(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 nonexponent 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 lower case) 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.

Answer:

$$\textit{Ada\_int} \longrightarrow \textit{digit}\ ((\ \texttt{\_} \mid \epsilon)\ \textit{digit})^*$$
$$\textit{extended\_digit} \longrightarrow \textit{digit} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f} \mid \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F}$$
$$\textit{Ada\_extended\_int} \longrightarrow \textit{extended\_digit}\ ((\ \texttt{\_} \mid \epsilon)\ \textit{extended\_digit})^*$$
$$\begin{aligned}\textit{Ada\_FP\_num} \longrightarrow\ & ((\textit{Ada\_int}\ ((\ \texttt{.}\ \textit{Ada\_int} \mid \epsilon))\\ & \mid (\textit{Ada\_int}\ \texttt{\#}\ \textit{Ada\_extended\_int}\\ & ((\ \texttt{.}\ \textit{Ada\_extended\_int}) \mid \epsilon)\ \texttt{\#}))\\ & (((\texttt{e} \mid \texttt{E})(\texttt{+} \mid \texttt{-} \mid \epsilon)\ \textit{Ada\_int}) \mid \epsilon)\end{aligned}$$

(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 complicatedthan 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.)

Answer: $\textit{digit}^+\ \texttt{\#}^*\ (\ \texttt{.}\ \ \texttt{\#}^* \mid \epsilon) \mid \textit{digit}^*\ \texttt{.}\ \textit{digit}^+\ \texttt{\#}^*$

(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.)

Answer:

$$\textit{nzdigit} \longrightarrow \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{4} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$$
$$\textit{digit} \longrightarrow \texttt{0} \mid \textit{nzdigit}$$
$$\textit{group} \longrightarrow \texttt{,}\ \textit{digit}\ \textit{digit}\ \textit{digit}$$
$$\textit{number} \longrightarrow \texttt{\$*}^*(\texttt{0} \mid \textit{nzdigit}\ (\epsilon \mid \textit{digit} \mid \textit{digit}\ \textit{digit})\ \textit{group}^*)\ (\epsilon \mid \texttt{.}\ \textit{digit}\ \textit{digit})$$

```
            case incomment2 :
               switch (c) {
                  case '*' :
                     state = Rcomment;
                     break;
                  default : break;
                     /* state remains incomment2 */
               } break;
            case Rcomment :
               switch (c) {
                  case ')' :
                     state = start;
                     mapping = true;
                     break;
                  case '*' : break;
                     /* state remains Rcomment */
                  default :
                     state = incomment2;
               } break;
         }
         if (mapping && (('A' <= c) && ('Z' >= c)))
            putchar(c - 'A' + 'a');
         else putchar(c);
      }
   }
```

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

Answer: Here is a modified version of the LR expression grammar from Section 2.1.3:

1. *expression* $\longrightarrow$ *term* | *expression add_op term*
2. *term* $\longrightarrow$ *factor* | *term mult_op factor*
3. *factor* $\longrightarrow$ *primary* | *primary* `**` *factor*
4. *primary* $\longrightarrow$ *identifier* | *number* | `-` *primary* | `(` *expression* `)`
5. *add_op* $\longrightarrow$ `+` | `-`
6. *mult_op* $\longrightarrow$ `*` | `/`

Note that the second alternative of production 3 is right recursive, whereas the corresponding parts of productions 1 and 2 are left recursive.

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

decl $\longrightarrow$ `ID` *decl_tail*
decl_tail $\longrightarrow$ `,` *decl*
$\longrightarrow$ `: ID ;`

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

Answer: By definition, a grammar is LL(1) if it can be parsed by an LL(1) parser. It can be parsed by an LL(1) parser if no conflicts arise in the creation of the parse table. In this grammar, no symbols generate ϵ, so the table can be built entirely from FIRST sets; FOLLOW sets do not matter. There is only one symbol, *decl_tail*, with more than one production, and the FIRST sets for the right-hand sides of those productions are distinct ({ , } and { ; }). Therefore no conflicts arise.

2.9 Consider the following grammar:

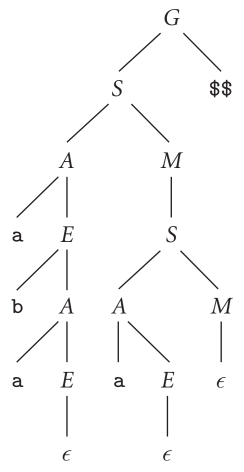
$$
\begin{array}{lcl}
G & \longrightarrow & S \ \$\$ \\
S & \longrightarrow & A \ M \\
M & \longrightarrow & S \mid \epsilon \\
A & \longrightarrow & \texttt{a} \ E \mid \texttt{b} \ A \ A \\
E & \longrightarrow & \texttt{a} \ B \mid \texttt{b} \ A \mid \epsilon \\
B & \longrightarrow & \texttt{b} \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

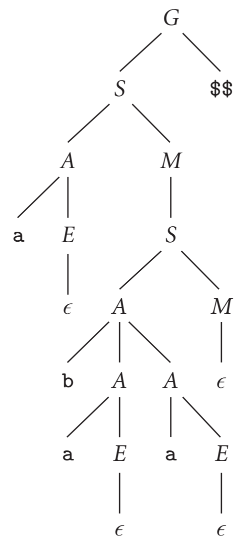
(a) Describe in English the language that the grammar generates.

