Problems from the Textbook

Chapter 1 PROBLEMS

1. How long is the shortest possible Java program?
2. What is the advantage of organizing a compiler into a sequence of passes?
3. Describe in words the set {b, c}*.
4. What does the set {}* contain?
5. Is it true that $x^* = \{x\}^*$ for any string x?
6. Under what circumstances are P and $\sim P$ mutually exclusive?
7. What does $P \mid Q = P$ imply?
8. What does $P \cap Q = P$ imply?
9. If P = {b} and Q = {bb, c}, then what does $P^* \cap Q^*$ equal?
10. If $A = \{\lambda$, b}, how many distinct strings are in AA? List them.
11. Is x^* always an infinite set? If not, give an example for which it is not infinite.
12. Does b*c* = {b, c}*? Justify your answer.
13. Represent the set ϕ|{λ}|bbbc(bbbc)* with a regular expression that does not use the | operator.
14. Using exponent notation, represent the set b*c*b*.
15. Write a regular expression for the set of all strings over the alphabet {b, c} containing exactly one b.
16. Write a regular expression for the set of all strings over the alphabet {b, c} containing at least one b.
17. Write an expression using exponent notation for the set $(\mathrm{b}^i\mathrm{c}^i\mathrm{d}^j : i \geq 0, j \geq 0\ \} \cap \{\mathrm{b}^p\mathrm{c}^q\mathrm{d}^q : p \geq 0$ and $q \geq 0\}$ without using the $\cap$ operator.
18. Is (b*c*)* = {b, c}*? If not, provide a counterexample.
19. List all the strings in {b, cc}* that are of length 3.
20. Does (b*|b*ccc)* = {b, ccc}*? Justify your answer.

21. Is concatenation distributive over union. That is, for all sets of strings A, *B*, C, does *A*(*B* | *C*) = *AB* | AC?

22. Is the star operation distributive over union. That is, for all sets of strings *A*, *B*, does (*A*|*B*)* = *A**|*B**?

23. Suppose *X*, *A*, and *B* are sets of strings, $\lambda \notin B$, and *X* = *A*|*XB*, what can be concluded about *X*? Hint: does *X* contain *AB*?

24. Does *xA* always equal {*x*}*A* where *x* is a string and *A* is a set of strings?

25. The parser in a compiler does not need the tokens corresponding to white space and comments. Yet, syntax errors may occur if white space and comments are removed from the source program. Explain this apparent contradiction.

26. Write a regular expression that defines the same language as b*c*∩c*d*.

27. Write a regular expression that defines the same language as {b, cc}* ∩ c*.

28. Write a regular expression that defines the same language as (bb)* ∩ (bbb)*.

29. Write a regular expression for the set of all strings over the alphabet {b, c} containing an even number of b's.

30. Write a non-extended regular expression that defines the same language as (~b)*, where the universe is (b|c|d)*.

31. Write a non-extended regular expression that defines the same language as ~({ b, c }*), where the universe is (b|c|d)*.

32. Prove that any finite language is regular.

33. Describe in English the strings in (bb|cc|((bc|cb)(bb|cc)*(bb|cc)))*

34. Is (((b))) a regular expression over the alphabet {b, c}?

35. Is () a regular expression over the alphabet {b, c}?

36. Give three regular expressions that define the empty set.

37. Suppose the alphabet for regular expressions consists of the symbols b, c, the backslash, the vertical bar, the single quote, and the double quote. Give an unambiguous regular expression that specifies the set consisting of b, c, the backslash, the vertical bar, the single quote, and the double quote.

38. Convert (b|c?)+ to an equivalent non-extended regular expression.

39. Show that extended regular expressions are not more powerful than regular expressions. That is, show that any language that can be defined by an extended regular expression can also be defined by a non-extended regular expression.

Chapter 2 Problems

1. Write a grammar that generates the language {}.

2. Write a grammar that generates the language {λ}.

3. Write a grammar that generates `bbb*cc*` that uses directlyleft recursive productions.

4. Write a regular grammar that generates `bbb*c*`.

5. Write a regular grammar that generates `b*|c*|d*`.

6. Write a grammar that generates the language consisting of all strings of `b`'s in which successive `b`'s are separated by at least one comma. Successive commas are allowed. For example, `b,b,,,b,b,,b`.

7. Write a regular grammar equivalent to

```
1) S → Sb
2) S → c
3) S → d
4) S → e
```

8. Write a regular grammar equivalent to

```
1) S → bcdS
2) S → cbaS
3) S → bbA
4) A → bbc
```

9. Write a grammar which generates a list in which each element is either `b`, `c`, or `d`, and in which successive elements are separated by exactly one comma. Your grammar should also generate the null string. Sample strings:

```
λ
b
b,b,b,c,d,b
```

10. Write a grammar which generates $\{b^{2i}c^{i} : i \geq 0\}$.

11. Given arbitrary grammars G1 and G2, show how to construct grammars G3 and G4 such that L(G3) = L(G1)|L(G2) and L(G4) = L(G1)L(G2).

12. Describe the language generated by

```
1) S → SS
2) S → bSc
3) S → cSb
```

```
4) S → λ
```

13. Write a regular grammar that generates (bcd)*eee.

14. Write a right linear grammar that generates (bcd)*eee.

15. Prove that a finite language is always regular.

16. Convert the following grammar to a right linear grammar:

```
1) S → Sbcd
2) S → fg
```

17. Write a grammar that generates $\{b^i c^{i+j} d^j : i \geq 1, j \geq 2\}$.

18. Describe the language generated by

```
1) S → bB
2) B → cC
3) C → dS
4) C → d
```

19. How many distinct derivations of bcde are possible using

```
1) S → BCDE
2) B → b
3) C → c
4) D → d
5) E → e
```

20. Write a grammar that generates all properly nested parenthesized strings. Some examples of strings in the language are

```
()
()()()
(()())
((()))()()(()())
```

21. Write a grammar that generates $\{b^i c^j : i > 2, j \geq 3\}$.

22. How many times (in terms of *i*, *j*, and *k*) and in what order must the productions below be used to generate the string $b^i c^j d^k$?

```
1) S → bSd
2) S → bA
3) A → bAc
4) A → c
```

23. Write a grammar that generates (`bc`*)+.

24. Prove that any grammar that generates the language {$b^i c^i$: $i \geq 0$ } cannot contain the production `S → bS`, where `S` is the start symbol.

25. Write a regular grammar that generates {$b^i c^i$: $i \leq 4$ } by counting.

26. Write a regular grammar equivalent to G2.36.

27. Let L denote the language defined by the following grammar:

```
1) S → bcS
2) S → d
```

Modify this grammar so that it generates the language L | {λ}.

28. Give a summary of the proof that the language *PAIRED* in section 2.7 is not regular (see Section 17.12).

29. Give a regular expression for the language defined by G2.30 to which `S → λ` has been added.

30. Give a summary of the proof that the language *TRIPLED* in Section 2.7 is not context-free (see Section 4.10).

31. Prove that the language consisting of arbitrarily long non-repeating lists is not context-free.

32. Convert the following grammar to a regular grammar:

```
1)S → bcdeS
2)S → edcb
```

33. If you make no assumptions about the terminal alphabet for the grammar below, can you determine how many strings are in the language it generates?

```
S → b | c
```

If you know the terminal alphabet is {`b`, `c`}, how many strings are in the language?

34. Give a regular expression that defines the same language as the following grammar:

```
1)b → "b" b
2)b → "b"
```

35. Is production 1 in G2.32 necessary?

Chapter 3 PROBLEMS

1. How many different parse trees does `bbbbb` have in grammar G3.7?

2. How many different words can be generated by the following grammar? Show the parse tree for each word.

```
 1) S → CBADE
 2) A → N
 3) B → R
 4) C → E
 5) D → I
 6) E → b
 7) E → λ
 8) I → t
 9) N → r
10) R → e
```

3. Convert the following grammar to an equivalent grammar that contains no left recursion:

```
1) S → Sb
2) S → Sc
3) S → Sd
4) S → b
5) S → c
6) S → d
```

4. Convert the following grammar to an equivalent unambiguous grammar:

```
1) S → SbS
2) S → e
```

5. Using a "one-way street" production, convert the following grammar to an equivalent unambiguous grammar:

```
1) S → bS
2) S → Sbe
3) S → d
```

6. Show that the deletion of all productions in a grammar containing unreachable nonterminals never creates new dead nonterminals.

7. Convert the following grammar to an equivalent grammar that contains no left recursion:

```
1) S → Ad
2) S → d
3) A → Bb
4) A → b
5) B → Sc
6) B → c
```

8. Convert the following grammar to an equivalent grammar with no lambda productions:

```
1) S → BD
2) B → bBBBDd
3) B → λ
4) D → e
```

9. Convert the following grammar to an equivalent grammar that contains no left recursion:

```
1) S → Af
2) S → fA
3) A → Bb
4) A → bB
5) B → Cc
6) B → cC
7) C → dS
8) C → Sd
9) C → e
```

10. Convert G3.8 to an equivalent grammar with no unit productions.

11. Convert the following grammar to an equivalent grammar that has no lambda productions:

```
1) S → BcD
2) B → BB
3) B → b
4) B → λ
5) D → DdD
6) D → dD
7) D → λ
```

12. Can a string have more than one leftmost derivation in a grammar but only one rightmost derivation? Justify your answer.

13. Eliminate the useless nonterminals in the following grammar:

```
1) S → AB
2) A → CD
3) C → CC
4) C → λ
5) D → DdD
6) D → eE
7) E → ee
```

14. Does the following procedure correctly eliminate unit productions:

For every pair of nonterminals `A` and `Z` such that `Z` can be derived from `A` using only unit productions, add the production `A → x` for every non-unit production `Z → x`. After adding all such productions, delete all unit productions.

Note that this procedure never creates new unit productions.

15. Another technique for eliminating unit productions can be based on the following observation: Suppose a grammar contains the production `A → B`, which replaces `A` with `B`. Instead of generating `A` and then replacing it with `B`, why not generate `B` directly. Then the unit production would not be needed. Eliminate the unit productions in G3.20 using this technique. Does the technique ever fail?

16. Find an algorithm that for every pair of nonterminals `A`, `B` in a context-free grammar determines if

$$A \overset{+}{=>} B$$

How could such an algorithm be used in eliminating unit productions from a grammar?

17. Convert the following grammar to an equivalent grammar that has only five productions:

```
1) S → d
2) S → bB
3) S → Be
4) S → cc
5) B → bB
6) B → Be
7) B → cc
```

18. Convert the following grammar to an equivalent grammar that has only seven productions:

```
1)   S → BCD
2)   S → BC
3)   S → BD
4)   S → CD
5)   S → B
6)   S → C
7)   S → D
8)   S → λ
9)   B → bBBb
10)  B → bBb
11)  B → bb
12)  C → cC
13)  C → c
14)  D → Dd
15)  D → d
```

19. Is the following grammar ambiguous?

```
1) S → bbbS
```

```
2) S → Sa
3) S → d
```

20. Convert the following grammar to an equivalent unambiguous grammar:

```
1) S → bSd
2) S → bS
3) S → Sd
4) S → c
```

21. Prove that any context-free grammar can be converted to an equivalent grammar that has at most one lambda production.

22. Convert the following grammar to an equivalent grammar that contains no left recursion. Process productions both in the order given and in the reverse order.

```
1) S → Ag
2) S → g
3) A → Sb
4) A → Bd
5) A → b
6) B → Sd
7) B → Ae
8) B → f
```

23. When eliminating left recursion, in what order should the groups of productions be processed to minimize the total number of productions in the final grammar?

