Both scanners and parsers can be built by hand if an automatic tool is not available. Hand-built scanners are simple enough to be relatively common. Hand-built parsers are generally limited to top-down recursive descent, and are generally used only for comparatively simple languages (e.g., Pascal but not Ada). Automatic generation of the scanner and parser has the advantage of increased reliability, reduced development time, and easy modification and enhancement.

Various features of language design can have a major impact on the complexity of syntax analysis. In many cases, features that make it difficult for a compiler to scan or parse also make it difficult for a human being to write correct, maintainable code. Examples include the lexical structure of Fortran and the `if ... then ... else` statement of languages like Pascal. This interplay among language design, implementation, and use will be a recurring theme throughout the remainder of the book.

2.6 Exercises

2.1 Write regular expressions to capture

(a) Strings in C. These are delimited by double quotes (`"`), and may not contain newline characters. They may contain double quote or backslash characters if and only if those characters are "escaped" by a preceding backslash. You may find it helpful to introduce shorthand notation to represent any character that is *not* a member of a small specified set.

(b) Comments in Pascal. These are delimited by `(*` and `*)`, as shown in Figure 2.6, or by `{` and `}`.

(c) Floating-point constants in Ada. These are the same as in Pascal (see the definition of *unsigned_number* in Example 2.2 [page 41]), except that (1) an underscore is permitted between digits, and (2) an alternative numeric base may be specified by surrounding the non-exponent part of the number with pound signs, preceded by a base in decimal (e.g., `16#6.a7#e+2`). In this latter case, the letters `a..f` (both upper- and lowercase) are permitted as digits. Use of these letters in an inappropriate (e.g., decimal) number is an error but need not be caught by the scanner.

(d) Inexact constants in Scheme. Scheme allows real numbers to be explicitly *inexact* (imprecise). A programmer who wants to express all constants using the same number of characters can use sharp signs (`#`) in place of any lower-significance digits whose values are not known. A base-ten constant without exponent consists of one or more digits followed by zero of more sharp signs. An optional decimal point can be placed at the beginning, the end, or anywhere in between. (For the record, numbers in Scheme are actually a good bit more complicated than this. For the purposes of this exercise, please ignore anything you may know about sign, exponent, radix, exactness and length specifiers, and complex or rational values.)

(e) Financial quantities in American notation. These have a leading dollar sign (`$`), an optional string of asterisks (`*`—used on checks to discourage fraud), a string of decimal digits, and an optional fractional part consisting of a decimal point (`.`) and two decimal digits. The string of digits to the left of the decimal point may consist of a single zero (`0`). Otherwise it must not start with a zero. If there are more than three digits to the left of the decimal point, groups of three (counting from the right) must be separated by commas (`,`). Example: `$**2,345.67`. (Feel free to use "productions" to define abbreviations, so long as the language remains regular.)

2.2 Show (as "circles-and-arrows" diagrams) the finite automata for parts (a) and (c) of Exercise 2.1.

2.3 Build a regular expression that captures all nonempty sequences of letters other than `file`, `for`, and `from`. For notational convenience, you may assume the existence of a **not** operator that takes a set of letters as argument and matches any *other* letter. Comment on the practicality of constructing a regular expression for all sequences of letters other than the keywords of a large programming language.

2.4 (a) Show the NFA that results from applying the construction of Figure 2.8 to the regular expression *letter* (*letter* | *digit*)*.

(b) Apply the transformation illustrated by Example 2.12 to create an equivalent DFA.

(c) Apply the transformation illustrated by Example 2.13 to minimize the DFA.

2.5 Build an ad hoc scanner for the calculator language. As output, have it print a list, in order, of the input tokens. For simplicity, feel free to simply halt in the event of a lexical error.

2.6 Build a nested-`case`-statements finite automaton that converts all letters in its input to lowercase, except within Pascal-style comments and strings. A Pascal comment is delimited by `{` and `}`, or by `(*` and `*)`. Comments do not nest. A Pascal string is delimited by single quotes (`'` ... `'`). A quote character can be placed in a string by doubling it (`'Madam, I''m Adam.'`). This upper-to-lower mapping can be useful if feeding a program written in standard Pascal (which ignores case) to a compiler that considers upper- and lowercase letters to be distinct.