(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.

Answer: The grammar generates all strings of `a`'s and `b`'s (terminated by an end marker), in which there are more `a`'s than `b`'s. There are two possible parse trees for the specified string:





The grammar is ambiguous, and hence not LL(1). A top-down parser would be unable to decide whether to predict an epsilon production when E is at the top of the stack.

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.

Local variables in Fortran 77 may be allocated on the stack, rather than in static memory, in order to minimize memory footprint and to facilitate interoperability with standard tools (e.g., debuggers).

Various aspects of code generation may be delayed until link time in order to facilitate whole-program code improvement.

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.

Answer: Here are a few possibilities:

(a) In Ada, if procedure `P` declares a local variable named `x`, then a global variable also named `x` will be live but not in scope when executing `P`.

(b) In Modula-2, a global variable declared in a module is live but not in scope when execution is not inside the module.

(c) In C, a static variable declared inside a function is live but not in scope when execution is not inside the function.

(d) in C++, non-public fields of an object of class `C` are live but not in scope when execution is not inside a method of `C`.

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

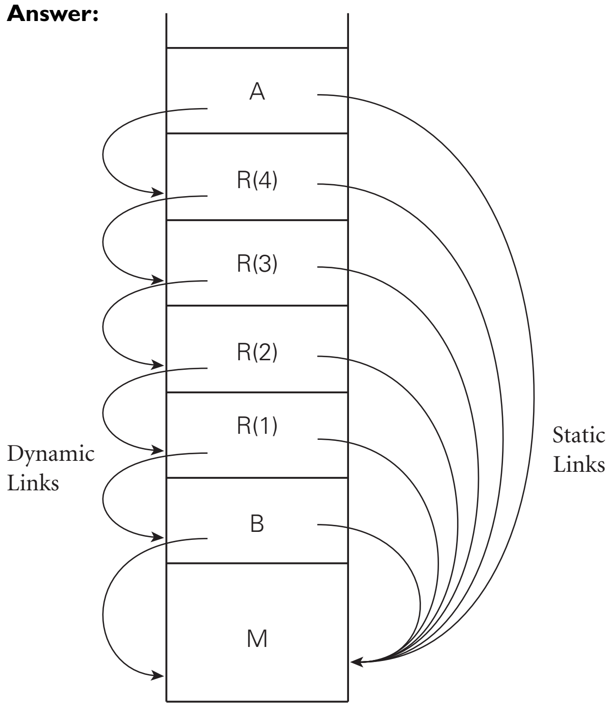
```
procedure main
    g : integer

    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?

Answer: `9 4 2 1 4`.

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

Answer:



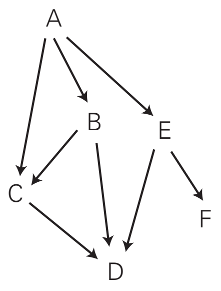
(c) Explain how A finds g.

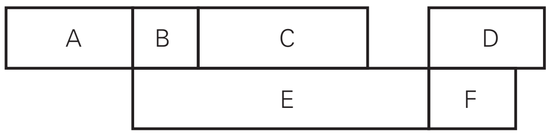
Answer: It dereferences its static link to find the stack frame of B. Within this frame it finds B's static link at a statically known offset. It dereferences that to find the stack frame of main, within which it finds g, again at a statically known offset. (This assumes that g is a local variable of main, *not* a global variable.)

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);
```





3.11 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 (*)?

Answer: P, B, X, Q, R, A (parameter to R), C (parameter to R), and Z. Note that the parameters and local variable of Q are not in scope, nor is P's parameter A.

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].)

Answer:

```
(let ((a (lambda (n) 1))
      (b (lambda (n) 2)))
  (let ((a (lambda (n) (if (zero? n) 3 (b (- n 1)))))
        (b (lambda (n) (if (zero? n) 4 (a (- n 1))))))
    (list (a 5) (b 5))))
```

The standard `let` evaluates the second element of all tuples before creating any new names, so the inner `a` and `b` both call the outer, constant functions, and the code evaluates to to `(2 1)`. If we replace the inner `let` with `let*`, then the inner `a` refers to the outer `b`, but the inner `b` refers to the *inner* `a`, and the code evaluates to `(2 2)`. If we replace the inner `let` with `letrec`, then the inner `a` and `b` refer to each other, and the code evaluates to `(4 3)`.

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.)