24. Write a program that determines the useless nonterminals in a context-free grammar.

25. Write an ambiguous grammar that defines

$$\{a^i b^i c^j : i, j \geq 0\} \mid \{a^p b^q c^q : p, q \geq 0\}$$

Try writing an unambiguous grammar for the language. Warning: this language is an inherently ambiguous context-free language. That is, no unambiguous context-free grammar exists for it.

26. Consider the procedures described in Sections 3.7 and 3.8 for eliminating lambda and unit productions. Is it possible that adding productions to make one production unnecessary may make another production previously made unnecessary once again necessary? Justify your answer.

27. Is the following grammar ambiguous:

```
1) S → SS
2) S → dA
3) A → bB
4) B → cS
```

28. Can we use the procedure in Section 3.7 for eliminating lambda productions to determine the nullable nonterminals in a grammar?

29. Convert the following grammar to an equivalent grammar that has no unit productions:

```
1) S → Bd
2) B → C
3) B → b
4) C → D
5) C → c
6) D → E
7) D → d
8) E → e
```

Do this conversion two ways: Use the approach described in Problem 3.14, and use the approach described in Section 3.8.

30. Convert the following grammar to an equivalent grammar that is not ambiguous:

```
1) S → bS
2) S → cS
3) S → Sd
4) S → Se
5) S → f
```

Chapter 4 Problems

1. Add the operations subtraction and division to grammars G4.1, G4.2, G4.3, and G4.4.
2. Draw the parse tree for `b+(((b+c)))` using grammar G4.4.
3. Convert G4.3 to a grammar that implies a right-to-left evaluation of like operations.
4. Write a grammar that generates all the strings over the alphabet { b, c} in which the number of b's equals the number of c's. (Hint: You need only four productions.)
5. Draw the parse tree of `b*c*d+c*d+d` using G4.3, G4.4, and G4.5.
6. Write a grammar that generates arithmetic expresions in postfix notation (i.e., the operator appears after the two operands). Use left recursion. Parentheses are never needed in postfix notation. Why? How are operator precedence and associativity handled in postfix notation?
7. Rewrite G4.1 in BNF and extended BNF.
8. Rewrite G4.2 in BNF and extended BNF.
9. Represent G4.5 using syntax diagrams.
10. Represent G4.10 using syntax diagrams.
11. Draw the parse tree for `b+c*d^e` using G4.5.
12. Convert the following grammar to an equivalent essentially non-contracting grammar:

```
1) S → BCD
2) S → bS
3) B → bBB
4) B → λ
5) C → c
6) D → D
7) D → λ
```

13. Convert the following grammar to an equivalent essentially non-contracting grammar:

```
1) S → A
2) A → B
3) B → C
4) C → λ
5) C → c
```

14. Suppose G is a grammar that does not generate the null string. Does adding `S → λ` to G (where S is the start symbol) add the null string and only the null string to L(G)?. Explain.

15. Prove that the language $\{b^i c^i d^i e^i : i \geq 0\}$ is not context-free.

16. Prove that the set of all strings over the alphabet {b, c, d} in which the number of b's is equal to the number of c's, and the number of c's is equal to the number of d's is not context-free. Note that the b's, c's, and d's can appear in any order in this language.

17. Prove that the language of non-repeating lists in section 2.11 is not context-free.

18. Prove that the language $\{ww : w \in (b \mid c)^*\}$ is not context-free.

19. Prove that the language $\{b^i c^j d^k : i > j > k\}$ is not context-free. Hint: use $z = b^{n+2}c^{n+1}d^n$ and pump both up and down.

20. Prove that the language $\{b^i c^j d^i e^j : i, j \geq 0\}$ is not context-free.

21. Can a language that is not context-free have the pumping property?

22. Does it follow from our discussion in Section 4.10 that if an infinite language has the pumping property, it is necessarily a context-free language?

23. Show that the context-free languages are closed under union. That is, let L_1 and L_2 be arbitrary context-free languages. Show that $L_1 \mid L_2$ is also a context-free language.

24. Let $L_1 = \{b^i c^j d^j : i, j \geq 0\}$ and $L_2 = \{b^i c^i d^j : i, j \geq 0\}$. What is the language $L_1 \cap L_2$. Are the context-free languages closed under intersection? That is, does the intersection of two context-free languages always yields a context-free language?

Chapter 5 PROBLEMS

1. Describe the language defined by

```
1) S  → dB
2) B  → bB
3) bB → AB
4) AB → Ab
5) Ab → Ba
6) dA → dc
```

2. Write a context-free grammar equivalent to G5.1.

3. Write a context-sensitive grammar that defines $\{b^i c^i d^i : i \geq 1\}$.

4. Write a context-sensitive grammar that defines $\{ww : w \in (b|c)^*\}$.

5. Write a context-sensitive grammar that defines $\{b^i c^j d^k : i < j < k\}$.

6. Write a context-senstive grammar that defines *QUADRUPLED* = $\{b^i c^i d^i e^i : i \geq 0\}$

7. Devise an algorithm that will work with any essentially non-contracting grammar that will determine if an arbitrary string is generated by the grammar. Does your algorithm also work for unrestricted grammars?

8. Try using the same technique we used to convert a contracting context-free grammar to a non-contracting grammars on contracting unrestricted grammars. Why does this technique fail?

9. Convert the production `WXYZ → ZYXW` to an equivalent set of genuine context-sensitive productions (i.e. productions which specify a left and/or right context).

10. Contracting productions increase the power of a grammar to generate languages. In a context-free grammar, contracting productions increases power minimally—they allow the grammar to generate the null string. On the other hand, in an unrestricted grammar, contracting productions substantially increase the power of the grammar. Why this difference?

11. Describe an unrestricted language that is not a context-sensitive language. You will probably have to consult a textbook on formal languages to answer this question (look under the topic of Turing machines or recursive languages).

Chapter 6 Problems

1. Rewrite the program in Fig. 6.12, using the `empty()` method in the `Stack` class in place of the bottom-of-stack marker `$`. Test your program with `cbd` and `cbb`.

2. Rewrite the program in Fig. 6.12 so that it uses the `int` 0 instead of `'#'` to signal the end of input. Why is using 0 better?

3. Using a regular expression, specify the language specified by G6.1.

4. Imitating Fig. 6.1, construct step-by-step the parse tree for `dbbdc` using G6.1.

5. Imitating Fig. 6.5, show the parse of `dbbdc` using G6.1.

6. Construct the parse table for

```
1) S → bSb
2) S → cAc
3) A → bAA
4) A → cASAb
5) A → dcb
```

7. Implement the stack parser for the grammar in problem 6.6. Test your parser with the strings `cdcbc`, `bcdcbcb`, `cbdcbdcbc`, `ccdcbcdcbcdcbbc`, `cdcbbb`, `cdcb`, and λ.

8. Construct the parse table for the following grammar:

```
1) S → bSc
2) S → d
```

9. Implement the stack parser for the grammar in problem 6.8. Test your parser with the strings `d`, `bdc`, `bbdcc`, `b`, `c`, `bbcd`, and `bcdd`.

10. Implement the stack parser for the following grammar:

```
1) S → bcdefg
```

11. What should you do to a grammar before using it to construct a parser? Consider, for example,

```
1) S → bABCD
2) S → c
3) A → bA
4) B → bB
5) C → bC
6) D → d
```

12. Show that the following grammar is LL(2):

```
1) S → bbbS
2) S → b
```

13. Give an example of a grammar that is LL(4) but not LL(3).

14. Show that the following grammar is not LL(k) for any k:

```
1) S → Ab
2) S → Ac
3) A → bA
4) A → λ
```

15. What operations should a top-down parser take when it is using a production whose right side starts with a nonterminal. Consider, for example, `D → ABC`.

16. Give the grammar that corresponds to the following table:

	b	c	#
S	pop push(B) push(B) advance	advance	
B		pop advance	
$			accept

17. Give an example of a RR(1) grammar (for those of us who like to read backwards).

18. Write a parser for G6.2 that uses the backtracking technique. Test your program with `ddc`, `cdbdbdc`, `ddd`, and `bcc`.

19. Write a parser for G6.2 that uses the lookahead technique. Test your program with `ddc`, `cdbdbdc`, `ddd`, and `bcc`.

20. Write a Java method that traverses a binary tree in depth-first order using the same general technique that we used in a stack parser.

21. Write a grammar for `b*c*d*`, design a stack parser based on your grammar, and write a program that implements your parser. Test your parser with `b`, `c`, `d`, `bc`, `bd`, `cd`, `bcd`, λ, `bcdb`, `cb`, and `db`.

22. An alternative approach a stack parser can take is to push the entire right side of production in reverse order (rather than everything up to but not including the leading terminal symbol). In what way, would our stack parser have to be modified to handle this alternative approach? Give the parse table for G6.1 using this alternative approach.

23. Modify the program in Fig 6.12 so that it constructs the parse tree as it parses the input string. When the parse is completed, the program should display every node of the constructed parse tree by traversing it in depth-first order.

24. Is this statement true: An LL(1) grammar is LL(k) for all k ≥ 1.

25. Is the following grammar LL(k) for some k? Justify your answer.

```
1) S → bB
2) S → bC
3) B → bB
4) B → d
5) C → bC
6) C → e
```

Chapter 7 PROBLEMS

1. Determine the selection sets for G7.18, G7.19, G7.20, and G7.21.

2. a) Determine the selection sets for

```
1) S → Ad
2) A → Bf
3) B → Cb
4) C → Dc
5) D → e
```

   b) Construct the parse table for this grammar.

   c) Show the sequence of input-stack configurations that occurs when your stack parser operates on the input strings `ecbfd` and `ecbff`.

   d) Implement the stack parser.

3. Same as question 7.2 but for the input strings `d` and `dd` and the grammar

```
1) S → A
2) A → B
3) B → C
4) C → d
```

4. Same as question 7.2 but for the input strings `bccdddeeee` and `bccdddeee` and the grammar

```
1) S →  BCDE
2) B →  b
3) C →  cc
4) D →  ddd
5) E →  eeee
```

5. Same as question 7.2 but for the input strings `bcdcde`, `bcde`, `bde`, `e`, and `bcd` and the grammar

```
1) S →  ABCe
2) A →  bB
3) A →  λ
4) B →  cC
5) B →  λ
6) C →  d
7) C →  λ
```

6. Same as question 7.2 but for the input string `bcdcd` and grammar

```
1) S → ABC
2) A → bB
```

```
3) A → λ
4) B → cC
5) B → λ
6) C → d
7) C → λ
```

7. Same as question 7.2 but for the input strings `d` and `dd` and the grammar

```
1) S → ABCD
2) A → λ
3) B → λ
4) C → λ
5) D → d
```

8. Same as question 7.2 but for the input string λ and `d` and the grammar

```
1) S → ABCD
2) A → λ
3) B → λ
4) C → λ
5) D → λ
```

9. Is the following grammar LL(1)?

```
1) S → λ
2) S → Ad
3) A → bAS
4) A → λ
```

10. Determine the selection sets for

```
1) S → ABC
2) A → bAS
3) A → λ
4) B → cASB
5) B → λ
6) C → AB
```

11. Convert the following grammar to an equivalent LL(1) grammar by eliminating left recursion:

```
1) S → S+S
2) S → S*S
3) S → b
```

12. Convert the following grammar to an equivalent LL(1) grammar:

```
1) S → SS+
2) S → SS*
3) S → b
```

Note: + and * are terminal symbols in this grammar.