2.7 Give an example of a grammar that captures right associativity for an ex ponentiation operator (e.g., `**` in Fortran).

2.8 Prove that the following grammar is LL(1).

$$\begin{array}{lcl}
\textit{decl} & \longrightarrow & \texttt{ID}\ \textit{decl\_tail} \\
\textit{decl\_tail} & \longrightarrow & \texttt{,}\ \textit{decl} \\
 & \longrightarrow & \texttt{:}\ \texttt{ID}\ \texttt{;}
\end{array}$$

(The final `ID` is meant to be a type name.)

2.9 Consider the following grammar.

$$\begin{array}{lcl}
G & \longrightarrow & S\ \texttt{\$\$} \\
S & \longrightarrow & A\ M \\
M & \longrightarrow & S \mid \epsilon \\
A & \longrightarrow & \texttt{a}\ E \mid \texttt{b}\ A\ A \\
E & \longrightarrow & \texttt{a}\ B \mid \texttt{b}\ A \mid \epsilon \\
B & \longrightarrow & \texttt{b}\ E \mid \texttt{a}\ B\ B
\end{array}$$

(a) Describe in English the language that the grammar generates.

(b) Show a parse tree for the string `a b a a`.

(c) Is the grammar LL(1)? If so, show the parse table; if not, identify a prediction conflict.

2.10 Consider the language consisting of all strings of properly balanced parentheses and brackets.

(a) Give LL(1) and SLR(1) grammars for this language.

(b) Give the corresponding LL(1) and SLR(1) parsing tables.

(c) For each grammar, show the parse tree for `([]([]))[](())`.

(d) Give a trace of the actions of the parsers on this input.

2.11 Give an example of a grammar that captures all the levels of precedence for arithmetic expressions in C. (*Hint*: This exercise is somewhat tedious. You probably want to attack it with a text editor rather than a pencil, so you can cut, paste, and replace. You can find a summary of C precedence in Figure 6.1 [page 237]; you may want to consult a manual for further details.)

2.12 Extend the grammar of Figure 2.24 to include `if` statements and `while` loops, along the lines suggested by the following examples.

```
abs := n
if n < 0 then abs := 0 - abs fi

sum := 0
read count
while count > 0 do
    read n
    sum := sum + n
    count := count - 1
od
write sum
```

3.3 Give two examples in which it might make sense to delay the binding of an implementation decision, even though sufficient information exists to bind it early.

3.4 Give three concrete examples drawn from programming languages with which you are familiar in which a variable is live but not in scope.

3.5 Consider the following pseudocode, assuming nested subroutines and static scope.

```
procedure main
    g : integer

    procedure B(a : integer)
        x : integer

        procedure A(n : integer)
            g := n

        procedure R(m : integer)
            write_integer(x)
            x /:= 2 -- integer division
            if x > 1
                R(m + 1)
            else
                A(m)

        -- body of B
        x := a × a
        R(1)

    -- body of main
    B(3)
    write_integer(g)
```

(a) What does this program print?

(b) Show the frames on the stack when A has just been called. For each frame, show the static and dynamic links.

(c) Explain how A finds g.

3.6 As part of the development team at MumbleTech.com, Janet has written a list manipulation library for C that contains, among other things, the code in Figure 3.18.

(a) Accustomed to Java, new team member Brad includes the following code in the main loop of his program.

```
list_node *L = 0;
while (more_widgets()) {
    insert(next_widget(), L);
}
L = reverse(L);
```

3.7 Rewrite Figures 3.7 and 3.8 in C.

3.8 Modula-2 provides no way to divide the header of a module into a public part and a private part: everything in the header is visible to the users of the module. Is this a major shortcoming? Are there disadvantages to the public/private division (e.g., as in Ada)? (For hints, see Section 9.2.)