Answer: If we think of a `raise/throw` statement as containing a reference to a handler, then entry into a code block with a handler for exception *E* is very much like the elaboration of a dynamically scoped declaration. The new handler hides any previous handler for the same exception. The old handler "declaration" becomes visible again when execution leaves the nested scope.

Given that all the languages listed in the question use static scoping for all other names, the dynamic scoping explanation of handlers is not particularly appealing—hence the usual description in terms of static scoping within subroutines and "an exceptional return" across subroutines. Like the two descriptions of dynamic scope (Exercise 3.14), however, the two ways of modeling exceptions are equivalent.

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?

Answer: With shallow binding, `set_x` and `print_x` always access `foo`'s local `x`. The program prints

```
1 0  2 0  3 0  4 0
```

With deep binding, `set_x` accesses the global `x` when `n` is even and `foo`'s local `x` when `n` is odd. Similarly, `print_x` accesses the global `x` when `n` is 3 or 4 and `foo`'s local `x` when `n` is 1 or 2. The program prints

```
1 0  5 2  0 0  4 4
```

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?

(b) What does it print if the language uses dynamic scoping with deep binding?

(c) What does it print if the language uses dynamic scoping with shallow binding?

Answer: (a) 3; (b) 4; (c) 1.

3.18 In Section 3.6.3 we noted that while a single `min` function in Fortran would work for both integer and floating-point numbers, overloading would be more efficient, because it would avoid the cost of type conversions. Give an example in which overloading does not seem advantageous—one in which it makes more sense to have a single function with floating-point parameters, and perform coercion when integers are supplied.

Answer: The following function in C uses Heron's formula to compute the area of a triangle with sides of length `a`, `b`, and `c`:

```
double area(double a, double b, double c) {
    double s = (a + b + c) / 2;
    return sqrt(s * (s - a) * (s - b) * (s - c));
}
```

Answer: 16-bit mode in the ARM and MIPS product lines is intended to support the embedded market, where low power consumption and modest memory requirements are particularly important. Though the ARM was a 32-bit processor from the outset, and the MIPS has had 64-bit registers since 1991, designers realized that they could save significant amounts of energy and code space, at a comparatively modest cost in performance, by cutting back to 16 bits whenever possible. Applications can switch in and out of 16-bit mode dynamically, using smaller and simpler instructions for the less performance-critical portions of the code.

5.10 Consider the following code fragment in pseudoassembler 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?).

Answer: A and B are arrays of pointers to structures. Each pointer is four bytes long. We are interested in the value of fields at an offset of 12 bytes from the beginning of the structure. C source code might look something like `S := A[k]->f + B[k]->f;`.

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

Answer: There are flow dependences from instruction 1 to instruction 4 (r1); 2 to 5 (r4), 3 to 8 (r6), 4 to 5 and 4 to 8 (r2), 5 to 6 (r3), 6 to 7 (r3), 7 to 11 (r5), 8 to 9 (r3), 9 to 10 (r3), 10 to 11 (r7), and 11 to 12 (r3).

There are anti-dependences from 6 to 8, 9, and 11; 7 to 8, 9, and 11; 9 to 11; and 10 to 11. All of them are carried by r3.

There are output dependences from 5 to 6, 8, 9, and 11; 6 to 8, 9, and 11; 8 to 9 and 11; and 9 to 11. Again, all are carried by r3.

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

Answer: There are three load delays to worry about, after instructions 6, 9, and 10. The best we can do, unfortunately, is to move an instruction down into the delay slot of instruction 6 (which now becomes instruction 5):

```
1.    r1 := I
2.    r4 := &A
3.    r2 := r1 × 4
4.    r3 := r4 + r2
5.    r3 := *r3
6.    r6 := &B           –– filled this slot
7.    r5 := *(r3 + 12)
8.    r3 := r6 + r2
9.    r3 := *r3          –– still a delay after this load
10.   r7 := *(r3 + 12)   –– and this one
11.   r3 := r5 + r7
12.   S := r3
```

There will still be delays after instructions 9 and 10.

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

Answer: Yes: if we introduce a new register name (r8) for the reuse of r3 at instruction 8 then we can move the second load downward, where it becomes instruction 9:

```
1.    r1 := I
2.    r4 := &A
3.    r2 := r1 × 4
4.    r3 := r4 + r2
5.    r3 := *r3
6.    r6 := &B
7.    r8 := r6 + r1
8.    r8 := *r8
9.    r5 := *(r3 + 12)   –– filled this slot
10.   r7 := *(r8 + 12)   –– still a delay after this load
11.   r3 := r5 + r7
12.   S := r3
```

Without a longer window to consider, there is still no way to eliminate the delay after instruction 10.

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?

Answer: Backward compatibility. If we eliminate delayed loads from an architecture, old programs will still run correctly; the `nops` will be harmless. (Of course, new programs will not run correctly on old machines, because they lack the `nops`.) By contrast, if we eliminate delayed branches from an architecture, then old programs will no longer run correctly: they won't execute the instructions in the delay slots.

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?