13. Suppose corner substitution is performed on a grammar that is already LL(1). Is the resulting grammar always LL(1)?

14. Suppose substitution (not necessarily corner substitution) is performed on a grammar that is already LL(1). Is the resulting grammar always LL(1)?

15. Convert the following grammar to an equivalent LL(1) grammar by left factoring:

```
1) S → BCe
2) S → Bd
3) S → BCc
4) B → b
5) C → c
```

16. Convert the following grammar to an equivalent LL(1) grammar by left-right factoring:

```
1) S → BCe
2) S → Bde
3) S → BCee
4) B → b
5) C → c
```

17. Convert the following grammar to an equivalent LL(1) grammar:

```
1) S → Bd
2) S → bcd
3) B → bB
4) B → λ
```

18. Give an example of an unambiguous grammar that is not LL(k) for any k.

19. Show that the language defined by the following grammar has no LL(1) grammar that defines it:

```
1) S →  aSa
2) S →  bSb
3) S →  λ
```

20. Give an example like G7.30 in which disambiguating works. Hint: Consider the control structures in Java.

21. Does disambiguating work for

```
1) S →  bS
2) S →  Sb
```

```
3) S →  c
```

22. Suppose the end-of-input symbol were always included in the selection set of any production whose right side is nullable. Would the top-down parser using these selection sets still work—that is, ultimately accept all and only input strings generated by the grammar?

23. Construct the FIRST/FOLLOW graph for G7.6. Using the graph, determine the FIRST and FOLLOW sets for each nonterminal.

24. Construct the FIRST/FOLLOW graph for G7.7. Using the graph, determine the FIRST and FOLLOW sets for each nonterminal.

25. Construct the FIRST/FOLLOW graph for G7.8. Using the graph, determine the FIRST and FOLLOW sets for each nonterminal.

26. Construct the FIRST/FOLLOW graph for G7.9. Using the graph, determine the FIRST and FOLLOW sets for each nonterminal.

27. Construct the FIRST/FOLLOW graph for G7.10. Using the graph, determine the FIRST and FOLLOW sets for each nonterminal.

28. Construct the FIRST/FOLLOW graph for G7.11. Using the graph, determine the FIRST and FOLLOW sets for each nonterminal.

29. Construct the FIRST/FOLLOW graph for G7.12. Using the graph, determine the FIRST and FOLLOW sets for each nonterminal.

30. Construct the FIRST/FOLLOW graph for

```
1) S →  ABC
2) S →  BC
3) S →  C
4) A →  aAb
5) A →  B
6) B →  dAb
7) B →  C
8) C →  cd
9) C →  c
```

    Using the graph, determine the FIRST and FOLLOW sets for each nonterminal.

31. If a grammar G is LL(k), does there necessarily exist an LL(1) grammar equivalent to G?

32. Show that for an arbitrary nonterminal `A`, if FOLLOW(`A`) = {}, then `A` is useless.

33. In an LL(1) grammar, can more than one production with the same left side have a nullable right side? Justify your answer.

34. Determine the selection sets for

```
1)  S → eBbCD
2)  S → λ
3)  B → bBc
4)  B → BSe
5)  B → λ
6)  C → CC
7)  C → d
8)  C → λ
9)  D → SdD
10) D →  λ
```

35. Suppose a grammar has two productions of the form

```
1) A →  Ax
2) A →  y
```

Show that these productions do not have mutually disjoint selection sets when FIRST(y) is empty.

36. Construct the parse table for the following grammar:

```
1) S →  CDe
2) C →  cC
3) C →  λ
4) D →  dD
5) D →  λ
```

Optimize the `S` row of the operational table as much as possible.

37. Determine the selection sets for

```
1) S →  BCD
2) B →  BcCc
3) B →  λ
4) C →  CSc
5) C →  λ
6) D →  dBf
```

38. Consider the following grammar:

```
1)  S →  ABSdC
2)  S →  ABC
3)  A →  BbSAfA
4)  A →  Ba
5)  A →  λ
6)  B →  BeB
7)  B →  Sc
```

```
8) B →  λ
9) C →  SgC
10)C →  λ
```

Given that A, B, and C are nullable, does the second production, S → ABC, imply that FIRST(A), FIRST(B), and FIRST(C) are subsets of FIRST(S)? What other relations on the FIRST sets do the other productions imply? What can you say about all the FIRST sets? What does the production S → ABC imply about FOLLOW(S), FOLLOW(A), FOLLOW(B), and FOLLOW(C). What other relations on the FOLLOW sets do the other produtions imply? What can you say about all the FOLLOW sets. What is the selection set for each productions of the grammar?

Chapter 8 PROBLEMS

1. Construct the full and abbreviated parse tables for the following grammars:

   a)
   ```
   S → b
   S → c
   ```

   b)
   ```
   S → λ
   ```

   c)
   ```
   S → ABCD
   A → λ
   B → λ
   C → λ
   D → λ
   ```

   d) G7.1

   e) G7.3

   f) G7.8

   g) G7.12

2. Implement the improved table driven stack parser described in Section 8.4.

3.
   a) Show the nondeterministic parse table for

   ```
   S → bSb
   S → b
   ```

   b) Show a successful parse of `bbb`.

   c) Give a grammar that defines the same language whose parse table is deterministic. Show its parse table.

   d) Show a parse of `bbb` corresponding to your parser in part c.

4. Is it possible to construct a stack parser for $\{\texttt{ww} : \texttt{w} \in \{\texttt{b}, \texttt{c}\}^*\}$?

5. Is it possible to construct a deterministic stack parser for $\{\texttt{b}^i\texttt{c}^j\texttt{d}^k : i = j \text{ or } i = k\}$?

6. Construct a deteministic parse table for for $\{\texttt{b}^i\texttt{c}^{2i} : i \geq 0\}$. Show a parse of `bcc`.

7. Construct a deterministic parse table for $\{\texttt{b}^i\texttt{c}^i : i \geq 1\}|\{\texttt{c}\}$. Show a parse of `bcc` and `c`.

8. Are there any regular languages for which no deterministic stack parser exist? Try to find one such regular language.

Chapter 9 PROBLEMS

1. Implement a recursive-descent parser for

```
1) E → +EE
2) E → -EE
3) E → *EE
4) E → /EE
5) E → b
6) E → c
7) E → d
```

   Test your program with `b`, `c`, `d`, `+bc`, `-/bc*cd`, `+-*/bcbcd`, `bc`, `+b`, `*bcd`, and λ.

2. Implement a recursive-descent parser for

```
1)  S → BCD
2)  B → bB
3)  B → λ
4)  C → cC
5)  C → λ
6)  D → dD
7)  D → λ
```

   Test your program with λ, `b`, `c`, `d`, `bc`, `bd`, `cd`, `ccc`, `bcb`, `db`, and `dc`.

3. Implement a recursive-descent parser for the same language processed by the stack parser in Fig. 6.12. Test your program with `cbd`, `bcbdcbcbcd`, `cbdd` , `bcd`, `ddd`, and λ.

4. Implement a recursive-descent parser for the same language processed the stack parser in Fig. 8.4. Test your program with `fbb`, `fbbcc`, `fccdec`, `bb`, `fbbed`, and λ.

5. Implement a recursive-descent parser for G4.4. Test your program with `b, (b), (((b)), (b+c*d), (b+c)*d, bb, b), )b(,` and λ.

6. What is the `B()` method corresponding to

```
B → bBbBbB
B → cccc
B → e
```

7. Write both a recursive and a non-recursive `S()` method for

```
S → bS
S → cS
S → d
```

8. Write the `S()` method corresponding to

```
S → bSc
S → λ
```

9. Implement a top-down non-recursive parser for the grammar in problem 8. Do this in two ways:

   1) Implement a stack parser.
   2) Eliminate the recursion in the `S()` method of a recursive-descent parser.

   Test your two programs with λ, `bc`, `bbcc`, `bcb`, `bbccc`, `bbc`, `ccbb`, and `bcbc`. Compare your two parsers. In what ways are they similar?

10. Implement a recursive-descent parser for the language `{`*w*`d`*w*^{r} `:` *w* `∈ (b|c)*}`. Note: *w*^{r} is the reverse of *w*. Test your program with `d`, `bcdcb`, `cdc`, `bcdcbb`, `bccdcb`, and λ.

11. Convert the following grammar to an LL(1) grammar by left factoring.

```
1)  S → bcC
2)  S → bcDf
3)  C → dD
4)  D → eD
5)  D → λ
```

12. Write the `S()`, `C()`, `D()`, and `R()` methods for the grammar in problem 11. Note: `R()` corresponds to the nonterminal `R` that left factoring introduces.

13. Incorporate the code body of `R()` in problem 12 into `S()`.

14. Rewrite G4.4 in extended BNF.

15. Implement a recursive-descent parser for G4.4. Do not use recursion in your methods for `termList` and `factorList`. Test your program as specified by problem 5.

16. Write a Java program that creates a linked list and then traverses it in its natural order and in reverse order using recursive methods.

17. Write a Java program that displays the contents of an array in bottom-to-top order and top-to-bottom order using recursive methods.

18. In a LL(1) grammar, there can never be more than one production with the same left side that is nullable. Why? In a recursive-descent parser, we do not have to test if the current token is in the selection set for this nullable production. Why? Thus, the only selection sets we have to compute are the selection sets for the non-nullable productions. What does this imply about the computation of FOLLOW sets?

19. Will a recursive-descent parser always detect trailing garbage at the end of the input if the parser performs selection set tests for every production?

20. Does a parser that performs a selection set test for every production produce better error messages than a parser the performs a selection set test only when it is absolutely necessary?

21. Show the structure of the code for

```
S : (Q)*"d"
```

22. Show the structure of the code for

```
S : (Q)*
```

23. Show the structure of the code for

```
S: (A)*"b" | (B)*"d" | (C)*"d"
```

Assume `A` and `C` but not `B` are nullable.

24. What is the problem with converting the following grammar to parser code:

```
S: (B)*"d"
B: "b" | "d"
```

25. Write the recursive method corresponding to

```
S → bS
S → λ
```

Write the iterative method corresponding to

```
S : (b)*
```

In what way will the two methods behave differently?

26. Implement the parser for the following extended BNF grammar:

```
S : B* "e"? D+
B : "b" B | "c"
D : "d" L
L : "d" L | λ
```

Use recursion in your `B()` and `L()` methods. Use loops (for `B*` and `D+)` in your `S()` method. Test your parser with the following input strings:`d`, `ed`, `cd`, `ced`, `bced`, `bbceddd`, `b`, `bedb`, λ

27. Under what circumstances can a call of the `advance()` method be used in place of a call of the `consume` method . Is the result more efficient code? Consider the following grammar:

```
1) S → b
2) S → c
3) S → dBe
4) B → bB
5) B → d
```

Chapter 10 PROBLEMS

1. Show the parse tree for `bccd` in G10.2.

2. Show the parse tree for `+*/bcbc` in G10.4

3. Write a translation grammar whose input language is `bb*`. Each output string should be the same as the input string, but with a comma separating successive `b`'s. For example, if the input string is `"bbb"`, the output string should be `"b,b,b"`.

4. Implement the parser/translator corresponding to your answer for problem 3. Test it with `b`, `bb`, `bbb`, `dd`, and λ.