3.9 Consider the following fragment of code in C.

```
{   int a, b, c;
    ...
    {   int d, e;
        ...
        {   int f;
            ...
        }
        ...
    }
    ...
    {   int g, h, i;
        ...
    }
    ...
}
```

Assume that each integer variable occupies four bytes. How much total space is required for the variables in this code? Describe an algorithm that a compiler could use to assign stack frame offsets to the variables of arbitrary nested blocks, in a way that minimizes the total space required.

3.10 Consider the design of a Fortran 77 compiler that uses static allocation for the local variables of subroutines. Expanding on the solution to the previous question, describe an algorithm to minimize the total space required for these variables. You may find it helpful to construct a *call graph* data structure in which each node represents a subroutine and each directed arc indicates that the subroutine at the tail may sometimes call the subroutine at the head.

3.11 Consider the following pseudocode.

```
procedure P(A, B : real)
    X : real

    procedure Q(B, C : real)
        Y : real
        ...

    procedure R(A, C : real)
        Z : real
        ...                        -- (*)
    ...
```

Assuming static scope, what is the referencing environment at the location marked by (*)?

3.12 Write a simple program in Scheme that displays three different behaviors, depending on whether we use `let`, `let*`, or `letrec` to declare a given set of names. (*Hint*: To make good use of `letrec`, you will probably want your names to be functions [`lambda` expressions].)

3.13 Consider the following pseudocode.

```
x : integer        -- global

procedure set_x(n : integer)
    x := n

procedure print_x
    write_integer(x)

procedure first
    set_x(1)
    print_x

procedure second
    x : integer
    set_x(2)
    print_x

set_x(0)
first()
print_x
second()
print_x
```

What does this program print if the language uses static scoping? What does it print with dynamic scoping? Why?

3.14 Consider the programming idiom illustrated in Example 3.20. One of the reviewers for this book suggests that we think of this idiom as a way to implement a central reference table for dynamic scope. Explain what is meant by this suggestion.

3.15 If you are familiar with structured exception-handling, as provided in Ada, Modula-3, C++, Java, C#, ML, Python, or Ruby, consider how this mechanism relates to the issue of scoping. Conventionally, a `raise` or `throw` statement is thought of as referring to an exception, which it passes as a parameter to a handler-finding library routine. In each of the languages mentioned, the exception itself must be declared in some surrounding scope, and is subject to the usual static scope rules. Describe an alternative point of view, in which the `raise` or `throw` is actually a reference to a *handler*, to which it transfers control directly. Assuming this point of view, what are the scope rules for handlers? Are these rules consistent with the rest of the language? Explain. (For further information on exceptions, see Section 8.5.)

3.16 Consider the following pseudocode.

```
x : integer        -- global

procedure set_x(n : integer)
    x := n

procedure print_x
    write_integer(x)

procedure foo(S, P : function; n : integer)
    x : integer := 5
    if n in {1, 3}
        set_x(n)
    else
        S(n)
    if n in {1, 2}
        print_x
    else
        P

set_x(0); foo(set_x, print_x, 1); print_x
set_x(0); foo(set_x, print_x, 2); print_x
set_x(0); foo(set_x, print_x, 3); print_x
set_x(0); foo(set_x, print_x, 4); print_x
```

Assume that the language uses dynamic scoping. What does the program print if the language uses shallow binding? What does it print with deep binding? Why?

3.17 Consider the following pseudocode.

```
x : integer := 1
y : integer := 2

procedure add
    x := x + y

procedure second(P : procedure)
    x : integer := 2
    P()

procedure first
    y : integer := 3
    second(add)

first()
write_integer(x)
```

(a) What does this program print if the language uses static scoping?

op $\in \{<, \le, =, \ne, >, \ge\}$. Suppose we subtract B from A, using two's complement arithmetic. For each of the six conditions, indicate the logical combination of condition-code bits that should be used to trigger the branch. Repeat the exercise on the assumption that A and B are signed, two's complement numbers.

