on automata theory and finite-state machines of all kinds that cover topics such as this. Try [Br1], [Co], [DeDe], [HoUl], or [LePa], for example. These are also good sources for supplementing the material in this chapter. This subject in general (including Turing machines, computability, and computational complexity), is usually called "the theory of computation," and, again, there are numerous books with essentially this title; see [Si] for a fairly recent and readable one that takes you from the beginning to a fairly advanced level.

8. Again, there are entire books on this subject (see [PrDu], for instance). The Game of Life was invented by the British mathematician John H. Conway, and was the subject of several articles in Martin Gardner's *Scientific American* column during the 1970s. Three of them are collected in [Ga2], which also mentions other books and articles. See also [BeCo], which covers lots of solitaire and two-person games, as well as Life.

9. See standard references on automata theory, such as those mentioned in Writing Project 7.

10. See standard references on automata theory, such as those mentioned in Writing Project 7.

11. Turing's first article is [Tu2], and it is actually quite readable. Keep in mind as you read it that real computers had not yet been invented.

12. See standard references on automata theory, such as those mentioned in Writing Project 7.

13. This actually opens up the door to most of the important modern-day research in theoretical computer science. For an elementary, nontechnical account, see [Gr2]. See the references given in Writing Project 4 for more detail. The big question is whether deterministic Turing machines can compute functions as efficiently as nondeterministic ones. You should definitely consult [GaJo], the classic work in this area.

14. See the references given for Writing Project 13.

15. See standard references on automata theory, such as those mentioned in Writing Project 7.

16. Searching under "lambda calculus" or "recursive function theory" in your library should turn up a place to start.

17. See standard references on automata theory, such as those mentioned in Writing Project 7.

18. See standard references on automata theory, such as those mentioned in Writing Project 7.