5. Write a translation grammar whose input language is `(b|c)*`. Each output string should be the input string with its characters rearranged so that all the `b`'s precede all the `c`'s.

6. Implement the parser/translator corresponding to your answer for problem 5. Test it with λ, `b`, `c`, `cbcb`, and `cccbbbb`.

7. Write a translation grammar whose input language is `(b|c)*.` Each output should be the number of `b`'s minus the number of `c`'s. For example, if the input string is `bbbc`, the output should be 2; if the input string is `cbccccb`, the output should be `-3`.

8. Write a translation grammar whose input language is `(b|c)(,(b|c))*.` This language consists of non-null lists of `b`'s and `c`'s with successive letters separated by a comma. The output string should be the same as the input string, but with its first element removed and appended on the end. For example, if the input string is `"b,b,c"`, the output string should be `"b,c,b"`.

9. Implement a parser/translator corresponding to your answer in Problem 10.8. Test it with `"b"`, `"c"`, `"b,c"`, `"c,b"`, and `"b,c,c,c"`.

10. Write a translation grammar that translates an arithmetic expression in infix notation to its corresponding decimal value. Limit your grammar to the addition and multiplication operations.

11. Implement the parser/translator corresponding to your answer for problem 10. Test it with `21`, `((22))`, `3*(4+5)`, and `3*4+5`.

12. Write a translation grammar that translates a list of unsigned integer constants to the maximum constant on the list. For example, if the input is `54 341 6 17` then the output is `341`. Assume successive constants in your input are separated by at least one white space character.

13. Implement the parser/translator corresponding to your answer for problem 12. Test it with the list given in problem 12.

14. Write a translation grammar that translates a list of unsigned integer constants into another list that contains each number from the input list followed by `"bigger"`, `"smaller"`, or `"equal"` depending on its size

relative to the average of all the numbers. For example, the input 1 98 99 is translated to `1 smaller 98 bigger 99 bigger.`

15. Implement the parser/translator corresponding to your answer for problem 14. Test it with the list given in problem 14.

16. Construct the parse tree for the input

```
x, y, z: int;
```

using G10.11.

17. On line 314 in the prefix expression compiler in Fig. 10.10, the parser converts the image of an unsigned integer constant to type `int`. Would it be better if the token manager performed this conversion and supplied the resulting `int` value to the parser via the token object the token manager returns?

18. Would the program in Fig. 10.10 work correctly if it used the following version of the `getNextChar()` method:

```
private void getNextChar()
{
  if (currentColumn = inputLine.length())

    if (inFile.hasNextLine())
    {
      inputLine = inFile.nextLine();
      currentLine++;
      currentColumn = 0;
      currentChar = inputLine.charAt(currentColumn++);
    }
    else
      currentChar = EOF;
  else
      currentChar = inputLine.charAt(currentColumn++);
}
```

19. Convert the following grammar to a translation grammar that translates each input string to its maximum nesting level:

```
1) S → (S)
2) S → λ
```

For example, the null string should be translated to `0`; the string `"(())"` should be translated to `2`.

Provide three translations grammars:

1) Count the left parentheses as you recurse "down" to the center of the input string. As you recurse "up", return the final count.

2) Count the right parentheses as you recurse "up".

3) Like approach 1, except do not return the final count as you recurse "up". Instead, display the final count when you reach the center of the string.

20. Implement the parser/translators for your answers in problem 19. Test with the input strings λ, `()`, `(())`, `((`, and `())`.

21. Implement the parser/translator corresponding to G10.4. Test your program with the input strings `b`, `+bc`, `+b*cd`, `+*/-bbcde`, `+b`, `+bcd`, and λ.

22. Convert G10.8 to a translation grammar that translates prefix expressions to infix expressions.

23. Implement the parser/translatior for your answer in problem 22. Test with the input strings `b`, `c`, `d`, `+bc`, `/cd`, `+b+cd`, `*b-cd`.

24. Write a translation grammar that defines `b*dd*` and translates each input string by outputting each `b` in the input but only the first `d`. For example, `bbdd` should be translated to `bbd`, and `ddd` should be translated to `d`.

25. Why is the `getToken()` method in Fig. 10.10 in the parser rather than in the token manager?

26. Why are tokens chained by the parser in Fig. 10.10 rather than by the token manager?

Chapter 11 PROBLEMS

1. Translate to an assembly language program:

```
       b = 1;
       c = 2;
       e = 3;
       d = b - c;
       e = e*e/d;
       println(b);
       println(c);
       println(d);
       println(e);
```

   Do not optimize. For example, do not initialize `b`, `c`, and `e` to 1, 2, and 3, respectively in their `dw` statements. Instead, initialize them to 0 in their `dw` statements. Place your program in a file named `p1101.a`. Assemble and run with

```
a p1101.a
e p1101 /c
```

2. Assemble and run the following program. What happens?

```
start:      p        5
            dout
            ja        start
```

   The `ja` instruction is the "jump always" instruction.

3. Translate the following statements to assembly language. Do not optimize. For example, for the first statement, you should provide assembly code that performs all the specified additions. Specifically, after pushing the address of `x`, push the operands, 1, 2, …, 10. Then perform the specified additions, followed by a `stav` instruction. Use the `pc` instruction to push each constant.

```
      x = (1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + (9 + 10)))))))));
      print("x = ");
      println(x);
```

   Place your program in a file name `p1103.a`. Assemble and run with

```
a p1103.a
e p1103 /c
```

   Change your program so that it pushes the constants with the `pwc` instruction instead of the `pc` instruction. Assemble and run. Compare the two log file reports. Which version is better? Why?

4. Translate the following statements to assembly language. Do not optimize. For example, for the first statement, you should provide assembly code that performs all the specified additions. Specifically, after pushing the address of `x`, push 1 and 2, then add. Then push 3 and add. Continue in this fashion, until all ten numbers have been added. Use the `pwc` instruction to push the constants.

```
      x = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10;
      print("x = ");
      println(x);
```

   Place your program in a file name `p1104.a`. Assemble and run with

```
a p1104.a
e p1104 /c
```

5. Compare the efficiencies of the assembly language programs for problems 3 and 4. Are they different?

6. Assemble and run the following program. What happens? Why?

```
      dout
      halt
```

7. Which bits of the `sp` register are used to determine where in memory an item is pushed? All 16 bits?

8. The stack pointer initially contains 0. What is the address of the memory location that receives the first value pushed? Hint: the stack pointer is decremented by 1 before each push operation.

9. Without using the `mult` instructions, write the most efficient assembly language sequence you can that multiplies 23 by 48 and displays the result. Display the result without any labels. Assemble and run with

```
a p1109.a
e p1109 /c
```

   Hint*:* You may want to use the `dupe` instruction. It duplicates the top of the stack. For example, if 5 is on top of the stack and you execute `dupe`, then an additional 5 is pushed onto the stack.

10. Which instruction below is better? Why?

```
      pc          5
```

or

```
      p           @5
```

   where `@5` is defined with

```
@5:         dw          5
```

   Which instruction below is better? Why?

```
      pwc        5
```
or
```
      p          @5
```

where `@5` is defined shown above.

11. Write an assembly language program that prompts the user for two integers, adds the two integers, and displays the sum. You program should produce output that looks exactly like that in the sample session below:

    ```
    Enter integer
    1
    Enter integer
    2
    Sum = 3
    ```

    After outputting the sume, your program should position the cursor at the beginning of the line that follows the sum. Place your program in a file name `p1111.a`. Assemble and run with

    ```
    a    p1111.a
    e    p1111 /c
    ```

12. Suppose you delete the `entry` directive in the program in Section 11.11. What happens when the program is executed?

Chapter 12 PROBLEMS

1. Copy `S1.java` to `S2.java`. Then replace every occurrence of "`S1`" in `S2.java` with "`S2`". Enter your name in `S2.java` on lines 10 and 30. Modify `S2.java` so that it meets the specifications for S2 described in Section 12.12. Compile your S2 compiler with

   ```
   javac S2.java
   ```

   Compile `S2.s` (which is in the J1 Software Package) with your S2 compiler with

   ```
   java S2 S2
   ```

   Assemble the output file `S2.a` created by your S2 compiler with

   ```
   a S2.a
   ```

   Finally run the executable program in `S2.e` created by the assembler with

   ```
   e S2 /c
   ```

   Submit to your instructor `S2.java`, `S2.a` and `S2.`<*family name*>`.log` (the log file that the `e` program creates when it runs `S2.e`).

2. Is it possible to output the `dw` for a variable as soon as that variable is encountered in the source program. If that is the case, can we eliminate the symbol table from S1? Hint: see the discussion of caret lines in Section 11.12.

3. Give two examples where a null statement in Java is useful.

4. S1 does not close the input file. It this a bug? What problem results if S1 does not close the output file?

5. In the translation grammar for the assignment statement, we emit the `stav` instruction before we parse the semicolon at the end of the statement (see line 32 in Fig. 12.2). Is the reverse order possible? Is the reverse order better?

6. If we were to change our target assembly language, we would have to change the parser in S1. That is, our parser implementation is dependent on the target language. Is it possible to implement the parser in S1 so so that it is largely independent of the target language? What would be the advantage of such an implementation?

7. Copy `S1.java` to `S1207.java`. Change all occurrences of "`S1`" in `S1207.java` to "`S1207`". Then rewrite `S1207.java` so that the `expr()` method corresponds to the production

   ```
   expr → term ("+" term)*
   ```

   and the `term()` method corresponds to the production

```
term → factor ("*" factor)*
```

That is, use loops instead of recursion for these productions. Test your new compiler by entering

```
javac S1207.java
java S1207 S1
a S1.a
e S1 /c
```

8. What error message does S1 generate if the source program is

```
x = 3; &
```

What error message does S1 generate if the source program is

```
x = 3; y
```

9. To support comments, you have to check in `getNextChar()` if the current character and the character that follows it are both `'/'`. Before you check for the second `'/'`, do you have to make sure that you are not already at the end of `inputLine`? Could an index-out-of-bounds exception occur?

10. What does your S2 compiler do when it compiles

```
x = 32768;
y = 5555555555;
```

Assemble the resulting assembly language program. What happens? Why?

11. Why are the assignments to `endLine` and `endColumn` inside the loop rather than after the loop on lines 171, 172, 187, and 188 in S1? Would not executing the assignment to `endLine` and `endColumn` once suffice?

12. The token managers in S1 and S2 perform a serial search to determine if an identifier token is actually a keyword (see lines 195 to 198 in S1). An alternative approach is a binary search of a table preloaded with all the keywords in ascending order. In S1, there is only one keyword `("println")`. Thus, is does not make sense to use the more complicated binary search in place of a serial search. How many keywords would justify using a binary search?

13. Copy `S2.java` to `S1213.java`. Change all occurrences of `"S2"` in `S1213.java` to `"S1213"`. Modify `S1213.java` so that it using a binary search to detect keywords (see Problem 12.12). Test your compiler as specified in Problem 12.1.

14. Examine the assembly code that your S2 compiler produces for the source code in `S2.s`. Try to write by hand more efficient code. Assemble with `a` and run with `e` using the `/c` command argument. Examine the performance statistics in the log file. How much better is your hand-written assembly code?