5.6 We implied in Section 5.4.1 that if one adds a new instruction to a non-pipelined, microcoded machine, the time required to execute that instruction is (to first approximation) independent of the time required to execute all other instructions. Why is it not strictly independent? What factors could cause overall execution to become slower when a new instruction is introduced?

5.7 Suppose that loads constitute 25% of the typical instruction mix on a certain machine. Suppose further that 15% of these loads miss in the on-chip (primary) cache, with a penalty of 40 cycles to reach main memory. What is the contribution of cache misses to the average number of cycles per instruction? You may assume that instruction fetches always hit in the cache. Now suppose that we add an off-chip (secondary) cache that can satisfy 90% of the misses from the primary cache, at a penalty of only 10 cycles. What is the effect on cycles per instruction?

5.8 Many recent processors provide a *conditional move* instruction that copies one register into another if and only if the value in a third register is (or is not) equal to zero. Give an example in which the use of conditional moves leads to a shorter program.

5.9 The 64-bit AMD Opteron architecture is backward compatible with the x86 instruction set, just as the x86 is backward compatible with the 16-bit 8086 instruction set. Less transparently, the IA-64 Itanium is capable of running legacy x86 applications in "compatibility mode." But recent members of the ARM and MIPS processor families support *new* 16-bit instructions as an *extension* to the architecture. Why might designers have chosen to introduce these new, less powerful modes of execution?

5.10 Consider the following code fragment in pseudo-assembler notation.

```
1.    r1 := K
2.    r4 := &A
3.    r6 := &B
4.    r2 := r1 × 4
5.    r3 := r4 + r2
6.    r3 := *r3           -- load (register indirect)
7.    r5 := *(r3 + 12)    -- load (displacement)
8.    r3 := r6 + r2
9.    r3 := *r3           -- load (register indirect)
10.   r7 := *(r3 + 12)    -- load (displacement)
11.   r3 := r5 + r7
12.   S := r3             -- store
```

(a) Give a plausible explanation for this code (what might the corresponding source code be doing?).

(b) Identify all flow, anti-, and output dependences.

(c) Schedule the code to minimize load delays on a single-pipeline, in-order processor.

(d) Can you do better if you rename registers?

5.11 With the development of deeper, more complex pipelines, delayed loads and branches have become significantly less appealing as features of a RISC instruction set. Why is it that designers have been able to eliminate delayed loads in more recent machines, but have had to retain delayed branches?

5.12 Some processors, including the PowerPC and recent members of the x86 family, require one or more cycles to elapse between a condition-determining instruction and a branch instruction that uses that condition. What options does a scheduler have for filling such delays?

5.13 Branch prediction can be performed statically (in the compiler) or dynamically (in hardware). In the static approach, the compiler guesses which way the branch will usually go, encodes this guess in the instruction, and schedules instructions for the expected path. In the dynamic approach, the hardware keeps track of the outcome of recent branches, notices branches or patterns of branches that recur, and predicts that the patterns will continue in the future. Discuss the tradeoffs between these two approaches. What are their comparative advantages and disadvantages?

5.14 Consider a machine with a three-cycle penalty for incorrectly predicted branches and a zero-cycle penalty for correctly predicted branches. Suppose that in a typical program 20% of the instructions are conditional branches, which the compiler or hardware manages to predict correctly 75% of the time. What is the impact of incorrect predictions on the average number of cycles per instruction? Suppose the accuracy of branch prediction can be increased to 90%. What is the impact on cycles per instruction?

Suppose that the number of cycles per instruction would be 1.5 with perfect branch prediction. What is the percentage slowdown caused by mispredicted branches? Now suppose that we have a superscalar processor on which the number of cycles per instruction would be 0.6 with perfect branch prediction. Now what is the percentage slowdown caused by mispredicted branches? What do your answers tell you about the importance of branch prediction on superscalar machines?

5.15 Consider the code in Figure 5.6. In an attempt to eliminate the remaining delay and reduce the overhead of the bookkeeping (loop control) instructions, one might consider *unrolling* the loop—that is, creating a new loop in which each iteration performs the work of k iterations of the original loop. Show the code for $k = 2$. You may assume that n is even and that your target