15. Copy `S1.java` to `S1215.java`. Change all occurrences of `"S1"` in `S1215.java` to `"S1215"`. Then rewrite `S1215.java` so that it uses the following optimization technique: Evaluate constant expressions at compile time rather than at run time. This technique is called ***constant folding***. For example, translate the

following statement

```
x = 1 + 2;
```

by evaluating the expression at compile time to get 3. Then generate the assembly code

```
pc        x
pwc       3
stav
```

Without this optimization technique, S2 would generate assembly code that performs the addition at run time:

```
pc       x
pwc      1
pwc      2
add
stav
```

Run your optimized S1215 compiler against `S1.s`. Examine the performance statistics in the log file. Is the code generated more efficient that the code generated by S1?

16. The `main` method in S1 outputs the compiler/author information to the output file (see line 30 in Fig. 12.3). Would it be better for the code generator's constructor to output this information?

17. Copy `S1.java` to `S1217.java`. Change every occurrence of `"S1"` in `S1217.java` to `"S1217"`. Then rewrite `S1217.java` so that it compiles directly to machine code. That is, it should output a machine code file (with extension "`.e`") rather and an assembly code file. Test your compiler by entering

```
javac S1217.java
java S1217 S1
e S1 /c
```

The machine code file that your S1217 compiler produces should start with the ASCII code for the capital letter "`T`" followed by the machine code in binary.

The best way to implement S1217 is to have its code generator enter machine code into an `int` array whose contents are then outputted at the end of the compile. Let's call this array `code`. Note that the index of an instruction in the `code` array is also the address of that instruction.

When the code generator enters an instruction that references a variable into the `code` array, it should enter its instruction's index into the symbol table entry for that variable. For example, suppose the instruction at index 0 references `x`. Then 0 should be recorded in the symbol table entry for `x`. If the instruction at index 4 also references `x`, then 4 should also be recorded in the symbol table entry for `x`. Thus, the symbol table entry for `x` will contain a list of indices—an index for every instruction that references `x`.

For each symbol in the symbol table, the `endcode()` method should

1) Insert the index of the next available slot in the `code` array into every instruction that references that symbol. `endcode()` can determine where these instructions are in the code array by examining the list of indices for that symbol table entry.

2) Insert 0 into the `code` array (this action allocates the variable and gives it an initial value).

After processing the symbol table, `endcode()` should then output in binary the ASCII code for the capital letter `"T"` (54 in hex) followed by the contents of the `code` array.

For example, consider the source program

```
x = 1;
println(x);
```

The corresponding machine and assembly code is

```
1009        pc       x        ; x = 1
F700 0001   pwc      1
F300        stav

0009        p        x        ; println(x);
FFFD        dout
100A        pc       '\n'
FFFB        aout
FFFF        halt
0000        dw       0
```

When the code generator in S1217 enters the machine code for the intial `pc` instruction into the code array, it does not know the address of `x`. Thus, it enters the machine code for a `pc` instruction with zero in its address field. For the same reason, the `p` instruction will have 0 in its address field. Then, when `endcode()` starts processing the symbol table, the `code` array will contain

```
1000  <----  0 in address of pc  instruction
F700
0001
F300
0000  <----  0 in address of p instruction
FFFD
100A
FFFB
FFFF
      <----  x will go here.  This slot has index 9.
```

At this point, the zero value for `x` has not yet been entered into the code array. Thus, the next available slot in the code array corresponds to `x`. This slot has index 9. Thus, the address of `x` is 9. When `endcode()` processes the symbol table entry for `x`, it has to insert 9 (the address of `x`) into all the instructions in the code array that reference x. Using the symbol table entry for x, endcode() can determine that the instructions at indices 0 and 4 reference x. Thus, encode() then inserts (by adding) 9 into the code array at indices 0 and 4. It then enters 0 into the next slot of the code array (this is the slot for x). Finally, endcode() outputs in binary the ASCII code for the capital letter "T" (54 in hex) followed by the contents of the code array.

18. Could the code that produces the token trace in the S1 compiler be placed in the parser rather than in the token manager? Could the code that outputs the source code as comments be placed in the parser rather than in the token manager?

19. Copy S1.java to S1219.java. Change every occurrence of "S1" in S1219.java to "S1219". Then rewrite S1219.java so that so that it uses the pc instruction in place of the pwc instruction wherever possible. Using your modified compiler, compile the source code in S1.s. How does the modified compiler compare with the original S1 compiler with respect the size and execution time of the generated target code? Next, further modify your compiler so that it uses the p instruction in place of the pwc instruction. A p instruction that replaces a pwc instruction requires a dw for the constant. For example, the following pwc instruction,

```
          pwc    -1
```

can be replaced by

```
          p      @_1
```

where @_1 is defined with

```
@_1:      dw     -1
```

How does this modification affect the size or execution time of the S1.s program?

20. The genEx method in S1 returns a RuntimeException to its caller (which the caller throws). Would it be better if genEx threw the exception rather than return it?

21. The error messages on line 461 and 487 in S1 include the right parenthesis even when a right parenthesis is not an expected token for the given input. For example, when S1 compiles the following program

```
x = 2
```

it displays the error message

```
Expecting "*", "+", ")", or ";"
```

even though the right parenthesis is not expected after 2 (because there is no preceding left parenthesis). Modify S1 so that it does not include the right parenthesis in an error message if it is not among the expected tokens.

22. Could the kind value of single-character tokens be the character itself? What changes would be required to S1? How does this approach compare with the approach used by S1.

23. Is it possible for line 379 in S1 to be executed? If so, give a source program that will cause it to be executed.

24. Would the following replacement for the `statementList()` method in S1 work correctly? In what way, if any, would S1 behave differently?

```
private void statementList()
{
  switch(currentToken.kind)
  {
    case ID:
    case PRINTLN:
      statement();
      statementList();
      break;
    default:
      break;
  }
}
```

25. Modify S1 so that on a parsing error, a `ParsingException` rather than a `RuntimeException` is thrown, where `ParsingException` is a class of your own creation. What is the advantage of doing this?

Chapter 13 PROBLEMS

1. Implement S2 as described in Chapter 12 using JavaCC. Call your JavaCC-generated compiler S2j. You should

   a. Copy `S1j.jj` to `S2j.jj`.

   b. Insert your name into `S2j.jj` on lines 25 and 43.

   c. Replace the translation grammar in `S2j.jj` with `S2.tg`.

   d. Add an entery to the `SKIP` block that defines a single-line comment. Hint: a single-line comment consists of two slashes, followed by zero or more occurrences of any character except \n and \r.

   e. Add entries to the `TOKEN` block for the new tokens: `"print"`, `"/"`, `"{"`, `"}"`.

   f. Add actions so that source code is output as comments (see Section 13.10).

   g. Test your compiler by entering

```
javacc S2j.jj
javac S2j.java
java S2j S2
a S2.a
e S2 /c
```

   Submit to your instructor `S2j.jj`, `S2.a`, and `S2.`<*family name*>`.log` (the log file that the `e` program creates).

2. Compile

```
x = ? 5;
```

   with the S1j compiler. Then delete the `ERROR` entry in the `TOKEN` block in `S1j.jj`. With this modified S1j compiler, compile the above statement. How do the error messages of the two versions of S1j compare?

3. Move the `ERROR` entry in the `TOKEN` block in S1j to the beginning of the `TOKEN` block. Compile `S1.s` with the modified S1j compiler. What happens? Why?

4. Is this translation grammar acceptable to JavaCC?

```
void S(); {}
{
   {int x;}    // declare local variable as action
   x = B()
   {System.out.println(x);}
}
int B(): {}
```

```
{
   {return 5;}
}
```

5. A JavaCC input file consists of several components: the `PARSER_BEGIN/PARSER_END` block, the `SKIP` block, the `TOKEN` block, and the translation grammar. Can these components be placed in any order? Experiment with JavaCC to determine your answer.

6. Represent all the lists in the translation grammar in `S1j.jj` using the star or plus operators. Compare the resulting code with that produced by that in the original `S1j.jj`. Is it smaller? Is it faster?

7. Write a JavaCC extended regular expression that defines all strings with one or more occurrences of the substring `"bc"`. Test your expression using JavaCC against the following input:

```
b, c, bc, cb, bbbbbbccccccc, cccbbb
```

8. Write a JavaCC extended regular expression that defines all strings with exactly one occurrence of the substring `"bc"`. Test your expression using JavaCC against the following input:

```
b, c, bc, cb, bccbc, bcbbbbbbbbbbbbbc
```

9. Write a JavaCC extended regular expression that defines a Java multiple-line comment (i.e., a comment bracketed with `"/*` and `*/"` that can span lines). Assume nested comments are not allowed. Test your expression using JavaCC.

10. Determine if command line arguments for JavaCC are case sensitive.

11. Write a JavaCC extended regular expression that defines a quoted string that can span multiple lines as long each newline is backslashed. Test your expression using JavaCC.

12. Suppose an expression in the `SKIP` block and an expression in the `TOKEN` block both match the same string. Which expression has precedence? Does the order in which the `SKIP` and `TOKEN` blocks appear matter?

13. Implement the compiler in Fig. 10.10 using JavaCC. Call your JavaCC version `Fig1013j.jj`. Test by entering the expression

```
+ * - / 10 9 8 7 6
```

14. In what ways does S1j function differently from S1 from Chapter 12?

15. What are the disadvantages of using JavaCC compared to writing the compiler yourself?

16. Fix the following grammar with a `LOOKAHEAD` directive:

```
void S(): {}
{
   B() "b" "c"
}
```

```
void B(): {}
{
   ("b" "c")*
}
```

17. Fix the following grammar with a `LOOKAHEAD` directive:

```
void S(): {}
{
   ("b" "c")+ "b" "d"
}
```

18. Can the last alternative for a non-terminal be applied without first performing a selection set test if the alternative is not nullable?

19. Show the structure of the code corresponding to

```
void S(): {}
{
("b"|"c")?
}
```

20. Show the structure of the code corresponding to

```
void S(): {}
{
("b" "c")+ "b" "b"
}
```

Repeat but with `LOOKAHEAD(2)` inserted at the choice point.

21. Suppose we delete line 206 in Fig. 13.14. What would the S1j compiler do if the source program is

```
x = 5;
7 = 8;
y = 9;
```

22. What will the following regular expression match:

```
<WHAT: ~[]~[]>
```

23. Create a Java program with JavaCC that inputs a text file and outputs the same text file with a line number inserted at the beginning of each line. For example, if the input file contains

```
aaa
bbb
.
.
.
```

```
jjj
```

then the output file would be

```
  1 aaa
  2 bbb
  .
  .
  .
 10 jjj
```

The line numbers in the output file should be right justified and have a field width of 3.

24. Create a program with JavaCC that inputs a text file and outputs the number of words in the file. Test your program with a file that contains

    Let's, you and I , have a really good time studying compilers.

    Your program should produce a count of 11 for this file.

25. Compile the following program using the S1j compiler:

    ```
    x = - 32768;
    println(x);
    ```

    Then assemble. Try the do the same with

    ```
    x = 5 - 32768;
    println(x);
    ```

    What happens? Why?

26. How are the symbol table and token manager objects passed to the parser in S1? In S1j?

27. Can an identifier for a regular expression be used before its definition in a `Token` block in a JavaCC input file? For example, is the following sequence legal:

    ```
    <UNSIGNED: (<DIGIT>)+>
    <DIGIT: ["0"-"9"]>
    ```

28. Would it be correct to test only `getToken(2).kind` on line 3 in Fig. 13.22? Try a test case with JavaCC to confirm your answer.

29. Can syntactic and semantic lookahead in JavaCC be placed at the multiple choice points in a list of alternatives? Confirm your answer with a test case.

30. Can you test the `image` field of a token instead of the `kind` field to determine the token type? For example, can you do this:

```
if (t.image.equals("+"))
{
...
}
```

What are the disadvantages of this approach?

31. Does the JavaCC-generated parser perform a selection set test for a non-terminal if there is only one production in the grammar with that non-terminal on the left side? Determine your answer by inspecting the code JavaCC generates for this case.

32. What happens if you move the `SKIP` block in `S1j.jj` to after the `TOKEN` block?

33. Why does the parser rather than the token manager produce the token trace?

34. Replace the `SimpleCharStream` class that JavaCC generates with a class you write. It should implement the interface `CharStream`, which JavaCC generates if the `USER_CHAR_STREAM` option is set to true.

Chapter 14 PROBLEMS

1. Implement the hand-written S3 compiler or the equivalent JavaCC-generated S3j compiler by extending your S2 or S2j compiler. Test your compiler by entering

```
javac S3.java    or     javacc S3j.jj
java S3 S3              javac S3j.java
a S3.a                  java S3j S3
e S3 /c                 a S3.a
                        e S3 /c
```

Submit to your instructor your source file (`S3.java` or `S3j.jj`), the assembly file created by your compiler, and the log file. Do not generate a token trace on the final version you submit. Note: you will probably get `OVER LIMIT` warnings when you execute the compiled program because your compiler does not produce optimal code for the unary minus operator (see problem 2).

2. Modify Fig. 14.4 so that two unary minus operators separated by one or more unary plus operators are optimized out. For example, the statement

```
y = - + + -x;
```

should be translated to

```
pc        y
p         x   // no neg instructions needed
stav
```

Hint: Change line 30 in Fig. 14.4 so that it matches one or more unary plus signs. Also change line 31 so that if the current token at that point is a unary minus, `factor()` is called but the not `emitInstruction("neg")`. This action optimizes out the current token unary minus along with the unary minus on line 15. If, on the other hand, the current token at line 31 is not a unary minus, then both `factor()` and `emitInstruction("neg")` should be called (for this case, we need the `neg` instruction for the unary minus on line 15). The selection sets for both of these alternatives will include the unary minus sign. Thus, the modified grammar will not be LL(1). However, you can still use it for deterministic parsing: Simply perform the first alternative whenever it is consistent with the current token. This approach is precisely what JavaCC-generated parsers do when they encounter a choice—they perform the first choice given in the grammar. Thus, if you are using JavaCC to implement S3j, you can use Fig. 14.4 modified as we have described. However, you may want to add `LOOKAHEAD(1)` directives to suppress the "choice conflict" messages that JavaCC will generate (see Section 13.11). You will need two `LOOKAHEAD(1)` statements—one for the choice conflict on the modified line 30 and one for the choice conflict on the modified line 31.

3. What does your S3 compiler do if a string is specified in the source program that is missing the terminating quote?

4. Execute the following Java code. Do `p` and `q` point to the same object?

```
String p, q;
p = "hello";
```

```
q = "hello"
if (p == q)
   System.out.println("p, q sharing one string object");
else
   System.out.println("p, q not sharing");
```

5. Execute the following C++ code. Do `p` and `q` point to the same string?

```
char *p, *q;
p = "hello";
q = "hello"'
if (p == q)
   cout << "p, q sharing one string";
else
   cout << "p, q not sharing";
```

6. A unary minus preceding a constant is detected and handled by the `factor()` method. Why not have the `getNextToken()` detect and handle a unary minus preceding a constant. For example, for `-5000`, it would return the token `<INTEGER>`, whose image is `"-5000"`.

7. Do these productions indicate a good way to add support for unary plus and minus?

```
      factor → <UNSIGNED>
      factor → <ID>
      factor → "(" expr ")"
      factor → "+" factor
      factor → "-" factor
```

   Is the language defined any different from the source language for S3?

8. What error message does S3 generate if the source program is

```
x = 3; &
```

   If the source program is

```
x = 3; y
```

9. When happens if during the execution of the readint statement, the user enters a non-integer or an integer too big?

`10.` If you define `<STRING>` is `S3j.jj` with

```
<STRING: "\" (~["\n", "\r"])* "\"">
```

   what happens when you compile

```
println("ABC"DEF");
```

11. Using EBNF and no recursion, define the cascaded assignment statement.

12. Compile the following statements with a C or C++ compiler:

```
y = - +  - + - - - -x;
```

    When you invoke the compiler, specify the command line argument that causes the compiler to output an assembly listing of the translated program. From this listing, determine if the generated code performs any negation operations. Repeat with a Java compiler. To get the JVM assembly code for a Java class file `X.class`, enter

```
javap –c X
```

13. Can you think of any reason why consecutive unary minus signs should *not* be optimized out—that is, not be translated to any code. For example, is there any reason why a compiler should generate six negation operations for the statement in Problem 12? What changes to Fig. 14. 4 are required if consecutive unary minus signs are not to be optimized out?

14. Copy your version of S3 or S3j to `S1414.java` or `S1414j.jj`, respectively. Then extend `S1414.java` or `S1414j.jj` so that it supports the increment operator (++) and the decrement operator (`--`). Test your compiler by entering

```
java S1414 S1414         or     java S1414j S1414
a S1414.a                       a S1414.a
e S1414 /c                      e S1414 /c
```

Chapter 15 PROBLEMS

1. Implement the S4 or S4j compiler. Test your compiler by entering

```
javac S4.java   or   javacc S4j.jj
java S4 S4           javac S4j.java
a S4.a               java S4j S4
e S4 /c              a S4.a
                     e S4 /c
```

   Also enter

```
java S4 p1501  or  java S4j p1501
```

   Your compiler should recover from all the errors in `p1501.s` (this file is in the J1 Software Package). Submit `S4.java` or `S4j.jj`, the assembly files produced by your compiler, and the log files to your instructor.

2. Modify `S1j.jj` so that the S1j compiler can handle both single-line comments that start with `"//"` and multi-line comments that are bracketed with `"/*"` and `"*/"`. Your modified S1j compiler should not comment the assembly code file with the source code it is translating. Test your modified compiler by compiling the file `p1502.s` (in the J1 Software Package) by entering

```
javacc S1j.jj
javac S1j.java
java S1j p1502
a p1502.a
e p1502 /c
```

   Submit to your instructor your modified `S1j.jj`, `p1502.a`, and `p1503.`<*family name*>`.log` (the log file that the e program creates).

3. Further modify `S1j.jj` from Problem 15.2 so that the S1j compiler's output includes as comments the source code in the input file (see Section 13.11). Test your modified S1j compiler as specified in problem 2. Use a `SPECIAL_TOKEN` block to capture the comments in `p1502.s` so you can output them to the assembly file.

4. Write the translation grammar for the `do-while` statement.

5. Using your S4/S4j compiler, compile the following program:

```
x = -32768;
y = 32767;
z = 5 - 32768;
```

   What error messages, if any, does your compiler produce?

6. If the `do-while` loop below were legal, what ambiguity would we encounter?

```
do
  x = x + 1;
  y = y + 1;
while (x);
```

7. Why does C, C++, and Java have a `do-while` but not a `do-until` statement.

8. Write a translation grammar for a `loop` statement defined by

```
loopStatement → "loop" expr statement
```

When executed, a `loop` statement will execute its body (a statement) the number of times equal to the value of the expression following the keyword `loop`. If the value of the expression is less than or equal to 0, then the statement is not executed.

9. Write out the assembly code corresponding to the code below that your S4/S4j compiler should generate. Then check your answer by compiling the code with your compiler and examining the output file.

```
if (b)
if (c)
if (d)
d = 1;
else
d = 2;
else
c = 1
else
b = 1;
```

10. Write out the assembly code corresponding to the code below that your S4/S4j compiler should generate. Then check your answer by compiling the code with your compiler and examining the output file.

```
while (a)
while (b)
while (c)
{
   a = 0;
   b = 0;
   c = 0;
}
```

11. Suppose a compiler labels a point in the assembly code it generates with multiple labels. For example,

```
@L3:
@L2:
@L1:
      pc         x
      .
      .
      .
```

Do these multiple labels cause a space or time inefficiency in the executable code to which the assembly code is translated?

12. Write a non-ambiguous grammar for the `if` statement. Is your grammar suitable for top-down parsing?

13. Compute the selection set for the production

```
elsepart → λ
```

in the grammar for the S4 compiler. Explain why it includes `";"`, `"{"`, and `"}"`.

14. Describe what happens when the token manager defined by Fig. 15.7 processes the input string `"A\nB"`. What matches the backslash? What matches the `"n"`?

15. The productions for the `if` statement are not LL(1). Why does JavaCC accept the `if` statement productions without complaint?

16. What happens when S4/S4j, modified with error recovery as described in Section 15.12 and 15.13, translates

```
if (x)
{
x = 5;
else
x = 6;
```

17. Give a source program for which the error recovery mechanism as described in Section 15.12 and 15.13 does a poor job.

18. Where does your S4j compiler detect the error in

```
println("hello
goodbye");
println("yes");
```

19. Would it be all right to move the `MORE` block on lines 15 to 18 to after the `TOKEN` block on lines 20 to 53 in Fig. 15.7? Justify your answer.

20. Would error recovery work correctly if `program()` were changed to

```
void program(): {}
{
   (statement())*
   {codeGen.endCode();}
   <EOF>
}
```

Justify your answer.

21. Modify your S4/S4j compiler so that error recovery is at the `statementList` level rather than at the `statement` level. Run your modified compiler against `p1521.s` and `p1501.s`. Compare with the error recovery for your original S4/S4j compiler. Any differences?

22. Will this definition of a string constant

    ```
    <STRING: "\"" (~["\n", "\r"])* "\"">
    ```

    match the entire string in

    ```
    println("abc\"cde");
    ```

    Use JavaCC to check your answer. Does this definition of a string correctly handle embedded quotes?

Chapter 16 PROBLEMS

1. `Implement the hand-written S5 compiler or the S5j JavaCC version. A partial translation grammar for S5 (Fig. 16.6) is available in the J1 Software Package in S5.tg. Test your S5 or S5j compiler by entering`

   ```
   javac S5.java    or     javacc S5j.jj
   java S5 S5a             javac S5j.java
   java S5 S5b             java S5j S5a
   a S5a.a S5b.a sup       java S5j S5b
   e S5a /c                a S5a.a S5b.a sup
                           e S5a /c
   ```

   Submit `S5.java` or `S5j.jj`, the assembly files created by your compiler and the log file to your instructor.
   Also test error recovery by entering

   `java S5 p1601` or `java S5j p1601`

2. What happens if you link `S5a.o` and `S5b.o` without `sup.o`? Does the link complete? If so, what happens when you execute the resulting program?

3. Create an `a.l` library that contains the `S5b.o` module created in problem 1. Then link the `S5a.o` with `S5b.o` in the library by entering

   ```
   a S5a sup
   ```

   Test the resulting `S5a.e` executable module.

4. `Implement the hand-written S6 compiler or the S6j JavaCC version. Test your S6 or S6j compiler by performing the following steps:`

   a. `Compile S6b.s with your S6 or S6j compiler.`
   b. `Using the l program, create a library containing the object module S6b.o obtained from S6b.s.`
   c. `Enter`

      ```
      a S6a.a sup
      e S6a /c
      ```

   `Submit S6.java or S6j.jj, S6a.a, S6b.a and the log file to your instructor.`

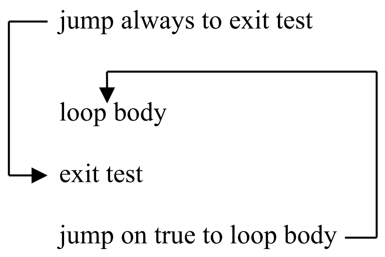
5. Would it be difficult for the parser to generate code to push the values of the arguments in a function call in right-to-left order? How would this approach affect the implementation of `parameterList()`? See problem 17.

6. Describe how you would implement `parameterList()` non-recursively.

7. Describe how you would implement the `for` statement.

8. Why does the compiler prefix the labels it generates with `"@"`?

9. An alternate structure for the `while` loop is



In what way is this structure superior to the structure we have been using for the `while` loop? Incorporate this structure into your S5 compiler. Test your modified S5 compiler with `S5a.s` and `S5b.s`, as specified in Problem 16.1.

10. Create an `a.l` library that contains three modules. One displays `"left"`; one displays `"middle"`; one displays `"right"`. Write an assembly language program that displays

```
left middle right
```

by calling these three modules. Place your program in a file named `p1610.a`. Assemble and run with

```
a p1610.a
e p1610 /c
```

11. Modify `S5.java` or `S5j.jj` so that a local variable can be declared anywhere within a function definition body. The scope of a local variable should start at the point of declaration and extend to the end of the function body. Test your new compiler as specified in problem 1. Also use your compiler to compile `p1611.s`, which is in the J1 Software Package. Assemble, link with `sup.o` and run with

```
a p1611.a sup
e p1611 /c
```

12. Modify your compiler from problem 11 so that the scope of a local variables ends at the end of the block in which it is declared. Thus, if it declared within a compound statement, it ends at the end of the compound statement. Test your new compiler as specified in problem 1. Also use your compiler to compile `p1612.s`, which is in the J1 Software Package. Assemble, link with `sup.o`, and run with

```
a p1612.a sup
e p1612 /c
```

13. Add support to your S5 or S5j compiler for the scope resolution operator `"::"`. When this operator precedes an identifier, it indicates that the identifier should be interpreted as the globally defined one even if it
is within the scope of an identically-named local identifier. Test your new compiler as specified in problem 1. Also use your compiler to compile `p1613.s`, which is in the J1 Software Package. Assemble, link with `sup.o`, and run with

```
a p1613.a sup
e p1613 /c
```

14. Suppose the `find` method for a `LOCAL` symbol returned its negated index, but for a non-`LOCAL` symbol returned its non-negated index. Could `push()` and `pushAddress()` then be implemented more efficiently. If so, how?

15. Are the local entries in the symbol table always grouped together at the bottom of the symbol table. If so, can you make the method `localRemove` more efficient?

16. Would it be better to have all local entries in the symbol table in a separate table? In that case, how would the `find` method work?

17. Modify your S5 or S5j compiler so that it pushes arguments in right-to-left order (the same order C and C++ uses). Then the relative address of the parameters are numbered starting from 2 from left to right. Test your compiler as specified in Problem 16.1.

18. Is the any reason *not* to use the star operator to define the list of statements in `statementList()` (lines 151 to 158 in Fig. 166)?

19. Evaluate the effectiveness of the error recovery mechanism you implemented in S5/S5j. Extend your recovery mechanism so that it recovers for most errors.

Chapter 17 Problems

1. What is the difference between a non-deterministic FA and an FA that is not deterministic?

2. Construct a DFA that defines `b*c*d*`. Write a computer program that simulates its operation. Test it with λ, `b`, `c`, `d`, `bc`, `bd`, `cd`, `bcd`, `bbdd`, `bcb`, `bcdc`, and `dc`.

3. Construct a DFA that defines `b*cb*`. Write a computer program that simulates its operation. Test it with `c`, `bc`, `cb`, `bcb`, `bbcb`, `bb`, `bcbc`, and `bccb`.

4. Construct a 4-state NFA then defines all strings over {`b`, `c`} that end in `bcc`. Using the subset algorithm, convert your NFA to an equivalent DFA.

5. Construct a 4-state NFA that defines the set of strings over {`b`, `c`} that contains at least one occurrence of `bcc`. Using the subset algorithm, convert your NFA to an equivalent DFA.

6. Construct a DFA that defines the set of strings that contain at least one occurrence or `bcc` and at least one occurrence of `ccb`.

7. Construct a DFA that defines C-type strings. In C-type strings, embedded quotes and line separators are permitted as long as they are immediately preceded by a backslash. Thus, the string

   ```
   "A\"B\\C\
   D"
   ```

   is legal. Convert your DFA to a regular expression.

8. Convert the following grammar to an NFA. Then convert your NFA to a DFA using the subset algorithm.

   ```
   S → bS
   S → bB
   B → bB
   B → cC
   C → c
   ```

9. Convert the following grammar to a NFA. Then convert your NFA to a DFA using the subset algorithm.

   ```
   S → bB
   B → bS
   B → b
   B → cC
   C → cC
   C → λ
   ```

10. Convert the DFA in Fig. 17.11 to a regular grammar.

11. Convert the DFA in Fig. 17.14 to a regular grammar.

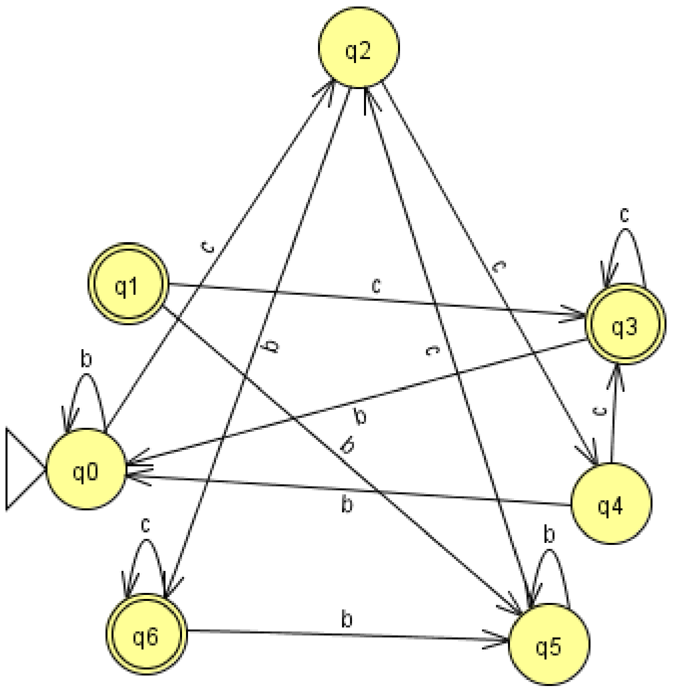
12. Using the construction technique in Section 17.10, construct the NFA for `b|c|d`.

13. Using the construction technique in Section 17.10, construct the NFA for `b***`.

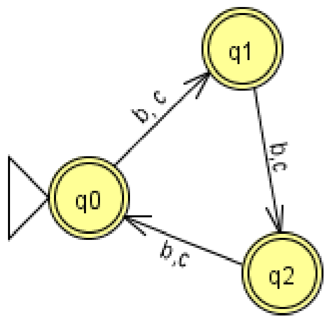
14. Using the construction technique in Section 17.10, construct the NFA for `bcd`.

15. Using the construction technique in Section 17.10, construct the NFA for `b*|(c*)|d*`.

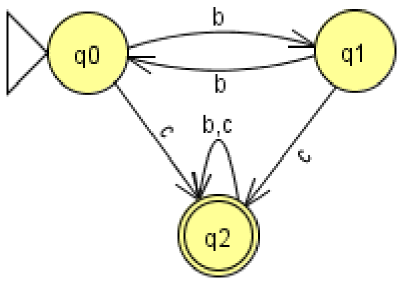
16. Construct the minimal DFA for



17. Construct the minimal DFA for



18. Construct the minimal DFA for



19. Prove that the set of all strings over {b, c} in which the number of b's equals the number of c's is not regular.

20. Show that the regular languages are closed under union, intersection, and complementation.

21. Prove that $\{b^ic^j : i \neq j\}$ is not regular. Hint: take the complement of this language and intersect it with `b*c*`. What language do you get? Also see problem 20.

22. Prove that $\{b^ic^j : i > j\}$ is not regular.

23. Prove that $\{b^i : i$ is a perfect square$\}$ is not regular.

24. Prove that $\{b^i : i$ is a prime$\}$ is not regular.

25. Prove that the union of a non-regular language and a finite language is always non-regular. That is, you can never change a non-regular language into a regular one by adding only a finite number of strings.

Chapter 18 PROBLEMS

1. Implement G1. Test your program with `b`, `bc`, `b|c`, `b*`, `((b))`, `bc|b*`, `b|`, `*b`, `b)`, `(b`, `b|*c`, `λ.` Be sure to surround the regular expressions that contain the star operator with quotes when you enter them on the command line. Also enter `b\` (`b` followed by a backslash), both with and without surrounding quotes.

2. Implement `G2`. Test your program with the regular expressions from Problem 18.1.

3. Implement `G3`. Test your program by entering

```
java G3 xyz grep.txt > p1803a.txt
java G3 "n.t|bo*t" grep.txt > p1803b.txt
java G3 "b\*t" grep.txt > p1803c.txt
```

`grep.txt` is in the J1 Software Package. Hand in your `G3` program, `p1803a.txt`, `p1803b.txt`, and `p1803c.txt`.

4. In the implementation of grep described in this chapter, every state has an `acceptState` field. An alternate approach is to represent an NFA with an object that has both the start state and the accept state. Then each state would not have to have an `acceptState` field. Rewrite your implementation of `G3` using this approach. How does it compare with the original approach?

5. Add support to your implementation of `G3` for the `+` (one or more) and `?` (zero or one) operators. Call this version `G4`. Test your program by entering

```
java G4 xyz grep.txt > p1805a.txt
java G4 "n.t|bo*t" grep.txt > p1805b.txt
java G4 "b\*t" grep.txt > p1805c.txt
java G4 "n.?t grep.txt > p1805d.txt
java G4 "bo+t" grep.txt > p1805e.txt
```

`grep.txt` is in the J1 Software Package. Hand in your `G4` program, `p1805a.txt`, `p1805b.txt`, `p1805c.txt`, `p1805d.txt`, and `p1805e.txt`.

6. Add support of the command line arguments `-v` (invert match) and `-n` (line numbers) to your `G3` program. Call this version `G5`. If `-v` is specified, `G5` displays every line *not* matched by the given regular expression. If `-n` is specified, `G5` displays the line number within the input file of every line displayed. Test your program by entering

```
java G5 -v -n xyz grep.txt > p1806a.txt
java G5 -n -v xyz grep.txt > p1806b.txt
```

`grep.txt` is in the J1 Software Package. Hand in your `G5` program, `p1806a.txt`, and `p1806b.txt`.

7. Would it make sense to convert the NFA that your grep program creates to a DFA and then use the DFA for pattern matching? How would you represent the DFA?

8. Would it be better to use an array representation of the NFA rather than a linked structure? Describe the structure of the array that you would use.

9. Implement a grep program that supports JavaCC-type regular expressions.

10. Using the standard grep program (for example, the grep on a Linux system), determine what strings are matched by the following regular expressions:

    `b\|c` (do not enter with enclosing quotes)
    `"b\|c"` (enter with enclosing quotes)
    `"b|c"` (enter with enclosing quotes)

    Now repeat using the standard egrep program.

Chapter 19 Problems

1. Implement the hand-written `R1` compiler or the equivalent JavaCC-generated `R1j` compiler. `R1` is `S1` modified to generate code in the register instruction set. Test your compiler by entering

   ```
   javac R1.java    or          javacc R1j.jj
   java R1 S1                    javac  R1j.java
   a S1.a                        java R1j S1
   e S1 /c                       a S1.a
                                 e S1 /c
   ```

   Start by copying `S1.java` to `R1.java` or `S1j.jj` to `R1j.jj`. Then modify `R1.java` or `R1j.jj`. Because you are compiling `S1.s`, the performance statistics that the e program generates will be relative to the S1 compiler. You will probably find that R1 is over the limit in size but under the limit in execution time.

2. Compare your `S1` (or `S1j`) compiler with your `R1` (or `R1j`) compiler with respect to size and execution time of the generated code. Does `R1` generates inefficient code relative to the code generated by S1?

3. Do temps have to be unique? Or can they be re-used?

4. What is the minimum number of temps required for each of the following statements:

   ```
   x = b + c + d + e + f;
   x = b + (c + (d + (e + f)));
   ```

5. Implement `R2` or `R2j`, the register instruction version of `S2` or `S2j`. Test your program with `S2.s`.

6. Implement `R3` or `R3j`, the register instruction version of `S3` or `S3j`. Test your program with `S3.s`.

7. Implement `R4` or `R4j`, the register instruction version of `S4` or `S4j`. Test your program with `S4.s`.

8. Implement `R5` or `R5j`, the register instruction version of `S5` or `S5j`. Test your program with `S5a.s` and `S5b.s`. Be sure to link with `rup.o`, not `sup.o`. `rup.o` is the start-up code for the register instruction set.

9. Implement `R6` or `R6j`, the register instruction version of `S6` or `S6j`. Test your program with `S6a.s` and `S6b.s`. Be sure to link with `rup.o`, not `sup.o`. `rup.o` is the start-up code for the register instruction set.

10. Implement the register instruction set version of `G1`. Test as specified in problem 1 inf Chapter 18.

11. Implement the register instruction set version of `G2`. Test as specified in problem 2 in Chapter 18.

12. Implement the register instruction set version of `G3`. Test as specified in problem 3 in Chapter 18.

Chapter 20 PROBLEMS

1. Implement R1a by extending R1 so that it replaces `ld` with `ldc` wherever possible. Test your compiler with `S1.s`.

2. Implement R1b by extending R1a so that it supports the re-use of temps. Test your compiler with `S1.s`.

3. Implement R1c by extending R1b so that it supports constant folding. Test your compiler with `S1.s`.

4. Implement R1d by extending R1c so that it supports register allocation. Test your compiler with `S1.s`.

5. Implement R1e by extending R1c (not R1d) so that it supports peephole optimization. Test your compiler with `S1.s`

6. Incorporate the following optimizations into the S1 compiler:

   a) Use the `pc` instruction in place of `pwc` instruction wherever possible.

   b) Use the `p` instruction in place of those instances of the `pwc` instruction not replaced by the pc instruction. For example,

   ```
   pwc      -5
   ```

   should be replaced with

   ```
   p        @_5
   ```

   where `@_5` is defined with

   ```
   @_5:      dw        -5
   ```

   c) Constant folding

   Call this compiler S1c.

7. Create a chart that shows the size and execution time of the `S1.s` program as created by the S1, S1c (see Problem 6), R1, R1a, R1b, R1c, R1d, and R1e compilers. How do the S1 and S1c compilers compare with the register instruction set compilers?

8. Suppose the following code is compiled with a compiler that uses the register allocation technique described in Section 20.5:

```
x = 7;
while(x - 5)
{
   x = x - 1;   // ac variable has index of x
}
```

```
y = x;           // no ld x emitted
```

After the assignment statement in the `while` loop is compiled , the `ac` variable used in register allocation will contain the index of `x`. If this value is still in the `ac` variable when the assignment statement following the `while` loop is compiled, the compiler will not emit a `ld` instruction that loads from `x` for this statement (because the `ac` variable indicates that the value of `x` is already in the `ac` register). But will the `ac` register have the value of `x` at this point during execution time? Describe how you would modify the register allocation scheme so that `ld` instructions are emitted whenever they are necessary.

9. In the register instruction set, why is it better to use a `ld` instruction with a `dw` than a `ldw` instruction without a `dw`? Similarly, in the stack instruction set, why is it better to use a `p` instruction with a `dw` than a `pwc` instruction without a `dw`?

10. What code is generated by `R1c` for

```
x = 2 + 3 + y;
x = 2 + y + 3;
x = y + 2 + 3;
```

11. Extend the constant folding mechanism in R1c so it can fold variables whose values can be determined at compiler time. For example, the assembly code for

```
x = 5;
y = x + 7;
```

should be

```
ldc      5
st       x
ldc      12   ; x and 7 folded to get 12
st       y
```

In this example, the compiler can determine the value of `x` in the second assignment statement. Thus, it folds this value with 7, to get 12. Call your compiler R1cc. Test your compiler with `S1.s`. How does it compare with R1c?

12. The code R1d produces for the following program

```
x = 3;
x = x + 5;
```

is

```
            ldc      3         ; version 1
            st       x
            add      @5
            st       x
            halt
x:          dw       0
```

```
@5:          dw       5
```

Because `x` is assigned the constant 3 before it is used, would it make sense to initialize `x` to 3 in its `dw` and then dispense with the instruction that stores into `x`? If the compiler did this, it would generate

```
             ld       x         ; version 2
             add      @5
             st       x
             halt
x:           dw       3
@5:          dw       5
```

Serially reusable code is code that be used by multiple users as long as one user finishes before the next one starts. In other words, a fresh copy of the program does not have to be loaded into memory for each user. Which version above is serially reusable?

==

In the following problems, the letter in a compiler's name indicates the optimizations supported:

a	use of `ldc` in place of `ld` wherever possible
b	a optimizations plus re-use of temps
c	b optimizations plus constant folding
d	c optimizations plus register allocation
e	c optimizations plus peephole optimization

The number in a compiler's name indicates the level of language support. 1, 2, 3, 4, and 5 corresponds to the levels supported by the S1, S2, S3, S4, and S5 compilers, respectively.

13. What assembly code should be generated by R4c for

```
x = 5;
while (x)
{
   x = x - 1;
   y = y + 1;

}
x = y;
do
{
   y = y - 1;
   x = x - 1;
} while (x);
```

14. What assembly code should be generated by R4c for

```
if (x)
    y = 3;
```

```
z = y;
if (x)
   y = 4;
else
   x = y;
```

15. What assembly code should be generated by R5c for

```
extern int x, y;
void f()
{
      x = 5;
      g();
      y = x;
}
```

16. Implement R2a, R2b, R2c, R2d, and R2e compilers. Test your compilers with `S2.s`.

17. Implement R3a, R3b, R3c, R3d, and R3e compilers. Test your compilers with `S3.s`.

18. Implement R4a, R4b, R4c, R4d, and R4e compilers. Test your compilers with `S4.s`.

19. Implement R5a, R5b, R5c, R5d, and R5e compilers. Test your compilers with `S5a.s` and `S5b.s`.

20. Implement R6a, R6b, R6c, R6d, and R6e compilers. Test your compilers with `S6a.s` and `S6b.s`.

21. What is line 12 in Fig. 20.9 executed only for temps?

Chapter 22 PROBLEMS

1. Construct the SLR(1) finite automaton for G22.9 and its corresponding parse table. Show the parse of `b`, `b+b`, `b+b+b`, `b++`, `+b`, and `b+b+`.

2. Construct the SLR(1) finite automaton for G22.10. Modify the parse table so that it always shifts at a shift/reduce conflict. Which strings in the language of the grammar can the corresponding parser successfully parse? Which strings can it not parse successfully?

3. Construct the SLR(1) finite automaton for G22.12 and its corresponding parse table. Are there any conflicts?

4. Construct the SLR(1) parsing table for

```
E → E + T
E → T
T → T * F
T → F
F → b
F → ( E )
```

   Show the parse of `b*(b+b)`. Implement your parser using Java. Test your parser with `b+b`, `b*b`, `b+b*b`, `b*(b+b)`, `bb`, `b*`, `b**b`, `b*b+`.

5. Using Java implement the SLR(1) parser for

```
E → E + <UNSIGNED>
E → <UNSIGNED>
```

   Add actions along with any necessary data structures that allow your parser to compute the value of the input expression. Test your program with `2`, `2+3`, and `2+3+4`.

6. Construct the SLR(1) parser for

```
S → b
S → Sd
S → Bde
B → b
```

   Are there any conflicts? Construct the LR(1) parser for the grammar. Are there any conflicts? Is this grammar SLR(1)? Is it LR(1)?

7. Show that the merging of states in the process of constructing a LALR(1) parser never creates a shift/reduce conflict.

8. Construct the SLR(1) parse table for G18.1 in Chapter 18.

9. Are LR(1) grammars ever ambiguous? Justify your answer.

10. Construct the LR(1) finite automata and the LALR(1) finite automaton for

```
S → BB
B → Bb
B → c
D → Dd
D → e
```

Chapter 23 PROBLEMS

1. In a yacc-generated parser, a value can be passed up the parse tree and down the parse tree (see Section 23.6). Can a value be passed across the parse tree? That is, can a value associated with a symbol in a production be passed to a symbol or an action to the right in the same production?

2. Implement the yacc version of the S2 compiler.

3. Implement the yacc version of the S3 compiler.

4. Implement the yacc version of the S4 compiler.

5. Implement the yacc version of the S5 compiler.

6. Implement the yacc version of the G3 grep program in Chapter 18.

7. Implement the yacc version of the R1 compiler.

8. Implement the yacc version of the R1a compiler.

9. Implement the yacc version of the R1b compiler.

10. Implement the yacc version of the R1c compiler.

11. Implement the yacc version of the R1d compiler.

12. Implement the yacc version of the I1 compiler.

13. Implement the yacc version of the I2 compiler.

14. Implement the yacc version of the I3 compiler.

15. Implement the yacc version of the I4 compiler.

16. Implement the yacc version of the CI1 interpreter.

17. Implement the yacc version of the CI2 interpreter.

18. Implement the yacc version of the CI3 interpreter.

19. Implement the yacc version of the CI4 interpreter.

20. Create the parse table for the following grammar:

```
S → bSc
S → λ
```

Determine the parse table yacc creates for this grammar by invoking yacc with the `-v` command line

argument.
Compare your parse table with yacc's table.

21. On line 52 in In Fig. 23.5, `yylval` is set to null if the current token is not an unsigned integer. Would it be better if `yylval` were assigned a reference to a `parserVal` object?

22. Are the values assigned to `endLine` and `endColumn` ever used in `S1y.y` (Fig. 23.11)?

23. On line 96 in Fig. 23.11, can `yytext` be used in place of `$1.sval`?

24. Create a program with jflex that outputs a file identical to the input file except that each run of spaces is replaced by a single space. Test your program with the input file `p2324.txt` in the J1 Software Package.

25. Create a program with jflex that capitalizes the first letter of every sentence. Test your program with the input file `p2325.txt` in the J1 Software Package.

26. Create a program with jflex that displays the character count, word count, and line count for the input file. Test your program with the input file `p2326.txt` in the J1 Software Package.

27. Create a program with jflex that displays the percent of words in the input file that contain at least one letter "e". Test your program with the input file `p2327.txt` in the J1 Software Package.